# Spring Data JPA

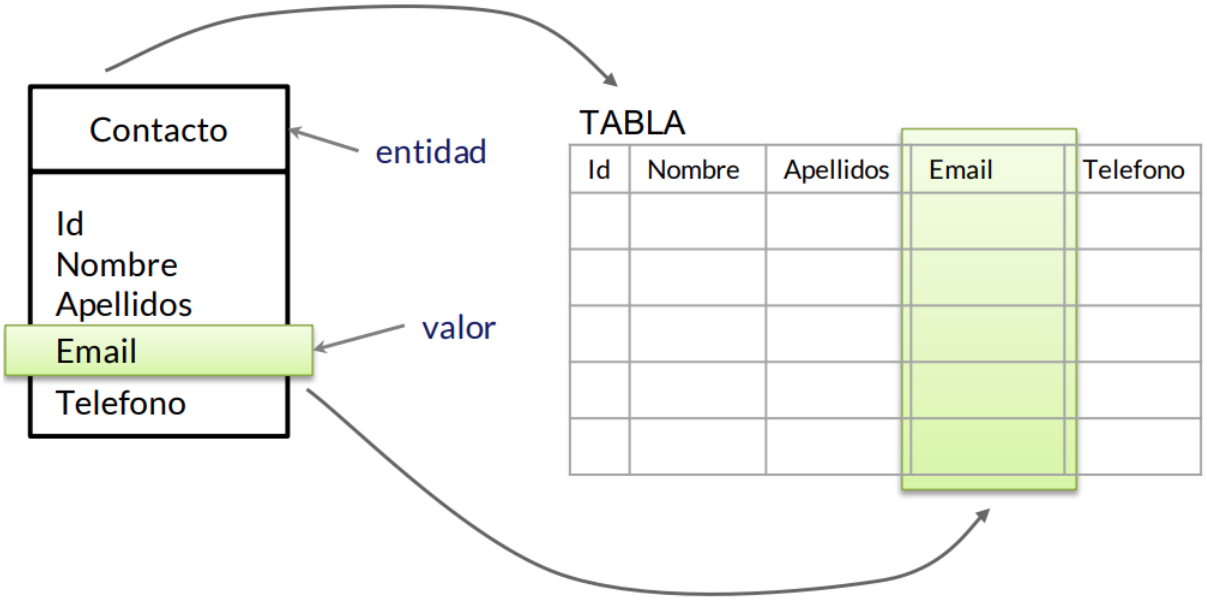
En Java usamos objetos (n-dimensiones)
Las base de datos, tablas (2-dimensiones)

¿Cómo salvar este desfase?
Con **ORM** (Object-Relational Mapping)

## Spring Data

- Ofrece un modelo consistente de programación para acceder a datos.
- Facilita el uso de bases de datos relacionales y NoSQL.
- Proyecto paraguas para muchos subproyectos.

## Mapeo de Objetos a Tablas




## Entidad

- Clase (POJO) Java con @Entity
- Debe tener (al menos) un atributo que lo identifique (Clave Primaria en bases de datos) con @Id.

Con @Entity se mapea con una tabla que se llama igual que la clase.

Añadiendo @Table antes de la clase, podemos cambiar el nombre de la tabla.

Con @Column encima del atributo podemos cambiar el nombre de una columna.

## Requisitos de la clase Entidad (Documentación Hibernate)

- La clase de entidad debe anotarse con la `javax.persistence.Entity` anotación (o indicarse como tal en el mapeo XML).
- La clase de entidad debe tener un **constructor sin argumentos** público o protegido. También puede definir constructores adicionales.
- La clase de entidad debe ser una **clase de nivel superior**.
- Una **enumeración o interfaz no** se puede designar como una **entidad**.
- La clase de entidad **no** debe ser **definitiva**. Ningún método o variable de instancia persistente de la clase de entidad puede ser final.
- Si una instancia de entidad se va a utilizar de forma remota como un objeto separado, la clase de entidad debe implementar la `interfaz Serializable`.
- Tanto las clases abstractas como las concretas pueden ser entidades. Las entidades pueden ampliar las clases que no son entidades así como las clases de entidades, y las clases que no son entidades pueden ampliar las clases de entidades.
- El estado persistente de una entidad está representado por variables de instancia, que pueden corresponder a propiedades de estilo JavaBean. Se debe acceder directamente a una variable de instancia sólo desde dentro de los métodos de la entidad por la propia instancia de la entidad. El estado de la entidad está disponible para los clientes solo a través de los métodos de acceso de la entidad (métodos getter / setter) u otros métodos comerciales.

**Hibernate**, sin embargo, no es tan estricto en sus requisitos. Las diferencias de la lista anterior incluyen:

- La clase de entidad debe tener un constructor sin argumentos, que puede ser público, protegido o con visibilidad de paquete. También puede definir constructores adicionales.
- La clase de entidad *no necesita* ser una clase de nivel superior.
- Técnicamente, Hibernate puede persistir clases finales o clases con métodos de acceso de estado persistente final (getter / setter). Sin embargo, generalmente no

es una buena idea, ya que al hacerlo evitará que Hibernate pueda generar proxies para la carga diferida de la entidad.

- Hibernate no restringe al desarrollador de aplicaciones exponer variables de instancia y hacer referencia a ellas desde fuera de la propia clase de entidad. La validez de tal paradigma, sin embargo, es discutible en el mejor de los casos.

Dependencias: Spring Web, H2 Database, JPA y Lombok

Spring Data JPA busca en la carpeta "resources" algún archivo .sql y lo trata de ejecutar automáticamente.

## HashCode() e Equals()

La nada más absoluta

## Asociaciones

La lógica de negocio son los **procesos propios** de un negocio o programa. Un proceso propio son todos los pasos que se siguen en una función o proceso (suena redundante, pero es que es así).
Esta parte del programa pertenece a la capa de servicios (@Service).

