# 5.3. Spring Data JPA

Acceso a Datos

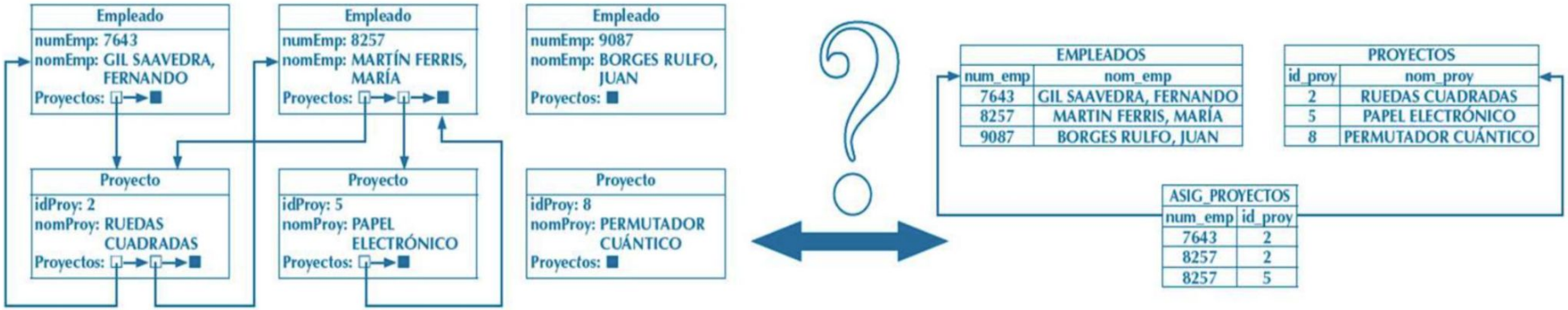
Alejandro Roig Aguilar
alejandro.roig@iesalvarofalomir.org

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 Hibernate.

