Hibernate

Framework

Objetivos

Conceptos fundamentales

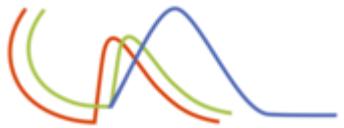
- Que es un framework de ORM (Object Relational Mapping)
- ¿Qué es hibernate?
- ¿Qué es JPA Data Spring?

Características

- Sesion
- Entidades
- Asociaciones
- Consultas

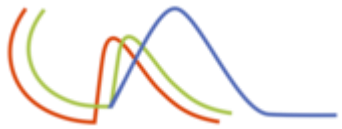
Introduccion

- Motivación:
  - Dos paradigmas diferentes : programación orientada a objetos y bases de datos relacionales.
  - El modelo relacional trata con relaciones, tuplas y conjuntos(matemático por naturaleza).
  - Paradigma orientado a objetos trata con objetos, sus atributos y las relaciones entre objetos.
  - GAP Objeto-Relacional.



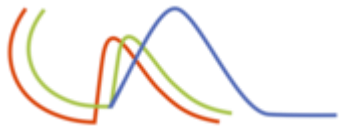
Introducción …

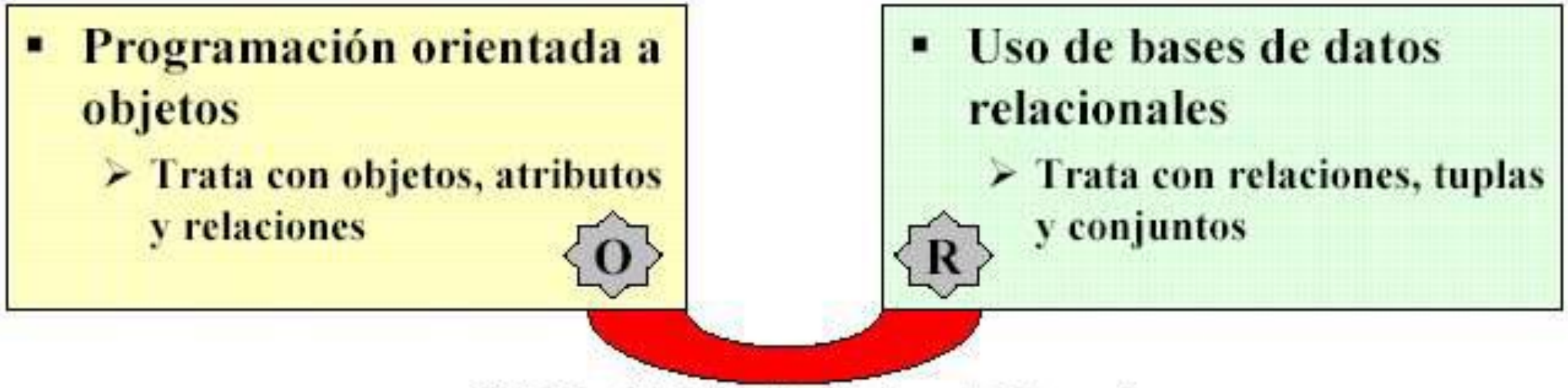
- Ej: Para hacer los objetos persistentes se requiere una conexión JDBC, crear una sentencia SQL y copiar todos los valores de las propiedades sobre un PreparedStatement o en una cadena SQL.
  - ¿Y las asociaciones?
  - ¿Y si el objeto contiene a su vez a otros objetos?
  - ¿Y las claves foráneas?



Introducción …

- Limitantes
  1. **Granuralidad**: atributos no primitivos se mapean en una columna. Ejemplo clase “dirección”
  2. **Subtipos**: No hay herencia en las BD SQL.
  3. **Identidad**: ¿Cómo realizar la igualdad, con el operador ==, la operación equals o la clave primaria?
  4. **Asociaciones**: En POO como referencias a objetos y colecciones de objetos, en BBDD como claves foráneas. Asociación muchos a muchos (BBDD no implementa la cardinalidad n-m).
- Solución -> ORM (Object-Relational Mapping)



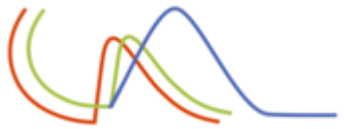


ORM: Object-Relational Mapping

- **Problema**: un 35% del código de una aplicación para realizar la correspondencia O ↔ R
- **Solución**: utilizar una ORM, por ejemplo Hibernate

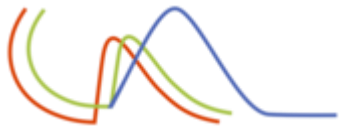
Características ORM

- Carácterísticas:
  - Un ORM cubre el GAP objeto-relacional.
  - Sólo hay que definir la correspondencia entre las clases y las tablas una sola vez (indicando que propiedad se corresponde con que columna, que clase con que tabla,etc…)
  - Utiliza POJO's (Plain Old Java Objects) en la aplicación y los hace persistentes con una sola instrucción:
    - orm.save(myObject)
  - Permite leer y escribir directamente en la BBDD usando POJO



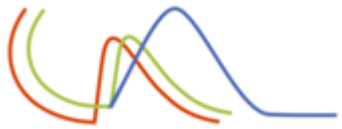
Características ORM …

- ORM es el middleware en la capa de persistencia que gestiona la persistencia.
- Esto implica cierta penalización en el rendimiento
- También hay sobrecarga en la gestión de los metadatos del mapeo, pero este coste es menor que el producido cuando se escribe a mano.



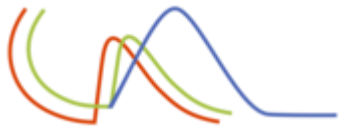
Elementos de un ORM:

- Un **API** para realizar las operaciones básicas CRUD (create, read, update, delete) en objetos persistentes.
- Un lenguaje para especificar **consultas** de objetos y propiedades.
- Facilidades para definir el **mapeo de los metadatos**.
- **Optimizaciones**



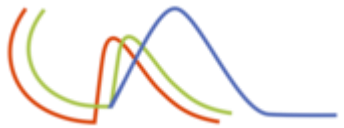
Beneficios

- **Productividad**: Ahorra mucho trabajo engorroso y repetitivo
- **Mantenibilidad**: Al haber menos código la aplicación es más mantenible. Además, evita el acoplamiento en el diseño del modelo de negocio y el de persistencia, cuando este último se hace a mano.
- **Rendimiento**: ORM realiza muchas optimizaciones, algunas dependientes de la BD en particular.
- **Independencia del proveedor de BBDD**: La aplicación es independiente de una BD particular o un dialecto específico SQL.



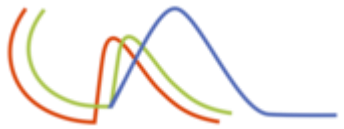
JPA

- JPA es el acrónimo de Java Persistence API y es el estándar de los frameworks de persistencia.
- Hibernate es una implementación de este estándar.



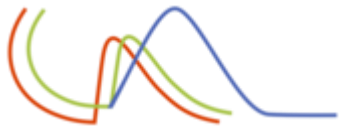
¿Qué es Hibernate?

- Hibernate es un ORM de libre distribución.
- Un framework maduro y completo.
- Puede utilizarse en cualquier contexto de ejecución (no necesita un contenedor especial).
- Facilita el mapeo de atributos entre una base de datos relacional y el modelo de objetos de una aplicación, mediante anotaciones o archivos declarativos (XML)
- No utiliza técnicas como generación de código a partir de descriptores del modelo de datos o manipulación de bytecodes en tiempo de compilación ni obliga a implementar interfaces específicos.
- Basado en el mecanismo de reflexión de Java .

