Programación Frontend y Backend

BLOQUE ACCESO A DATOS

JPA

GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, COMERCIO Y TURISMO
EOI Escuela de organización industrial
Unión Europea
Fondo Social Europeo
Iniciativa de Empleo Juvenil
El FSE invierte en tu futuro
SISTEMA NACIONAL DE GARANTÍA JUVENIL

01 Conceptos Básicos

GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, COMERCIO Y TURISMO
EOI Escuela de organización industrial

Unión Europea
Fondo Social Europeo
Iniciativa de Empleo Juvenil
El FSE invierte en tu futuro

SISTEMA NACIONAL DE GARANTÍA JUVENIL

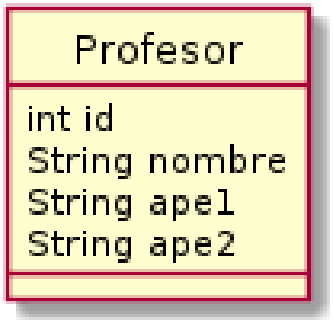
Hibernate

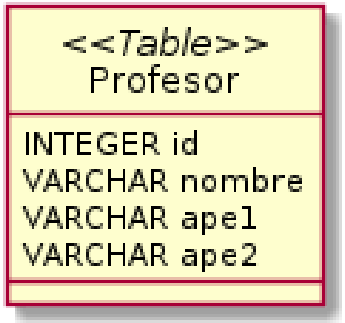
Hibernate es una herramienta de mapeo objeto-relacional (**ORM**) que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante archivos declarativos (XML) o anotaciones en los beans de las entidades que permiten establecer estas relaciones.

GOBIERNO DE ESPAÑA · MINISTERIO DE INDUSTRIA, COMERCIO Y TURISMO · EOI Escuela de organización industrial
Unión Europea · Fondo Social Europeo · Iniciativa de Empleo Juvenil · El FSE invierte en tu futuro
SISTEMA NACIONAL DE GARANTÍA JUVENIL

Hibernate es:

- **ORM** (Object Relational Mapper). Facilita el mapeo entre objetos de una BD relacional y objetos de aplicación. El código Java que, dado un objeto Profesor , genera una fila en la tabla Profesor o la borra o la actualiza, etc, es un ORM.





GOBIERNO DE ESPAÑA MINISTERIO DE INDUSTRIA, COMERCIO Y TURISMO
EOI Escuela de organización industrial
Unión Europea
Fondo Social Europeo
Iniciativa de Empleo Juvenil
El FSE invierte en tu futuro
SISTEMA NACIONAL DE GARANTÍA JUVENIL

Hibernate es:

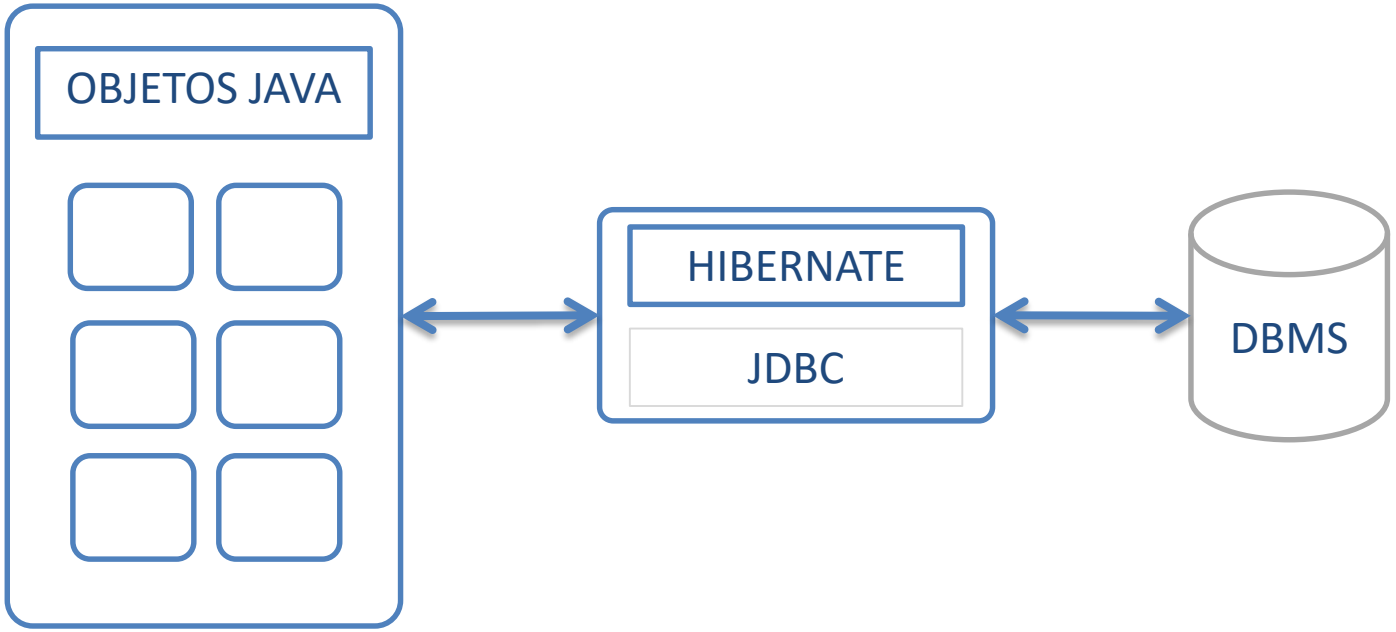
- **ORM**
  - Cuando usamos **JDBC** directamente tiene muchos problemas, ya que para crear el mantenimiento de una tabla, tenemos que crear una estructura de código que se repite bastante con cada una de las tablas.

