



# UD-11: Gestión del Modelo de datos con Doctrine

Desarrollo Web en Entorno Servidor  
Curso 2020/2021





## Mapeo objeto-relacional

- ORM (Object Relational Mapping, mapeo objeto-relacional) es un modelo de programación que consiste en convertir a objetos todos los elementos de una base de datos relacional, y viceversa (la conversión o mapeo se realiza en ambos sentidos).
- **Ventajas de usar un ORM:** facilidad y velocidad de uso, abstracción de la BD usada, más seguridad de la capa de acceso a datos contra ataques, etc.
- **Inconvenientes de usar un ORM:** en entornos con gran carga, poner una capa más podría mermar el rendimiento, aprender el lenguaje del ORM, etc.
- Usaremos Doctrine, pero hay otros ORM para PHP (Propel, Rocks, Torpor...).
- Doctrine, por tanto, no opera con tablas y filas, sino con entidades y objetos.
- La principal ventaja de usar un ORM es aislar la aplicación del gestor de BD que elijamos (MySQL, Oracle, PostgreSQL...), ya que a nivel de aplicación usaremos objetos, y será Doctrine quien conecte con la BD elegida y transforme los objetos para adaptarlos a la misma.

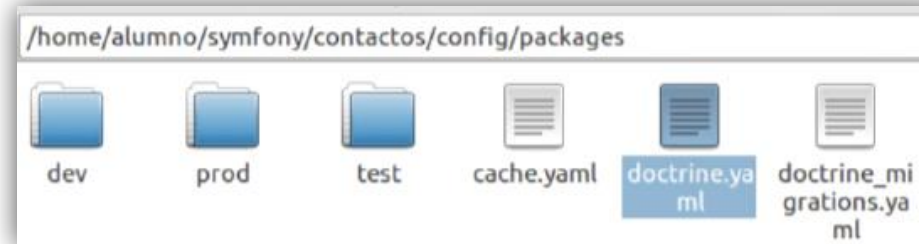


## Configuración básica de Doctrine

- Para usar Doctrine, tenemos que indicar cómo conectar al servidor de BD que vayamos a utilizar. Configuraremos estos parámetros de conexión en el archivo .env de nuestro proyecto, el cual está oculto en la raíz del proyecto.
- Este es un archivo donde se definen ciertas variables propias de entorno. En nuestro caso, definimos una llamada DATABASE\_URL, con una URL donde se especifican la dirección y puerto de conexión a la BD, el login y password necesarios para acceder, y el nombre de la BD a la que conectar.
- Por ejemplo, para una base de datos MySQL, la estructura general será ésta:  
`DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/db_name`
- Un ejemplo de conexión con usuario *root* y contraseña vacía es el siguiente:  
`DATABASE_URL=mysql://root@127.0.0.1:3306/contactos`

# Configuración básica de Doctrine

- Modifica doctrine.yaml en contactos/config/packages e introduce lo siguiente:



```
1 doctrine:
2     dbal:
3         # configure these for your db server
4         driver: 'pdo_mysql'
5         server_version: '5.7'
6         charset: utf8mb4
7         default_table_options:
8             charset: utf8mb4
9             collate: utf8mb4_unicode_ci
10
11         url: '%env(resolve:DATABASE_URL)%'
12
```



## Crear base de datos


- Si la BD aún no existe, Doctrine puede crearla por nosotros. Para ello, escribimos el siguiente comando usando la consola, desde la carpeta principal de nuestro proyecto:

`php bin/console doctrine:database:create`

- Se creará una BD vacía (sin tablas, de momento) cuyo nombre es el de la variable de entorno de la diapositiva anterior. En nuestro ejemplo, **contactos**.
- Es como “*crear BD vacía llamada como la última parte de DATABASE\_URL*”.
- *Si ya existe una BD llamada **contactos**, bórrala y vuelve a intentarlo.*

```
PS C:\xampp\htdocs\contactos> php bin/console doctrine:database:create
Could not create database `contactos` for connection named default
An exception occurred while executing 'CREATE DATABASE `contactos`':

SQLSTATE[HY000]: General error: 1007 Can't create database 'contactos'; database exists
```



```
PS C:\xampp\htdocs\contactos> php bin/console doctrine:database:create
Created database `contactos` for connection named default
PS C:\xampp\htdocs\contactos> █
```