**Repository → @Service → @Controller**

**¡¡ OJO !¡** : Clase Abstracta es aquella de la que no se pueden crear instancias.

La clase de servicio base tiene que ser abstracta.

**Teoría sobre genéricos:**
https://docs.oracle.com/javase/tutorial/java/generics/index.html

En nuestra clase BaseService, la cual tiene que ser abstracta, debemos definir los siguientes genéricos:

```
public abstract class BaseService<T,ID,R extends JpaRepository<T,ID>> {
    |
```

- **T** → Tipo de la entidad
- **ID** → Tipo del ID
- **R** → Tipo del repositorio

Inyectamos la dependencia del repositorio de la siguiente forma:

```
@Autowired
protected R repositorio;
```

En este caso usaremos @Autowired para no tener que crear constructores. Además, su acceso será protegido de forma que podrán acceder a él las clases dentro del mismo paquete y aquellas que extiendas a BaseService.

Todos los métodos que añadamos a BaseService estarán disponibles en los demás servicios gracias a la herencia.

Para crear el servicio de Alumno, deberemos de crearlo bajo el directorio services creado, para lo cual lo añadiremos en el directorio base y modificaremos su ruta, obligando a IDEA a que mueva nuestro fichero de ruta

```
package com.salesianostriana.dam.springdata.services;
```

```
@Service
public class AlumnoService extends BaseService<Alumno,Long, AlumnoRepository> {
```

La clase servicio que creemos para nuestra entidad irá anotada con @Service y extenderá a BaseService.

- **¿Para qué sirve la clase BaseService?**
  Para no tener que usar los repositorios directamente en los controladores.

**Teoría asociaciones de Hibernate:**
https://docs.jboss.org/hibernate/orm/5.6/userguide/html_single/Hibernate_User_Guide.html#associations

## Bidireccional (ManyToOne y OneToMany)

Opción que recomienda y le gusta a Luismi.

Sólo implementar la asociación One to Many = ERROR!

La mejor es realizar la bidireccionalidad a través de consultas. Estas consultas se realizan del lado OneToMany.

Ejemplo: Conocer los alumnos de un curso.

***public List<Alumno> findByCurso( )***

Se recomienda el uso de List para colecciones.

Cuando implementamos la asociación bidireccionalmente, es necesario añadir a la anotación @OneToMany el atributo mappedBy = "nombre del atributo anotado con @ManyToOne de la entidad opuesta" .

```java
// En mappedBy ponemos el atributo de la clase de enfrente que esté anotado con @ManyToOne
@OneToMany(mappedBy = "curso")
private List<Alumno> alumnos;
```

```java
@ManyToOne
private Curso curso;
```

Para que no salga un error a la hora de implementar la lista de alumnos en este caso, habría que inicializar la lista directamente al declararla.

```java
@OneToMany(mappedBy="curso")
private List<Alumno> alumnos = new ArrayList<>();
```

Para guardar posteriormente los datos de ambas asociaciones, usamos unos métodos llamados **"helper".** Estos nos permiten añadir los datos en ambos extremos de la asociación a la vez, ahorrando tiempo y esfuerzo. Estos métodos es mejor implementarlos en la parte de la entidad propietaria **(es decir, el lado NO mapeado)**, aunque tenga menos sentido semánticamente, pero tiene más lógica así. Básicamente, el que no tenga como atributo la lista de elementos de la otra clase.

La colección de entidades en lado Uno de la asociación es virtual (mapeada), es decir, no existe a nivel de base de datos, si no que Hibernate nos la proporciona. Esto quiere decir que el único lado que se guarda en la base de datos es el lado Muchos, por ello es el lado propietario. Por esta razón, esta base de datos está normalizada.

```
public void addCurso(Curso c) {
    this.curso = c;
    c.getAlumnos().add(this);
}

public void removeCurso(Curso c) {
    c.getAlumnos().remove(this);
    this.curso = null;
}
```

Después de escribir estos métodos, debemos ejecutar el método en el Main o en el Init Data si lo tenemos.

Tendríamos que crear un nuevo Alumno y añadirlo al Curso de la siguiente forma:

***alumno.addCurso(2dam);***

Como la asociación es bidireccional, no sería necesario realizar ninguna otra operación para establecer el Alumno en el Curso.

**No poner comillas dobles al nombre de las columnas en las sentencias SQL en H2.**

Si se acompaña un número con "L", el valor numérico se convierte en tipo Long. ⇒ Útil para usar los IDs de las tablas.

***dam2.ifPresent(c -> System.out.println("Curso: " + c.getNombre( ) ) );***

*dam2 debe ser un Optional para usar este método.*

# FETCHING

Un **fetching** es una estrategia para cargar asociaciones.

Un fetching tipo **LAZY** ejecutará las consultas adicionales si la conexión con la base de datos está abierta y se requiera el recurso explícitamente. Es el fetching por defecto en aquellas asociaciones **x To Many = donde hay una colección.**

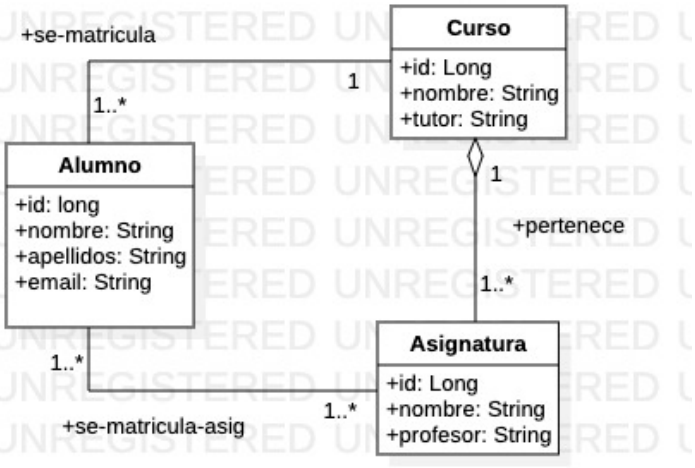
Un fetching tipo **EAGER** ejecutará las consultas adicionales siempre, trayendo todos los recursos asociados, se hayan requerido o no. Es el fetching por defecto en aquellas asociaciones **x To One.**