  - Lo normal es que para evitar toda esta repetición de código, los programadores empecemos a hacer nuestras utilidades que nos ahorren tiempo y esfuerzo, así es como se creó ORM y lo acabó implementando Hibernate, por lo que podemos decir que Hibernate es un conjunto de clases que nos ayudan a hacer **CRUD** sobre nuestra BD.

Hibernate es:

- **Implementación de JPA,** JPA es la API de persistencia de Java que define las interfaces para acceder a bases de datos como lo es JDBC.
- **JPQL** Lenguaje de Consultas orientado a objetos
- **Anotaciones** Configuración y Mapeo de objetos.

Hibernate



Ventajas

- ✓ **Automatiza la persistencia (base de datos relacionales – POO)**
- ✓ **Automatiza las relaciones (base de datos relacionales – Objetos POO relacionales)**
- ✓ **Soporte de consultas (HQL)**
- ✓ **Permite las consultas en SQL nativo**
- ✓ **Abstrae del tipo de base de datos**
- ✓ **Automatiza el mapeo entre Objetos Java – DBMS**
- ✓ **Reduce el coste de mantenimiento**

Inconvenientes

- ✓ **Curva de aprendizaje**
- ✓ **Algunas queries (muy poquitas) no se puede hacer exactamente igual que con JDBC**

02 Anotaciones

GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, COMERCIO Y TURISMO
EOI Escuela de organización industrial

Unión Europea
Fondo Social Europeo
Iniciativa de Empleo Juvenil
El FSE invierte en tu futuro

SISTEMA NACIONAL DE GARANTÍA JUVENIL

@Entity Marca una clase como Entidad de base de datos. Desde ese momento, dicha clase "representará" una entidad en nuestro modelo que tiene que ser persistida en la base de datos. Se pone justo encima de la clase que hemos declarado (Entity).

```
@Entity
public class Alumno {
        private int id;
        private String nombre;

        public Alumno() { }
```

Implica:

Todas las propiedades privadas (con getters/setters)

Una clase con constructor público sin argumentos, y ningún otro.

@Table Nos permite definir detalles de la tabla que se usará para persistir la **@Entity** que es representada por la clase.

- Nombre de la tabla
- Catálogo al que pertenece
- Esquema en el que definirla
- Restricciones de unicidad

```
@Entity
@Table(name = "alumnos", schema="mi_prefijo")
public class Alumno {
        private int id;
        private String nombre;

        public Alumno() { }
```

@Id Como ya sabemos cada tabla va a tener una clave principal o clave primaria e Hibernate se va a encargar de la gestión de dichas claves mediante la anotación @Id. Se permiten @Id de campos múltiples

@GeneratedValue Define la estrategia de generación de identificadores mediante dos parámetros:

- strategy(GenerationType.AUTO):
  GenerationType.TABLE
  GenerationType.SEQUENCE
  GenerationType.IDENTITY

- generator: Se usa para definir el nombre de **@GenericGenerator** (definimos un generador propio de sequencias en Java).

