

**Módulo X**

# **SPRING DATA JPA II**



## **MASTER DE SPRING FRAMEWORK Y SPRING BOOT**

Impartido por Rafael Álvarez Martínez

18 Abril 2022

# CONTENIDOS

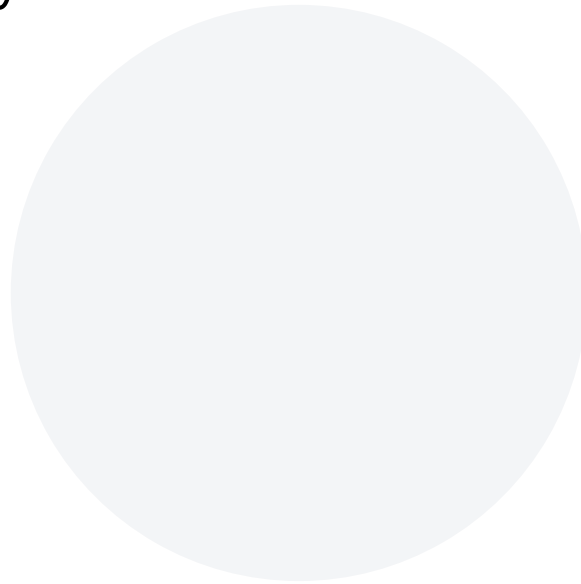
- Estableciendo relaciones entre las tablas.
- Relaciones de uno a uno - @onetoone.
- Relaciones de muchos a uno - @manytoone.
- Relaciones de muchos a muchos - @manytomany.
- CascadeType.

# ESTABLECIENDO RELACIONES ENTRE LAS TABLAS

- Cada clase tendrá una tabla en la BD.
- Es necesario tener relaciones entre estas tablas.
- Por ejemplo, los Clientes realizan Pedidos y los Pedidos contienen Productos.
- Estas relaciones deben ser representadas en la BD:
  - uno a uno (1:1) - @ONETOONE
  - uno a muchos (1:M) - @ONETOMANY
  - muchos a uno (M:1) - @MANYTOONE
  - muchos a muchos (M:M) - @MANYTOMANY

# ERRORES

- **detached entity passed to persist**
  - Se está intentando insertar en la BD un objeto con el mismo ID que uno que ya existe.



- **Tarea 0:** TAREAS/TareaCero

- Crea un proyecto con las dependencias MySql Driver, Spring Boot Dev Tools, Spring Data JPA. La clase main implementará CommandLineRunner
- Crea la BD **academia** y en modo **update**. Las clases:
  - **Estudiante** (Integer id, String nombre, String apellidos).
  - **Matricula** (Integer id, String curso\_academico)
  - **Curso** (Integer id, String nombre).
  - **Asignatura** (Integer id, String nombre)
- Crea los repositorios para cada clase.

- **Tarea 0:** TAREAS/TareaCero

- **Relaciones:**

- Estudiante – Matricula → OneToOne
- Estudiante – Curso → ManyToOne
- Estudiante – Asignatura → ManyToMany
- En el método run llena las tablas con una instancia de cada clase relacionada con las clases con las que tiene relación.

- Un registro de una entidad A se relaciona sólo con un registro de la entidad B y viceversa.
  - Por Ejemplo, un Coche tiene asignado una Matrícula y una Matrícula pertenece sólo a un Coche.
- Los datos de esta relación suelen meterse en una de las dos tablas.
- Se desaconseja su uso por motivos de rendimiento.
- En el mundo empresarial este tipo de relación no se usa optándose siempre por la relación @ManyTone.

- **Ejemplo 1:** EJEMPLOS/EjemploOneToOne

- Un Usuario tiene un Dni y un Dni es de un Usuario.
- @JoinColumn establece el nombre de la columna para el dni dentro de la tabla Usuario que guardará la relación.

```
Dni.java X
2 public class Dni {
3     @Id
4     @GeneratedValue(strategy = GenerationType.IDENTITY)
5     private Integer id;
```

```
Usuario.java X
@OneToMany(cascade = CascadeType.ALL)
@JoinColumn(name = "dni_id", unique = true)
private Dni dni;
```



- **Ejemplo 1:** EJEMPLOS/EjemploOneToOne
  - Las tablas que se crean en la BD son estas.

