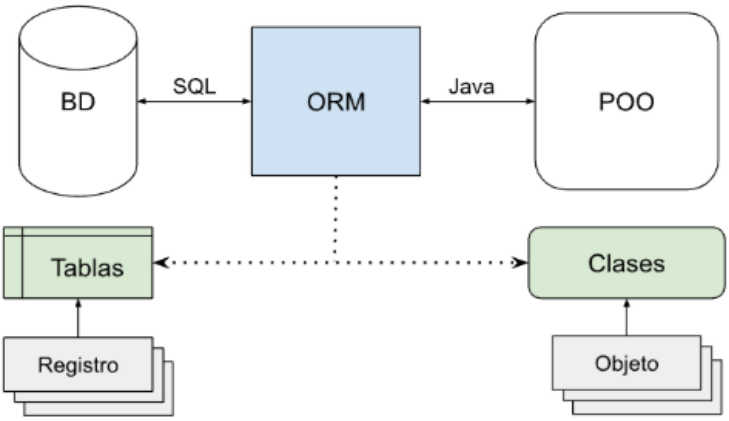
Conceptos importantes

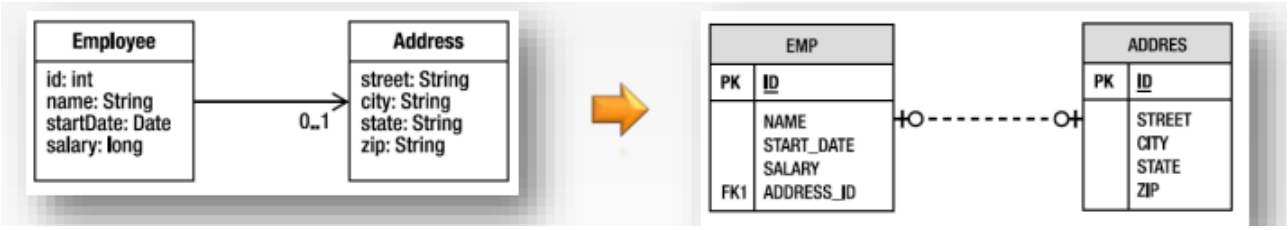
✓ **ORM** (Object-Relational mapping) es una técnica para convertir nuestras clases en una base de datos relacional y viceversa. Esto se conoce como mapeo objeto relacional.



✓ JPA: Java Persistence API

- Es la especificación de ORM para Java.
- Utiliza internamente JDBC
- Define un conjunto de interfaces y reglas, pero no proporciona una implementación concreta.
- JPA introduce anotaciones como @Entity, @Table, @Id, @GeneratedValue, entre otras, para mapear clases Java a tablas de bases de datos.

Ejemplo de mapeo entre clases y tablas:



✓ **Hibernate**:

Hibernate es una implementación concreta de la especificación JPA. Es decir, Hibernate cumple con las reglas e interfaces definidas por JPA.

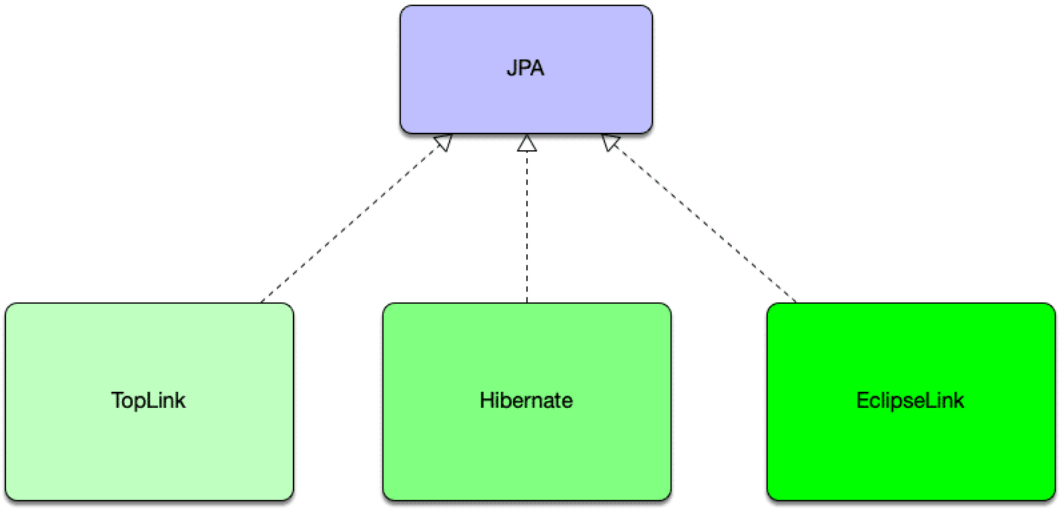
Importante:
JPA es un estándar, una especificación, mientras que Hibernate es una implementación concreta de la especificación JPA. Es decir, Hibernate cumple con las reglas e interfaces definidas por JPA.