GenerationType.AUTO no se recomienda el uso, ya que dejas a hibernate elegir.

GenerationType.TABLE usa una tabla de la base de datos para gestionar las claves

GenerationType.SEQUENCE La base de datos debe soportar la definición de secuencias.

GenerationType.IDENTITY Debe tener soporte para el tipo de columna IDENTITY ★

```
@Entity
@Table(name = "alumnos")
public class Alumno {
        @Id
        @GeneratedValue(strategy=GenerationType.AUTO)
        private int id;
        private String nombre;

        public Alumno() { }
```

@Column Define propiedades de la columna donde se mapeará un campo de un objeto dentro de la base de datos, las propiedades más comunes son:

- name: Define el nombre de la columna en la base de datos.
- length: Define la longitud del campo en la base de datos (muy útil para Strings).
- nullable: Define que la columna pueda contener valores nulos.
- unique: Permite poner una restricción de unicidad en dicha columna.

```
@Entity
@Table(name = "alumnos")
public class Alumno {
        @Id
        @GeneratedValue(strategy=GenerationType.AUTO)
        private int id;

        @Column(name = "first_name", unique="true")
        private String nombre;
```

@NotEmpty Comprueba que el campo de tipo String, Collection, Mapa o Array no sean nulos o estén vacíos.

```
@Entity
@Table(name = "alumnos")
public class Alumno {
            @Id
            @GeneratedValue(strategy=GenerationType.AUTO)
            private int id;

            @Column(name = "first_name", unique="true")
            @NotEmpty
            private String nombre;
```

@Temporal Sirve para afinar la forma en que se van a representar en la base de datos. Se utiliza con campos del tipo JAVA:

- Date
- Time
- Timestamp
- Calendar

Utilizaremos los siguientes parámetros:

- TemporalType.DATE
- TemporalType.TIME
- TemporalType.TIMESTAMP

@DateTimeFormat Se usa junto con @Temporal, pero no pertenece a Hibernate. Es parte de Spring Framework. No tiene nada que ver con ninguna validación, simplemente especifica el formato para las fechas.

```
@Entity
@Table(name = "alumnos")
public class Alumno {
        @Id
        @GeneratedValue(strategy=GenerationType.AUTO)
        private int id;

        @Column(name = "first_name", unique="true")
        @NotEmpty
        private String nombre;

        @Column(name = "birth_date")
        @Temporal(TemporalType.DATE)
        @DateTimeFormat(pattern = "yyyy/MM/dd")
        private Date fechaDeNacimiento;
```

@Digits Lo usamos para especificar y validar la cantidad de digitos de un número. Posee dos atributos principales:

- Integer: Define el número de digitos en la parte entera que se pueden tener
- Fraction: Define el número de digitos en la parte decimal

```
@Digits(integer = 2, fraction = 0)
private int edad;
```

@Min / @Max Definen los valores máximos y mínimos para un atributo dado.

```
@Digits(integer = 2, fraction = 0)
@Min(value = 18, message = "Hay que ser mayor de edad")
@Max(value = 67, message = "No se puede ser jubilado")
private int edad;
```

@Length Define la longitud máxima y mínima para una String.

```
@Column(name = "first_name", unique="true")
@NotEmpty
@Length(min = 3, max = 100, message="El nombre debe tener entre 3 y 100 caracteres")
private String nombre;
```

03 Relaciones

GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, COMERCIO Y TURISMO
EOI Escuela de organización industrial