### Usuarios

id	apellidos	direccion	edad	nombre	dni_id
1	Perez	calle de la luz, 21	34	Pedro	1

### Dnis

id	codigo	numero
1	123	123454678

1 Usuario sólo tiene 1 DNI

- **Tarea 1:** TAREAS/TareaUno

- Crea un proyecto con las dependencias de Spring Web, Thymeleaf, Validation, MySql Driver, Spring Boot Dev Tools, Spring Data JPA.
- Crea la BD **optica** y haz que siempre se cree el esquema al iniciar.
- Crea las clases:
  - **Paciente** (Integer id, String nombre, String apellidos, Graduacion graduacion). Los atributos String tendrán un máximo de 50.
  - **Graduacion** (Integer id, double izquierdo, double derecho). Los atributos double valor mínimo de 0.
- Establece las relaciones para que un Paciente sólo tenga una Graduacion.

- **Tarea 1:** TAREAS/TareaUno
  - Crea las interfaces **IPacienteRepo** y **IGraduacionRepo** que extiendan de CrudRepository
  - Crea la vista **form.html**
    - Tendrá un **formulario** para introducir los pacientes junto con su graduación.
    - Se mostrará una **tabla** BootStrap con todos los pacientes.
    - Añade un botón en cada fila para **Editar** sus datos y otro botón para **Eliminar** el paciente.

- **Tarea 1:** TAREAS/TareaUno

- Crea el controlador **OpticaController** con los endpoints:

- Get:

- / **muestra** todos los pacientes.

- /**editar**/*{id}* muestra el form para editar un paciente.

- /**eliminar**/*{id}* elimina el paciente con el id pasado.

- Post:

- /**editar** cambia los datos del paciente en la BD.

- Inserta 5 registros de cada entidad y relacionalas entre ellas al guardar.

- **Tarea 1:** TAREAS/TareaUno

## Pacientes

Id	Nombre	Apellidos	Graduacion Izquierdo	Graduacion Derecho		
1	Nombre1	Ape1	1.0	2.0	Editar	Eliminar
2	Nombre2	Ape2	2.0	3.0	Editar	Eliminar
3	Nombre3	Ape3	3.0	4.0	Editar	Eliminar
4	Nombre4	Ape4	4.0	5.0	Editar	Eliminar
5	Nombre5	Ape5	5.0	6.0	Editar	Eliminar



## Formulario

Id

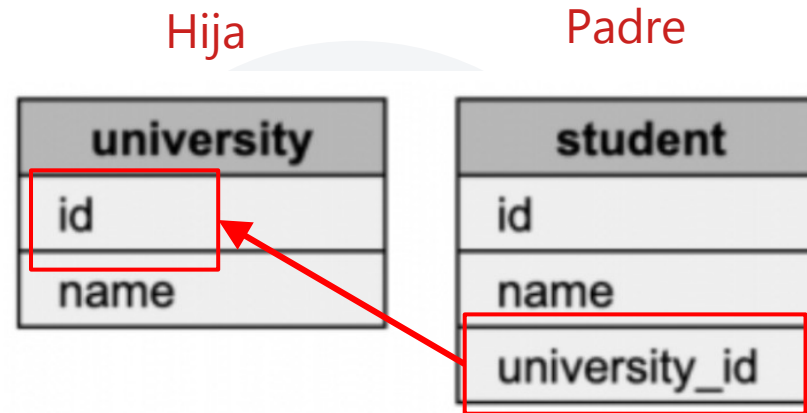
Nombre

Apellidos

Graduacion Izq

Graduacion Der

- La tabla origen (padre) tiene un campo que hace referencia a la clave primaria de la tabla destino (hija).