Además de JPA, Hibernate ofrece características adicionales que no están especificadas en JPA, como la caché de segundo nivel, la validación automática de esquemas, y estrategias de recuperación y manipulación de datos más avanzadas.

Proporciona su propio lenguaje de consulta, HQL (Hibernate Query Language), además de JPQL (Java Persistence Query Language) definido por JPA.

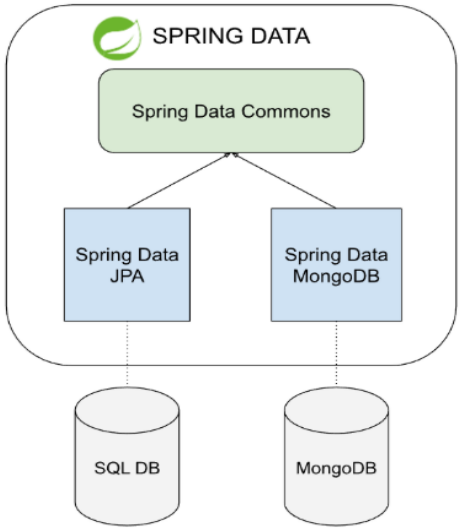
¿Existen más implementaciones de JPA?

Sí, Hibernate no es la única implementación de JPA. Ejemplos: EclipseLink (desarrollado por Eclipse), TopLink (desarrollado por Oracle), OpenJPA (desarrollado por Apache),…



✓ Spring Data

Spring Data es un proyecto dentro del ecosistema **Spring Framework** que facilita el acceso y la manipulación de datos en aplicaciones Java, proporcionando una capa de abstracción sobre diferentes tecnologías de almacenamiento.



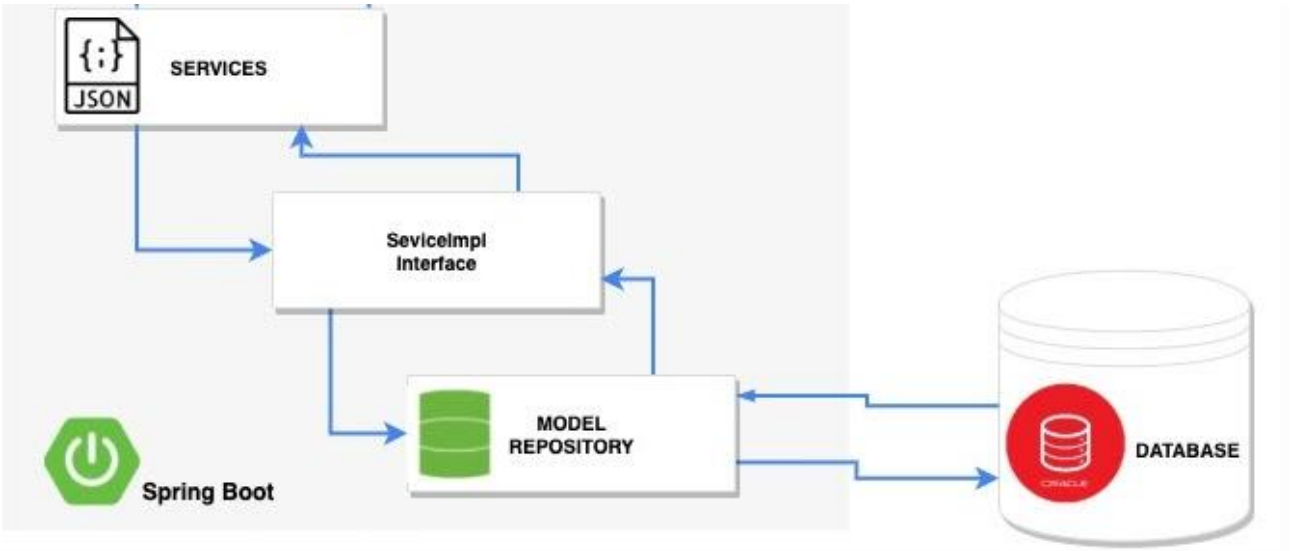
✓ **Spring Data JPA:** Se encarga de la integración con JPA/Hibernate para bases de datos SQL.

- Conversión automática entre objetos Java y el esquema de la base de datos
- Creación de consultas en base a métodos en interface

Estructura de paquetes en src/main/java:

- **paquete modelo:** Clases o entidades que se mapean: Ej Cliente.java

- **paquete servicio:**
  - interfaz servicio: Ej ClienteService.java
  - clase servicio: Ej ClienteServiceImpl.java
- **paquete repository:**
  - interfaz dao: Ej ClienteDAO.java



Paquete modelo: Mapeo de Entidades - Anotaciones

Creamos una clase con los atributos de la tabla que vamos a mapear. Es necesario tener los set, get y constructor vacío. Los constructores con parámetros son opcionales.

```
public class Cliente {

    private int id;
    private String nombre;
    private String apellidos;
}
```

Tenemos que especificar con anotaciones el nombre de la tabla y atributos: (import javax.persistence)

Anotaciones a nivel de clase:

- **@Entity** : Indicamos que la clase es una entidad que vamos a mapear.
- **@Table**(name=clientes): Indicamos con qué tabla de BD vamos a mapear. Si ambos nombres son iguales, no hace falta indicarlo.