## Creación de entidades

- Las entidades son las clases que van a componer el modelo de datos de nuestra aplicación. Por ejemplo, en nuestra aplicación de contactos, necesitaremos una entidad/clase llamada Contacto que almacene los datos de cada contacto (código, nombre, teléfono y email), como las tablas.
- Para crear una entidad, empleamos el siguiente comando desde terminal:  
`php bin/console make:entity`
- Se iniciará un asistente que nos irá pidiendo información para la entidad:
  - Nombre de la clase o entidad
  - Propiedades o atributos de la clase. Para cada uno, pedirá el nombre, el tipo de dato, el tamaño del campo, si admite nulos...
  - Para terminar, pulsaremos Intro.



# Creación de entidades

Por ejemplo, para el caso de nuestra entidad Contacto, el proceso quedaría así:

```
Class name of the entity to create or update (e.g. TinyPuppy):
> Contacto
created: src/Entity/Contacto.php
created: src/Repository/ContactoRepository.php
Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-running
this command.
New property name (press <return> to stop adding fields):
> nombre
Field type (enter ? to see all types) [string]:
> string
Field length [255]:
> 255
Can this field be null in the database (nullable) (yes/no) [no]:
> no
updated: src/Entity/Contacto.php
Add another property? Enter the property name (or press <return>
to stop adding fields):
> telefono
Field type (enter ? to see all types) [string]:
> string
```

```
Field length [255]:
> 15
Can this field be null in the database (nullable) (yes/no) [no]:
> no
updated: src/Entity/Contacto.php
Add another property? Enter the property name (or press <return>
to stop adding fields):
> email
Field type (enter ? to see all types) [string]:
> string
Field length [255]:
> 255
Can this field be null in the database (nullable) (yes/no) [no]:
> no
updated: src/Entity/Contacto.php
Add another property? Enter the property name (or press <return>
to stop adding fields):
>
Success!
```

**(Aquí se ha pulsado intro para terminar)**

## Creación de entidades

- Como resultado, se generará de forma automática una clase Contacto (Contacto.php) dentro de la carpeta src/Entity.
- Por defecto se añade un atributo id, que es clave primaria y autoincremental. Se podría cambiar esto, claro. Los otros atributos que aparecen son los que pusimos en el terminal: nombre, telefono y email. Como podrás ver, se generan de forma automática los métodos get y set de cada uno de ellos.

```
8  /**
9   * @ORM\Entity(repositoryClass=ContactoRepository::class)
10  */
11  class Contacto
12  {
13      /**
14       * @ORM\Id
15       * @ORM\GeneratedValue
16       * @ORM\Column(type="integer")
17       */
18      private $id;
19
20      /**
21       * @ORM\Column(type="string", length=255)
22       */
23      private $nombre;
24
25      public function getId(): ?int
26      {
27          return $this->id;
28      }
```



## Creación de entidades

- Una vez hemos definido la entidad, podemos generar la correspondiente tabla en la base de datos. Para ello, escribimos este comando:

```
php bin/console make:migration
```

- Lo que hace este comando es cotejar los cambios entre nuestro modelo de entidades y el esquema de la BD, generando un archivo que volcará esos cambios a la BD. Por consola se nos informará de dónde está este archivo para que lo comprobemos (carpeta **src/Migrations**).
- Si todo es correcto, con este comando se reflejarán los cambios en la BD:

```
php bin/console doctrine:migration:migrate
```

- Tras ejecutarlos, veremos en PHPMyAdmin las tablas creadas con sus filas en base a las entidades que tengamos.

## Editar o añadir entidades

Podemos editar la clase de la entidad para añadir, modificar o borrar campos. Para añadir campos, ejecutaremos de nuevo el comando `make:entity`, indicando el mismo nombre de clase a modificar, y especificando los nuevos campos.

Tras realizar cambios en las entidades deseadas, hay que volver a generar una nueva migración con los comandos vistos en la diapositiva anterior.

Por defecto, Doctrine agrega un campo `id` a las entidades, que es autonumérico y clave primaria. Si preferimos elegir otro campo como clave primaria, hay que:

1. Eliminar el atributo `id` y su getter correspondiente de la entidad
2. Añadir la siguiente anotación al atributo que hayamos elegido como PK:

```
/**
 * @ORM\Id()
 * ...
 */
```

Si quisiéramos una clave primaria compuesta por más de un campo, añadiríamos esta anotación en cada campo que forme parte de la clave primaria.