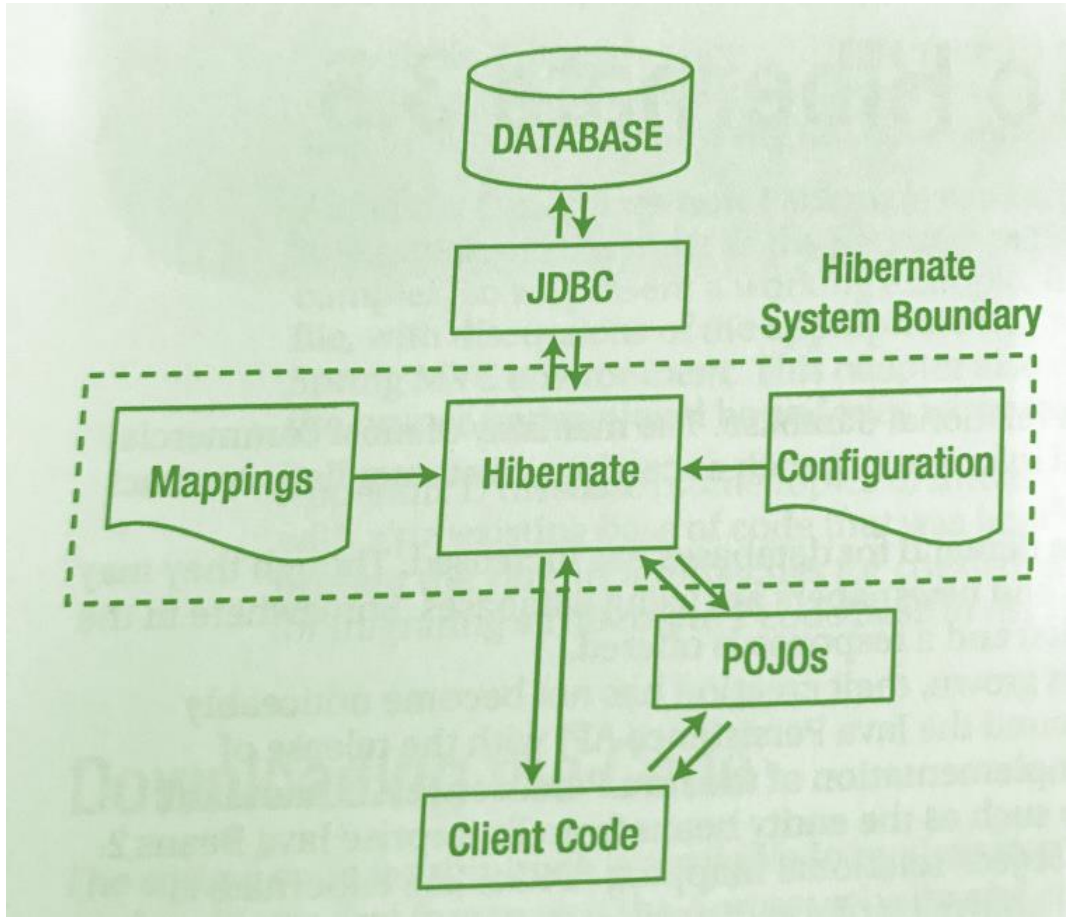


Características

- No intrusivo (estilo POJO).
- Buena documentación, comunidad amplia y activa.
- Transacciones, caché, asociaciones, polimorfismo, herencia, persistencia transitiva, estrategias de fetching.
- Potente lenguaje de consulta (HQL): subqueries, outer joins, ordering, proyección, paginación.
- Fácil testeo (basado en pojos).
- Uso de anotaciones o ficheros XML de mapeo, de donde se obtiene toda la información para realizar las operaciones CRUD.
- Uso del estándar JPA o configuración propia no-estándar

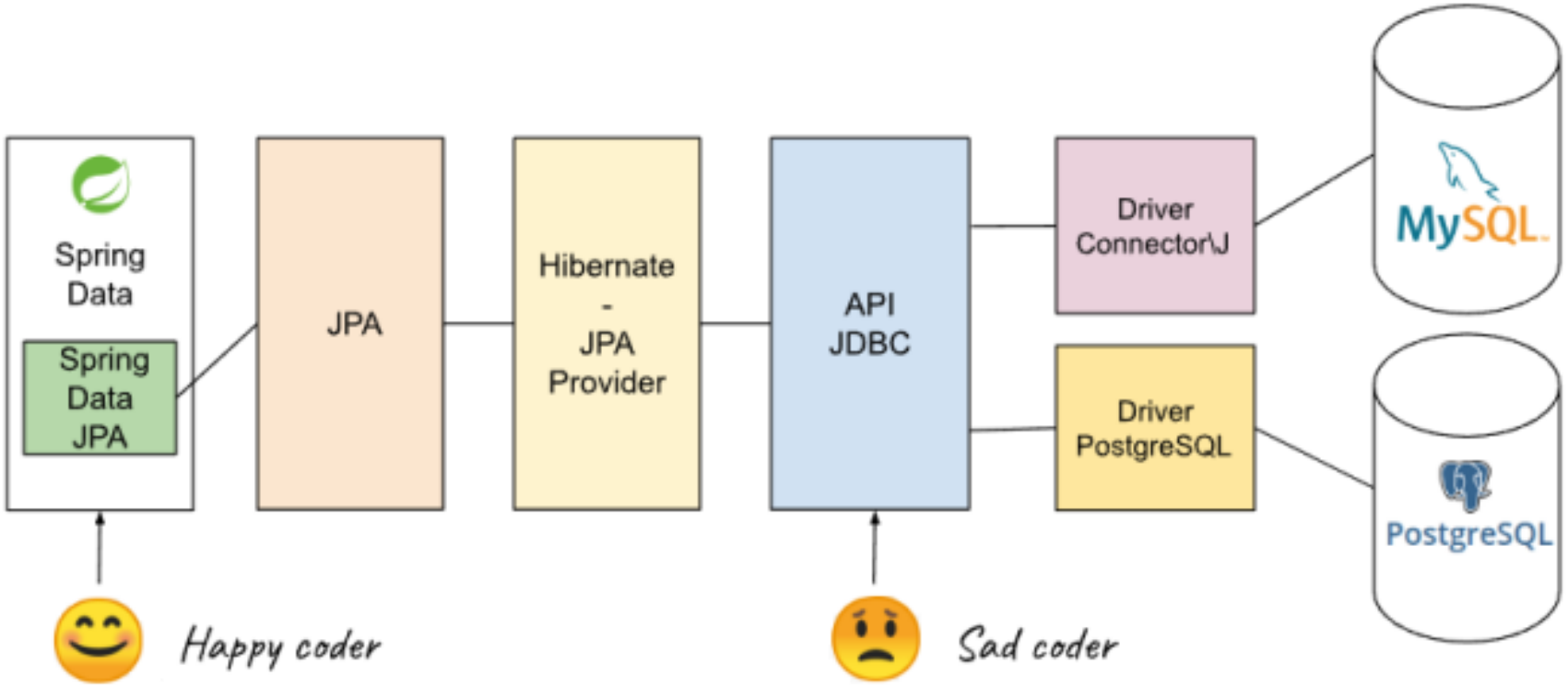


Hibernate


DATABASE
JDBC
Hibernate
System Boundary
Mappings
Hibernate
Configuration
POJOs
Client Code

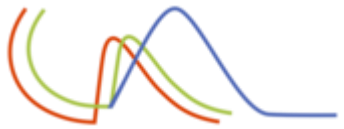
Spring Data JPA

- Spring Data JPA es un subproyecto de Spring Framework que proporciona una capa de abstracción adicional y funcionalidades adicionales para facilitar el desarrollo de aplicaciones JPA (Java Persistence API). Su objetivo principal es simplificar el acceso a los datos y reducir la cantidad de código boilerplate necesario al interactuar con la capa de persistencia.



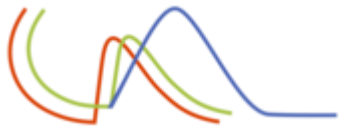
Spring Data JPA

- se basa en los conceptos de repositorio y consulta para manejar la interacción con la base de datos.
- Proporciona **anotaciones** e **interfaces** que permiten definir interfaces de repositorio personalizadas, que extienden las interfaces proporcionadas por Spring Data JPA, como CrudRepository, JpaRepository o PagingAndSortingRepository.
- Estas interfaces predefinidas ofrecen métodos comunes para realizar operaciones CRUD (crear, leer, actualizar y eliminar) y consultas en la base de datos.



Sesión - Hibernate

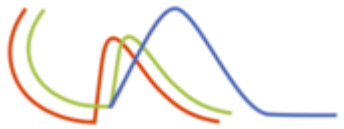
- Comunicación con el motor de Hibernae mediante Session.
- Se requiere una instancia de sesión para cada tipo de BD.
- Una sesión hace de cache de objetos cargados.
- Objetos tipo transient y tipo persistent.
  - Transient: Objetos que sólo existen en memoria y no en un almacén de datos.
  - Persistent: Objetos ya almacenados y por tanto persistentes.
- Necesidad de crear y cerrar explícitamente las sesiones de Hibernate.