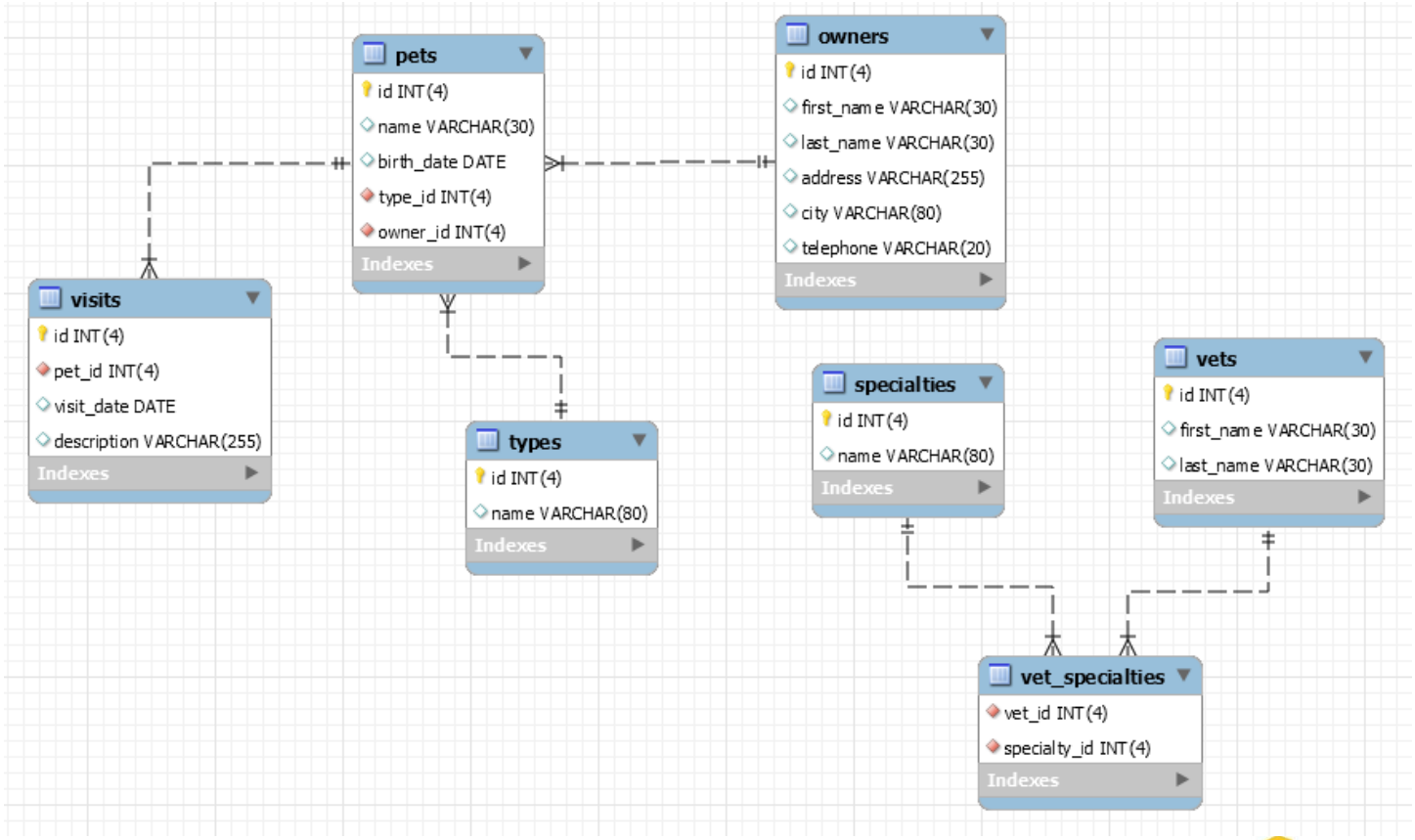
Unión Europea
Fondo Social Europeo
Iniciativa de Empleo Juvenil
El FSE invierte en tu futuro