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

Anotaciones por cada atributo:

- **@Column**(name="nombre del campo de BD") : Si el nombre del atributo y el campo de la tabla son iguales, sólo se indica @Column.

```
@Column(name="id")
private int id;
@Column(name="nombre")
private String nombre;
@Column(name="apellidos")
private String apellidos;
```

- **@Id**: Es necesario en el campo que es clave primaria de la tabla.
- **@GeneratedValue:** Se especifica en el atributo que es clave de la tabla.
  Indica que el valor de la PK se genera automáticamente. Se coloca junto a @Id. Tiene varios atributos opcionales para indicar cómo se genera el valor de la PK:
  - GenerationType.AUTO: La estrategia de generación se selecciona automáticamente según el proveedor de persistencia.
  - **GenerationType.IDENTITY:** Utiliza una columna de identidad en la base de datos para generar valores únicos.
  - GenerationType.SEQUENCE: Utiliza una secuencia en la base de datos para generar valores únicos. Se suele utilizar con bases de datos que soportan secuencias.
  - GenerationType.TABLE: Utiliza una tabla específica en la base de datos para generar valores únicos. Es una opción menos común.

```java
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name="id")
private Integer id;
@Column(name="nombre")
private String nombre;
@Column(name="apellidos")
private String apellidos;
```

Constructores con y sin parámetros: source/generate constructors

```java
public Cliente() {
    super();
}


public Cliente(String nombre, String apellidos) {
    super();
    this.nombre = nombre;
    this.apellidos = apellidos;
}
```

Métodos set/get: Set/get: source/generate setters/getters

Paquete Servicio - Anotaciones

Creamos la **interfaz ClienteServicio** y la **clase ClienteServicioImpl.java**
En la interfaz ClienteServicio debemos tener métodos que llamen a los métodos del repositorio. Copiamos los métodos de la interfaz del DAO y los insertamos en la interfaz del servicio.
En la clase **ClienteServicioImpl** añadimos los métodos no implementados.

A nivel de clase:

- **@Service:** Indicamos que esta clase es el servicio.

A nivel de métodos:

- **@Transactional:** Con esta anotación indicamos que la gestión de transacciones es automática. Esto significa que todas las operaciones deben completarse en su totalidad o no ejecutarse en absoluto, garantizando la integridad de los datos en caso de errores o fallos.
  @Transactional le indica al framework que debe gestionar la transacción para el método o la clase anotada. El framework se encargará de abrir, comprometer (commit) o revertir (rollback) la transacción según el éxito o fallo de la operación.

Si se coloca @Transactional a nivel de clase, todos los métodos de la clase serán transaccionales.

Paquete Repository - Anotaciones

Creamos una **interfaz que extiende a JpaRepository.**
Ejemplo:

public interface SongsRepository extends JpaRepository { Song, Integer}

Donde: Song es la clase entidad y Integer es el tipo de la clave primaria de la clase.

Consultar: https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html

Métodos principales de JpaRepository:

Método	Descripción
<S extends T> S **save**(S entity)	Inserta o actualiza (INSERT si el id es nuevo, UPDATE si existe).
<S extends T> List<S> **saveAll**(Iterable<S> entities)	Inserta o actualiza en lote.
Optional<T> **findById**(ID id)	Busca por ID.
boolean **existsById**(ID id)	Verifica si existe un registro con ese ID.
List<T> **findAll**()	Obtiene todos los registros.
List<T> **findAllById**(Iterable<ID> ids)	Busca por una lista de IDs.
long **count**()	Devuelve el número total de registros.
void **deleteById**(ID id)	Elimina un registro por ID.
void **delete**(T entity)	Elimina un registro pasando la entidad.
void **deleteAll**(Iterable<? extends T> entities)	Elimina un lote de registros.
void **deleteAll**()	Elimina todos los registros.

Métodos derivados de JpaRepository: Spring Data JPA genera la consulta automáticamente según el nombre del método, siguiendo esta estructura:

[Acción]By[Atributo][Condición][Conectores(lógicos)][Orden]

Donde:

1. Acción → Lo que quieres obtener:
   - find → devuelve entidades.
   - read / get → sinónimos de find.
   - count → devuelve un número.
   - delete / remove → borra entidades.
   - exists → devuelve un booleano.