Para evitar operaciones innecesarias y mejorar la eficiencia del acceso a la base de datos, cambiaremos el tipo de fetching del lado One a EAGER. Esto se hará siempre y cuando sea necesario, **no** es la mejor estrategia a utilizar siempre, ya que puede afectar negativamente a la eficiencia de la base de datos en ciertos casos.

# ASOCIACIONES MANY TO MANY

Una asociación ***muchos-a-muchos*** es un tipo de *relación* entre clases, donde una instancia de uno de los tipos conectados se puede asociar a muchas instancias del otro tipo, y viceversa. Uno de los ejemplos clásicos suele ser el de las clases Alumno y Asignatura, mediante la cual un Alumno puede estar matriculado muchas Asignaturas, y una Asignatura puede tener matriculado a muchos Alumnos.

Aunque esta asociación tenga la misma *multiplicidad* en ambos extremos, también puede tratarse de forma bien unidireccional, o bien bidireccional.



En el lado propietario debemos tener una colección de entidades del otro lado (generalmente un List<Entity>), anotado con **@ManyToMany**.

```
@ManyToMany
List<Entity> myList = new ArrayList<>();
```

Este tipo de asociaciones genera una nueva tabla en el DDL, cuya clave primaria está compuesta por las claves primarias de ambas tablas como claves externas. Esta tabla se conoce como tabla Join.

Creamos la clase Asignatura, con sus respectivos repositorio y servicios. Mientras, en la clase Alumno, creamos una nueva lista de asignaturas con la anotación **@JoinTable**:

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

Podríamos añadir a la notación @JoinTable una opción name = “matriculaciones” para modificar el nombre de la tabla generada.

Una vez modificada la asociación, añadiríamos los métodos helpers, en el caso de que esta asociación fuera bidireccional.

```
/** MÉTODOS HELPERS **/

public void addAsignatura(Asignatura a) {
        asignaturas.add(a);
        a.getAlumnos().add(this);
}

public void removeAsignatura(Asignatura a) {
        asignaturas.remove(a);
        a.getAlumnos().remove(this);
}
```

Si fuese unidireccional, haremos un alumno.getAsignaturas().addAll() y posteriormente guardaríamos ese alumno con alumnoService.edit(alumno).

## ASOCIACIONES MANY TO MANY BIDIRECCIONALES

En Asignaturas, crearíamos un atributo que sea una lista de Alumnos, mapeado por asignaturas.

Más tarde, en la entidad propietaria (Alumno), colocaríamos los métodos helpers de la asociación bidireccional con Asignatura.

En estos métodos, hacemos un if para comprobar que, tanto la lista de alumnos, como la de asignaturas, no están vacías o son null.

Mostraríamos las asignaturas con esta línea de código (adolescente vs. mili en Ceuta):

**asignaturas.forEach( alumno -> alumno.addAsignatura(asignatura) );**

**asignaturas.forEach(alumno::addAsignatura);**

Si hay un método que nos sirve para varias consultas, podemos implementarlo directamente en BaseService, como el saveAll() para guardar todos los alumnos.

Para evitar ciertos problemas, ponemos el fetch tipo eager en las dos partes de la asociación.

---

### Inciso

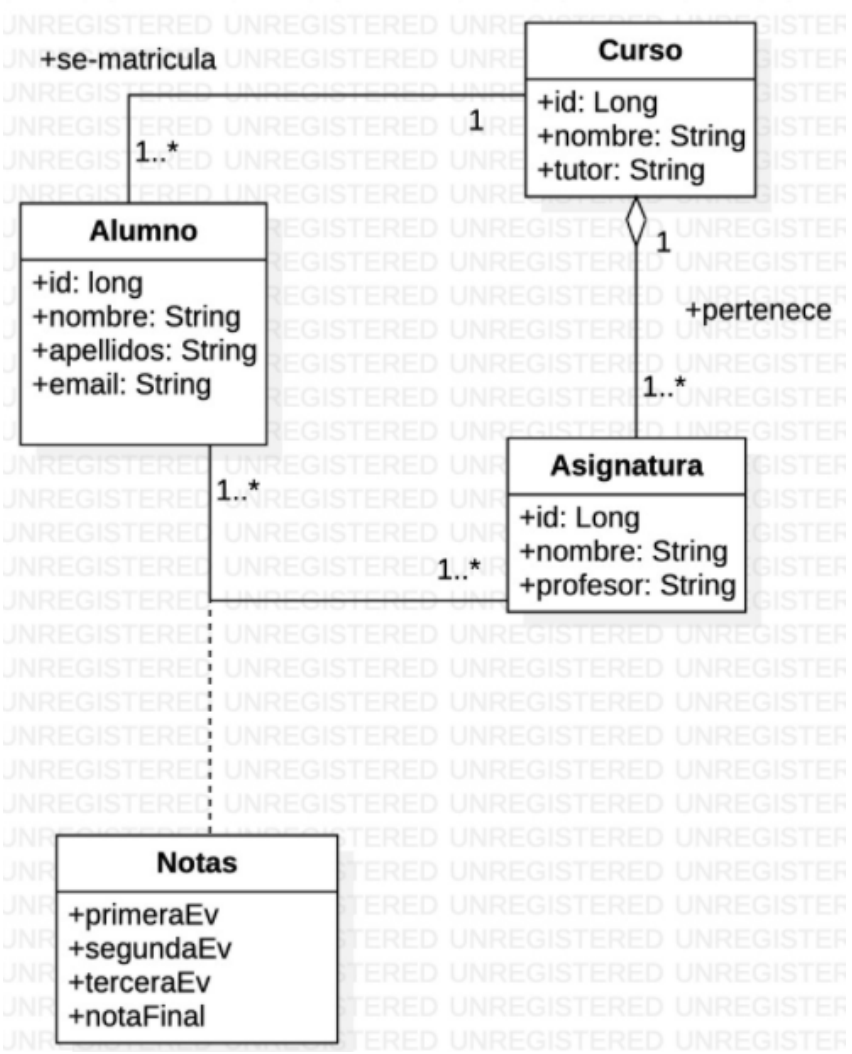
Para depurar:

- Añadir una nueva flag en configuración de ejecución, dentro de command line:
- ***spring-boot:run -Dspring-boot.run.fork=false***
- Añadir puntos de ruptura.
- Pulsar la cucaracha.

---

## ASOCIACIÓN MANY TO MANY CON CLASE DE ASOCIACIÓN

Podemos romper nuestras asociaciones @ManyToMany con una nueva entidad que haga de clase de asociación, haciendo que se queden como dos asociaciones ManyToOne.




**@Builder.Default**: la podemos poner a nivel de atributo. Si no se le proporciona nada, se toman los valores que tienen anotados el Builder por defecto.

## Tipos Embeddable

En alguna ocasión, puede interesarnos agrupar un cierto conjunto de valores:

Supongamos que queremos manejar la **localización** de una determinada **Oficina**.

Una **localización** está formada por una *dirección*, una *ciudad* y un *país*. Realmente no queremos tratar una localización como una entidad; su ciclo de vida siempre estará asociado al de la Oficina correspondiente.

Nos puede interesar agrupar todo, por ejemplo, para dar un tratamiento integral.

JPA nos ofrece la anotación **@Embeddable**, que nos permite generar una clase que será *encajable* ( *incrustable*, *embebible*) en otra entidad.

Estos tipos incrustables nos permiten definir una clave primaria compuesta.

Para hacer esto primero crearíamos una clase que implemente Serializable, cuyos atributos son los ids de ambas clases que generan la clase de asociación.

Posteriormente, en nuestra clase de asociación, crearíamos un ID cuyo tipo es el de la clase anterior.

```java
@Embeddable
public class NotasPK {

    long alumno_id;
    long asignatura_id;

}
```

```java
public class MatriculaNotas implements Serializable {

    @EmbeddedId @Builder.Default
    private NotasPK id = new NotasPK();
```

De esta forma podemos tratarlo como un todo y definir sus atributos a la vez.

**@EmbeddedId** define el ID de nuestra clase incrustada como clave primaria de la misma, lo cual nos va a permitir tratarla como clave externa de la clase de asociación.

La usaremos principalmente para representar claves compuestas.

Rompemos la asociación Many To Many en dos asociaciones One To Many, donde el lado de muchos se encuentra en ambas relaciones en el lado de la clase de asociación.

Para representar esto en código, deberemos de crear ambas asociaciones en la clase de asociación:

```java
public class MatriculaNotas implements Serializable {

    @EmbeddedId @Builder.Default
    private NotasPK id = new NotasPK();

    @ManyToOne
    @MapsId("alumno_id")
    @JoinColumn(name="alumno_id")
    private Alumno alumno;

    @ManyToOne
    @MapsId("asignatura_id")
    @JoinColumn(name="asignatura_id")
    private Asignatura asignatura;
```

**@MapsId("nombre_atributo")** le indica al programa que busque el id de la entidad dentro de la clase creada para guardar las claves externas de la asociación, donde se han definido los atributos("nombre_atributo") como claves primarias. Este atributo va a tener el mismo valor que el Long que hace de id de NotasPK.

Para definir la bidireccionalidad, crearemos dos relaciones **@OneToMany** en las clases antiguas (Alumno y Curso).

```java
@Builder.Default
@OneToMany(mappedBy="alumno" , fetch = FetchType.EAGER)
private List<MatriculaNotas> matriculaNotas = new ArrayList<>();
```

```java
@OneToMany(mappedBy="asignatura", fetch = FetchType.EAGER)
private List<MatriculaNotas> matriculaNotas = new ArrayList<>();
```

Asimismo, debemos crear los métodos helper dentro de la clase de la asociación, en este caso, notas. Tendremos que crear los métodos para cada una de las entidades a cada lado, o crearlos para ambas a la vez (esto sería más complicado si las asociaciones fueran entre más entidades).

Insertar imágenes después

## ASOCIACIÓN ONE TO MANY CON COMPOSICIÓN

Una asociación de composición puede venir representada por el siguiente diagrama:


Esta asociación es una asociación ***uno-a-muchos*** que podemos tratar bidireccionalmente, para que su gestión sea más eficiente; sin embargo, el hecho de que sea de composición le da un tinte algo especial.

En esta asociación, el *componente* es una entidad débil; no tiene sentido su existencia fuera del ámbito de una instancia de un *compuesto*. Por tanto, la gestión de cada componente debe ir asociada a la gestión del compuesto.

Por ejemplo, con las entidades Pedido y LineaDePedido, solo necesitamos crear el repositorio y los servicios de Pedido, que sería el compuesto de la asociación entre ambos.

### Operaciones en cascada

JPA nos permite realizar operaciones en cascada con entidades.