Sesión - Hibernate

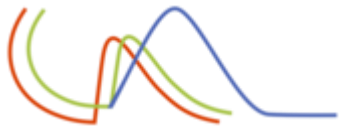
- Una sesión siempre va a pertenecer a un mismo thread de ejecución (el que pertenece a la ejecución de un método de negocio para un usuario determinado)

- Es decir (escenario aconsejable), en un entorno multiusuario y por tanto multithread habrá por tanto múltiples sesiones simultáneas, cada una perteneciente a su correspondientes thread y con su contexto de objetos en caché, transacciones, etc

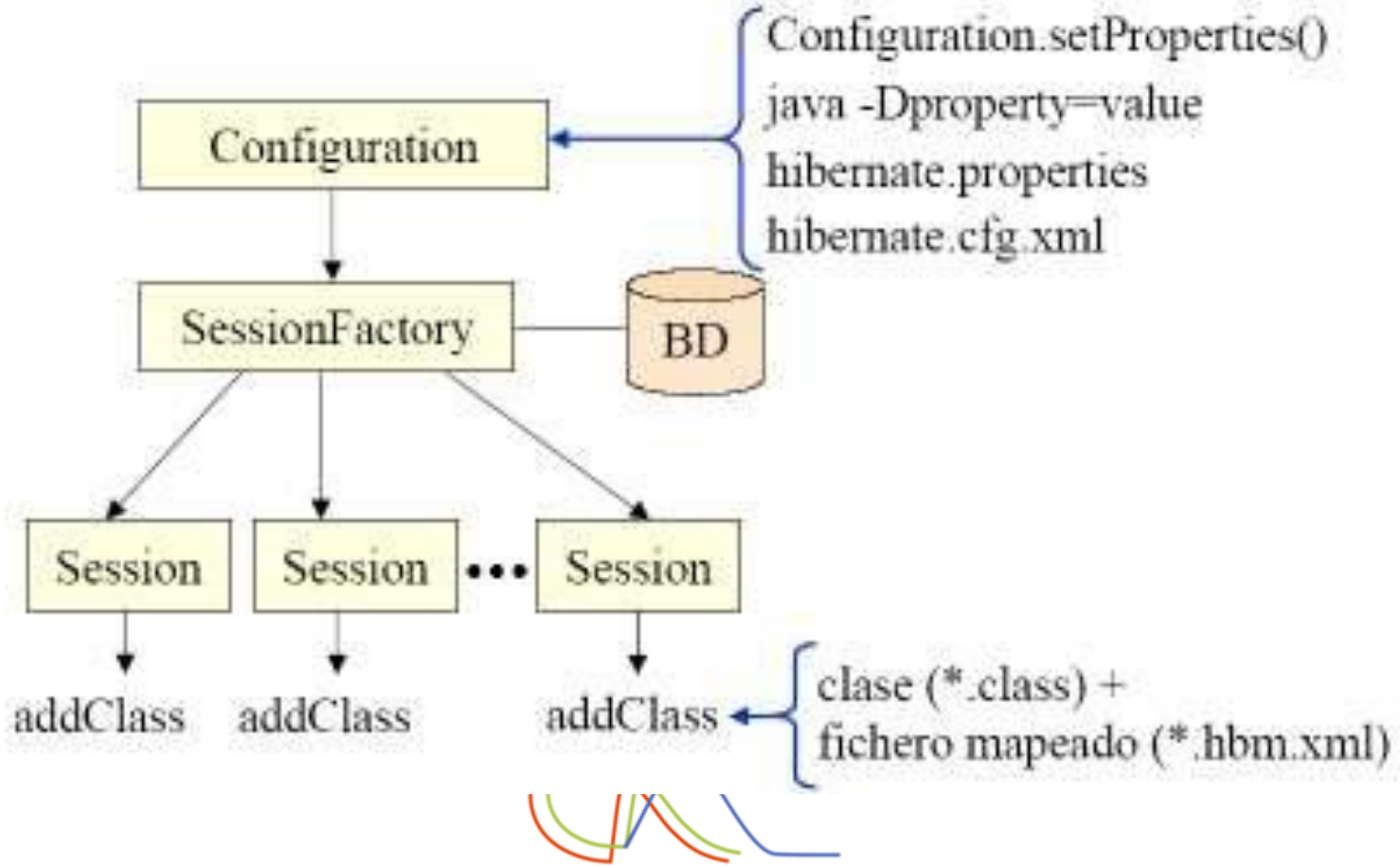


SessionFactory

- SessionFactory para crear instancias de sesiones y realizar operaciones comunes a los diferentes threads: gestión de una caché compartida entre threads, etc…
  - openSession(), evict(Class persistentClass), …
- ¿Qué sucede si en un entorno de múltiples hilos se accede a un mismo objeto desde dos sesiones diferentes?
  - Una instancia de un objeto persistente no es compartida por dos sesiones (dos instancias dentro de la misma máquina virtual Java para un “mismo” objeto de datos)

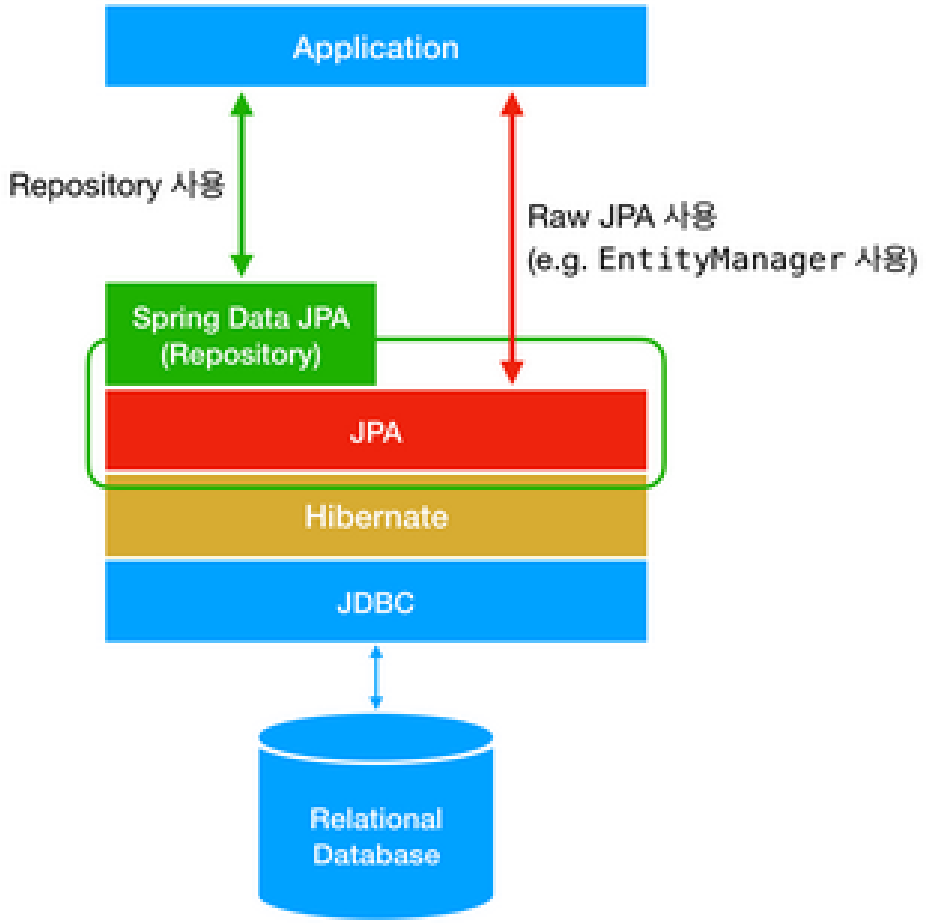


Configuración



Hibernate con Spring Data JPA

- El manejo de sesiones se simplifica gracias a la integración de Spring Framework.
- Spring Tools proporciona soporte para administrar automáticamente las sesiones de Hibernate a través del uso de transacciones y administración de contexto de persistencia.
- Esto se logra a través de:
  - La configuración de la sesión de Hibernate en el application.properties.
  - La inyección de dependencias.
  - El uso de transacciones declarativas.



Enfoque de generación de código

Code-First

Cuando es un proyecto nuevo, se programan las clases en java y se generan las tablas.

Database-First

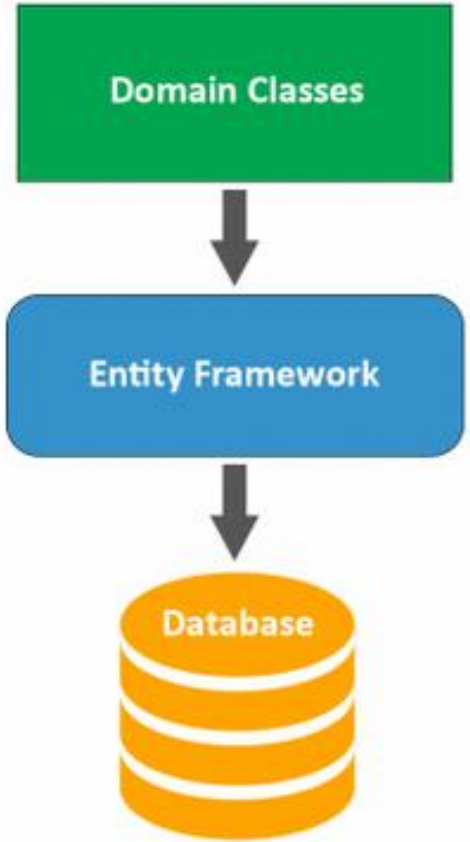
Cuando ya está creada la BBDD en nuestro proyecto