2. By → Obligatorio para empezar las condiciones.

3. Atributo → Nombre EXACTO del atributo de la entidad (respeta mayúsculas en camelCase).
   - Ej.: Nombre, Precio, Estado, FechaRegistro.

4. Condición u Operador → Define cómo comparar el atributo:
   - Is, Equals → igual (por defecto no hace falta ponerlo).
   - Containing → LIKE %valor%
   - StartingWith → LIKE valor%
   - EndingWith → LIKE %valor
   - GreaterThan, LessThan, Between, In, NotIn, Before, After, etc.
   - IgnoreCase → ignora mayúsculas/minúsculas.

5. Conectores lógicos → And, Or para unir condiciones.

6. Orden → Puedes ordenar los resultados:
   - OrderBy[Atributo][Asc|Desc]
   - También se puede usar Top1, First1, etc., antes del By

Ejemplos:

Operador	Ejemplo de método	Equivalente SQL
Is / Equals	findBy*Nombre*(String nombre)	WHERE nombre = ?
Not	findBy*Nombre***Not**(String nombre)	WHERE nombre <> ?
IsNull	findBy*Nombre***IsNull**()	WHERE nombre IS NULL
IsNotNull / NotNull	findBy*Nombre***IsNotNull**()	WHERE nombre IS NOT NULL
True	findBy*Activo***True**()	WHERE activo = TRUE
False	findBy*Activo***False**()	WHERE activo = FALSE
Before	findBy*Fecha***Before**(Date fecha)	WHERE fecha < ?
After	findBy*Fecha***After**(Date fecha)	WHERE fecha > ?
GreaterThan	findBy*Precio***GreaterThan**(Float precio)	WHERE precio > ?
GreaterThanEqual	findBy*Precio***GreaterThanEqual**(Float precio)	WHERE precio >= ?
LessThan	findBy*Precio***LessThan**(Float precio)	WHERE precio < ?
LessThanEqual	findBy*Precio***LessThanEqual**(Float precio)	WHERE precio <= ?
Between	findBy*Precio***Between**(Float min, Float max)	WHERE precio BETWEEN ? AND ?
Like	findBy*Nombre***Like**(String nombre)	WHERE nombre LIKE ?
NotLike	findBy*Nombre***NotLike**(String nombre)	WHERE nombre NOT LIKE ?
StartingWith	findBy*Nombre***StartingWith**(String prefijo)	WHERE nombre LIKE 'prefijo%'
EndingWith	findBy*Nombre***EndingWith**(String sufijo)	WHERE nombre LIKE '%sufijo'
Containing	findBy*Nombre***Containing**(String texto)	WHERE nombre LIKE '%texto%'
OrderBy	findBy*Nombre***OrderBy***Precio***Asc**(String nombre)	ORDER BY precio ASC
In	findBy*Id***In**(List<Long> ids)	WHERE id IN (?, ?, ?)
NotIn	findBy*Id***NotIn**(List<Long> ids)	WHERE id NOT IN (?, ?, ?)
Top / First	findTop1By**OrderBy***Precio***Desc**()	ORDER BY precio DESC LIMIT 1
Distinct	find**Distinct**By*Nombre*(String nombre)	SELECT DISTINCT …
And	findBy*Nombre***And***Estado*(String nombre, String estado)	WHERE nombre = ? AND estado = ?
Or	findBy*Nombre***Or***Estado*(String nombre, String estado)	WHERE nombre = ? OR estado = ?
Count	**countBy**Nombre(String nombre)	SELECT COUNT(*) WHERE estado = ?
ContainingIgnoreCase	findByNombre**ContainingIgnoreCase**(String nombre)	WHERE UPPER(nombre) LIKE UPPER('%?%')

Otras consultas: JPQL

Se hacen las consultas sobre las entidades de JAVA no de Base de datos

1. **Consulta con varios filtros**

```
@Query("SELECT c FROM Cliente c WHERE c.nombre = :nombre AND c.apellidos = :apellidos")
List<Cliente> buscarPorNombreYApellidos(@Param("nombre") String nombre, @Param("apellidos") String apellidos);
```

2. **Consulta con LIKE**

```
@Query("SELECT c FROM Cliente c WHERE c.nombre LIKE %:nombre%")
List<Cliente> buscarPorNombreLike(@Param("nombre") String nombre);
```

3. **Consulta con ordenación**

```
@Query("SELECT c FROM Cliente c ORDER BY c.apellidos ASC")
List<Cliente> buscarTodosOrdenados();
```

4. **Conteo**

```
@Query("SELECT COUNT(c) FROM Cliente c WHERE c.nombre = :nombre")
long contarPorNombre(@Param("nombre") String nombre);
```

5. GroupBy

```
@Query("SELECT c.apellidos, COUNT(c) FROM Cliente c GROUP BY c.apellidos")
List<Object[]> contarClientesPorApellido();
```