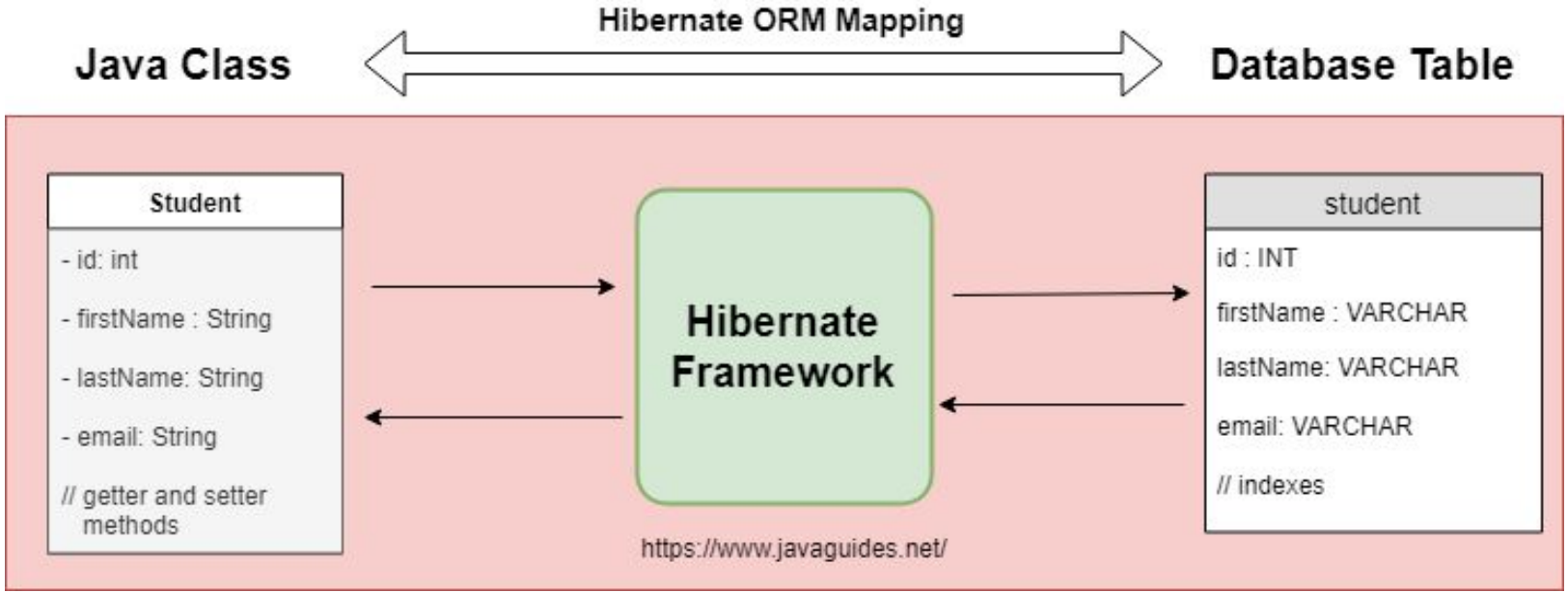
# JPA vs ORM

JPA (Jakarta Persistence API) es la especificación para persistir, leer y gestionar data desde los objetos Java a la base de datos.

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

Hibernate es un ORM que ofrece una implementación de JPA.

# JPA vs ORM


Hibernate ORM Mapping
Java Class
Database Table
Student
- id: int
- firstName : String
- lastName: String
- email: String
// getter and setter methods
Hibernate Framework
student
id : INT
firstName : VARCHAR
lastName: VARCHAR
email: VARCHAR
// indexes
https://www.javaguides.net/

# Spring Data JPA

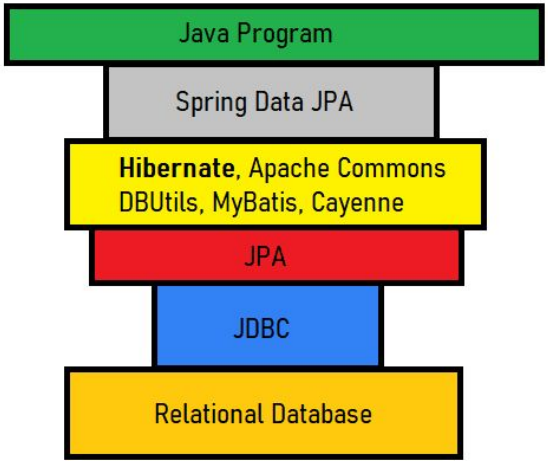
Entre las principales ventajas de Spring Data JPA se encuentran:

- Reduce gran parte del código repetitivo que normalmente se necesita para interactuar con una base de datos relacional, lo que hace que el código sea más limpio y fácil de mantener.
- Proporciona una capa de abstracción sobre la base de datos, lo que permite a los desarrolladores escribir código que sea independiente de la base de datos subyacente.
- Soporte para repositorios específicos, paginación y ordenamiento de forma fácil y sencilla, lo que facilita la implementación de características avanzadas en una aplicación.
- Integración con otras tecnologías de Spring, como Spring MVC y Spring Boot, facilitando la construcción de aplicaciones web.

# Spring Data JPA

Spring Data JPA es parte de Spring Framework. No es una implementación de JPA como Hibernate, sino una abstracción para reducir la complejidad de la integración con bases de datos relacionales desde aplicaciones Java.

Spring Data JPA puede utilizar Hibernate, Eclipse Link u otra implementación.