# @MANYTOONE

- Mapear la propiedad de la clase propietaria (clase padre) de la relación con la anotación @ManyToOne y en la clase hija el atributo al que hace referencia debe anotarse con @OneToMany(mappedBy="propiedad")

**Hija**

```
@Entity
@Table(name = "university")
public class University {

    @OneToMany(mappedBy = "university", cascade =
CascadeType.ALL, orphanRemoval = true)
    private List<Student> students;
```

**Padre**

```
@Entity
@Table(name = "students")
public class Student {

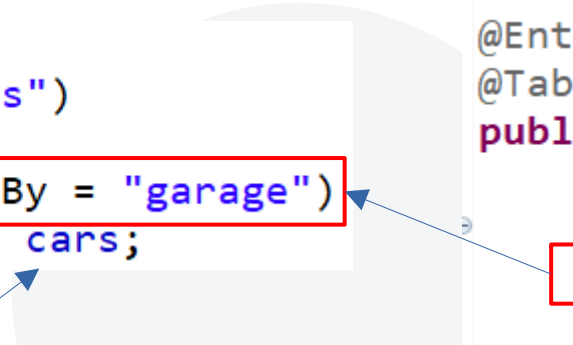
    @ManyToOne()
    @JoinColumn(name = "university_id")
    private University university;
```

Propiedad **"university"** establece la relación.

- **FetchType:** Establece el tipo de carga de los datos desde la BD en los atributos de las clases:
  - EAGER: Se cargarán los datos en cuanto se cree la entidad.
  - LAZY: Se cargarán los datos cuando se haga uso del atributo que esté en la relación.



- **Ejemplo 2:** EJEMPLOS/EjemploManyToOne
  - Muchos coches (Padre) pueden estar en un garaje (Hija).



```
@Entity
@Table(name = "garajes")
public class Garage {
    @OneToMany(mappedBy = "garage")
    private List<Car> cars;
}

@Entity
@Table(name = "coches")
public class Car {
    @ManyToOne
    @JoinColumn(name = "garage_id")
    private Garage garage;
}
```

Propiedad **"garage"** establece la relación.

FetchType.LAZY = cars se llenará con los datos de la BD cuando sea usado. usado

- **Ejemplo 2:** EJEMPLOS/EjemploManyToOne
  - La tablas que se crean en la BD son estas.

Coches

id	color	marca	modelo	garage_id
1	Rojo	Opel	Zafira	1
2	Azul	Renault	Laguna	1

Muchos Coches pueden  
tener 1 mismo Garaje

Garajes

id	direccion	nombre
1	Avenida de la alegría, 7	Veloz

- **Tarea 2:** TAREAS/TareaDos

- Crea un proyecto con las mismas dependencias que la Tarea 1.
- La BD se llamará **discografica**.
- Crea la clase **Album** (Integer id, String titulo, String compositor, List<Cancion> canciones)
- Crea la clase **Cancion** (Integer id, String titulo, Duration duracion)
- Incluye las anotaciones para que un Album pueda tener muchas Canciones y una Cancion esté relacionada con un Album

- **Tarea 2:** TAREAS/TareaDos
  - Crea las interfaces **IAlbumRepo** e **ICancionRepo** que extiendan de JpaRepository.
  - En **ICancionRepo** añade el método **findByAlbumId** para poder obtener todas las canciones que pertenezcan a un Album.
  - Crea la clase **DiscograficaService** con los métodos crud y que usen los repositorios.

- **Tarea 2:** TAREAS/TareaDos
  - Crea las vistas:
    - **cancion.html** mostrará las canciones de un album en una tabla BootStrap. Tendrá los botones para **Editar** y **Eliminar**.
    - **album.html** mostrará los albums en una tabla BootStrap y los botones para **Ver** las canciones, **Editar** y **Eliminar** el Album.
    - Cada vista tendrá un formulario para **Editar**.
  - Crea **DiscoController** con los endpoints para manejar la edicion, eliminacion y listado.

- **Tarea 2:** TAREAS/TareaDos

## Pacientes

Id	Titulo	Compositor	Canciones		
1	Titulo1	Compositor1	Ver	Editar	Eliminar
2	Titulo2	Compositor2	Ver	Editar	Eliminar
3	Titulo3	Compositor3	Ver	Editar	Eliminar
4	Titulo4	Compositor4	Ver	Editar	Eliminar
5	Titulo5	Compositor5	Ver	Editar	Eliminar