# Insertar objetos a la tabla

```
/**
 * @Route("/contacto/insertar", name="insertar")
 */
public function insertar()
{
    $contacto = new Contacto();
    $contacto->setNombre("Inserción de prueba");
    $contacto->setTelefono("900110011");
    $contacto->setEmail("insercion.de.prueba@contacto.es");

    $entityManager = $this->getDoctrine()->getManager();
    $entityManager->persist($contacto);
    $entityManager->flush();

    return new Response("Contacto insertado con id " . $contacto->getId());
}
```

Se crea un objeto Contacto y se "llean" sus campos

Obtenemos el objeto entity manager

Se hace persist y flush. La idea es que con persist se van preparando los queries, y con flush se lanzan todas. Ambas instrucciones irán juntas.

- ¡Ojo! Por sí sola, la instrucción **persist** no actualiza la BD. Hasta el **flush** no cambia.
- Esta forma de terminar (con Response) no es la que usamos. Esto es para practicar.

## Detectar errores en la inserción

- Si ocurre algún error en la inserción (por ejemplo, porque algún campo es nulo y no puede serlo, o porque se duplica la clave primaria), se provocará una excepción al llamar al método **flush**.
- Trataremos este error capturando la excepción y generando una respuesta:

```
$entityManager->persist($objeto);  
try  
{  
    $entityManager->flush();  
}  
catch (\Exception $e)  
{  
    return new Response("Error insertando objeto");  
}
```

- Es importante y no cuesta nada hacerlo.

## Obtención de objetos

Para obtener **objetos** de una **entidad**, tenemos dos formas diferentes que usaremos dependiendo de la mayor o menor dificultad de la consulta:

- **Consultas simples** → Métodos del repositorio de la clase.
  - Este archivo es como un asistente que nos ayuda a obtener objetos de una clase.
  - Se genera al crear la entidad. Por ejemplo: [src/Repository/ContactoRepository.php](#)
  - Por defecto, tiene los métodos [findAll](#), [find](#), [findBy](#) y [findOneBy](#)
  - En las siguientes diapositivas veremos dichos métodos.
- **Consultas avanzadas** → Mediante ampliación del repositorio.
  - Modificar el archivo de repositorio a mano, añadiendo otros métodos mediante:
    - Lenguaje DQL: lenguaje alternativo para crear sentencias. Quizás, más sencillo. Veremos ejemplos.
    - Query Builder de Doctrine: quizás, más complicado. Veremos ejemplos.



## Métodos del repositorio: findAll y find

El método **findAll** (sin parámetros) obtiene todos los objetos de la colección.

```
$repositorio = $this->getDoctrine()->getRepository(Contacto::class);  
$contactos = $repositorio->findAll();
```

1. Obtenemos el repositorio
2. Accedemos al método **findAll** del repositorio para guardar todos los contactos en la variable \$contactos. Esto equivaldría a hacer “SELECT \* FROM Contacto”.

El método **find** localiza el objeto por la clave primaria recibida como parámetro (por defecto es el id, pero puede ser otro).

En el siguiente ejemplo, buscaríamos el contacto cuya id es 1:

```
$repositorio = $this->getDoctrine()->getRepository(Contacto::class);  
$contacto = $repositorio->find(1);
```

Accedemos al método **find** del repositorio para guardar en \$contacto el contacto cuy 'id' vale 1. Esto equivaldría a hacer “SELECT \* FROM Contacto where id=1”



## Métodos del repositorio: findBy

El método **findBy** localiza todos los objetos que cumplan los criterios de búsqueda pasados como parámetro. Siempre devuelve un array con todos los resultados coincidentes (aunque sólo devuelva un resultado):

```
$repositorio = $this->getDoctrine()->getRepository(Contacto::class);  
$contactos = $repositorio->findBy(["edad" => 18]);
```

Como ves, los criterios de búsqueda son un array asociativo en el parámetro. Esto equivaldría a hacer “SELECT \* FROM Contacto where **edad=18**”. Hemos accedido al repositorio y hemos usado el método **findBy** del repositorio para guardar en **\$contactos** todos los contactos cuya edad sea justamente 18. Podemos también poner más restricciones. Si quisiéramos buscar los contactos cuya edad sea 18 y cuyo nombre sea Pepe:

```
$contactos = $repositorio->findBy(["edad" => 18, "nombre" => "Pepe"]);
```