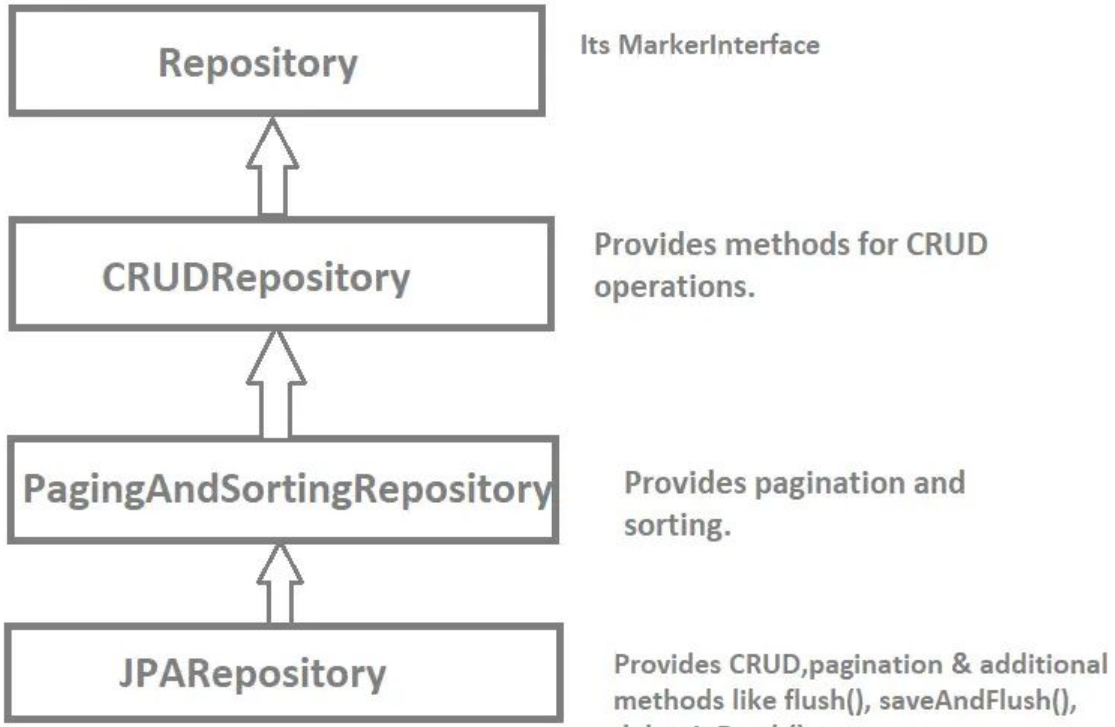
# Clases de repositorio

Un repositorio (@Repository) define una colección de métodos para acceder y manipular datos en una BD, abstrayendo la capa de acceso a datos y facilitando la implementación de operaciones CRUD.

Los tipos de repositorios más comunes en Spring Data JPA son:

- CrudRepository proporciona funciones CRUD
- PagingAndSortingRepository proporciona métodos para paginar y ordenar registros
- JpaRepository proporciona métodos relacionados con JPA, como vaciar el contexto de persistencia y eliminar registros en un lote.

# Clases de repositorio



# Consultas en el repositorio

Para crear consultas para los repositorios, Spring Data JPA ofrece opciones para personalizar consultas generadas automáticamente y crear consultas personalizadas.

- Métodos de la interfaz. Podemos utilizar los métodos que las interfaces anteriores ofrecen.
- Consultas personalizadas. Podemos utilizar la anotación @Query para definir consultas personalizadas utilizando JPQL o SQL. Podemos definir parámetros en la consulta con @Param.
- Consultas generadas automáticamente. Spring Data JPA genera automáticamente consultas utilizando el nombre del método y los parámetros definidos en el método.

# Mapeo de entidades

Una entidad en JPA es una clase persistente que 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.

# Mapeo de entidades

En Spring Data JPA se utilizan anotaciones JPA para mapear tablas con entidades. Las anotaciones se realizan en las entidades.

- @Entity → Indica que la clase se mapea a una tabla en la BBDD
- @Table → Indica el nombre de la tabla. @Table(name = “tabla”)
- @Id → Especifica el identificador de la entidad.
- @Column → Define los atributos de la entidad. Definimos el nombre de la columna con @Column (name = “columna”)

# Identificadores y @GeneratedValue

Utilizamos la anotación @GeneratedValue para especificar cómo se debe generar el identificador de una entidad:

- AUTO: Utiliza una estrategia predeterminada.
- IDENTITY: Utiliza una columna de identidad en la base de datos.
- SEQUENCE: Utilizando una secuencia de base de datos.
- TABLE: Utiliza una tabla de base de datos.
- NONE: Debe ser especificado manualmente.
- UUID: Utiliza un UUID.

Por ejemplo: @GeneratedValue(strategy = GenerationType.IDENTITY)

# Restricciones de integridad y validadores

Podemos definir restricciones de integridad para columnas siguiendo los validadores existentes en JPA.

