Conceptos importantes

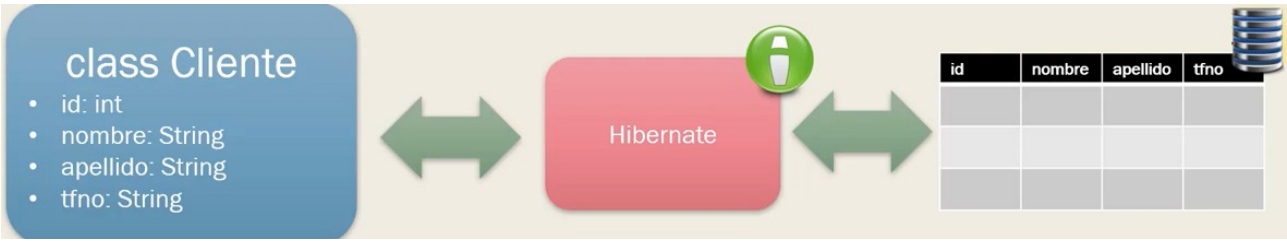
Hibernate:

Framework que facilita el acceso a BD desde aplicaciones Java. Permite leer, actualizar o eliminar información.

Utiliza O.R.M (Mapa de objeto relacional) para el acceso a datos. Es decir, accede a BD usando el concepto de POO, transformando las tablas en objetos: entidades con propiedades.

O.R.M automatiza el acceso a datos y de esta forma suprimimos la capa del lenguaje SQL (capa intermedia de abstracción).

Hibernate, lee una clase e interpreta que cada una de las propiedades corresponde a un campo de una tabla almacenada en una base de datos. Todas las transacciones sobre las tablas se harán de forma automática.



JPA: Java Persistence API

Es una especificación de Java para acceder, gestionar y persistir datos entre objetos Java y bases de datos relacionales, esto es usando ORM.

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.

Define una API estándar para realizar operaciones CRUD y consultas en la base de datos.

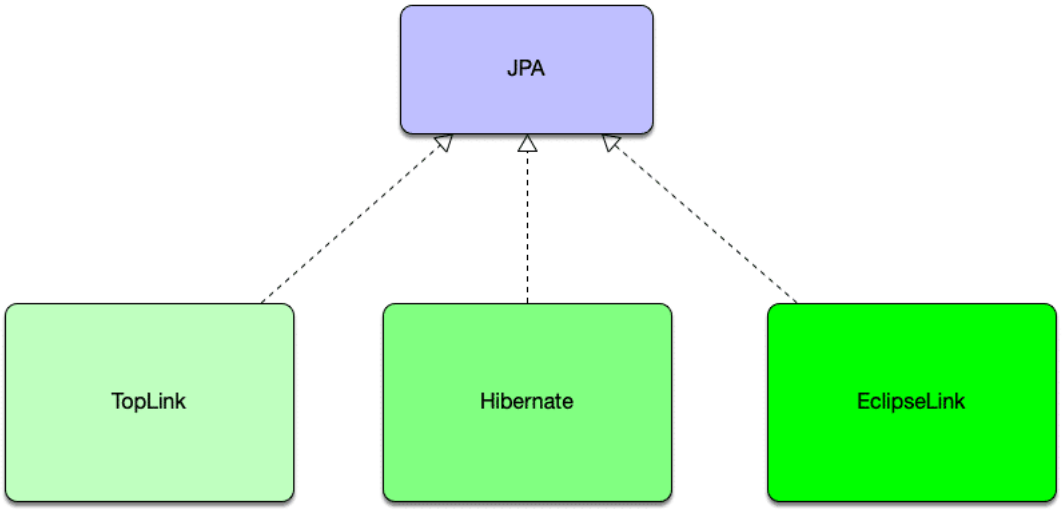
Importante:
JPA es un estándar, una especificación, mientras que Hibernate es una implementación concreta de la especificación JPA. Es decir, Hibernate cumple con las reglas e interfaces definidas por JPA.