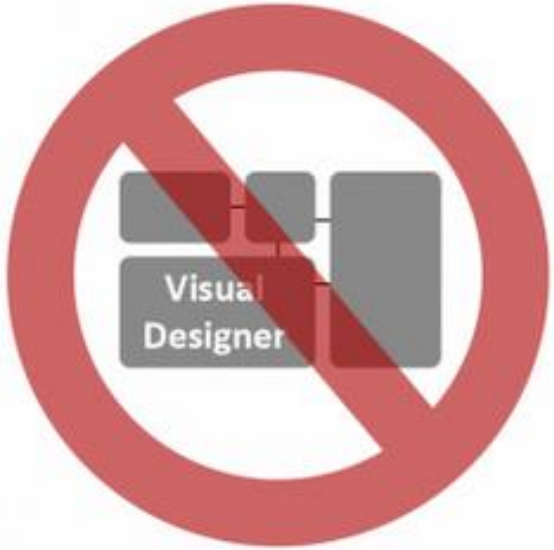
Model-Fist

Trabajar un modelo en una herramienta visual y generar ambos

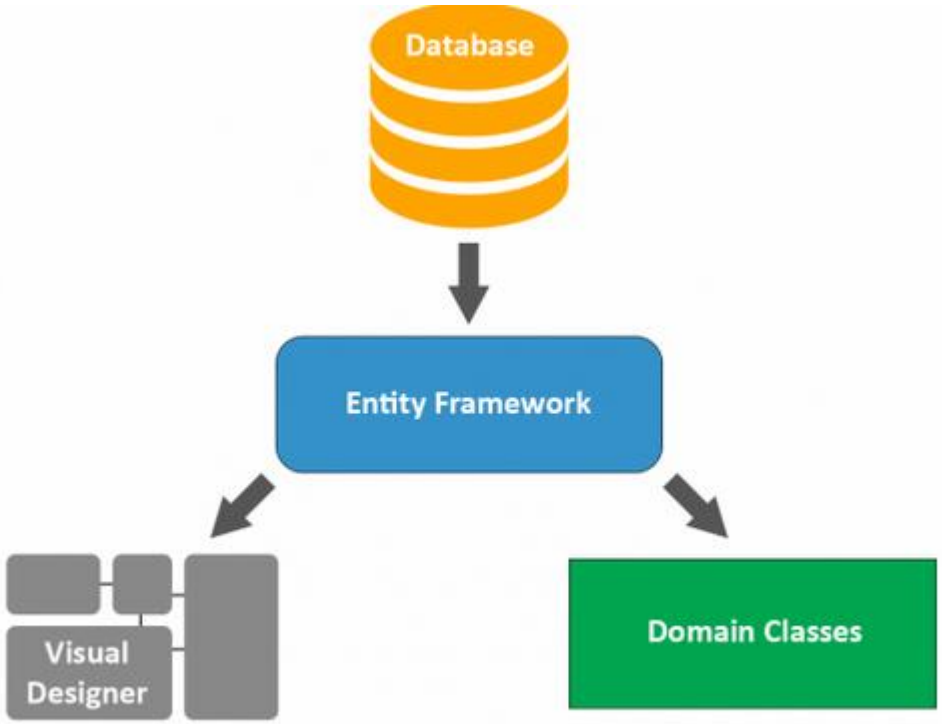
Codigo-First

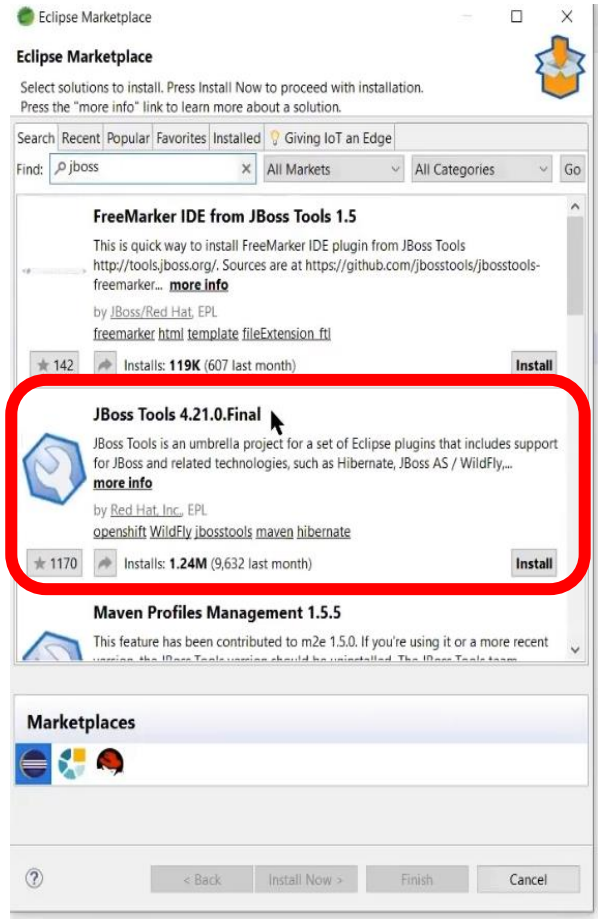
- Cuando es un proyecto nuevo, se programan las clases en java y se generan las tablas.

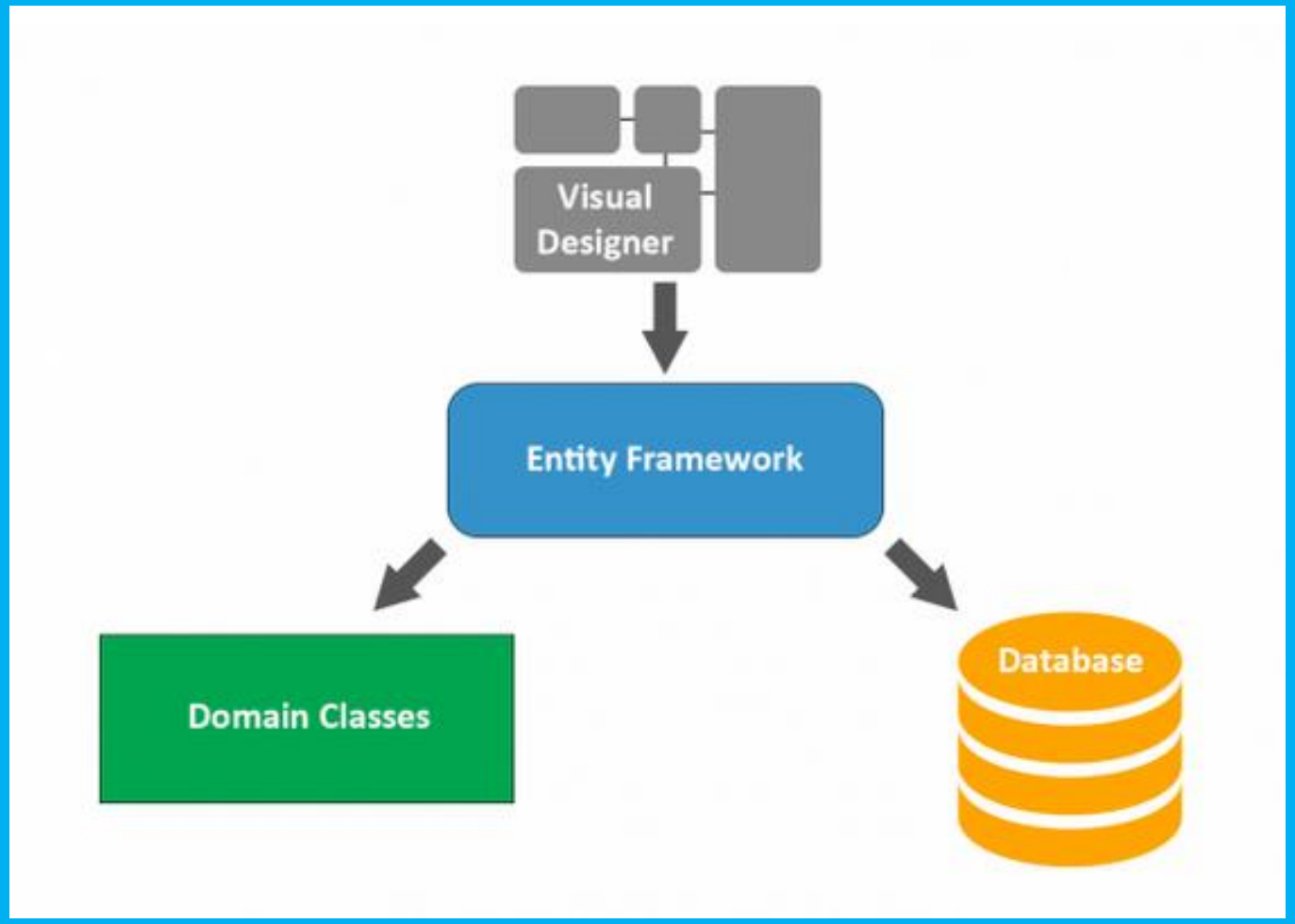




DataBase First






Visual
Designer
Entity Framework
Domain Classes
Database

Model First


Anotaciones

Entidad

- Una entidad se refiere a una clase Java que se mapea a una tabla en una base de datos relacional. Es una representación de un objeto persistente que puede ser almacenado, actualizado y recuperado de la base de datos.

```java
import jakarta.persistence.*;
import java.util.List;
no usages  TitaRuiz *
@Entity
@Table(name = "productos")
public class Producto {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int idProducto;
    no usages
    @Column(length=30, nullable = false)
    private String nombre;
    no usages
    @Column(length=60, nullable = false)
    private String descripcion;
    no usages
    private double precio;
    no usages
    private int cantidadExistencia;
```