¿Cómo conseguimos hacer estas operaciones? Las anotaciones como @OneToMany pueden recibir algunos argumentos además de mappedBy. Entre ellos están los siguientes:

- **Cascade**: Podemos indicar qué tipo de operaciones en cascada queremos que se realicen al trabajar con esta entidad. Debe ser un valor de la enumeración *javax.persistence.CascadeType*, a saber: ***ALL***, ***PERSIST***, ***MERGE***, ***REMOVE***, ***DETACH***.
- **orphanRemoval**: Propiedad booleana que permite indicar que si una entidad a la que hace referencia la anotación de asociación (por ejemplo, @OneToMany) pierde *su clave externa* (es decir, la entidad con la que está asociada, y por tanto *queda huérfana*) se eliminará.

*En el repositorio de Luismi, la clase “Compuesto” corresponde a Producto y la clase “Componente” corresponde a LineaDePedido*

## FETCHING.2

Como decíamos en apartados anteriores, JPA provee dos tipos de estrategia de fetching, es decir, de carga de datos de asociaciones (como OneToMany, ManyToMany, ...)

- FetchType.LAZY
- FetchType.EAGER

Al traer un usuario siempre trae todos los datos asociados, lo cual no siempre es conveniente (usando el fetch eager)

La segunda dificultad es que, en el fetching tipo EAGER, se hacen muchas llamadas a la base de datos con las consultas, ya que JPA rescata las entidades asociadas de las clases que estamos consultando. De esta forma, cuanto más datos consultemos, mayor será el número de sentencias que traerá.

### ¿Cómo solucionamos este problema?

## Grafos de entidad (*Entity Graphs*)

A través de grafos de entidad, los cuales nos permiten representar asociaciones entre entidades con distintos tipos de fetching en diferentes tipos de consulta.

Nos permiten establecer fetchings dinámicos, que aunque se eligen en tiempo de compilación, no necesariamente tienen que ser siempre los mismos.

Cuando una entidad tiene referencias a otras entidades podemos especificar el fetch mediante grafos de entidad para establecer qué atributos o propiedades se tomarán juntos. Para ello se utiliza la anotación **@NamedEntityGraph** a nivel de clase.

**@NamedAttributeNode**( atributo de la entidad → asociación de tipo LAZY ) Se usa para los atributos que forman el grafo.

```java
@NamedEntityGraph(
        name = "grafo-direccion-ciudad-usuario",
        attributeNodes = {
                @NamedAttributeNode("ciudad"),
                @NamedAttributeNode("usuario")
        }
)
@Entity
@NoArgsConstructor @AllArgsConstructor @Builder
@Getter @Setter
public class Direccion {

    @Id
    @GeneratedValue
    private Long id;

    private String tipo, calle, piso, codigoPostal;

    @OneToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "ciudad_id")
    private Ciudad ciudad;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "usuario_id")
    private Usuario usuario;
```

Con una estrategia estática de tipo **LAZY**, podemos definir un grafo de entidad, llamado *grafo-direccion-ciudad-usuario*. Este grafo de entidad contiene los atributos ciudad y el usuario, por tanto, estos objetos serán tomados de la base de datos conjuntamente con la direccion.

Si queremos obtener estos atributos, el método findAll ya no nos sirve, puesto que ejecutaría el *fetch* de tipo LAZY. Para asignar el grafo de entidad con nombre a una consulta tenemos dos estrategias:

- **Spring Data JPA**: hacerlo a través de la anotación @EntityGraph en los repositorios.
- **JPA**: hacerlo programáticamente a través de EntityManager.

Si lo hacemos con Spring Data JPA quedaría así:

```
public interface DireccionRepository extends JpaRepository<Direccion, Long> {

    @EntityGraph("grafo-direccion-ciudad-usuario")
    List<Direccion> findByIdNotNull();

}
```

```
Hibernate:
    select
        direccion0_.id as id1_1_0_,
        ciudad1_.id as id1_0_1_,
        usuario2_.id as id1_2_2_,
        direccion0_.calle as calle2_1_0_,
        direccion0_.ciudad_id as ciudad_i6_1_0_,
        direccion0_.codigo_postal as codigo_p3_1_0_,
        direccion0_.piso as piso4_1_0_,
        direccion0_.tipo as tipo5_1_0_,
        direccion0_.usuario_id as usuario_7_1_0_,
        ciudad1_.nombre as nombre2_0_1_,
        usuario2_.email as email2_2_2_,
        usuario2_.nombre as nombre3_2_2_,
        usuario2_.password as password4_2_2_,
        usuario2_.telefono as telefono5_2_2_
    from
        direccion direccion0_
    left outer join
        ciudad ciudad1_
            on direccion0_.ciudad_id=ciudad1_.id
    left outer join
        usuario usuario2_
            on direccion0_.usuario_id=usuario2_.id
    where
        direccion0_.id is not null
```

Podemos comprobar como, en lugar de realizar n+1 consultas, automáticamente ha generado una consulta de tipo JOIN.

Se puede tener más de un método diferente de consulta con el mismo grafo de entidad. Por ejemplo, para buscar las direcciones de un usuario o de una ciudad.

```
public interface DireccionRepository extends JpaRepository<Direccion, Long> {

    @EntityGraph("grafo-direccion-ciudad-usuario")
    List<Direccion> findByIdNotNull();

    @EntityGraph("grafo-direccion-ciudad-usuario")
    List<Direccion> findByUsuarioId(Long id);
}
```

También podemos definir múltiples grafos de entidad con nombre para una única entidad.

```
@NamedEntityGraphs({
        @NamedEntityGraph(
                name = "grafo-direccion-ciudad-usuario",
                attributeNodes = {
                        @NamedAttributeNode("ciudad"),
                        @NamedAttributeNode("usuario")
                }
        ),
        @NamedEntityGraph(
                name = "grafo-direccion-ciudad",
                attributeNodes = {
                        @NamedAttributeNode("ciudad"),
                }
        ),
        @NamedEntityGraph(
                name = "grafo-direccion-usuario",
                attributeNodes = {
                        @NamedAttributeNode("usuario")
                }
        ),
```

## Tipos de grafos de entidad

La anotación **@EntityGraph** tiene un parámetro type que puede tomar dos valores:

- **FETCH**. Es el valor por defecto. Cuando es elegido, los atributos indicados como nodos se tratan con el fetching de tipo EAGER, y los atributos que no están especificados como nodos, se tratan con fetching de tipo LAZY.
- **LOAD**. Cuando este tipo es seleccionado, los atributos especificados como nodos se tratan con fetching de tipo EAGER, y a los que no lo están se les aplica el fetching que tengan definido, o el tipo de fetching por defecto para esa asociación.