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),…



Requisitos para usar Hibernate con Spring Boot

- Instalar SGBD:

  - MariaDB. Nosotros usaremos este.
  - MySQL Workbench
  - PHPMyAdmin

- Librerías Hibernate (Mínimo Java 8): La dependencia de **Spring Data JPA** en Spring Boot, automáticamente incluye Hibernate como el proveedor JPA predeterminado. Viene con todas las bibliotecas necesarias para trabajar con JPA e Hibernate, facilitando la configuración y el uso de estas tecnologías en tu aplicación Spring Boot.

La dependencia se puede agregar tanto manualmente en el archivo pom.xml de Maven (copiamos las dependencias de la última versión estable y la incluimos en el fichero pom.xml), como con el asistente o inicializador de Spring Boot.

- Dependencias a driver JDBC: Usaremos **mariaDB**. Se puede agregar manualmente en el pom (Maven) o utilizando el asistente o inicializador de Spring Boot.

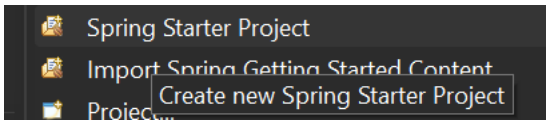
- Conexión BBDD: en application.properties configuramos la cadena de conexión.

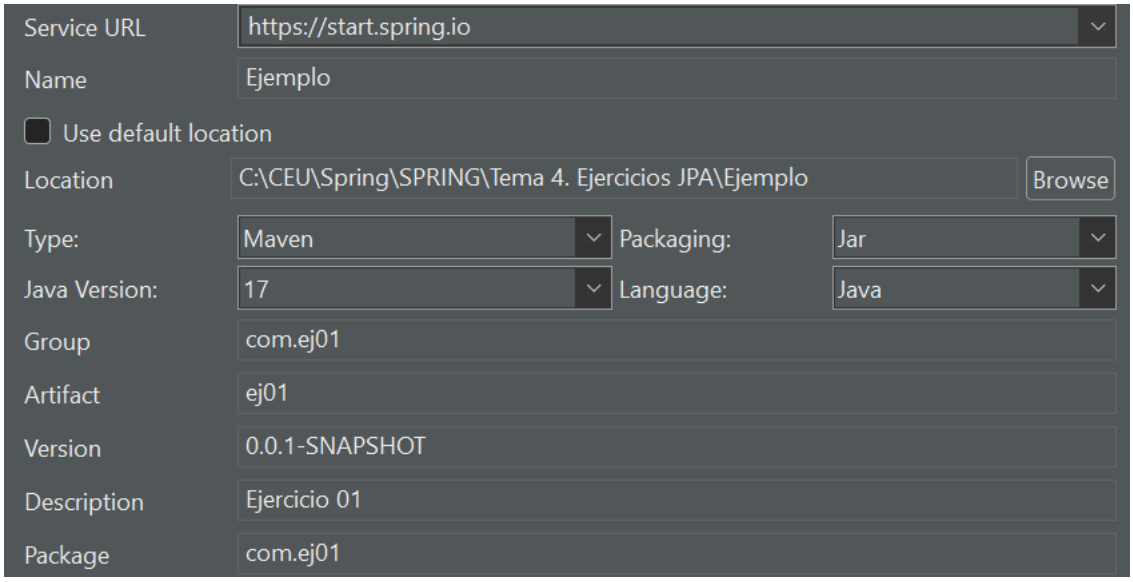
```
# Configuración de la base de datos
spring.datasource.url=jdbc:mariadb://localhost:3306/hibernate
spring.datasource.username=user
spring.datasource.password=password

#Permite ver las consultas SQL en la consola
spring.jpa.show-sql=true
```

#Habilita el formato de las consultas, haciendo que sean más legibles al agregar saltos de línea y #sangrías