Clave principal compuesta

- Se define una clase para la clave primaria con la anotación @Embeddable
- En la entidad se define un atributo del tipo definido aneriormente con anotación @EmbeddedId

1

```
1 usage   new *
@Embeddable
public class PersonaPK implements Serializable {

    // Solo los campos que forman la PK
        no usages
        @Column(name="telefono")
        private Long telefono;
        no usages
        @Column(name="nif")
        private String nif;



}
```

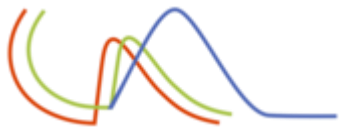
2

```
no usages   new *
@Entity
@Table(name="personas")
public class Persona implements Serializable {
    no usages
    private static final long serialVersionUID = -9014986003273703863L;

//  @Id
//  @GeneratedValue(strategy = GenerationType.IDENTITY)
//  private int id;
    no usages
    @EmbeddedId
    private PersonaPK pk;
    no usages
    @Column(nullable = false, length =20)
    private String nombre;
    no usages
    @Column(nullable = false, length =20)
    private String apellido;
    no usages
    private char sexo;
    no usages
    private int edad;
    no usages
    private LocalDate fechaNacimiento;
    no usages
    private String curriculum;
```

Estados de una entidad

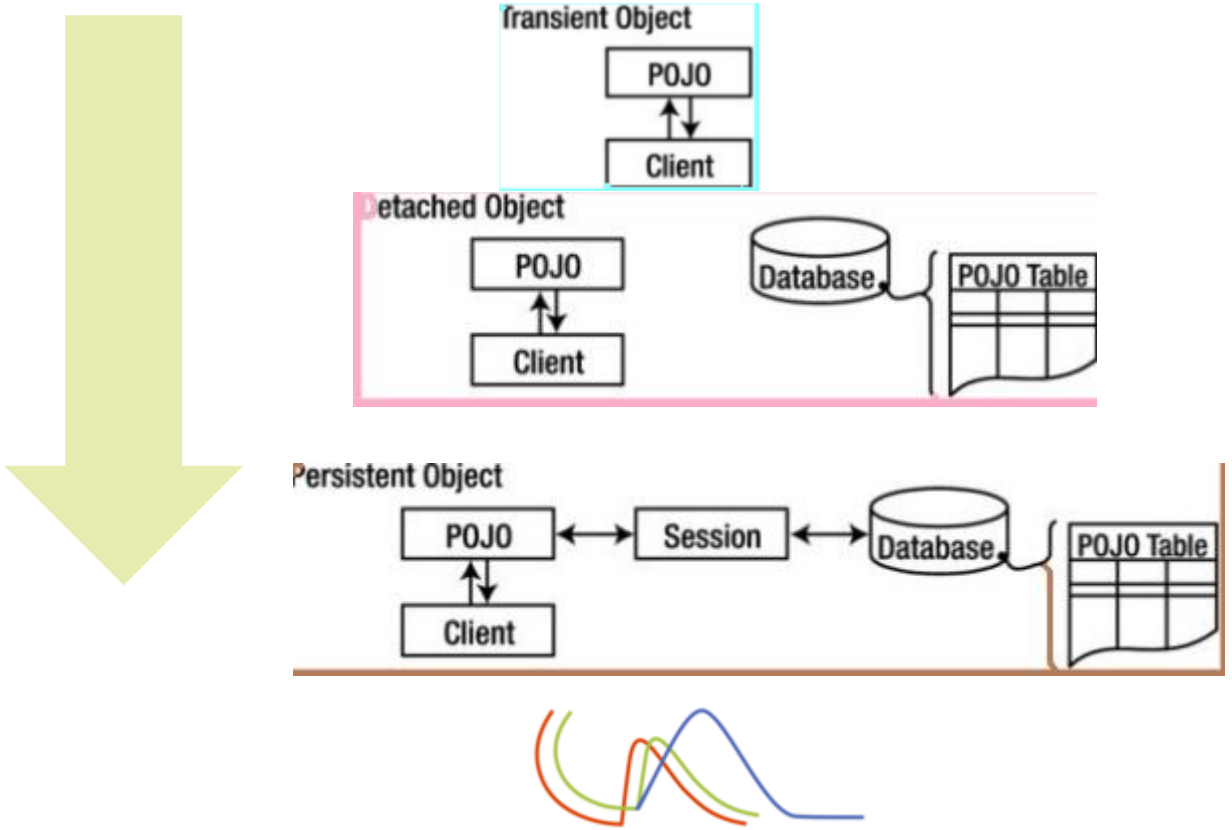
Los estados principales de una entidad en JPA son los siguientes:

- Transient (Transitorio): Cuando un objeto en memoria aún no está asociado con el contexto de persistencia y no se ha convertido en una entidad administrada. Es decir, el objeto no está siendo rastreado ni gestionado por JPA.
- Persistent (Persistente): Un objeto se convierte en persistente cuando se asocia con el contexto de persistencia, generalmente al persistirlo o recuperarlo a través de una operación de búsqueda. En este estado, los cambios realizados en el objeto se reflejan automáticamente en la base de datos cuando se realiza una sincronización.
- Detached (Separado): Un objeto se considera separado cuando ya no está asociado con el contexto de persistencia, generalmente porque se ha cerrado la sesión o se ha liberado la entidad de la gestión de JPA. Aunque el objeto ya no está siendo rastreado, aún puede contener datos válidos y ser utilizado de forma independiente.
- Removed (Eliminado): Este estado indica que la entidad está programada para ser eliminada de la base de datos en la próxima operación de sincronización con la base de datos. Una vez que se sincroniza, el objeto ya no existe en la base de datos.



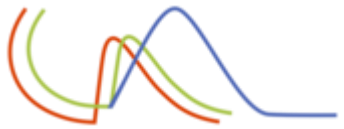
Estados de una entidad

- Una instancia de un objeto que es mapeado a hibernate puede estar en los siguientes estados:



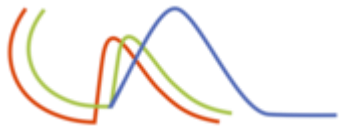
Asociaciones

- Estas anotaciones permiten definir y configurar cómo se relacionan las entidades entre sí en el modelo de datos.
- Para ello se tiene que conocer el propietario de la asociación.
- Principales anotaciones de asociación.
  - @OneToOne
  - @OneToMany
  - @ManyToOne
  - @ManyToMany



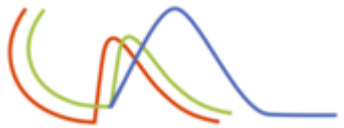
Marcando el propietario de las asociaciones

Typo de Asociación	Opción
One-to-one	Cualquier final puede ser el propietario, pero sólo uno, sino sería una dependencia circular
One-to-Many	El final de many debe ser realizado en el propietario de la asociación.
Many-to-One	Es lo mismo que el One-to-Many pero viéndolo desde la otra perspectiva.
Many-to-many	Cualquier final puede ser el propietario, pero sólo uno, sino sería una dependencia circular



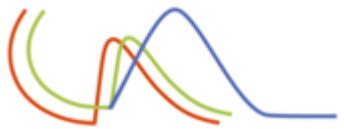
Atributo cascade para las asociaciones

- cascade = CascadeType.ALL: Con esta configuración, todas las operaciones de persistencia realizadas en la entidad principal se propagarán a la entidad asociada. Esto significa que si se guarda, se actualiza o se elimina la entidad principal, Hibernate también guardará, actualizará o eliminará automáticamente la entidad asociada.
- cascade = CascadeType.PERSIST: Con esta configuración, solo la operación de persistencia (guardado) realizada en la entidad principal se propagará a la entidad asociada. Si se guarda la entidad principal, Hibernate también guardará automáticamente la entidad asociada. Sin embargo, las operaciones de actualización o eliminación no se propagarán.
- cascade = CascadeType.MERGE: Con esta configuración, solo la operación de actualización (merge) realizada en la entidad principal se propagará a la entidad asociada. Si se actualiza la entidad principal, Hibernate también actualizará automáticamente la entidad asociada. Sin embargo, las operaciones de guardado o eliminación no se propagarán.
- cascade = CascadeType.REMOVE: Con esta configuración, solo la operación de eliminación realizada en la entidad principal se propagará a la entidad asociada. Si se elimina la entidad principal, Hibernate también eliminará automáticamente la entidad asociada. Sin embargo, las operaciones de guardado o actualización no se propagarán.
- cascade = CascadeType.REFRESH: Con esta configuración, la operación de refresco (refresh) realizada en la entidad principal se propagará a la entidad asociada. Si se refresca la entidad principal, Hibernate también refrescará automáticamente la entidad asociada. Las operaciones de guardado, actualización o eliminación no se propagarán.