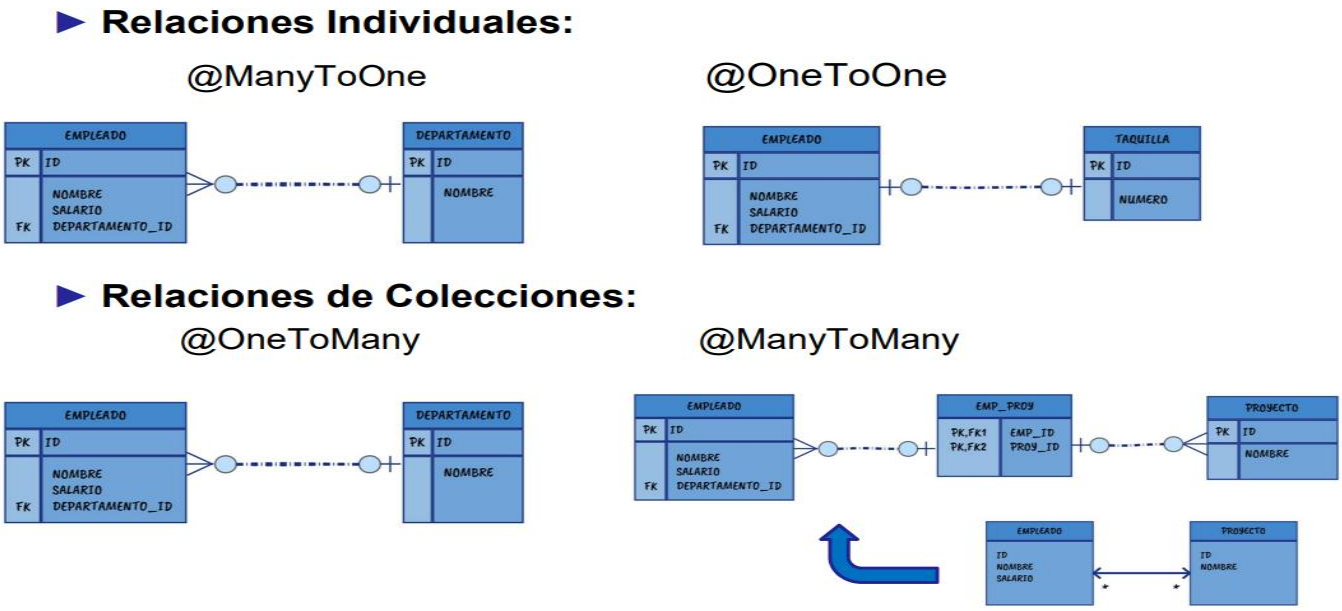
6. UPDATE

```
@Modifying
  @Query("UPDATE Cliente c SET c.direccion.ciudad = 'Sevilla' WHERE LOWER(c.nombre) LIKE 'a%'")
  int actualizarCiudadSevillaParaClientesConA();
```

Relación entre tablas one to one (unidireccional)

Supongamos que la tabla cliente se relaciona con la tabla dirección. Un cliente tiene una dirección y una dirección pertenece sólo a un cliente. Al ser unidireccional, el cliente conoce la existencia de su dirección pero dada una dirección, no es posible conocer el cliente.

La tabla cliente en bd tendrá un campo id_direccion que será una FK al campo id de la tabla



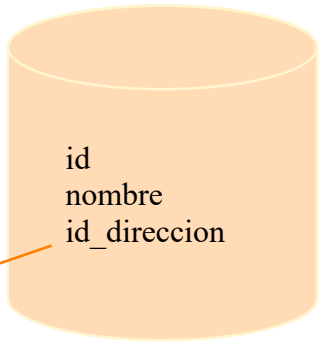
Dirección. ¿Cómo mapeamos esta relación de tablas?

```java
@Entity
public class Cliente {

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

    private String nombre;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "id_direccion")
    private Direccion direccion;
```



Clientes

- ✓ **@OneToOne:** Indica la relación entre las entidades, en este caso es de tipo One to One (1-1).
- ✓ **@JoinColumn:** Indica que id_direccion es la clave foránea que referencia a Direccion. Este nombre se usa como columna de unión en la base de datos.

Relación entre tablas one to one (bidireccional)

Si la relación entre tablas es bidireccional, tanto Cliente como Direccion tendrán referencias entre sí. Ahora, a partir de una dirección se conoce el cliente. ¿Cómo se mapea? En Cliente no hay que hacer nada, sólo en la clase Dirección: hay que indicar que el mapeo ya se ha realizado con Cliente a través del atributo dirección.

```java
@Entity
public class Cliente {

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

    private String nombre;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "id_direccion")
    private Direccion direccion;
```

```java
@Entity
public class Direccion {

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

    private String calle;
    private String ciudad;
    private String codigoPostal;

    @OneToOne(mappedBy = "direccion")
    private Cliente cliente;
```

La base de datos no cambia. Hibernate maneja la relación y la estructura de la base de datos, teniendo en cuenta que la tabla Direccion no necesita una columna adicional para la clave foránea en esta configuración. La entidad Cliente tiene la referencia a Direccion, y la entidad Direccion tiene una referencia a Cliente para completar la relación bidireccional.