## Formulario Album

Id

Titulo

Compositor

- **Tarea 2:** TAREAS/TareaDos

## Canciones

Id	Titulo	Duracion	Album		
1	titulo11	1'30"	Titulo1	Editar	Eliminar

## Formulario Cancion

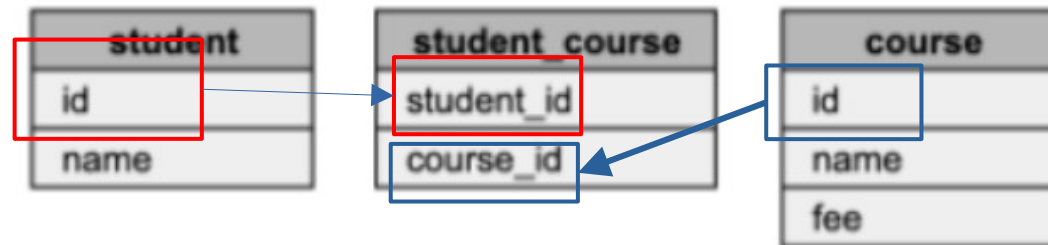
Id

Titulo

Duracion

- Una entidad A se puede relacionar con 1 o con muchas instancias de B y viceversa.
  - Por ejemplo, un Estudiante puede estar en un Curso y en un Curso pueden matricularse muchos Estudiantes.
- La relación creará una nueva tabla en la BD que almacenará las claves primarias de las clases relacionadas.

Nueva





- El propietario es la entidad Student.
- **@JoinTable** nombra la tabla nueva del mapeo (student\_course).
- **@JoinColumns** apunta a la tabla propietaria (student).
- **@InverseJoinColumns** apunta a la otra tabla (course).
- Usamos el cascade **Merge** y **Persist**, pero no **Remove** ya que si eliminamos un curso, no queremos eliminar los estudiantes asociados a él.
- Se usa **Set** y no **List** porque con List Hibernate vacia todo su contenido no solo las del objeto que queremos eliminar (Innecesario e ineficiente).

# @MANYTOMANY

Propietaria

```
@Entity
@Table(name="student")
public class Student {
```

```
    @ManyToMany(cascade = {
        CascadeType.PERSIST,
        CascadeType.MERGE
```

```
    })
```

```
    @JoinTable(
```

```
        name = "student_course",
```

```
        joinColumns = {@JoinColumn(name = "student_id")},
```

```
        inverseJoinColumns = {@JoinColumn(name =
```

```
            "course_id")}
```

```
    )
```

```
    private Set<Course> courses;
```

Inversa

```
@Entity
```

```
@Table(name="course")
```

```
public class Course {
```

```
    @ManyToMany(mappedBy = "courses")
    private Set<Student> students;
```



Propiedad **"courses"** establece la relación.

# @MANYTOMANY

- **Ejemplo 3:** EJEMPLOS/EjemploManyToMany

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

    @ManyToMany(cascade = { CascadeType.PERSIST, CascadeType.MERGE })
    @JoinTable(name = "clase_alumno",
        joinColumns = @JoinColumn(name = "clase_id"),
        inverseJoinColumns = @JoinColumn(name = "alumno_id"))
    private Set<Alumno> alumnos = new HashSet<>();
```

Propiedad "alumnos" establece la relación.