## Métodos del repositorio: findByOne

El método **findByOne** es igual que findBy, pero devuelve sólo el primer objeto que cumpla los criterios de búsqueda recibidos como parámetro. Por lo tanto, devuelve un objeto y no un array:

```
$repositorio = $this->getDoctrine()->getRepository(Contacto::class);  
$contacto = $repositorio->findByOne(["edad" => 18]);
```

Se usa como antes. Si quisiéramos el primer contactos de edad 18 y nombre Pepe:

```
$contacto = $repositorio->findByOne(["edad" => 18, "nombre" => "Pepe"]);
```

Los criterios de búsqueda se reciben también como array asociativo. Siguen también siendo criterios exactos: edad justo 18, y nombre "Pepe", tal cual.





## Métodos del repositorio: Resumen

Los métodos que incorpora el repositorio de una entidad por defecto son:

Método	Argumentos que recibe	¿Para qué sirve?	Devuelve un...
<b>findAll</b>	(ninguno)	Encontrar todos los objetos	Array numérico de objetos
<b>find</b>	Un valor de la clave primaria	Encontrar un objeto por su clave primaria	Objeto
<b>findBy</b>	Un array asociativo con criterios de búsqueda exactos	Encontrar todos los objetos que cumplan una/s condición/es	Array numérico de objetos
<b>findOne</b>	Un array asociativo con criterios de búsqueda exactos	Encontrar el primer objeto que cumpla una/s condición/es	Objeto



## Métodos del repositorio: Limitaciones

Con los 4 métodos anteriores podríamos encontrar, por ejemplo:

- Todos los contactos
- El contacto cuya clave primaria (id) es 3
- Los contactos que tengan exactamente 18 años
- Los contactos que tengan exactamente 18 años y se llamen exactamente "Pepe"
- El primer contacto que tenga exactamente 18 años
- El primer contacto que tenga exactamente 18 años y se llame exactamente "Pepe"

Pero, ¿si quisiéramos buscar, por ejemplo, los contactos mayores de 18 años? NO PODEMOS, CON ESTOS MÉTODOS. Los métodos que hay por defecto en el repositorio están muy limitados. Por ello, ahora veremos que habrá que ampliarlos a la hora de hacer consultas más avanzadas.

Para consultas más avanzadas, ampliaremos el repositorio añadiendo métodos manualmente. Vimos que hay dos opciones: Query Builder (objeto de Doctrine para crear sentencias) y el lenguaje DQL (sintaxis alternativa, parecida a SQL).



## Ampliar el repositorio: Query Builder

- En nuestra entidad Contacto, queremos buscar los contactos cuyo nombre contenga un cierto texto. Crearemos un método para esto que reciba el texto por parámetro. Para esto, podríamos editar el repositorio de la entidad (src/Repository/ContactoRepository.php), añadiendo un método manualmente y utilizar Query Builder.

```
public function findByName($text): array
{
    return $this->createQueryBuilder('con')
        ->andWhere('con.nombre LIKE :text')
        ->setParameter('text', '%'.$text.'%')
        ->getQuery()
        ->execute();
}
```

Lo que hace este código es buscar los contactos (con) cuyo nombre sea como el parámetro `text`, y luego especifica que dicho parámetro `text` es igual al parámetro que recibimos en el método, encerrado entre "%" para indicar que da igual lo que haya delante o detrás del texto.

- Se usa el **query builder** de Doctrine para construir la consulta con esta sintaxis específica. Primero se define un elemento que hemos llamado **con** (de Contacto) que se usará para referenciar las propiedades de los contactos, por ejemplo, en la cláusula **where** (con.nombre).



## Ampliar el repositorio: Query Builder

Tras eso, ya podríamos usar dicho método donde lo necesitemos. Por ejemplo, podemos modificar el método buscar de **ContactoController** para que busque contactos por nombre usando este nuevo método:

```
/**
 * @Route("/contacto/{texto}", name="buscar_contacto")
 */
public function buscar($texto)
{
    $repositorio = $this->getDoctrine()->getRepository(Contacto::class);
    $resul = $repositorio->findByName($texto);

    return $this->render('lista_contactos.html.twig', array('contactos' => $resul));
}
```

Así usaríamos el método **findByName** que hemos añadido manualmente al repositorio en la diapositiva anterior.

