# ORM

Desarrollo Web en Entorno Servidor

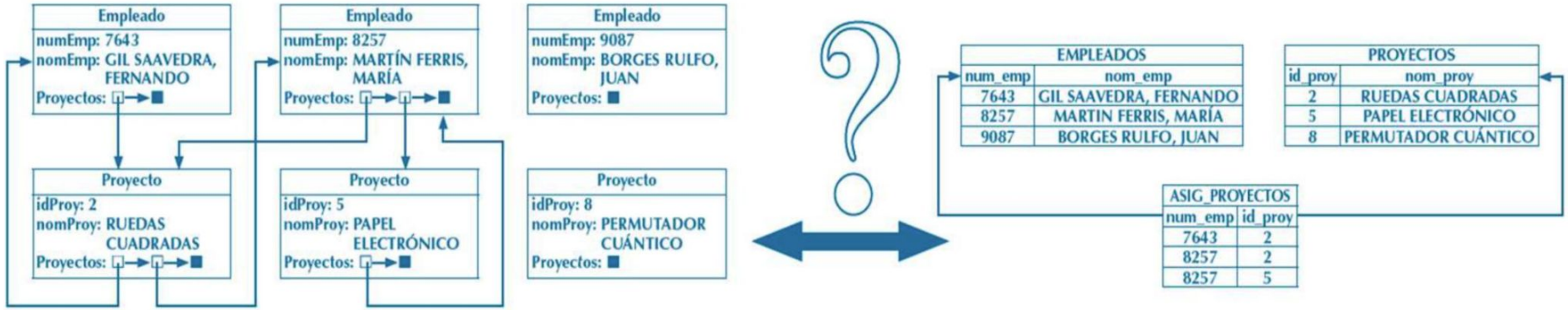
Ana Viciano Fabregat
ana.viciano@iesalvarofalomir.org

IES Álvaro Falomir
Curso 2023-2024

# Desfase objeto-relacional

Existen diferencias entre el paradigma de programación orientada a objetos y las bases de datos relacionales.

- El modelo relacional de la base de datos trata de relaciones y conjuntos.
- La POO trata los datos como objetos y asociaciones entre ellos.



# Desfase objeto-relacional

Esto conlleva un conjunto de problemas a resolver denominado desfase objeto-relacional, ante los que se han planteado varias soluciones:

- BD objeto-relacionales: Son BBDD relacionales con capacidad para gestionar objetos. Destacan Oracle y PostreSQL.
- Mapeo objeto-relacional (ORM): Es una solución más flexible con la ventaja de proporcionar soporte para múltiples BBDD. Existen múltiples herramientas, bibliotecas o frameworks para ORM, entre las que destaca JPA e Hibernate.

# Acceso a bases de datos relacionales

En el tema anterior usamos la API JDBC (Java DataBase Connectivity) para ejecutar operaciones sobre bases de datos desde Java, utilizando el dialecto SQL del modelo de BBDD utilizado.

Tuvimos que descargar los drivers que cada BBDD proporcionaba para implementar los interfaces del API y acceder a la BBDD.

JDBC ofrece interfaces para:

- Conectar a una BBDD
- Ejecutar una consulta
- Procesar los resultados

# Acceso a bases de datos relacionales

Como hemos visto, vincular los datos recuperados de la base de datos con una clase o insertar en un objeto en la base de datos es una tarea pesada y manual.

- Se debe de crear la consulta correspondiente en SQL
- Se deben vincular las columnas y datos para que funcione de forma correcta.

# Características ORM

Son muy utilizados y han reemplazado a las bases de datos orientadas a objetos (BDOO) al conseguir orientación a objetos en bases de datos relacionales con ellos.

Existen muchos ORM en diferentes lenguajes, por ejemplo:.