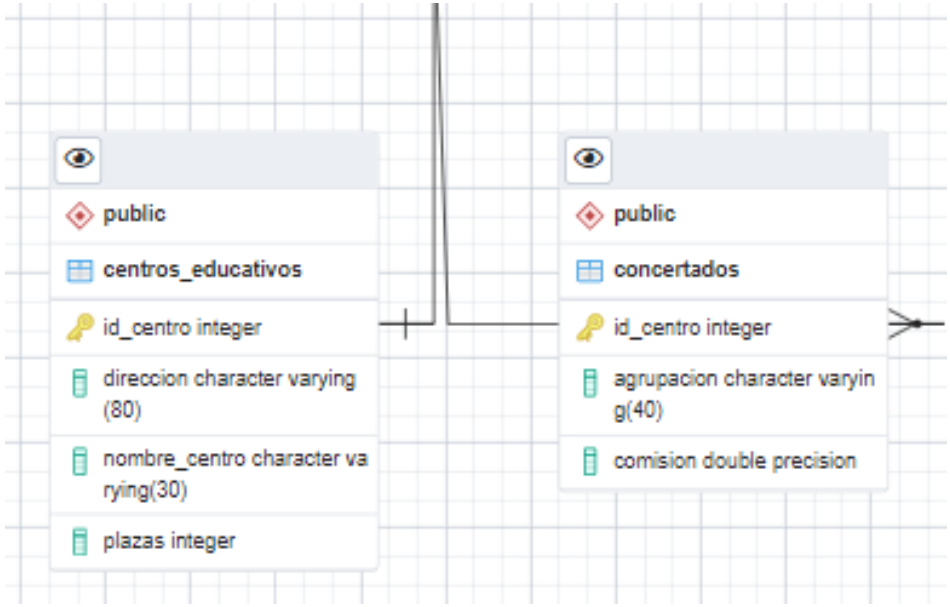
Atributo fetch para las asociaciones

- El atributo fetch determina la forma en que se carga la entidad asociada cuando se accede a la propiedad correspondiente. Puede tener dos valores:
- fetch = FetchType.LAZY: Con esta configuración, la entidad asociada se carga de forma diferida o perezosa, es decir, no se recupera automáticamente de la base de datos cuando se accede a la propiedad. La carga de la entidad asociada se produce solo cuando se accede a dicha propiedad por primera vez. Esto puede ser beneficioso en términos de rendimiento, ya que evita cargar innecesariamente entidades relacionadas si no se utilizan. Sin embargo, se debe tener cuidado al acceder a la propiedad fuera del contexto de persistencia, ya que podría producirse una excepción si no se ha inicializado adecuadamente.
- fetch = FetchType.EAGER: Con esta configuración, la entidad asociada se carga inmediatamente cuando se recupera la entidad principal. En otras palabras, la entidad asociada se recupera de la base de datos en el mismo momento en que se accede a la propiedad. Esto garantiza que la entidad asociada esté disponible de inmediato, pero puede tener un impacto negativo en el rendimiento si la entidad asociada no se utiliza con frecuencia.

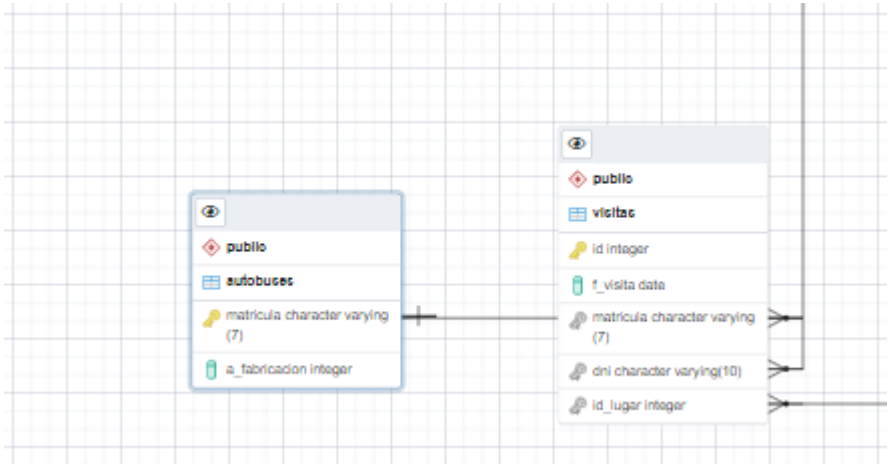


Uso de @OneToOne

```java
import jakarta.persistence.*;


1 usage  new *
@Entity
@Table(name = "concertados")
public class Concertado {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int idCentro;
    no usages
    private double comision;
    no usages
    @Column(length = 40)
    private String agrupacion;
    no usages
    @OneToOne
    @JoinColumn(name ="id_centro", nullable = false,
    foreignKey = @ForeignKey(name = "FK_concertados_centros"))
    private CentroEducativo ce;


}
```