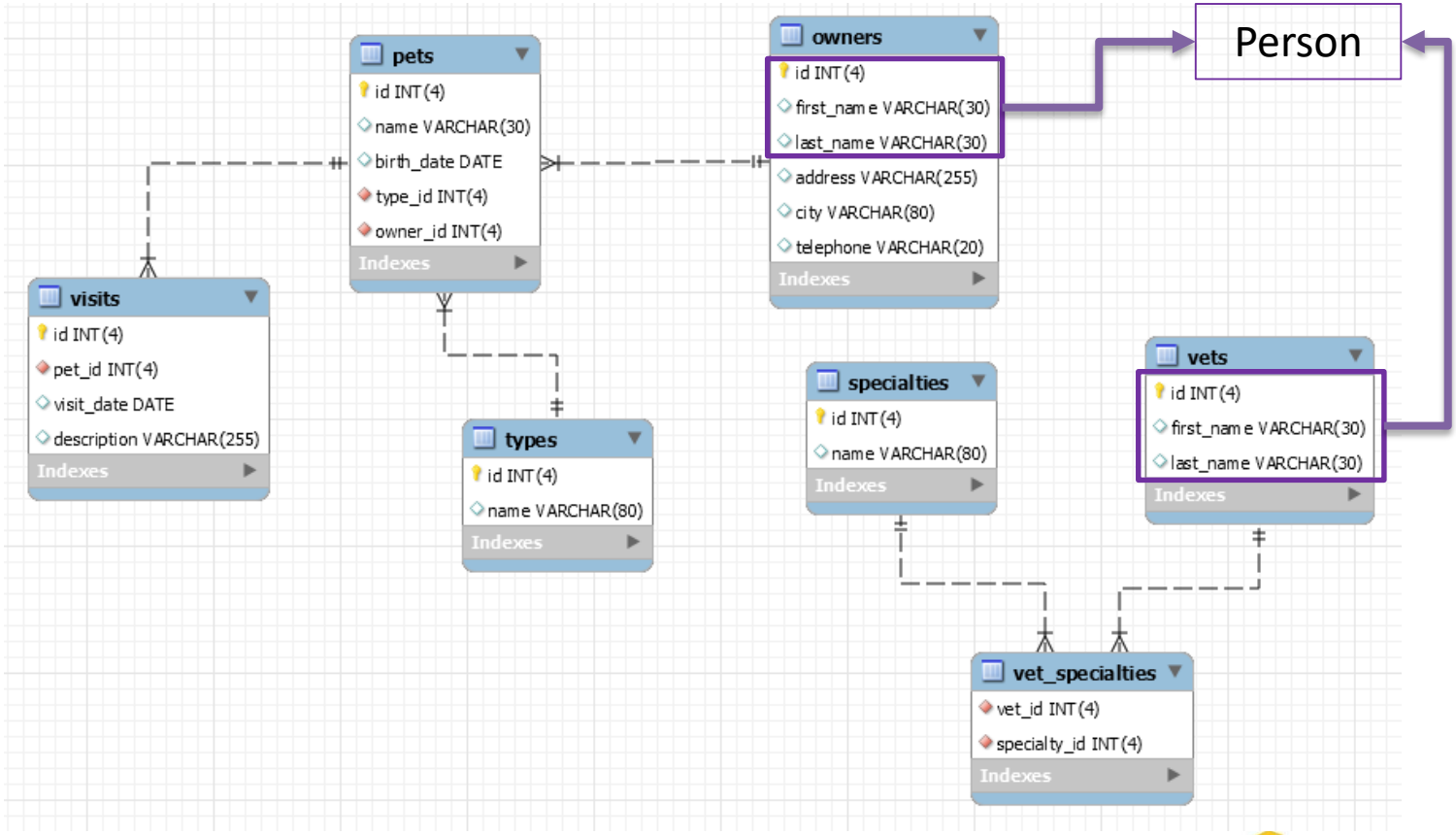
SISTEMA NACIONAL DE GARANTÍA JUVENIL

@MappedSuperClass Nos permite definir una clase abstracta de la que heredan otras entidades como “parte” del mapeado en cada tabla de las entidades. Es decir, serán los atributos comunes que tendrán varias clases sin necesidad de repetirlos.

Hibernate

Anotaciones



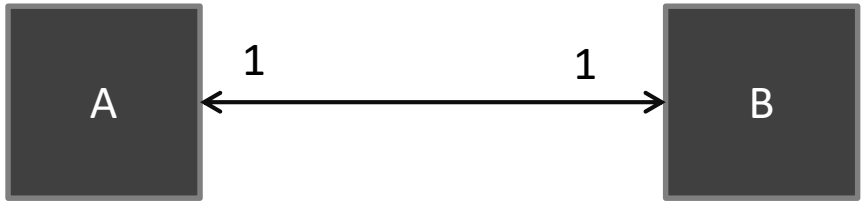

pets
id INT(4)
name VARCHAR(30)
birth_date DATE
type_id INT(4)
owner_id INT(4)
Indexes
owners
id INT(4)
first_name VARCHAR(30)
last_name VARCHAR(30)
address VARCHAR(255)
city VARCHAR(80)
telephone VARCHAR(20)
Indexes
Person
visits
id INT(4)
pet_id INT(4)
visit_date DATE
description VARCHAR(255)
Indexes
types
id INT(4)
name VARCHAR(80)
Indexes
specialties
id INT(4)
name VARCHAR(80)
Indexes
vets
id INT(4)
first_name VARCHAR(30)
last_name VARCHAR(30)
Indexes
vet_specialties
vet_id INT(4)
specialty_id INT(4)
Indexes

```java
@MappedSuperclass
public class Persona {
        @Id
        @GeneratedValue(strategy=GenerationType.AUTO)
        private int id;

        @Column(name = "first_name", unique="true")
        @NotEmpty
        @Length(min = 3, max = 100, message="El nombre debe tener entre 3 y 100 caracteres")
        private String nombre;
}

@Entity
@Table(name = "alumnos")
public class Alumno extends Persona {
        @Min(value=1)
        @Max(value=5)
        private int curso;
}
```