```
@Entity
@Table(name = "alumnos")    Inversa
public class Alumno {

    @ManyToMany(mappedBy = "alumnos")
    private Set<Clase> clases = new HashSet<>();
```



- **Ejemplo 3:** EJEMPLOS/EjemploManyToMany
  - Las tablas que se crean en la BD son estas.

Un alumno puede estar en muchas clases  
y  
Una Clase puede tener muchos alumnos.

clase\_alumno

clase_id	alumno_id
1	1
1	2
2	1
2	2

Alumnos

id	apellidos	dni	edad	nombre
1	Perez	1234567	34	Pedro
2	González	1234678	24	María

id	aforo	edificio	materia	nombre
1	150	Facultad de Ciencias	Biología comparada	Primero
2	80	Facultad de Ciencias	Ciencias del Mar	Segundo

Clases



- **Tarea 3:** TAREAS/TareaTres

- Crea un proyecto con las mismas dependencias que la Tarea 1.
- La BD se llamará **viajes**.
- Crea la clase **Viajero** (Integer id, String nombre, String apellidos, String dni, Set<Tren> trenes)
- Crea la clase **Tren** (Integer id, String matricula, Set<Viajero> viajeros)
- Incluye las anotaciones para que un Viajero pueda viajar en muchos trenes y un Tren tenga muchos viajeros. Tren: creará la tabla Join.

- **Tarea 3:** TAREAS/TareaTres
  - Crea las interfaces **IViajeroRepo** e **ITrenRepo** que extiendan de JpaRepository.
  - Crea la clase **ViajesService** con los métodos crud que usen los repositorios.

- **Tarea 3:** TAREAS/TareaTres

- Crea las vistas:

- **trenes.html :**

- Una tabla de Bootstrap mostrará todos los trenes.
      - Un botón para **Agregar** viajeros y otro para **Ver** los viajeros de cada tren.

- **viajeros.html:**

- Una tabla de Bootstrap mostrará todos los viajeros de un tren.
      - Un formulario que agregará un Viajero para el id del Tren recibido.

- **Tarea 3:** TAREAS/TareaTres
  - Crea **ViajesController** con los endpoints:
    - **/trenes** muestra los trenes en trenes.html
    - **/tren/{id}/viajeros** muestra los viajeros del tren con el id pasado
    - @Get - **/addViajero/tren/{id}** muestra form de viajeros.html
    - @Post - **/addViajero/tren/{id}** guarda un nuevo viajero para el tren.
  - Insertar al iniciar al menos 5 registros de Tren y Viajero.
  - Vuelve a crear la BD al iniciar una nueva ejecución.



- **Tarea 3:** TAREAS/TareaTres

localhost:8080/trenes

### Trenes

Id	Matricula	Viajeros	
1	matricula1	Ver	Agregar
2	matricula2	Ver	Agregar
3	matricula3	Ver	Agregar
4	matricula4	Ver	Agregar
5	matricula5	Ver	Agregar

localhost:8080/addViajero/tren/1

### Formulario Viajero

Nombre

Apellidos

Dni

Enviar

Trenes

localhost:8080/tren/1/viajeros

### Viajeros

Id	Nombre	Apellidos	Dni
6	Pedro	Perez	123456789
1	María	Gonzalez	123456789

Trenes

- **Tarea 4:** TAREAS/TareaCuatro

- A partir del proyecto de la TareaTres realiza las modificaciones para que se cumplan estos requisitos:
  - Un **Vagon** tendrá un número que lo identifique, una cantidad de asientos y una categoria (primera, segunda y mercancías)
  - Un **Asiento** tendrá un número que lo identifique.
  - Un **Vagon** tendrá muchos Asientos, pero un **Asiento** sólo estará en un Vagón.
  - Un **Tren** podrá tener varios vagones, pero un **Vagon** solo podrá estar en un Tren.
  - Un **Tren** podrá tener muchos **Viajeros** y un **Viajero** podrá viajar en muchos **Trenes**.

- **Tarea 4:** TAREAS/TareaCuatro
  - **vagon.html**
    - Mostrará los vagones que hay en un tren pasado.
    - Tendrá un formulario para crear vagones.
    - Los asientos se crearán al mismo tiempo que se crea el Vagon indicando un número máximo.
  - **trenes.html** mostrará un nuevo botón para **Ver** los Vagones del tren.

- **Tarea 4: TAREAS/TareaCuatro**

localhost:8080/trenes

## Trenes

Id	Matricula	Viajeros		Vagon	
1	matricula1	Ver	Agregar	Ver	Agregar
2	matricula2		Agregar	Ver	Agregar
3	matricula3		Agregar		Agregar
4	matricula4		Agregar		Agregar
5	matricula5		Agregar		Agregar

localhost:8080/addVagon/tren/1

## Formulario Vagon

Categoria

primera

Asientos

3

Enviar

Trenes

localhost:8080/tren/1/vagones

## Vagones

Id	Categoria	Asientos
1	primera	3

Trenes

Tabla

asientos

trenes

tren\_viajero

vagones

viajeros

# CASCADETYPE

- Es el mecanismo que permite replicar las acciones entre las entidades relacionadas en la BD.
- Por ejemplo, si borramos un Cliente también sería necesario eliminar su Dirección.
- Las operaciones en cascada de JPA / Hibernate se representan con el enum **javax.persistence.CascadeType**.

# CASCADETYPE

- Se debe añadir en la anotación de la relación con cascadeType

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

- Si tenemos las entidades A y B relacionadas para establecer la política debemos preguntarnos:
  - ¿Si insertamos A deberíamos insertar también B? SI, si no existe → PERSIST
  - ¿Si actualizamos A deberíamos actualizar también B? SI, si ha cambiado → MERGE
  - ¿Si borramos A deberíamos borrar también B? SI, es única → REMOVE

# CASCADETYPE

- Las operaciones más comunes son:
  - CascadeType.**ALL**: se aplican todos los tipos de cascada.
  - CascadeType.**PERSIST**: al guardar en la BD las entidades padre también se guardarán las entidades relacionadas.
  - CascadeType.**MERGE**: las entidades relacionadas se unirán al contexto de persistencia cuando la entidad propietaria se una.
  - CascadeType.**REMOVE**: al eliminar la entidad padre también se eliminan las entidades relacionadas.

# CASCADETYPE

- Las operaciones menos usadas son:
  - CascadeType.**REFRESH**: las entidades relacionadas actualizan sus datos desde la base de datos cuando la entidad propietaria se actualiza.
  - CascadeType.**DETACH**: las entidades relacionadas se separan del contexto de persistencia cuando ocurre una operación de separación manual.



# CASCADETYPE

- En el código siguiente al crear un **Cliente** se creará su **Direccion** y su **Factura**. Si se elimina un **Cliente** se eliminará su **Direccion**, pero no su **Factura**.

@Entity