- @Null, @NotNull, @NotEmpty y @NotBlank
- @AssertTrue y @AssertFalse
- @Min y @Max
- @Negative, @NegativeOrZero, @Positive y @PositiveOrZero
- @Size y @Digits
- @Past, @PastOrPresent, @Future, @FutureOrPresent
- @Pattern, @Email

# Marcas temporales

En JPA, podemos usar marcas temporales para registrar cuándo se creó o modificó una entidad. Su utilidad reside en poder realizar un seguimiento de los cambios en la base de datos y realizar auditorías.

- @EntityListeners se utiliza para especificar listeners de eventos de ciclo de vida de una entidad con @CreatedBy, @LastModifiedBy, @CreatedDate y @LastModifiedDate.
- @EnableJpaAuditing se utiliza para habilitar la auditoría en la aplicación Spring Boot de las entidades que estén anotadas con las anotaciones anteriores.

En muchos casos, se utilizarán juntos para realizar auditorías automáticas de entidades en una aplicación Spring Boot.

# Relaciones entre entidades

En Spring Data JPA, una relación es una asociación entre dos o más entidades. Hay varios tipos de relaciones que se pueden establecer entre entidades, entre los cuales se encuentran:

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

# Asociación OneToOne

```java
@Entity
public class Pais {

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

    @OneToOne
    @JoinColumn(name = "bandera_id")
    private Bandera bandera;
}

// JoinColumn se indica en la entidad
// owner y permite definir la columna
// con la clave ajena
```

```java
@Entity
public class Bandera {

    @Id
    @GeneratedValue(strategy =
        GenerationType.IDENTITY)
    private Long id;
    private List<String> colores;
    private LocalDate fecha;

    // bidireccional (Opcional)
    @OneToOne(mappedBy = "bandera")
    private Pais pais;
}

// mappedBy indica el atributo de la
// entidad owner en la relación
```

# Asociación OneToMany/ManyToOne

```
@Entity
public class Equipo {

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

   @OneToMany(mappedBy = "equipo")
   private List<Jugador> jugadores;
   . . .

}

// One Equipo To Many Jugador
```

```
@Entity
public class Jugador {

   @Id
    @GeneratedValue(strategy =
       GenerationType.IDENTITY)
   private Long id;
   private String nombre;
   private int numero;

   @ManyToOne
   @JoinColumn(name = "equipo_id")
   private Equipo equipo;
   . . .

}

// Many Jugador To One Equipo
```

# Asociación ManyToMany

```
@Entity
public class Estudiante {

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

   @ManyToMany(mappedBy =
“estudiantes”)
   private List<Curso> cursos;
   . . .

}
```

```
@Entity
public class Curso {

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

   @ManyToMany
   @JoinTable(
      name = "curso_estudiante",
      joinColumns = @JoinColumn(name = “curso_id”),
      inverseJoinColumns = @JoinColumn(name = "estudiante_id")
   )
   private List<Estudiante> estudiantes;
   . . .

}
```

# Composición

Podemos usar las anotaciones @Embedded y @Embeddable para definir una relación de composición entre entidades.

```
@Embeddable
public class Direccion {

    private String calle;
    private String ciudad;
    . . .

}

// Una entidad @Embeddable se va a
// embeber en otra entidad y no necesita
// una tabla ni identificador propios
```

```
@Entity
public class Persona {

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

    @Embedded
    private Direccion direccion;
    . . .

}
```

# Opciones de cascada

En Spring Data JPA, la cascada es una opción que te permite propagar una acción desde una entidad a otra relacionada.

Por defecto, la cascada no se habilita, pero se puede configurar mediante la anotación @CascadeType:

- CascadeType.ALL: Propaga todas las acciones a las entidades relacionadas.
- CascadeType.PERSIST: Propaga la persistencia a las entidades relacionadas.
- CascadeType.MERGE:Propaga la fusión a las entidades relacionadas.
- CascadeType.REMOVE: Propaga la eliminación a las entidades relacionadas.
- CascadeType.REFRESH: Propaga el refresco a las entidades relacionadas.
- CascadeType.DETACH: Propaga la desconexión a las entidades relacionadas.