## Subgrafos

Hacer asociaciones bidireccionales

## Fechas

Fechas de creación y edición → Fechas de auditoría.

https://turkogluc.com/spring-data-jpa-auditing/

Podemos hacer uso de algunas anotaciones como **@CreatedBy**, **@CreatedDate**, **@LastModifiedDate**, **@LastModifiedBy** para indicar a Spring JPA que complete dichos campos. Ejemplo:

```
@CreatedDate
private LocalDateTime fechaCreacion;

@LastModifiedDate
private LocalDateTime fechaModificacion;
```

Para poder hacer uso de esta funcionalidad necesitamos añadir a la clase Application la anotación **@EnableJpaAuditing**

```
@SpringBootApplication
@EnableJpaAuditing
public class Application {
```

Y a la entidad la anotación **@EntityListeners(AuditingEntityListener.class)**

```
@AllArgsConstructor @Builder
@EntityListeners(AuditingEntityListener.class)
public class Alumno implements Serializable {
```

```
@Entity
public class Category {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Long title;

    @CreatedBy
    private Long createdBy;

    @CreatedDate
    private LocalDateTime createdDate;

    @LastModifiedBy
    private Long lastModifiedBy;

    @LastModifiedDate
    private LocalDateTime lastModifiedDate;

    // getters and setter..
}
```

LocalDateTime > LocalDate

Crear un bean que implemente AuditoryAware<?> donde en el genérico va una referencia a una entidad.

# HERENCIA

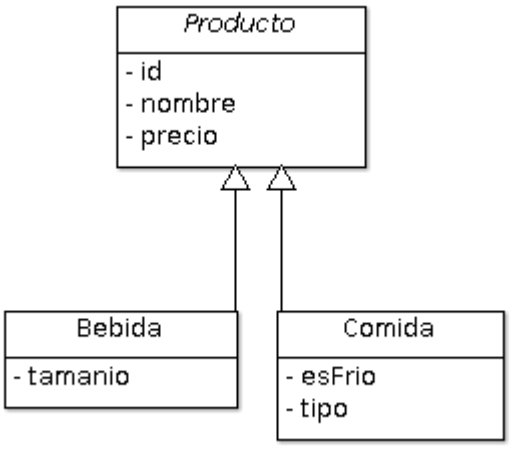
La herencia es un mecanismo de la Orientación a Objetos que nos permite indicar que dos objetos tienen un tipo de asociación especial, llamada *es-un*. De esta forma, creamos una clase a partir de otra.

JPA nos ofrece varios mecanismo para poder implementar la herencia, que nos servirán en función del problema a resolver y el resto de asociaciones que tengan las entidades:

- ***Mapped SuperClass***: nos permite utilizar una clase como base para su extensión, pero esta clase no será mapeada a la base de datos.
- ***Single Table***: se traslada la jerarquía de herencia como una sola tabla, utilizando una columna como *discriminante*.
- ***Joined***: se traslada una tabla para cada una de las clases de la jerarquía de la entidad, realizándose operaciones join.
- ***Table Per Class***: se traslada una tabla para cada una de las clases de la jerarquía de la entidad, pero sin uso de operaciones join.

## Herencia con @MappedSuperclass

Utilizaremos este primer tipo de herencia en aquellas situaciones en que tengamos más de una entidad que herede de una clase base, pero no necesitemos que esta sea una entidad. Es decir, si se da la circunstancia de que una serie de entidades comparten algunos atributos, y queremos tener estos en una clase base, pero esta (siendo o no abstracta) no queremos gestionarla como entidad (es decir, no vamos a hacer consultas sobre ella, no estará asociada con otra entidad, ...) podemos utilizar este tipo de herencia.




Sea el diseño de clases de nuestro ejemplo:

En este caso, la clase base Producto es abstracta, y no queremos mapearla como entidad.

En tal caso, podemos utilizar la anotación **@MappedSuperclass**:

```
@NoArgsConstructor
@MappedSuperclass
public abstract class Producto {

	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private long id;

	private String nombre;

	private float precio;

	public Producto(String nombre, float precio) {
		this.nombre = nombre;
		this.precio = precio;
	}
}
```

Como podemos observar, aunque no sea una entidad, podemos utilizar las anotaciones @Id y las complementarias, para establecer que uno de los atributos comunes a las dos clases derivadas, Bebida y Comida, será la clave primaria.

Como es una clase abstracta, no necesita todas las anotaciones de lombok que venimos utilizando hasta ahora. Tan solo añadimos @NoArgsConstructor, para que las clases derivadas también la puedan utilizar.

Al no ser entidad, no lleva @Entity ni tampoco tiene repositorios y servicios. Solo habría que crear dichas clases para sus clases hijas.