- Java → JPA
- Android → Room
- NodeJS → Sequlize
- .NET → Entity Framework
- PHP → Doctrine

# Características ORM

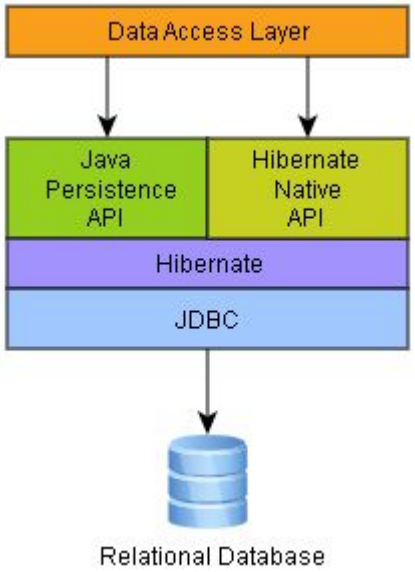
| Ventajas | Inconvenientes |
| --- | --- |
| Orientación a Objetos | Complejos de aprender |
| Reducen el tiempo de desarrollo | Múltiples soluciones en el mercado |
| Permite reutilizar código de forma simple | En ocasiones, documentación escasa |
| Evitan mezclar lenguajes en código (Java + SQL) | Complejidad para realizar consultas avanzadas |
| Sirven para muchos motores de BBDD | Aplicaciones más pesadas para el procesador por el tratamiento de los datos |
| Lenguajes propios para consultas | |

# JPA

JPA es la especificación para estandarizar ORMs en Java

Actualmente funciona mediante el uso de anotaciones en las clases, lo que simplifica la tarea.

- API para operaciones CRUD
- Lenguaje y API para consultas (JPQL)
- Elementos de optimización



# JPA y Maven

El uso de JPA implica incluir en nuestro proyecto 15 jars con más de cientos de paquetes y miles de clases.

Maven nos permite gestionar dependencias mediante un fichero descriptor xml (pom.xml).

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
    <version>3.1.4.</version>
</dependency>
```

# Instalar JPA

1. Crear un proyecto Maven (más adelante crearemos uno Spring)
2. Instalar la librería del conector de la BBDD (Postgres)
3. Instalar la librería de JPA, los datos están arriba.

Para esto necesitas:

- MVN Repository , donde tienes indexados las librerías (artefactos) para añadir al fichero pom.xml.
- Pom.xml (Project Object Model), es un archivo XML que tiene información sobre dependencias, configuraciones y otra información importante del proyecto Maven.

# Mapeo de entidades

Una clase de persistencia mapea una tabla de la base de datos.

Estas clases necesitan seguir una serie de reglas:

- Han de tener un constructor sin argumentos.
- Un atributo ha de ser utilizado como identificador de la clase, comportándose como clave primaria en la tabla de la BBDD.
- Declarar getters y setters para todos los atributos con los nombres de los métodos por defecto (getAtributo y setAtributo).
- Es preferible que la clase implemente la interfaz Serializable.

# Mapeo de entidades

JPA utiliza anotaciones para relacionar tablas con objetos Java.

Las anotaciones se realizan en las clases de persistencia.

- @Entity → Indica que la clase es una tabla en la BBDD
- @Table (name = "nombre_tabla") → Indica el nombre de la tabla
- @Id → Indica que un atributo es la clave primaria de la tabla
- @GeneratedValue(strategy = GenerationType.IDENTITY) Indica que es un valor con incremento automático desde la BBDD
- @Column (name = "nombre_columna") → Indica el nombre de una la columna en la tabla donde debe ser mapeado el atributo

# Entidad

```
@Entity
public class Producto {

        @Id @GeneratedValue
        private Long Id;

        private String nombre;
        private String descripcion;
        private float pvp;

        //… resto de métodos y atributos
}
```

# Control de nombres

@Entity se mapea con una tabla que se llame igual que la clase
Con @Table adicional, podemos cambiar el nombre
Los atributos también se mapean con su nombre
Con un @Column, podemos cambiar el nombre

```
@Entity
@Table(name="PROD")
public class Producto {

        @Column(name="prod_nombre")
    private String nombre;
        //… resto de métodos y atributos
}
```

# Anotación @Column

@Column nos permite definir otras propiedades:

- Nullable: si permite almacenar nulos
- Name: nombre de la columna en la BD
- Insertable, updatable: define si la entidad puede ser o no insertable o actualizable
- Lenght: tamaño que tendrá el campo en la BD.

# Mapeo de valores. Asociaciones

- JPA nos permite asociar dos entidades
- Debemos conocer la multiplicidad de dicha asociación
  - @ManyToOne: muchos a uno
  - @OneToMany: uno a muchos
  - @ManyToMany: muchos a muchos
  - @OneToOne: uno a uno