Relación 1 a 1




```
public class A {
        @OneToOne(fetch = FetchType.LAZY, mappedBy = "a", cascade = CascadeType.ALL)
        private B b;
}

public class B {
        @OneToOne(fetch = FetchType.LAZY)
        //Opcional @PrimaryKeyJoinColumn
         private A a;
}
```

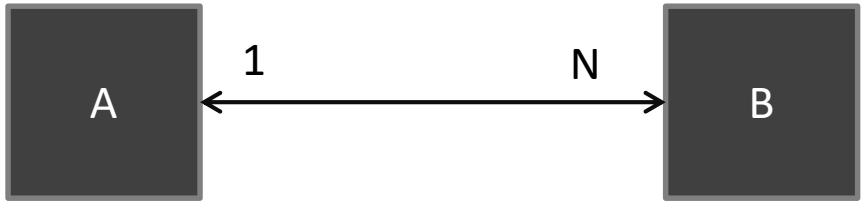
Anotaciones

Relación 1 a N


```
public class A {
        @OneToMany(cascade = CascadeType.ALL, mappedBy = "a")
        private List<B> collectionB;
}

public class B {
        @ManyToOne
        @JoinColumn(name = "ID_A", referencedColumnName = "ID")
        private A a;
}
```

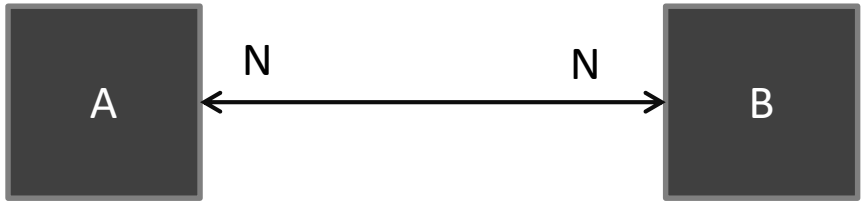
Relación 1 a N



@JoinColumn Indica que la entidad (N) tendrá la clave ajena de la entidad (1)

Cascade = CascadeType.ALL = Indica que se hacen las operaciones sobre los datos en cascada. Si borramos un objeto de dicha tabla, borraremos en cascada sus relacionados. Borrar una B, indicaría en este caso, borrar la A.

Relación N a N




```
public class A {
		@ManyToMany(fetch = FetchType.LAZY)
		@JoinTable(name="A_B", joinColumns = @JoinColumn(name="key_de_a"), inverseJoinColums = @JoinColumn(name="key_de_b")
		private Set<B> collectionB;
}

public class B {
		@ManyToMany(fetch = FetchType.EAGER, mappedBy="collectionB")
		private Set<A> collectionA;
}
```

EAGER VS LAZY

Cuando tenemos una relación y entre sus "miembros" se realiza Fetch de tipo **EAGER** (A contiene a B o una colección de B), cuando pedimos un dato de tipo A, además de A, nos vendrá también todo lo relativo a B.

Cuando tenemos una relación y entre sus "miembros" se realiza Fetch de tipo **LAZY** (A contiene a B o una colección de B), cuando pedimos un dato de tipo A, sólo nos llega la información relativa a A y hay que pedir expresamente la información relativa a B.