## Clases extendidas

El diseño de las clases extendidas sería como sigue:

```
@Getter @Setter @NoArgsConstructor
@EqualsAndHashCode(callSuper = true)
@ToString(callSuper = true)
@Entity
public class Bebida extends Producto{

        // Tamaño de la bebida expresado en centilitros
        private int tamanio;

        public Bebida(String nombre, float precio, int tamanio) {
                super(nombre, precio);
                this.tamanio = tamanio;
        }
}
```

```
@Getter @Setter @NoArgsConstructor
@EqualsAndHashCode(callSuper = true)
@ToString(callSuper = true)
@Entity
public class Comida extends Producto {

        private boolean esFrio;

        private String tipo;

        public Comida(String nombre, float precio, boolean esFrio, String tipo) {
                super(nombre, precio);
                this.esFrio = esFrio;
                this.tipo = tipo;
        }

}
```

De las clases extendidas podemos observar:

- Extienden la clase base, Producto.
- Definen un constructor con argumentos (todos, salvo el marcado con @Id) que necesita del constructor de la clase base (invocado con super).
- No tienen las anotaciones de lombok tal y como las venimos usando hasta ahora.
- Tanto en la anotación **@EqualsAndHashCode** como en **@ToString**, especificamos que el ***callSuper*** sea true. **NO LAS PONEMOS**

## SUPERBUILDER!

Cuando usamos herencia en java, nos encontramos con el problema de que al usar los métodos de @Builder, no nos permite acceder a los métodos y atributos de la herencia. Esto lo resolveremos de la siguiente manera:

https://www.baeldung.com/lombok-builder-inheritance

```java
@Getter
@AllArgsConstructor
public class Parent {
    private final String parentName;
    private final int parentAge;
}

@Getter
public class Child extends Parent {
    private final String childName;
    private final int childAge;

    @Builder
    public Child(String parentName, int parentAge, String childName, int childAge) {
        super(parentName, parentAge);
        this.childName = childName;
        this.childAge = childAge;
    }
}
```

Para diferenciar entre múltiples @Builder:

```
@Getter
public class Child extends Parent {
    private final String childName;
    private final int childAge;

    @Builder(builderMethodName = "childBuilder")
    public Child(String parentName, int parentAge, String childName, int childAge) {
        super(parentName, parentAge);
        this.childName = childName;
        this.childAge = childAge;
    }
}
```

En el Main de Prueba construiremos el objeto de la siguiente manera:

```
Student student = Student.studentBuilder()
  .parentName("Andrea")
  .parentAge(38)
  .childName("Emma")
  .childAge(6)
  .schoolName("Baeldung High School")
  .build();
```

Para resolver este problema de forma más sencilla, usamos @SuperBuilder:

```
@Getter
@SuperBuilder
public class Parent {
    // same as before...

@Getter
@SuperBuilder
public class Child extends Parent {
   // same as before...

@Getter
@SuperBuilder
public class Student extends Child {
   // same as before...
```

De esta forma detecta automáticamente cuando se trata de una clase padre o una clase hija.

Así, podríamos construir una instancia de un objeto de la siguiente manera:

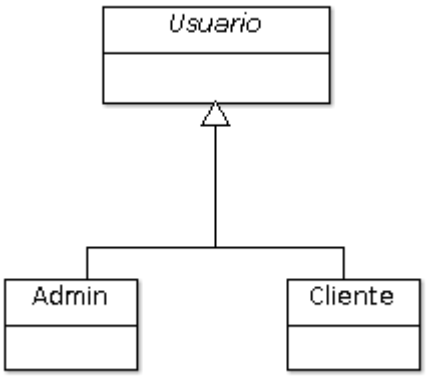
```
Student student = Student.builder()
  .parentName("Andrea")
  .parentAge(38)
  .childName("Emma")
  .childAge(6)
  .schoolName("Baeldung High School")
  .build();
```

## Herencia de tipo Single Table

Utilizaremos este segundo tipo de herencia en aquellas situaciones en las que tengamos una jerarquía de herencia en la que las clases hijas no añaden atributos nuevos, o añaden muy pocos (uno o dos, como mucho).

A nivel de base de datos, todas las clases de la jerarquía se mapean en una sola tabla.

Diseño de el ejemplo:




La única diferencia que tenemos con respecto al tipo de herencia anterior es la notación **@Inheritance** dentro de la clase base:

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class Usuario {
```

Para las clases hijas, deberemos de añadir la notación **@DiscriminationValue**

```
@Entity
@DiscriminatorValue("A")
public class Admin extends Usuario {
```

De esta forma JPA ha añadido una columna adicional en la base de datos, llamada **DTYPE**.

Esta columna es la que nos permitirá almacenar un valor que discrimine o diferencie los distintos tipos de Usuario.

Por defecto se usa el nombre de la entidad, pero podemos establecer un valor a través de la anotación **@DiscriminatorValue.**

## ¿Es necesario el repositorio / servicio de la clase base?

Si deseamos definir un repositorio base, del cual extenderemos, pero que no vamos a usar directamente, podemos usar la anotación **@NoRepositoryBean**. Esta anotación indica a Spring que no debe crear el bean para este repositorio.

```
@NoRepositoryBean
public interface BaseUsuarioRepository<T extends Usuario> extends JpaRepository<T, Long> {

        public T findByUsername(String Username);

}
```

Por tanto, los repositorios de las clases hijas extenderán a este repositorio.

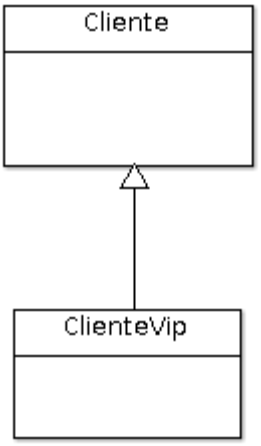
```
public interface ClienteRepository extends BaseUsuarioRepository<Cliente>{