Las **consultas derivadas** en Spring Data JPA permiten generar automáticamente las sentencias SQL o JPQL necesarias basándose únicamente en el **nombre del método** del repositorio. Cuando existe una **relación One-to-One**, Spring permite **navegar entre las entidades relacionadas** concatenando los nombres de los atributos.

Sintaxis general:

findBy<AtributoRelacion><CampoDeLaEntidadRelacionada>

Método derivado	Qué hace (en SQL/JPQL)
findByDireccionCiudad(String ciudad)	SELECT c FROM Cliente c JOIN c.direccion d WHERE d.ciudad = :ciudad
findByDireccionCalle(String calle)	WHERE d.calle = :calle
findByDireccionCiudadIgnoreCase(String ciudad)	Ignora mayúsculas/minúsculas en la comparación (LOWER(d.ciudad))
findByDireccionCiudadAndNombre(String ciudad, String nombre)	WHERE d.ciudad = :ciudad AND c.nombre = :nombre
findByDireccionCiudadOrNombre(String ciudad, String nombre)	WHERE d.ciudad = :ciudad OR c.nombre = :nombre
findByDireccionIsNull()	Clientes sin dirección (WHERE c.direccion IS NULL)
findByDireccionCiudadStartingWith("Se")	Ciudades que comienzan con "Se"

Método derivado	Qué hace (en SQL/JPQL)
findByDireccionCiudadContainingIgnoreCase("sev")	Ciudades que contengan "sev" sin importar mayúsculas
countByDireccionCiudad(String ciudad)	Número de clientes en una ciudad.

Relación entre tablas one to many (unidireccional)

Veamos un ejemplo. Imaginemos que un Cliente puede tener múltiples Pedidos, mientras que cada Pedido está asociado a un solo Cliente.

PASO 1: Crear la entidad principal (Cliente)

```java
@Entity
public class Cliente {

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

    private String nombre;

    @OneToMany
    @JoinColumn(name = "cliente_id")  // La clave foránea
    private List<Pedido> pedidos = new ArrayList<>();
```

- ✓ **@OneToMany:** Define la relación uno a muchos.
- ✓ **@JoinColumn:** Indica que cliente_id es la clave foránea que referencia a Pedido. Este nombre se usa como columna de unión en la base de datos.

PASO 2: Crear la entidad secundaria (Pedido)
La clase Pedido quedará así:

```java
@Entity
public class Pedido {

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

    private String descripcion;
```

En Base de datos tendríamos:

Importante: En el caso de una relación uno a muchos unidireccional, la clave foránea (cliente_id) se encuentra en la tabla del lado "muchos" (en este caso, Pedido), pero en el modelo de datos de la clase Pedido, no necesitas definir un atributo para la clave foránea cliente_id en la clase Pedido si la relación es unidireccional y el mapeo es gestionado solo por la clase Cliente.

CREATE TABLE Cliente (id INT AUTO_INCREMENT PRIMARY KEY, nombre VARCHAR(255));	CREATE TABLE Pedido (id INT AUTO_INCREMENT PRIMARY KEY, descripcion VARCHAR(255), cliente_id INT, FOREIGN KEY (cliente_id) REFERENCES Cliente(id));

NOTA: Cuando tienes una relación uno a muchos y haces una consulta que involucra esta relación, es posible que se produzcan problemas con duplicados si la relación no está correctamente mapeada. Para evitar estos problemas, se recomienda usar Set en lugar de List para las colecciones de una relación uno a muchos.

Relación entre tablas one to many (bidireccional)

A partir de un Cliente puedo conocer su lista de pedidos y a partir de un pedido puedo saber qué cliente le pertenece.
Hay que modificar la clase Cliente para especificar con **mappedBy** y el atributo donde está la FK en la clase Pedido.

```java
@Entity
public class Cliente {

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

    private String nombre;

    @OneToMany(mappedBy = "cliente")
    private Set<Pedido> pedidos = new HashSet<>();
```

En la clase pedido:

```java
@Entity
public class Pedido {

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

    private String descripcion;

    @ManyToOne
    @JoinColumn(name = "cliente_id")
    private Cliente cliente;
```

- ✓ **@ManytoOne:** Indica la relación entre las entidades. Indica que muchos pedidos pueden estar asociados a un solo cliente.
- ✓ **@JoinColumn:** Indica que cliente_id es la clave foránea que referencia a Cliente. Este nombre se usa como columna de unión en la base de datos.