# Opciones de cascada

```
@Entity
public class Equipo {

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

    @OneToMany(mappedBy = "equipo",
        cascade = CascadeType.ALL)
    private List<Jugador> jugadores;
    ...
}
```

```
@Entity
public class Jugador {

    @Id
    @GeneratedValue(strategy =
        GenerationType.IDENTITY)
    private Long id;
    private String nombre;
    private int numero;

    @ManyToOne(cascade =
        CascadeType.ALL)
    @JoinColumn(name = "equipo_id")
    private Equipo equipo;
    ...

}
```

# OrphanRemoval

La opción OrphanRemoval permite eliminar automáticamente las entidades “huérfanas” que ya no están relacionados con la entidad padre debido a su eliminación.

```
@Entity
public class Pedido {
    @Id
    @GeneratedValue(strategy =
        GenerationType.IDENTITY)
    private Long id;

    @OneToMany(orphanRemoval = true,
        cascade = CascadeType.ALL,
        mappedBy = "pedido")
    private List<Producto> productos;
}
```

```
@Entity
public class Producto {
    @Id
    @GeneratedValue(strategy =
        GenerationType.IDENTITY)
    private Long id;
    private String nombre;

    @ManyToOne
    private Pedido pedido;
}

// Si se elimina un Pedido, todos sus
// Productos asociados se eliminarán también
```

# Recursión infinita y asociación bidireccional

Las relaciones bidireccionales pueden ser peligrosas debido a varios problemas potenciales:

- Recursión infinita: En una relación bidireccional entre dos entidades, si no se configura correctamente la serialización puedes terminar con una recursión infinita.
- Actualizaciones inesperadas de la base de datos: En una relación bidireccional, hay que tener cuidado con qué entidad es la "propietaria" de la relación, ya que es la que Hibernate utiliza para determinar el estado de la relación en la BD.
- Rendimiento: Las relaciones bidireccionales pueden afectar el rendimiento de la aplicación si no se manejan correctamente.

# Recursión infinita y asociación bidireccional

¿Cuándo usar relaciones bidireccionales? Si necesitamos navegar por la relación en ambas direcciones en el código, una relación bidireccional puede ser la mejor opción.

¿Cuándo evitarlas? Si solo necesitamos navegar por la relación en una dirección.

En Spring Data JPA, cuando se establece una relación bidireccional entre dos entidades, se produce un problema de recursión infinita al serializar las entidades a JSON.

Una solución sencilla es especificar las propiedades a ignorar durante la serialización con la anotación @JsonIgnoreProperties.

# Recursión infinita y asociación bidireccional

```
@Entity
public class Equipo {

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

    @OneToMany(mappedBy = “equipo”,
        cascade = CascadeType.ALL)
    private List<Jugador> jugadores;
    . . .

}
```

```
@Entity
public class Jugador {

    @Id
    @GeneratedValue(strategy =
        GenerationType.IDENTITY)
    private Long id;
    private String nombre;
    private int numero;

    @ManyToOne
    @JoinColumn(name = “equipo_id”)
    @JsonIgnoreProperties("jugadores")

    private Equipo equipo;
    . . .

}
```

# FetchType

En JPA, los atributos y relaciones de una entidad pueden ser recuperados desde la base de datos de dos maneras distintas:

- Modo Eager, de inmediato al momento de obtener la entidad.
- Modo Lazy, bajo demanda, al ejecutar un getter del atributo.

Por defecto, cuando la relación se anota con **@OneToOne** o **@ManyToOne**, se carga en modo **Eager**.

De igual forma, cuando la relación está anotada con **@OneToMany** o **@ManyToMany**, se carga en modo **Lazy**.

Podemos cambiar este comportamiento en la definición de la relación, por ejemplo: **@OneToOne(fetch = FetchType.LAZY)**

# FetchType

LAZY

- Menor uso de memoria.
- N+1 consultas.

EAGER

- Menos probable que nos encontremos con algún tipo de excepción provocada por el mappeo.
- Mayor tiempo de carga de los objetos en memoria.
- Demasiados objetos innecesarios que podrían afectar el rendimiento de la aplicación.