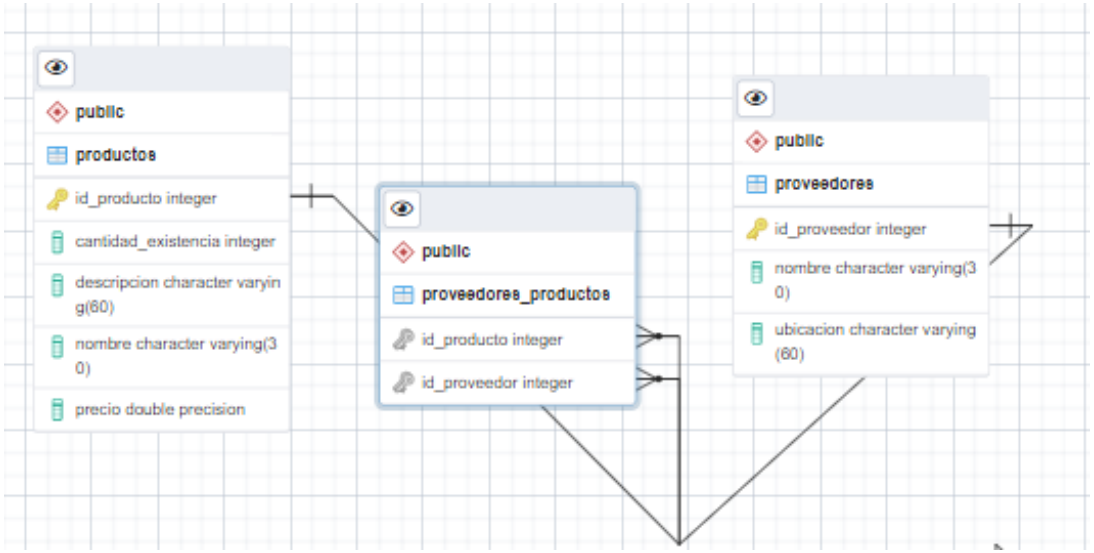
Uso de @ManyToOne

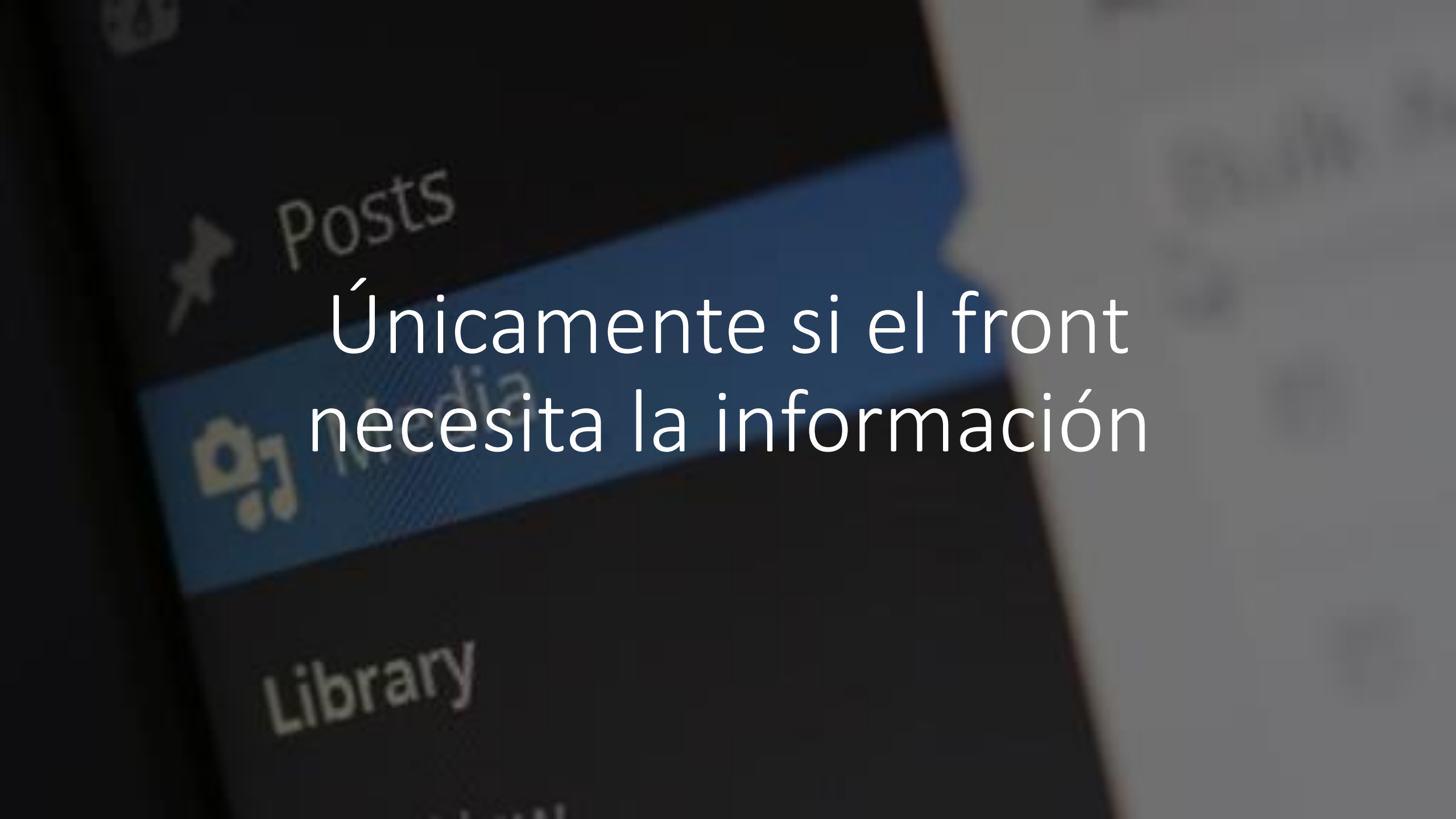



```
10 usages   TitaRuiz *
@Entity
@Table(name = "visitas")
@ToString(onlyExplicitlyIncluded = true)
public class Visita {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    no usages
    private LocalDate fVisita;


    no usages
    @JsonIgnore
    @ManyToOne(cascade={CascadeType.MERGE,CascadeType.PERSIST,
            CascadeType.REFRESH})
    @JoinColumn(name = "matricula", referencedColumnName = "matricula", foreignKey = @ForeignKey(name= "fk_visitas_autobus"))
    private Autobus autobus;
```

Uso de @ManyToMany

```java
no usages  TitaRuiz
@Entity
@Table(name = "productos")
public class Producto {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int idProducto;
    no usages
    @Column(length=30, nullable = false)
    private String nombre;
    no usages
    @Column(length=60, nullable = false)
    private String descripcion;
    no usages
    private double precio;
    no usages
    private int cantidadExistencia;

    no usages
    @ManyToMany
    @JoinTable(name = "proveedores_productos" ,
    joinColumns = @JoinColumn(name = "id_producto", referencedColumnName = "idProducto"),
    inverseJoinColumns = @JoinColumn(name = "id_proveedor", referencedColumnName = "idProveedor"))
    private List<Proveedor> proveedores;


}
```

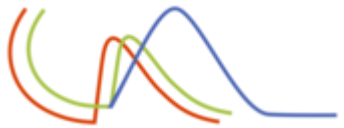




Únicamente si el front necesita la información

@OneToOne (opcional)

```java
import jakarta.persistence.*;

1 usage  new *
@Entity
@Table(name = "centros_educativos")
public class CentroEducativo {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int idCentro;

    no usages
    @Column(length = 30, nullable = false)
    private String nombreCentro;

    no usages
    @Column(length = 80, nullable = false)
    private String direccion;

    no usages
    private int plazas;

    no usages
    @OneToOne(mappedBy = "ce", cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    private Concertado c;
}
```

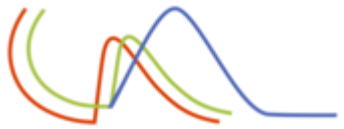


@OneToMany (opcional)

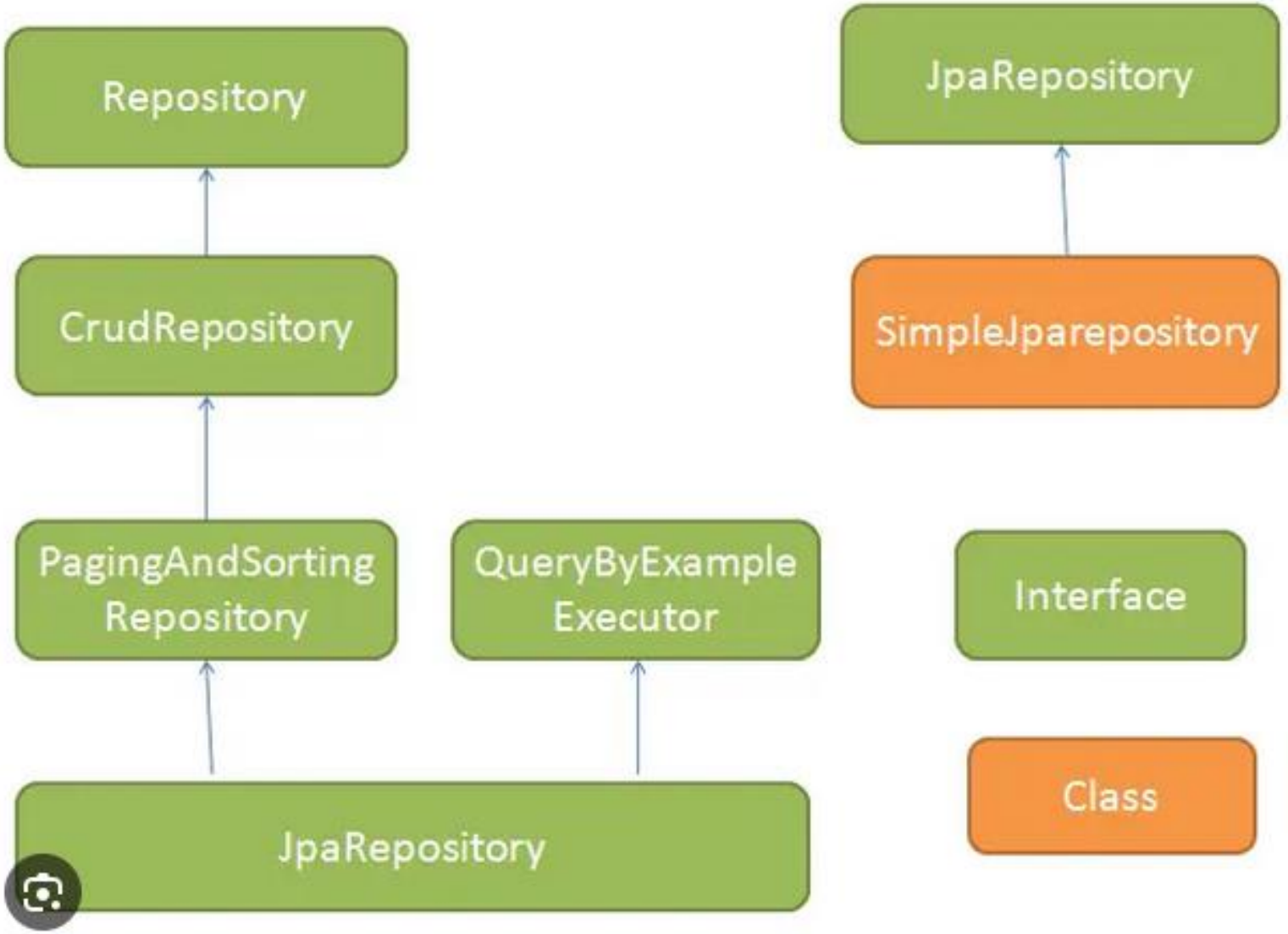
```java
@Entity
@Table(name = "autobuses")
public class Autobus {
    @Id
    @Column(length = 7)
    private String matricula;
    @Column(nullable = false)
    private Integer aFabricacion;

    @OneToMany(mappedBy = "autobus",
            cascade = CascadeType.ALL,
            orphanRemoval = true,
            fetch=FetchType.EAGER )
    private List<Visita> visitas;
```

@ManyToMany (opcional)

```java
@Entity
@Table(name = "proveedores")
public class Proveedor {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int idProveedor;
    @Column(length=30)
    private String nombre;
    @Column(length=60)
    private String ubicacion;
    @ManyToMany(mappedBy = "proveedores", cascade = CascadeType.ALL, fetch = FetchType.EAGER)
    List<Producto> productos;


}
```

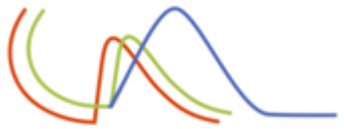


Interfaces JPA Data Spring



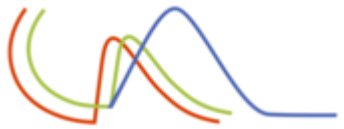
Interfaces JPA Data Spring

- Estas interfaces proporcionan una abstracción sobre la capa de persistencia y permiten realizar operaciones en las entidades de manera más sencilla y declarativa. Al extender estas interfaces en tus repositorios personalizados y utilizar las convenciones de nomenclatura adecuadas, Spring Data JPA generará automáticamente la implementación correspondiente y la inyectará en tu aplicación.
- Es importante tener en cuenta que estas interfaces son parte de la funcionalidad principal de Spring Data JPA, pero también existen otras interfaces y anotaciones adicionales que se pueden utilizar para personalizar aún más los repositorios y las consultas según tus necesidades específicas.