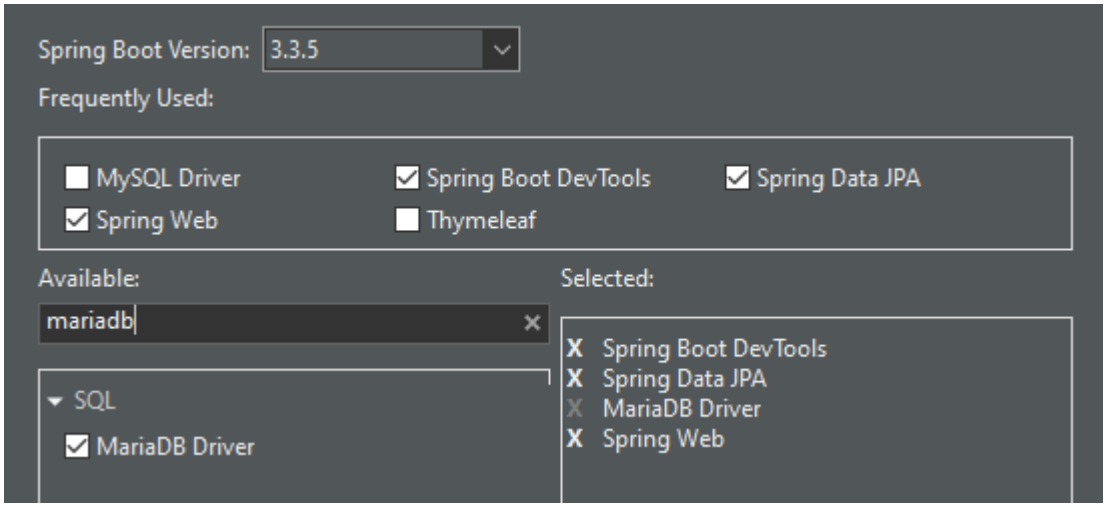
spring.jpa.properties.hibernate.format_sql=true

Ejemplo: Crear un proyecto de prueba en eclipse STS con Spring Starter Project:





Añadimos Spring Data JPA y el conector de mariadb para que se añadan las dependencias de hibernate y el driver.



Configurar la base de datos

1) En MySQL Workbench o MariaDB creamos una BD de prueba. Por ejemplo: hibernate

   CREATE DATABASE hibernate;

```
show databases;
use hibernate;
```

2) Crear usuario

   CREATE USER '**mi_usuario**'@'localhost' IDENTIFIED BY '**mi_contraseña**';
   GRANT ALL PRIVILEGES ON **nombre_BD**.* TO '**mi_usuario**'@'localhost';
   FLUSH PRIVILEGES;

Fichero de configuración

En **application.properties** de nuestro proyecto, configuramos los datos de la conexión:
NOTA: Usar el puerto donde esté instalada la BD.

Con MySQL:

```
spring.datasource.url=jdbc:mysql://localhost:3306/hibernate
spring.datasource.username=BELEN
spring.datasource.password=BELEN
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect
```