```
@Entity
public class Producto {

        @Id @GeneratedValue
        private Long Id;

        //… resto de atributos

        @ManyToOne
        private Categoria categoria;

}
```

# CRUD

Con JPA configurado e iniciado, y las entidades mapeadas, ya podemos realizar las operaciones fundamentales de una base de datos: CRUD.

- Create: Guardar un nuevo objeto en la base de datos.
- Read: Leer los datos de un objeto de la base de datos.
- Update: Actualizar los datos de un objeto de la base de datos.
- Delete: Borrar los datos de un objeto de la base de datos.

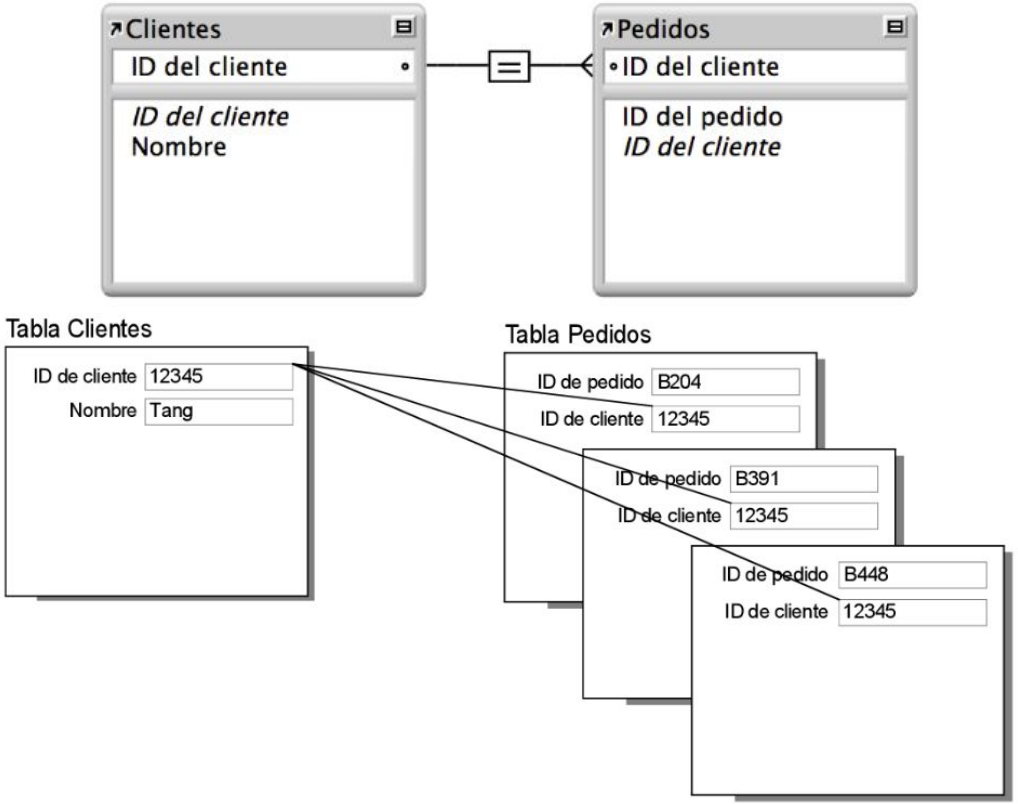