Relación entre tablas many to many (unidireccional)

Supongamos que, un cliente puede tener muchos productos y un producto puede ser asociado con muchos clientes. La relación se mantiene unidireccional desde Cliente hacia Producto, es decir, el Cliente conoce los productos que tiene, pero el Producto no tiene una referencia de vuelta al Cliente. En este tipo de relaciones, es necesaria una tabla de unión cliente_producto.

```java
@Entity
public class Cliente {

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

    private String nombre;

    @ManyToMany
    @JoinTable(
        name = "cliente_producto",
        joinColumns = @JoinColumn(name = "cliente_id"),
        inverseJoinColumns = @JoinColumn(name = "producto_id")
    )
    private Set<Producto> productos = new HashSet<>();
```

- ✓ **@ManytoMany:** Indica la relación entre las entidades. Indica que muchos pedidos pueden estar asociados a un solo cliente.

- ✓ **@JoinTable:** Especifica la tabla de unión cliente_producto que se utilizará para gestionar la relación muchos a muchos.

    - joinColumns: Define la columna en la tabla de unión que se refiere al Cliente.
    - inverseJoinColumns: Define la columna en la tabla de unión que se refiere al Producto.

```java
import javax.persistence.*;

@Entity
public class Producto {

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

    private String nombre;

    // No se define la relación desde Producto hacia Cliente
```

En este ejemplo, solo la entidad Cliente mantiene la relación con Producto. La entidad Producto no tiene un campo para la relación con Cliente, haciendo que la relación sea unidireccional.

La Estructura de la Base de Datos sería:

```
CREATE TABLE Cliente (
   id INT AUTO_INCREMENT PRIMARY KEY,
   nombre VARCHAR(255)
);

CREATE TABLE Producto (
   id INT AUTO_INCREMENT PRIMARY KEY,
   nombre VARCHAR(255)
);

CREATE TABLE cliente_producto (
   cliente_id INT,
   producto_id INT,
   PRIMARY KEY (cliente_id, producto_id),
   FOREIGN KEY (cliente_id) REFERENCES Cliente(id),
   FOREIGN KEY (producto_id) REFERENCES Producto(id)
);
```

Relación entre tablas many to many (bidireccional)

En este caso es necesario que en la clase Pedido haya un atributo con la lista de los clientes ya que se trata de una relación bidireccional. Con mappedBy se indica que el mapeo de hace con el atributo pedidos de Cliente.

```
@Entity
public class Pedido {

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

    private String descripcion;

    @ManyToMany(mappedBy = "pedidos")
    private Set<Cliente> clientes = new HashSet<>();
```

- ✓ **@ManyToMany(mappedBy = "pedidos")** indica que esta relación está mapeada en la entidad Cliente y se utiliza el nombre del campo de la otra entidad (pedidos).

La base de datos no se modifica.

USO DE CASCADE

En Hibernate, la opción de cascade se utiliza para simplificar operaciones en nuestro código. Determina si las operaciones de persistencia, actualización o borrado se propaga automáticamente hacia las entidades relacionadas. Si por ejemplo tenemos una relación usuario – dirección y queremos eliminar un usuario, su dirección también se debería eliminar. Pero esto no siempre es así.

Tipos de cascada que puedes usar para propagar operaciones entre entidades. Aquí están los principales:

1. CascadeType.PERSIST: Propaga la operación de persistencia. Cuando se **inserte** una entidad, las entidades asociadas también se persisten.
2. CascadeType.MERGE: Propaga la operación de mezcla (merge). Cuando se actualiza una entidad (merge), las entidades asociadas también se **actualizan**.
3. CascadeType.REMOVE: Propaga la operación de eliminación. Cuando se **elimina** una entidad, las entidades asociadas también se eliminan.
4. CascadeType.ALL: Aplica todas las operaciones de cascada (PERSIST, MERGE, REMOVE, REFRESH, DETACH).

Cómo se usa Cascade

El atributo cascade se define en una relación entre entidades. Se usa en las anotaciones que definen

la relación, como @OneToMany, @ManyToMany, @OneToOne, etc.

Ejemplo: Uso de Cascade en una Relación @OneToMany

Imaginemos que tenemos las entidades Cliente y Pedido con una relación uno a muchos (un cliente puede tener muchos pedidos). Supongamos que queremos que, al persistir un Cliente, también se persistan automáticamente todos los Pedidos asociados:

@OneToMany(cascade = CascadeType.ALL)

EAGER Y LAZY

En JPA , EAGER y LAZY son estrategias para definir cómo se deben cargar las entidades asociadas cuando se realiza una consulta a la base de datos. Estas estrategias son aplicables a relaciones entre entidades, como @OneToMany, @ManyToMany, @OneToOne, etc.