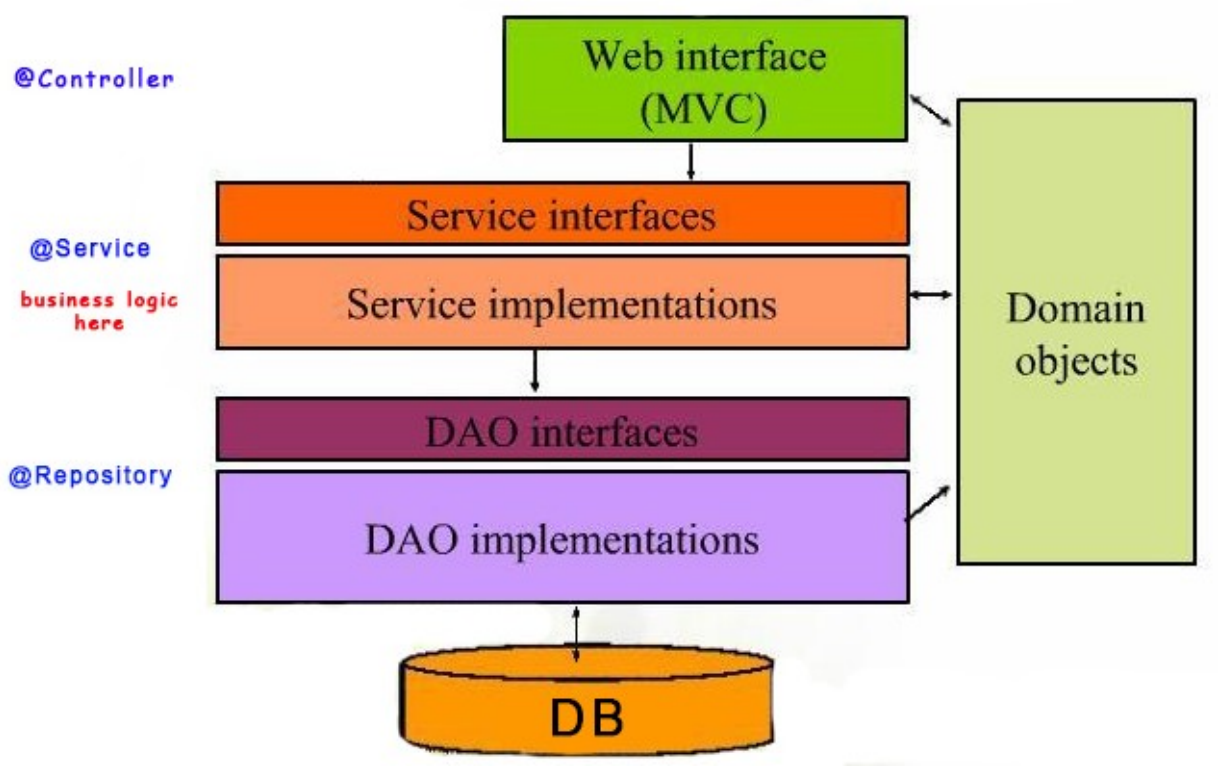
Con MariaDB:

```
spring.datasource.url=jdbc:mariadb://localhost:3307/hibernate
spring.datasource.username=belen
spring.datasource.password=belen
```

Actualizamos dependencias, botón derecho sobre el proyecto : Maven / Update Project
Para compilar por consola: mvn clean package -U

Estructura en src/main/java:

- **paquete controller**
  - clase controller: Ej ClienteController.java
- **paquete dao:**
  - interfaz dao: Ej ClienteDAO.java
  - clases dao: Ej ClienteDAOImpl.java
- **paquete modelo:** Clases que se mapean: Ej Cliente.java
- **paquete servicio:**
  - interfaz servicio: Ej ClienteService.java
  - clase servicio: Ej ClienteServiceImpl.java



Mapeo de Entidades - Modelo - 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)

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 {
```

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.

```
@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

```
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

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.

```
<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

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

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

    @Column
    private String nombre;

    @Column
    private String apellidos;
}
```

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;
```

- ✓ **@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 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.

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.

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.

```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:** Indica la relación entre las entidades, en este caso es de tipo One to Many (1-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.

  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:

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.

```
@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:

```
@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.

```java
@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 especificar cómo deben propagarse las operaciones realizadas sobre una entidad a las entidades asociadas. Esto es útil cuando se gestionan relaciones entre entidades, ya que permite que las operaciones en BD Tipos de Cascade

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;
```

Operaciones de base de datos – DAO - Anotaciones

Creamos la **interfaz ClienteDAO.java** en el paquete dao y definimos los métodos que usaremos para interactuar con la base de datos.

```
public interface ClienteDAO {

    List<Cliente> getClientes();

    Cliente get(int id);

    void save(Cliente cliente);

    void delete(int id);
}
```

Creamos **la clase ClienteDAOImpl** que implementará la interfaz anterior.
Una vez creada, sobre el nombre de la clase, pulsamos sobre 'add unimplemented methods'

A nivel de clase:

- **@Repository:** Indicamos que la clase es un DAO o repositorio jpa.