# Relaciones entre tablas

Hasta ahora hemos visto cómo trabajar con una sola tabla, pero ¿qué pasa con relaciones de varias tablas?

JPA soporta relaciones:

- 1–1 ó OneToOne
- 1–N ó OneToMany/ManyToOne
- N–M ó ManyToMany

# Asociación OneToMany



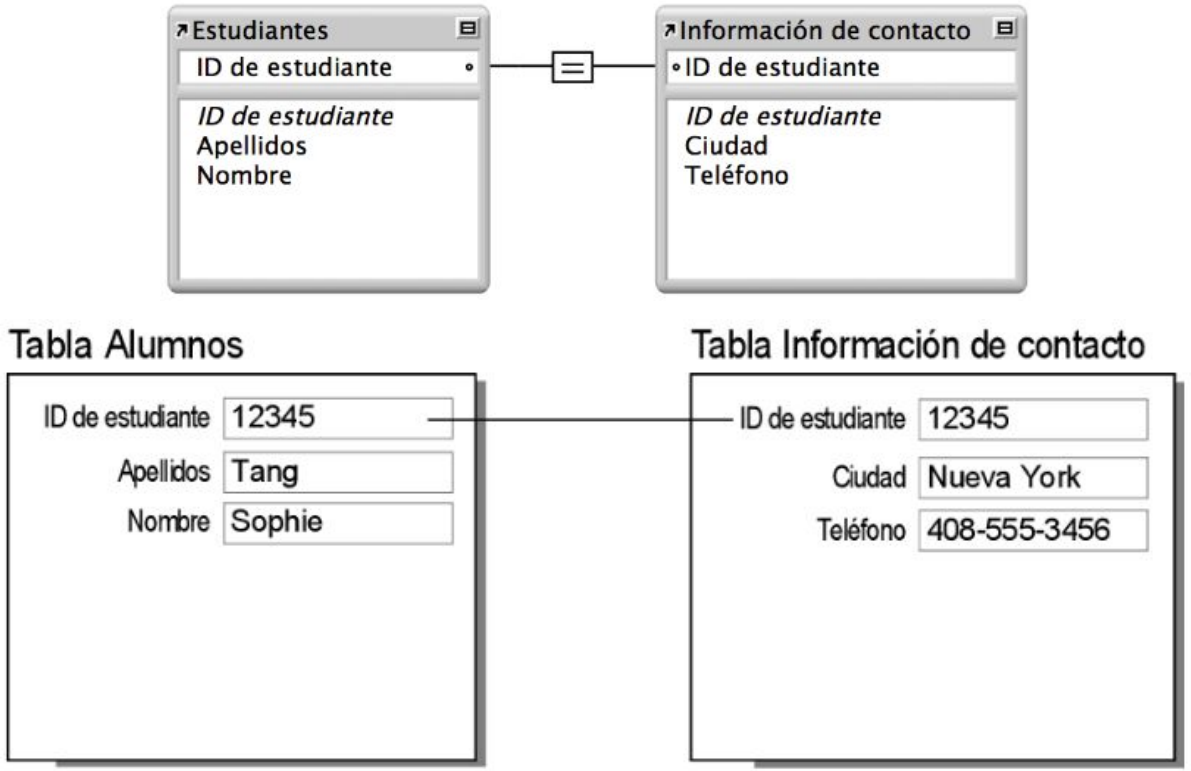
# Asociación OneToMany

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

  @Id
  private int id;
  private String nombre;

  @OneToMany(mappedBy = "cliente")
  private Set<Pedidos> pedidos;
  . . .

}
```

```
@Entity
@Table(name = "pedidos")
public class Pedido {

  @Id
  private int id;

  @ManyToOne(cascade =
{CascadeType.PERSIST, CascadeType.MERGE})

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

}
```