Para saber más sobre Query Builder, puedes consultar [este enlace](#).



## Ampliar el repositorio: Lenguaje DQL

- Otra opción para ampliar el repositorio es el lenguaje DQL (Doctrine Query Language).
- Es fácil de usar, ya que es muy parecido a SQL.
- Aunque aparezcan las palabras SELECT, DELETE y UPDATE no hay que olvidar que es un lenguaje para objetos, no para tablas. Por lo qué habrá pequeñas diferencias con SQL.
- Aunque se puede usar para eliminar y actualizar objetos, no se pueden insertar.
- Por ejemplo, este método obtendría **todos los contactos**:

```
public function todosLosContactos(): array
{
    $entityManager = $this->getEntityManager();
    $query = $entityManager->createQuery('SELECT con FROM App\Entity\Contacto con');
    return $query->getResult();
}
```

con es un alias que asignamos a la entidad Contacto

- Esta función hace lo mismo que **findAll()**. Por lo que solo nos sirve de ejemplo.



## Ampliar el repositorio: Lenguaje DQL

- Una vez creado el método anterior, bastaría con llamarlo desde el controlador correspondiente. Por ejemplo, así:

```
$repositorio = $this->getDoctrine()->getRepository(Contacto::class);  
$contactos = $repositorio->todosLosContactos();
```

Con esto, tendríamos un array de todos los contactos en \$contactos, para luego pasárselos a la vista, por ejemplo.

- Veamos otro ejemplo. Obtener los contactos con 18 años o más:

```
...  
$query = $entityManager->createQuery('SELECT con FROM App\Entity>Contacto con  
                                     WHERE con.edad >= 18'); // con.edad es objeto  
...
```

- Contactos mayores de una edad que tenemos en una variable del controlador.

```
public function mayoresDe($edad): array  
{  
    $entityManager = $this->getEntityManager();  
    $query = $entityManager->createQuery('SELECT con FROM App\Entity>Contacto con  
                                           WHERE con.edad >= :edad')  
    return $query->setParameter('edad', $edad)->getResult();  
}
```



## Ampliar el repositorio: Lenguaje DQL

- En el controlador correspondiente, la llamaríamos así:

```
$repositorio = $this->getDoctrine()->getRepository(Contacto::class);  
$contactos = $repositorio->mayoresDe($edad);
```

- El ejemplo de buscar un contacto por nombre, con DQL podría quedar así:

```
public function contactosPorTexto($texto): array  
{  
    $entityManager = $this->getEntityManager();  
    $query = $entityManager->createQuery('SELECT con FROM App\Entity>Contacto con  
                                         WHERE con.nombre LIKE :texto');  
    $query->setParameter(':texto', '%'.$texto.'%');  
    return $query->getResult();  
}
```

- Para saber más sobre DQL, puedes consultar [este enlace](#).

## Actualizar objetos

- Para actualizar un objeto en una BD, debemos seguir tres pasos:
  1. Obtener el objeto de la BD (por ejemplo, con un *find* dando su PK).
  2. Modificarlo con los setters del objeto.
  3. Hacer un flush para actualizar los cambios en la BD.
- Para actualizar, por ejemplo, los datos del contacto con id = 1, haríamos esto:

```
$entityManager = $this->getDoctrine()->getManager();  
$repositorio = $this->getDoctrine()->getRepository(Contacto::class);  
$contacto = $repositorio->find(1); // encontramos el objeto  
if ($contacto)  
{  
    $contacto->setNombre("Nuevo nombre"); // le cambiamos el nombre  
    $entityManager->flush(); // con esto, se reflejará en la BD  
}
```

Recuerda que ese método setNombre se creó automáticamente en libros/src/Entity/Contacto.php cuando hicimos la entidad Contacto





## Borrar objetos

- El borrado es similar a la actualización: también obtenemos el objeto, pero después llamamos al método **remove** para borrarlo, y finalmente a **flush**.
- Así borraríamos el contacto con id = 1:

```
$entityManager = $this->getDoctrine()->getManager();  
$repositorio = $this->getDoctrine()->getRepository(Contacto::class);  
$contacto = $repositorio->find(1);  
  
if ($contacto)  
{  
    $entityManager->remove($contacto);  
    $entityManager->flush();  
}
```

- Nuevamente, tanto en la actualización como en el borrado, el método **flush** puede provocar una excepción si la operación no ha podido llevarse a cabo.
- Podríamos tenerlo en cuenta para capturarla y generar una respuesta (con try-catch).
- TAMBIÉN PODRÍAMOS HACERLO CON DQL con la sintaxis DELETE de SQL.