```
@Repository
public class ClienteDAOImpl implements ClienteDAO {
```

Servicios - Anotaciones

Creamos la **interfaz ClienteServicio** y la **clase ClienteServicioImpl.java**
En la interfaz ClienteServicio debemos tener métodos que llamen a los métodos del DAO. 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.

Operaciones de Base de datos

Para interactuar con la base de datos, se utiliza la interfaz **EntityManager**. Actúa como un puente entre el código java y la base de datos. Nos permite realizar operaciones de BD: actualizar, eliminar, recuperar ...

La forma de utilizar este atributo privado es con la anotación @PersistenceContext (también se puede usar @Autowired):

```
@Repository
public class ClienteRepositoryImpl implements ClienteRepository{


    @PersistenceContext
    private EntityManager entityManager;
```

Las operaciones que cambian el estado de la base de datos (como inserciones, actualizaciones o borrado) deben realizarse dentro de una transacción por lo que lo más habitual es tener la anotación *@Transactional* en el servicio.

Inserción de entidades en BD

Para insertar en base de datos se usa el método **persist** de entityManager. Este método necesita como parámetro el objeto que vamos a insertar.
Ojo: si la entidad tiene dependencias, habrá que insertarlas manualmente de forma separadas, (primero las referenciadas), a no ser que tenga el atributo cascade.

```
@Override
public void insertaCliente(Cliente cliente) {

    //es necesario @transaccional en el servicio
    entityManager.persist(cliente);

}
```

NOTA: El servicio debe tener la anotación **@Transactional**.

Actualización de entidades en BD

Para actualizar una entidad se utiliza **merge**. Devuelve la nueva que está sincronizada con la base de datos. Si la entidad no tiene el atributo cascade, es necesario hacer las modificaciones de forma manual.

```
@Override
public void actualizarCliente(Cliente cliente) {

    entityManager.merge(cliente);
}
```

NOTA: El servicio debe tener la anotación @Transactional.

Consultar entidades de BD

Para consultar una entidad, usamos el método **createQuery** con dos parámetros:

1.- El primer parámetro es la consulta sobre la entidad.
2.- El segundo parámetro es la entidad que vamos a consultar:

```
@Override
public List<Cliente> getClientes() {


    Query<Cliente> query = (Query<Cliente>) entityManager.createQuery("select c from Cliente c", Cliente.class);
    List<Cliente> lista = query.getResultList();
    return lista;
}
```

También se permite la consulta “from Cliente”, aunque es menos recomendable.
Se utiliza getResultList() para ejecutar una consulta y obtener los resultados como una lista.

Si la consulta lleva parámetros, se usarán tantos métodos setParameter como parámetros haya:

```
// Crear la consulta JPQL
String jpql = "SELECT c FROM Cliente c WHERE c.nombre LIKE :nombre";

// Crear la consulta
Query<Cliente> query = (Query<Cliente>) entityManager.createQuery(jpql, Cliente.class);

// Configurar el parÃ¡metro
query.setParameter("nombre", "%" + nombre + "%");

// Ejecutar la consulta y obtener la lista de clientes
return query.getResultList();
```

Consulta una entidad por id

Utilizamos el método **find** pasándole el id de la entidad. Si no lo encuentra devuelve null.

```
@Override
public Cliente getCliente(Integer id) {

    Cliente cliente = entityManager.find(Cliente.class, id);
    return cliente;
}
```

Eliminar una entidad

Se utiliza **remove**. Para que funcione correctamente, la entidad debe estar en un estado gestionado por el contexto de persistencia antes de eliminarla, por lo que, si no lo está, hay que obtener la entidad con find. Una vez que la entidad ya está persistida, se puede eliminar utilizando remove. Si la entidad no tiene el atributo cascade, es necesario hacer las eliminaciones de forma manual.

```
@Override
public void deleteCliente(Integer id) {

    // necesario @transaccional en el servicio.
    Cliente c = getCliente(id);
    entityManager.remove(c);

}
```

NOTA: El servicio debe tener la anotación @Transactional.