Tipo de relación	JPA 2.x	Hibernate
@OneToMany	LAZY	LAZY
@ManyToOne	EAGER	LAZY
@ManyToMany	LAZY	LAZY
@OneToOne	EAGER	LAZY

04 Implementación

GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, COMERCIO Y TURISMO
EOI Escuela de organización industrial

Unión Europea
Fondo Social Europeo
Iniciativa de Empleo Juvenil
El FSE invierte en tu futuro

SISTEMA NACIONAL DE GARANTÍA JUVENIL

persistence.xml

```
1
2<?xml version="1.0" encoding="UTF-8"?>
3<persistence xmlns="http://java.sun.com/xml/ns/persistence"
4    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
6             http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
7    version="2.0">
8
9    <persistence-unit name="EJERCICIOPU">
10       <provider>org.hibernate.ejb.HibernatePersistence</provider>
11       <properties>
12         <property name="hibernate.connection.url" value="jdbc:mysql://localhost:3306/ej_eoi?serverTimezone=UTC" />
13         <property name="hibernate.connection.driver_class" value="com.mysql.jdbc.Driver" />
14         <property name="hibernate.connection.username" value="root" />
15         <property name="hibernate.connection.password" value="1234" />
16         <property name="hibernate.archive.autodetection" value="class" />
17         <property name="hibernate.show_sql" value="true" />
18      </properties>
19   </persistence-unit>
20</persistence>
21
```

Hibernate

Implementacion


```java
@Entity
public class Employee {

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

  @OneToMany(fetch = FetchType.LAZY, mappedBy = "employee")
  private List<Email> emails;

  // ...
}
```

```java
@Entity
public class Email {

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

  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = “employee_Id",
     referencedColumnName = "Id“)
  private Employee employee;

  // ...

}
```

Autonuméricos MYSQL

Entidad:

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

Tabla:

```
ID INT NOT NULL AUTO_INCREMENT
```

EntityManager

```
private static EntityManager em;

public static void main(String[] args) {

    EntityManagerFactory emf = Persistence.
            createEntityManagerFactory("EJERCICIOPU");
    em = emf.createEntityManager();
```

Persist

```
em.getTransaction().begin();
Producto p = new Producto("Tomate", 2);
em.persist(p);
```

Al tratarse de DML podremos hacer commit o rollback.

```
em.getTransaction().rollback();
em.getTransaction().commit();
```

Find

```
Cliente cli = null;
cli = em.find(Cliente.class, dni);
```

Este método solo podremos utilizarlo con la clave primaria o identificador del objeto.

Merge

```
Producto p = findProducto(1);
p.setPrecio(2);
em.getTransaction().begin();
em.merge(p);
```

Al tratarse de DML podremos hacer commit o rollback.

```
em.getTransaction().rollback();
em.getTransaction().commit();
```

Remove

```
Producto p = findProducto(3);
em.getTransaction().begin();
em.remove(p);
```

Al tratarse de DML podremos hacer commit o rollback.

```
em.getTransaction().rollback();
em.getTransaction().commit();
```

HQL – Hibernate Query Language

```
String hql = "FROM Employee";
Query query = em.createQuery(hql);
List<Employee> result = null;
result = query.getResultList();
return result;
```

HQL – Hibernate Query Language

```
String hql = "FROM Employee";
Query query = em.createQuery(hql);
List<Employee> result = null;
result = query.getResultList();
return result;

String hql = "FROM Email e WHERE e.employee.id = :parameter";
Query query = em.createQuery(hql);
query.setParameter("parameter",5);
List<Email> result = null;
result = query.getResultList();
return result;
```

Native Queries

```
Query q = em.createNativeQuery("SELECT a.firstname, a.lastname FROM Author a");
List<Object[]> authors = q.getResultList();

for (Object[] a : authors) {
    System.out.println("Author "
            + a[0]
            + " "
            + a[1]);
}
```

05 Ejercicios

GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, COMERCIO Y TURISMO
EOI Escuela de organización industrial

Unión Europea
Fondo Social Europeo
Iniciativa de Empleo Juvenil
El FSE invierte en tu futuro

SISTEMA NACIONAL DE GARANTÍA JUVENIL

Configuración

Nuevo Maven Project >> Simple Project >>

- **Group ID** es.eoi.jpa
- **Artifact ID** EjercicioJPA
- **Name** EjercicioJPA

Nos aseguraremos de tener Java 8 en el proyecto.

pom.xml

```
<dependency>
        <groupId>org.eclipse.persistence</groupId>
        <artifactId>javax.persistence</artifactId>
        <version>2.0.0</version>
</dependency>
<dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>4.2.8.Final</version>
</dependency>
<dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.11</version>
</dependency>
```