## Relaciones entre entidades

- Hasta ahora las operaciones que hemos hecho se han centrado en una única tabla o entidad (la entidad/tabla de contactos). Ahora veremos cómo trabajar con más de una tabla/entidad, relacionadas entre sí.
- Hay dos tipos principales de relaciones entre entidades:
  - **Uno a muchos:** en este tipo se englobarían las relaciones “uno a muchos”, “muchos a uno” y “uno a uno”, ya que en cualquiera de los tres casos, la relación se refleja añadiendo una clave ajena en una de las dos entidades que referencie a la otra.
  - **Muchos a muchos:** en este tipo de relaciones, se necesita de una tabla adicional para reflejar la relación entre las entidades.



## Relaciones entre entidades

- Definamos una relación uno a muchos en nuestra BD de contactos. Crearemos una entidad **Provincia**, con sólo un id autogenerado y un nombre (string):

```
php bin/console make:entity
Class name of the entity to create or update (e.g. AgreeableGnome):
> Provincia
created: src/Entity/Provincia.php
created: src/Repository/ProvinciaRepository.php
New property name (press <return> to stop adding fields):
> nombre
Field type (enter ? to see all types) [string]:
> string
Field length [255]:
> 255
Can this field be null in the database (nullable) (yes/no) [no]:
> no
updated: src/Entity/Provincia.php
Add another property? Enter the property name (or press <return> to stop adding fields):
>
Success!
```

- Tras generar la nueva entidad, la trasladamos a la BD con la migración:

```
php bin/console make:migration
php bin/console doctrine:migration:migrate
```



# Relaciones entre entidades

- Ahora, vamos a hacer que los contactos tengan una provincia asociada. Para ello, editamos la entidad Contacto y le añadimos un nuevo campo llamado provincia, que será de tipo “uno a muchos” (un contacto tendrá una provincia, y una provincia puede tener muchos contactos).

## php bin/console make:entity

Class name of the entity to create or update (e.g. DeliciousPuppy):

> **Contacto**

Your entity already exists! So let's add some new fields!

New property name (press <return> to stop adding fields):

> **provincia**

Field type (enter ? to see all types) [string]:

> **relation**

What class should this entity be related to?:

> **Provincia**

What type of relationship is this?

-----

Type Description

-----

ManyToOne Each Contacto relates to (has) one Provincia.

Each Provincia can relate/has to (have) many Contacto objects

OneToMany Each Contacto can relate to (have) many Provincia objects.

Each Provincia relates to (has) one Contacto

Each Provincia relates to (has) one Contacto

ManyToOne Each Contacto can relate to (have) many Provincia objects.

Each Provincia can also relate to (have) many Contacto objects

OneToOne Each Contacto relates to (has) exactly one Provincia.

Each Provincia also relates to (has) exactly one Contacto.

-----

Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]:

> **ManyToOne**

Is the Contacto.provincia property allowed to be null (nullable)? (yes/no) [yes]:

> **no**

Do you want to add a new property to Provincia so that you can access/update Contacto objects from it

- e.g. \$provincia->getContactos()? (yes/no) [yes]:

> **no**

updated: src/Entity/Contacto.php

Add another property? Enter the property name (or press <return> to stop adding fields):

>

Success!