Sobre Cascade

# Asociación OneToOne

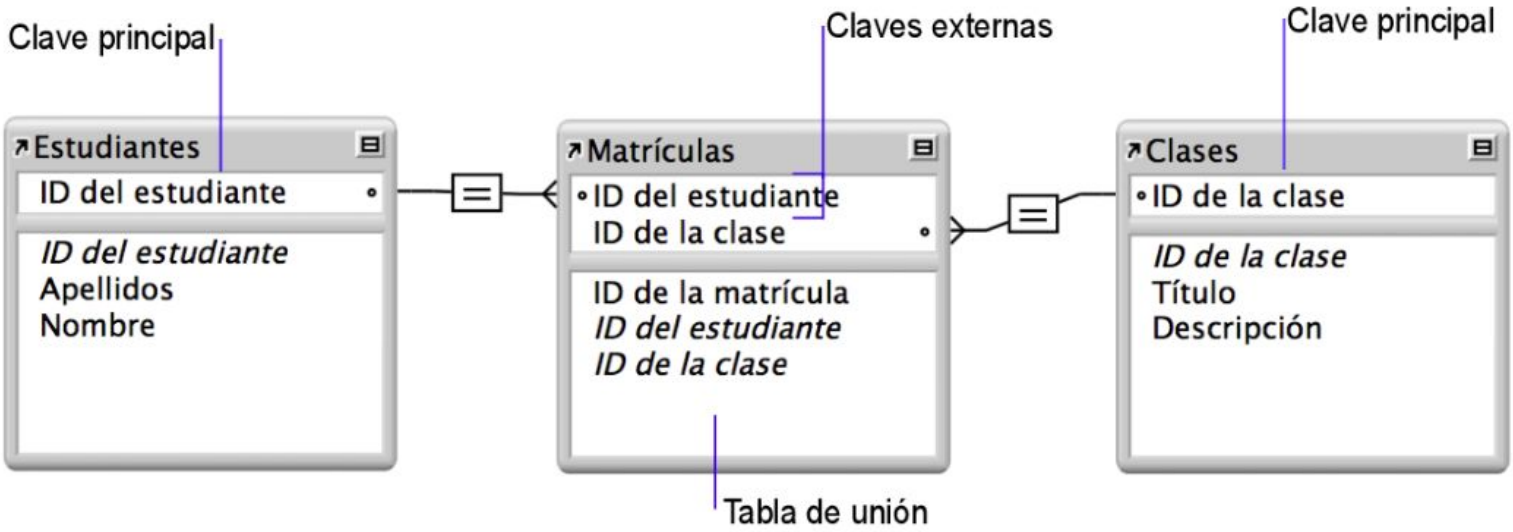


# Asociación OneToOne

```
@Entity
@Table(name = "estudiantes")
public class Estudiante {

  @Id
  private int id;
  private String nombre;
  private String apellidos;

  @OneToOne(cascade = CascadeType.ALL)

  @JoinColumn(name="estudiante_id")
  private Contacto contacto;
  . . .

}
```

```
@Entity
@Table(name = "contactos")
public class Contacto {

  @Id
  private int id;
  private String ciudad;
  private String telefono;


@OneToOne(mappedBy="estudiante")

private Estudiante estudiante;
  . . .

}
```

# Asociación ManyToMany



# Asociación ManyToMany

En una relación direccional @ManyToMany hay un lado dueño y un lado mappedBy. Es recomendable agregar métodos para añadir y eliminar entidades hijo. Bidireccional ManyToMany. Hibernate 6.1.6.

```
@Entity
@Table(name = "estudiantes")
public class Estudiante {

  @Id
  private int id;
  ...

  @ManyToMany(cascade =
{CascadeType.PERSIST, CascadeType.MERGE})

@JoinTable (joinColumns = @JoinColumn(name =
"student_id"),
  private Set<Clase> clases;

  public anyadirClase(Clase clase) { ... }
  public eliminarClase(Clase clase) { ... }

}
```

```
@Entity
@Table(name = "clases")
public class Clase {

  @Id
  private int id;
  ...

  @ManyToMany(cascade =
{CascadeType.PERSIST, CascadeType.MERGE},
mappedBy = "clases")
  private Set<Estudiante> estudiantes;


}
```

# Operaciones en Cascada

- CascadeType.ALL: se aplican todos los tipos de cascada.
- CascadeType.PERSIST: las operaciones de guardado en la base de datos de las entidades padre se propagarán a las entidades relacionadas.
- CascadeType.MERGE: las entidades relacionadas se unirán al contexto de persistencia cuando la entidad propietaria se una.
- CascadeType.REMOVE: las entidades relacionadas se eliminan de la base de datos cuando la entidad propietaria se elimine.
- CascadeType.REFRESH: las entidades relacionadas actualizan sus datos desde la base de datos cuando la entidad propietaria se actualiza.
- CascadeType.DETACH: se separan del contexto de persistencia todas las entidades relacionadas cuando ocurre una operación de separación manual.