```
public class Cliente {
```

```
    @OneToOne(cascade={CascadeType.PERSIST, CascadeType.REMOVE})
```

```
    Direccion direccion;
```

```
    @OneToOne(cascade={CascadeType.PERSIST})
```

```
    Factura factura;
```

```
}
```



# CUÁNDO USAR CADA CASCADETYPE

- **ALL:** Se debe evitar porque crea efectos indeseados en el modelo.
- **PERSIST:** cuando hay que crear la hija a la vez que el padre.
- **MERGE:** cuando hay que propagar la actualización entre hija y padre.
- **REMOVE:** usar con cuidado. Es seguro en relaciones one-to-one y one-to-many porque hay una relación de propiedad perfecta y la eliminación del padre implica la eliminación de la hija. Nunca se debe usar en las relaciones many-to-many
- **REFRESH:** para recargar una entidad desde la BD. Pero todo los datos de la entidad en memoria se descartan.
- **DETACH:** saca a un objeto de la persistencia para que no se almacene en la BD.

# PARA QUÉ SIRVE ORPHANREMOVAL

- La entidad hija se borrará cuando el padre ya no la referencia.
- Útil para no tener objetos huérfanos en la BD.
- Si **Cliente** deja de apuntar a una **Direccion** y ahora apunta a otra **Direccion** la **Direccion** inicial no se eliminaría. Con TRUE SI se borraría.

@Entity

public class Cliente {

@OneToOne(cascade={CascadeType.PERSIST,CascadeType.REMOVE},orphanRemoval=true)

Direccion direccion;

@OneToOne(cascade={CascadeType.PERSIST})

Factura factura;

}

- **Tarea 5:** TAREAS/TareaCinco
  - A partir de la TareaCuatro realiza las modificaciones para que se puedan **eliminar** los viajeros, los trenes, los vagones y los asientos.
  - La politica ante borrados es:
    - **Tren** si se borra se borrarán sus vagones, pero no los viajeros.
    - **Vagon** si se borra se borrarán sus asientos, pero no al revés.
  - Añade en **trenes.html** los botones para poder ver todos los **Viajeros**, **Vagones** y **Asientos** en tablas BootStrap.

- **Tarea 5: TAREAS/TareaCinco**

→ ↻ ⓘ localhost:8080/trenes

## Trenes

Id	Matricula	Eliminar	Viajeros	Vagon		
1	matricula1	Eliminar	Ver	Agregar	Ver	Agregar
2	matricula2	Eliminar		Agregar		Agregar
3	matricula3	Eliminar		Agregar		Agregar
4	matricula4	Eliminar		Agregar		Agregar
5	matricula5	Eliminar		Agregar		Agregar

Viajeros

Vagones

Asientos

→ ↻ ⓘ localhost:8080/asientos

## Asientos

Id	Vagon
1	1
2	1
3	1

Trenes