- ✓ **EAGER (Carga Ansiosa)**

  EAGER indica que la relación entre entidades debe cargarse inmediatamente cuando se consulta la entidad principal. Esto significa que cuando se recupera una entidad, también se recuperan automáticamente las entidades relacionadas.

  Uso Típico: Utiliza EAGER cuando sabes que siempre necesitarás los datos relacionados junto con la entidad principal. Por ejemplo, si siempre necesitas los detalles de los pedidos de un cliente cuando obtienes la información del cliente.

- ✓ **LAZY (Carga Perezosa)**

  LAZY indica que las entidades relacionadas deben cargarse solo cuando realmente se necesitan. Inicialmente, solo se carga la entidad principal y las relaciones se cargan bajo demanda.

  Uso Típico: Utiliza LAZY cuando las entidades relacionadas pueden no ser necesarias en todas las operaciones. Esto puede mejorar el rendimiento al evitar la carga de grandes cantidades de datos innecesarios.

Por defecto en JPA está configurado de este modo según el tipo de relación:

- OneToOne y ManyToOne: EAGER
- OneToMany y ManyToMany: LAZY

Relación	Modo
OneToMany	LAZY
ManyToMany	LAZY
OneToOne	EAGER
ManyToOne	EAGER

Para modificar el comportamiento se utiliza fetch:

```
@OneToMany(fetch = FetchType.EAGER, mappedBy = "autor")
private Set<Libro> libros = new HashSet<>();
```

Problemas con JOIN donde la columna no puede ser NULL

Es posible que, al insertar una entidad con una relación OneToMany que también se debe crear, Hibernate nos genere un error porque la columna por la que tenemos que hacer el JOIN (FK) no puede ser NULL o no tiene un valor predeterminado o por defecto.
Esto es porque Hibernate está intentando hacer insertar todos los registros y luego actualizar las claves foráneas.
La solución es indicar en nuestra anotación JoinColumn el atributo nullable = false. Ejemplo:

```
@OneToMany(cascade = CascadeType.ALL)
@JoinColumn(name = "id_pedido", nullable = false)
private List<PedidoLinea> lineas;
```

Probar aplicación

Usaremos la interfaz CommandLineRunner para poder ejecutar código después de que la aplicación se haya inicializado correctamente y antes de que el servidor comience a aceptar solicitudes HTTP.

Es necesario el método **run** que se invoca automáticamente cuando la aplicación arranca.

```
@SpringBootApplication
public class Ejercicio06Application implements CommandLineRunner {

    public static void main(String[] args) {
        SpringApplication.run(Ejercicio06Application.class, args);
    }

    @Autowired
    ClienteService clienteService;

    @Override
    public void run(String... args) throws Exception {
```

Lombok

Lombok es una biblioteca Java que ayuda a reducir el código repetitivo, como los métodos getter, setter, constructores, toString, equals, y hashCode, mediante anotaciones en tiempo de compilación. Es decir, no tenemos que especificar esos métodos. Al usar Lombok, puedes hacer que tu código sea más conciso y fácil de mantener.

Primero: Para usar Lombok hay que añadir la dependencia con Maven en el archivo pom.xml.

```xml
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.34</version>
    <scope>provided</scope>
</dependency>
```

Segundo: Hay que instalarlo en el IDE. En eclipse STS no viene incluido por defecto. Para ello hay que descargar el jar de Lombok desde su sitio oficial. Una vez descargado el jar de Lombok, ejecuta el siguiente comando en la terminal o haz doble clic en el archivo jar para abrir el instalador:
java -jar lombok.jar

El instalador de Lombok debería detectar automáticamente tu instalación de Eclipse STS. Si no, selecciona manualmente la ruta de instalación de Eclipse STS. Haz clic en el botón "Install/Update" para instalar Lombok en Eclipse STS.

Por último, reinicia Eclipse STS. En Help -> About Eclipse -> Installation Details -> Installed Software y verifica que Lombok esté. Si no lo estuviera, ve a Window -> Preferences -> Java -> Compiler -> Annotation Processing y asegúrate de que "Enable annotation processing" esté marcado.

Tercero: Utilizar anotaciones de Lombok en las clases. Anotaciones de Lombok:

1. **@Data**: Combina @Getter, @Setter, @ToString, @EqualsAndHashCode, y **@RequiredArgsConstructor** en una sola anotación.
2. **@NoArgsConstructor**: Genera un constructor sin argumentos.
3. **@AllArgsConstructor**: Genera un constructor con un argumento para cada campo en la clase.
4. **@Getter** y **@Setter**: Genera métodos getter y setter para cada campo.
5. **@ToString**: Genera un método toString.
6. **@EqualsAndHashCode**: Genera métodos equals y ha

```java
@Data
@Entity
@Table(name="clientes")
public class Cliente {

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

    @Column
    private String nombre;

    @Column
    private String apellidos;
}
```