}
```

## Herencia de tipo JOINED

También se conoce como *table per subclass*.

En este caso, JPA va a generar una tabla para la clase *base*, con todos sus atributos; y también una tabla para cada una de las subclases, que incluirá una clave externa a la tabla de la clase base, así como los posibles atributos propios que tenga.

Ejemplo:




En este caso, nos interesa poder tener instancias de ambas entidades: tendremos clientes y clientes vip.

En la clase base solo cambiaremos la notación de herencia.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Cliente {
```

Esta herencia trae un nuevo problema en las consultas, ya que al hacer un SELECT de la clase madre, nos traeríamos también la clase hija. Para evitarlo, utilizaremos en la consulta una palabra TYPE( ) dónde ponemos el nombre que le hemos dado a la clase en la consulta. Es decir, si en la consulta hemos puesto SELECT c FROM Cliente c, pondríamos TYPE(c).

```java
public interface ClienteRepository extends JpaRepository<Cliente, Long>{

        @Query("select c from Cliente c where TYPE(c) = Cliente")
        public List<Cliente> clientesNoVip();

}
```

# Consultas con Spring Data JPA: *Query methods*

## Consulta a través de métodos

Podemos crear métodos de consulta que utilizan esta estrategia siguiendo estas reglas:

- El nombre de nuestro método de consulta debe comenzar con uno de los siguientes prefijos: **find…By**, **read…By**, **query…By**, **count…By**, y **get…By**.
- Si queremos **limitar** el número de resultados devueltos, podemos agregar la palabra clave **First** o **Top** antes de By. Si queremos obtener más de un resultado, debemos agregar un valor numérico. Por ejemplo, **findTopBy** , findTop1By, findFirstBy y findFirst1By devuelven la primera entidad que coincida con los criterios de búsqueda especificados.
- Si queremos **seleccionar resultados únicos**, debemos agregar la palabra clave **Distinct** antes de By. Por ejemplo, **findTitleDistinctBy** o **findDistinctTitleBy** significa que queremos seleccionar todos los títulos únicos que se encuentran en la base de datos.
- Debemos **agregar los criterios de búsqueda** de nuestro método de consulta después de By . Podemos especificar los criterios de búsqueda mediante la combinación de expresiones de propiedades con las palabras clave admitidas.
- Si nuestro método de consulta especifica x condiciones de búsqueda, debemos agregar X parámetros al método . En otras palabras, el número de parámetros del método debe ser igual al número de condiciones de búsqueda. Además, los parámetros del método deben darse en el mismo orden que las condiciones de búsqueda.
- El tipo de retorno podrá ser:
    - Para consultas que devuelven un resultado:

        - **Tipo básico** . Nuestro método de consulta devolverá el tipo básico encontrado o nulo .
        - **Entidad** . Nuestro método de consulta devolverá un objeto entidad o nulo .
        - **Optional<T>** . Nuestro método de consulta devolverá un opcional que contiene el objeto encontrado o un opcional vacío.
    - Para consultas que devuelven más de un resultado:
        - **List<T>** : Nuestro método de consulta devolverá una lista que contiene los resultados de la consulta o una lista vacía.
        - **Stream<T>** : Nuestro método de consulta devolverá un stream que se puede utilizar para acceder a los resultados de la consulta o un stream vacío.

Es habitual necesitar la anotación **@Transactiona**l cuando utilicemos Stream<T> como tipo de retorno de una consulta en un servicio.

```
@Service
public class AlumnoServicio extends BaseService<Alumno, Long, AlumnoRepository>{

        // Resto del código

        /**
         * Obtiene desde el repositorio como mucho 3 alumnos cuyo primer o segundo
         * apellido sea el proporcionado.
         * @return Una lista con el nombre completo de los alumnos como String.
         */
        @Transactional
        public List<String> nombreAlumnosContieneApellido(String apellido) {
                return repositorio
                                .findTop3ByApellido1ContainsOrApellido2Contains(apellido, apellido)
                                .map(a -> a.getNombre() + " " + a.getApellido1() + " " + a.getApellido2())
                                .collect(Collectors.toList());
        }

}
```

## Specification y JpaSpecificationExecutor

Interfaces que usaremos con las Criteria Query. Dentro de ellas, destacan métodos como ToPredicate, que nos facilitará hacer consultas con WHERE.

https://dc722jrlp2zu8.cloudfront.net/media/attachments/academy_lesson/7370/02_Manejo_de_parámetros_en_el_query.pdf la presentación de luismi el enlace esta roto mipana

https://www.baeldung.com/rest-api-search-language-spring-data-specifications

## Proyecciones

Una proyección de una tabla es quedarnos con algunos elementos de una tabla en vez de con todos. Esto es útil a la hora de utilizar DTOs, puesto que nos ahorraremos utilizar conversores.

https://www.baeldung.com/spring-data-rest-projections-excerpts

No las realizaremos de esta forma.

https://www.baeldung.com/spring-data-jpa-projections

Tampoco las haremos así.

Las haremos basadas en clases DTO.

```
@Query("""
        select new com.salesianostriana.dam.dto.GetProductoDto(
            p.id, p.nombre, c.nombre
        )
        from Producto p LEFT JOIN p.categoria c
    """)
List<GetProductoDto> todosLosProductosDto( );
```

De esta forma podemos obtener los productos DTO sin necesidad de conversores, ya que la consulta nos devolverá un objeto con los campos de cada entidad que nos interese, en este caso, el id y el nombre del producto y el nombre de la categoría.

Esta consulta hará uso del constructor definido en GetProductoDto, el cual a su vez está definido por @AllArgsConstructor.

Otra de las ventajas es que las consultas son mucho más rápidas, por lo que ganamos en eficiencia.

Esta consulta habría que implementarla en su servicio correspondiente y luego invocarla desde el controlador.

Estos son consultas de solo lectura, por lo que no funcionan para guardas datos en la base de datos.

### Problema de CORS

https://www.baeldung.com/spring-cors