JpaRepository<T, ID>

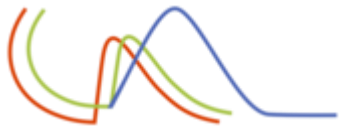
- La interfaz JpaRepository<T, ID> proporciona métodos predefinidos para realizar operaciones comunes en la base de datos, como guardar, recuperar, actualizar y eliminar entidades. Al extender esta interfaz en tus repositorios personalizados, heredas automáticamente estos métodos y los puedes utilizar sin necesidad de escribir el código de implementación.
- Aquí hay algunos ejemplos de los métodos proporcionados por JpaRepository<T, ID>:
  - save(entity): Guarda una entidad en la base de datos. Si la entidad ya existe, se realiza una actualización, de lo contrario, se realiza una inserción.
  - findById(id): Recupera una entidad por su identificador único.
  - findAll(): Recupera todas las entidades de un tipo específico.
  - delete(entity): Elimina una entidad de la base de datos.
  - deleteById(id): Elimina una entidad por su identificador único.
- Además de estos métodos, JpaRepository<T, ID> también proporciona métodos para realizar consultas personalizadas utilizando el lenguaje de consultas JPA (JPQL) o consultas nativas de SQL



```java
public interface IEmpleadoBasicoRepo extends JpaRepository<EmpleadoBasico, Integer> {

    1 usage   new *
    @Query("FROM EmpleadoBasico p WHERE p.edad=:edad")
    List<EmpleadoBasico> filtroPorEdad(@Param("edad") int edad);


    1 usage   new *
    @Query(value="Select * from Products where empleados_b_id=:edad", nativeQuery = true)
    List<EmpleadoBasico> filtroPorEdadNativa(@Param("edad") int edad);
```

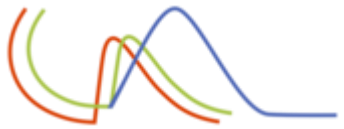
JPQL (Java Persistence Query Language)

- Lenguaje de queries orientado a objetos, en lugar de operar en las tablas y columnas trabaja con los objetos persistentes y sus propiedades será traducido al SQL dependiente de cada base de datos de manera automática y transparente.
- Se recomienda que se utilice este lenguaje en los proyectos para realizar mejor la portabilidad en caso de haber necesidad.
- Es más compacto ya que hace uso de las relaciones definidas en el mapeo.
- Es solo consulta.



EXPRESAR RUTAS

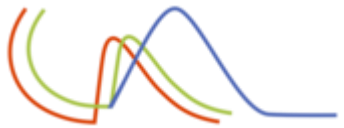
- Utilizamos el punto (.) como operador de navegación para expresar rutas.
- Al igual que ocurre en el lenguaje Java el punto nos permite especificar una propiedad dentro del objeto, pues en JPA lo utilizamos para concretar un campo dentro de la entidad.



CLAUSULA FROM

- define el dominio de la consulta, es decir, los nombres para las entidades que van a ser utilizados en la consulta. El operador AS es opcional.
- La sintaxis es la siguiente:

```
@Query("FROM Producto p")
List<Producto> toda();
```



LA CLÁUSULA SELECT

- La cláusula SELECT escoge qué objetos y propiedades devolver en el conjunto de resultados de la consulta.

```java
public interface IProductosRepo extends IGenericRepo<Producto, Short> {
    1 usage  new *
    @Query("select new com.example.prueba.dto.ProductoProyeccionDTO(p.id, p.nombre) from Producto p")
    List<ProductoProyeccionDTO> proyeccion();
```

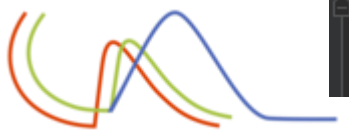
```java
@Data
@NoArgsConstructor
public class ProductoProyeccionDTO {
    private Short id;
    private String nombre;

    1 usage  new *
    public ProductoProyeccionDTO(Short id, String nombre) {
        this.id = id;
        this.nombre = nombre;
    }
}
```

FUNCIONES DE AGREGACIÓN

- Las consultas HQL pueden incluso retornar resultados de funciones de agregación sobre propiedades:

```
2 usages  new *
public interface IProductosRepo extends IGenericRepo<Producto, Short> {
    1 usage  new *
    @Query("select new com.example.prueba.dto.ProductoAgregacionDTO(sum(p.unidadesExistencia)) from Producto p")
    List<ProductoAgregacionDTO> agregacion();
```

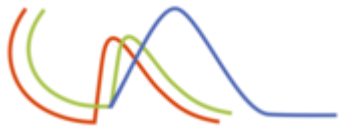
```
6 usages  new *
@Data
@NoArgsConstructor
public class ProductoAgregacionDTO {

    private Long suma;

    1 usage  new *
    public ProductoAgregacionDTO(Long suma) {
        this.suma = suma;
    }
}
```



CONSULTAS POLIMÓRFICAS

- Devuelve instancias no solamente de Cat, sino también de subclases como DomesticCat. Las consultas de Hibernate pueden nombrar cualquier clase o interfaz Java en la cláusula from. La consulta retornará instancias de todas las clases persistentes que extiendan esa clase o implementen la interfaz.

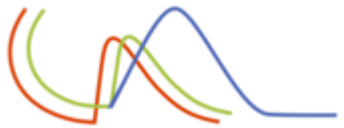
```
from Cat as cat
```



Clausula WHERE

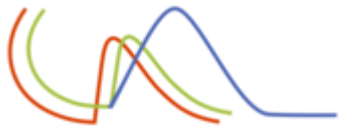
- La cláusula WHERE le permite filtrar los resultados de una consulta. Únicamente las entidades que responden a la condición de consulta especificada se recuperarán de la base de datos.

```
1 usage  new *
@Query("FROM Producto p WHERE p.nombre LIKE %:texto%")
List<Producto> filtroPorTexto(@Param("texto") String texto);
```



EXPRESIONES CONDICIONALES Y OPERADORES

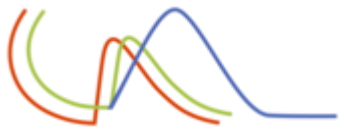
Tipo de Operador	Operador
Navegación	.
Signo unario	+. -
Aritméticos	*,/, +,-
Relacionales	=, >, >=, <, <=, <>, [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER [OF]
Lógicos	NOT, AND, OR
	concatenación de cadenas ...||... o concat(...,...)
Fecha	current_date(), current_time() y current_timestamp() second(...), minute(...), hour(...), day(...), month(...), and year(...)



LA CLÁUSULA ORDER BY

- La lista retornada por una consulta se puede ordenar por cualquier propiedad de una clase retornada o componentes:
- Los asc o desc opcionales indican ordenamiento ascendente o descendente respectivamente.

```
public interface IProductosRepo extends IGenericRepo<Producto, Short> {
    1 usage  new *
    @Query("select new com.example.prueba.dto.ProductoProyeccionDTO(p.id, p.nombre) from Producto p order by p.nombre desc ")
    List<ProductoProyeccionDTO> proyeccion();
```

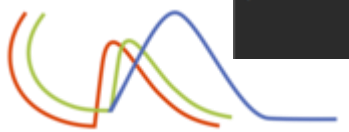


LA CLÁUSULA GROUP BY

- Una consulta que retorna valores agregados se puede agrupar por cualquier propiedad de una clase retornada o componentes:

```
2 usages  new *
public interface IProductosRepo extends IGenericRepo<Producto, Short> {
    1 usage  new *
    @Query("select new com.example.prueba.dto.ProductoAgrupacionDTO(p.categoriaId, avg(p.precioUnitario)) from Producto p group by p.categoriaId")
    List<ProductoAgrupacionDTO> agrupacionPromedio();
```

```
6 usages  new *
@Data
public class ProductoAgrupacionDTO {
    private Short categoriaId;
    private Double promedio;

    1 usage  new *
    public ProductoAgrupacionDTO(Short categoriaId, Double promedio) {
        this.categoriaId = categoriaId;
        this.promedio = promedio;
    }
}
```



SUBCONSULTAS

- Para bases de datos que soportan subconsultas, Hibernate soporta subconsultas dentro de consultas. Una subconsulta se debe encerrar entre paréntesis (frecuentemente por una llamada a una función de agregación SQL). Incluso se permiten subconsultas correlacionadas (subconsultas que se refieren a un alias en la consulta exterior).

```
from Cat as cat
where not exists (
    from Cat as mate where mate.mate = cat
)
```

```
from DomesticCat as cat
where cat.name not in (
    select name.nickName from Name as name
)
```