## Relaciones entre entidades

- Como puede verse, a la hora de elegir el tipo de campo, indicamos que es una relación (relation), en cuyo caso nos pide indicar a qué entidad está vinculada (Provincia, en este caso), y qué tipo de relación es (ManyToOne en nuestro caso, pero podemos elegir cualquiera de las otras tres opciones OneToMany, OneToOne o ManyToMany). También podemos comprobar que el asistente nos pregunta si queremos añadir un campo en la otra entidad para que la relación sea bidireccional (es decir, para que desde un objeto de cualquiera de las dos entidades podamos consultar el/los objeto(s) asociado(s) de la otra. En este caso hemos indicado que no, para simplificar el código.
- Tras estos cambios, realizaremos de nuevo la migración
- Llegados a este punto, ya tendremos el nuevo campo añadido en nuestra entidad Contacto y a la tabla contacto de la BD:

```
/**
 * @ORM\ManyToOne(targetEntity="App\Entity\Provincia")
 * @ORM\JoinColumn(nullable=false)
 */
private $provincia;
```

#	Name	Type	Collation	Attributes	Null	Default
<input type="checkbox"/> 1	id 🔑	int(11)			No	None
<input type="checkbox"/> 2	nombre	varchar(255)	utf8mb4_unicode_ci		No	None
<input type="checkbox"/> 3	telefono	varchar(15)	utf8mb4_unicode_ci		No	None
<input type="checkbox"/> 4	email	varchar(255)	utf8mb4_unicode_ci		No	None
<input type="checkbox"/> 5	provincia_id	int(11)			No	None



## Inserción de entidades relacionadas

- Ahora, ¿cómo podemos insertar un objeto en una entidad que depende de otra?
- Si quisiéramos insertar un contacto asignándole una provincia **que no existe**, haríamos:

```
$provincia = new Provincia();  
$provincia->setNombre("Alicante");
```

Creamos la provincia

```
$contacto = new Contacto();  
$contacto->setNombre("Cliente alicantino");  
$contacto->setTelefono("900220022");  
$contacto->setEmail("allicantino22@contacto.es");  
$contacto->setProvincia($provincia);
```

Creamos el contacto

“Metemos” la provincia en el campo del cliente

```
$entityManager = $this->getDoctrine()->getManager();  
$entityManager->persist($provincia);  
$entityManager->persist($contacto);  
$entityManager->flush();
```

Trasladamos todo a la BD

- Para insertar un contacto asignándole una provincia ya existente, no necesitaríamos crear una nueva provincia como antes. Bastaría con usar la línea siguiente:

```
$provincia = $repositorio->find(1);
```



## Búsqueda de entidades relacionadas

- ¿Cómo accedemos a los datos de una entidad desde la otra? Si buscamos en una entidad que está relacionada con otra, el acceso a esa otra entidad se hace desde la primera.
- Por ejemplo, si quisiéramos saber el nombre de la provincia del contacto con código 3, haríamos algo así:

```
$repositorio = $this->getDoctrine()->getRepository(Contacto::class);  
$contacto = $repositorio->find(3);  
$nombreProvincia = $contacto->getProvincia()->getNombre();
```

- Accedemos al campo provincia del contacto, que es una relación a un objeto Provincia, del cual sacamos el nombre.

# EJERCICIO 1



- En la aplicación de libros, crea una entidad llamada **Libro** con estos atributos:
  - ✓ **isbn** (string de tamaño 20). Este atributo deberá ser establecido como PK.
  - ✓ **titulo** (string de tamaño 255)
  - ✓ **autor** (string de tamaño 100)
  - ✓ **paginas** (integer)
- Ninguno de los campos anteriores admite nulos.
- Recuerda definir previamente la variable **DATABASE\_URL**, y modificar el fichero **doctrine.yaml**, crear la BD llamada Libros y luego migrar estos cambios a la BD, usando los comandos vistos en esta unidad.
- Después, modifica los archivos **InicioController.php** y **LibroController.php** para usar *Doctrine*.
- Añade un nuevo controlador **insertar (/libro/insertar)**, que inserte en la base de datos los cuatro libros de ejemplo que teníamos.
- Modifica los dos controladores (**listar\_libros**, **ficha\_libro**) que teníamos para que funcionen correctamente con la base de datos, usando en cada caso un método del repositorio.