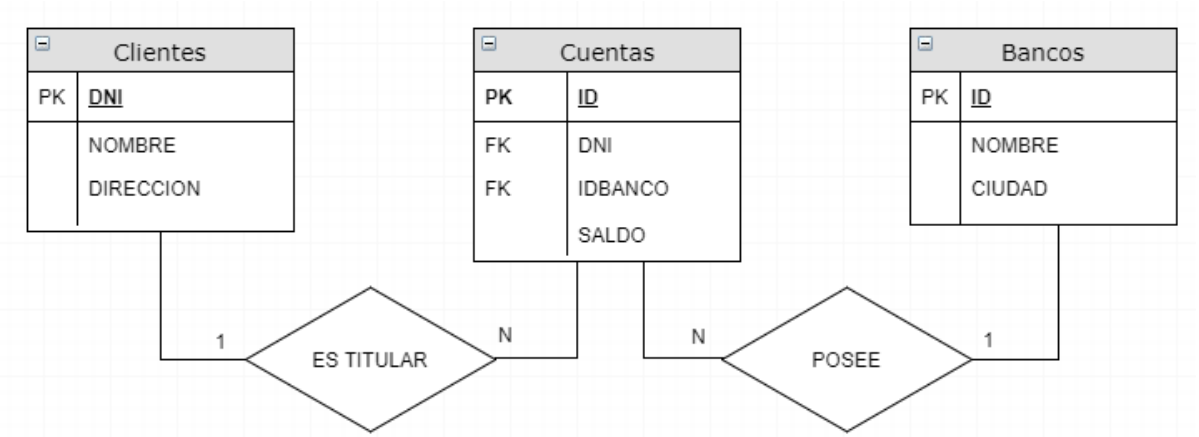
persistence.xml

En la carpeta src/main/resources crearemos un fichero xml llamado persistence.xml

```
1
2<?xml version="1.0" encoding="UTF-8"?>
3<persistence xmlns="http://java.sun.com/xml/ns/persistence"
4    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
6             http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
7    version="2.0">
8
9    <persistence-unit name="ClientesSucursalesPU">
10       <provider>org.hibernate.ejb.HibernatePersistence</provider>
11       <properties>
12         <property name="hibernate.connection.url" value="jdbc:mysql://localhost:3306/ej_eoi?serverTimezone=UTC" />
13         <property name="hibernate.connection.driver_class" value="com.mysql.jdbc.Driver" />
14         <property name="hibernate.connection.username" value="root" />
15         <property name="hibernate.connection.password" value="1234" />
16         <property name="hibernate.archive.autodetection" value="class" />
17         <property name="hibernate.show_sql" value="true" />
18       </properties>
19   </persistence-unit>
20</persistence>
21
```

Ejercicio

En la carpeta src/main/java crearemos un paquete llamado es.eoi.entity. Eliminaremos de nuestra base de datos el ejercicio **CLIENTES-SUCURSALES** y crearemos una versión mejorada:



Ejercicio

Una vez que hemos creado el modelo de datos crearemos las entidades:

- Cliente
- Banco
- Cuenta

Deben estar bien mapeadas y bien relacionadas.

*Los campos ID serán autonuméricos.

Ejercicio

Crearemos un paquete es.eoi.Service, crearemos una clase Service para cada una de nuestras entidades que nos permita hacer CRUD de nuestras Tablas.

Para probar que toda nuestra lógica funciona nos crearemos en es.eoi.app una clase llamada MundoBancario.java y en ella **construiremos un menú con opciones para hacer CRUD sobre las tablas al igual que hicimos en el ejercicio de JDBC**, en la que iremos seleccionado cada opción que nos interese hasta que pulsemos 0, al pulsar 0 finalizará el programa.

Ejercicio

Para probar el ejercicio, como mínimo debemos:

- Crear 5 Clientes
- Crear 5 Bancos
- Recuperar un Cliente por su Clave Principal
- Recuperar un Banco por su Clave Principal
- Modificaremos un Cliente
- Modificaremos un Banco

Ejercicio

- Eliminaremos un Cliente
- Eliminaremos un Banco
- Obtendremos la lista de todos los clientes.
- Obtendremos la lista de todos los bancos.
- Crearemos 5 cuentas (Habrán clientes con varias cuentas)
- Modificaremos cuentas
- Eliminaremos cuentas
- Obtendremos las cuentas por banco
- Obtendremos las cuentas por cliente
- Obtendremos un listado de todos los clientes y sus cuentas (tengan cuenta o no)