+ Opciones					isbn	titulo	autor	paginas
<input type="checkbox"/>	Editar	Copiar	Borrar	A001	Jarry Choped	JK Bowling	100	
<input type="checkbox"/>	Editar	Copiar	Borrar	A002	El señor de los palillos	JRR Tolquien	200	
<input type="checkbox"/>	Editar	Copiar	Borrar	A003	Los polares de la tierra	Ken Follonet	300	





- Añade un nuevo controlador a **LibroController** y llámalo **eliminarLibro**, que irá asociado a la URI **/eliminar/{isbn}** y eliminará de la BD el libro cuyo *isbn* sea el recibido. **Este controlador no irá asociado a ninguna vista.**
- Por lo tanto, no terminará con ningún *render*. Lo que hará en la última línea será “saltar” al apartado donde se ven todos los libros. En Symfony se hace así:

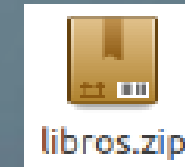
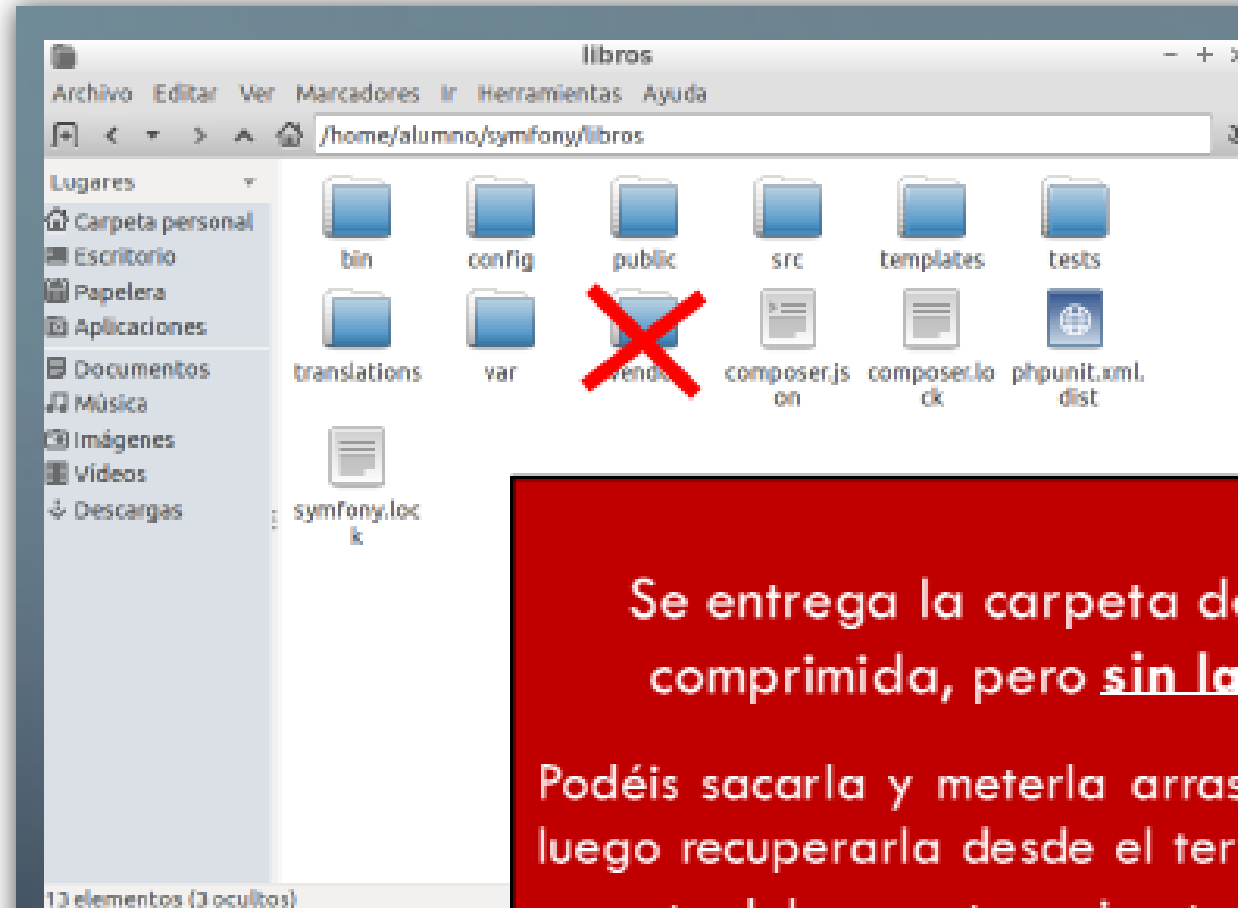
```
return $this->redirectToRoute('libros');
```

- NO SE PERMITE HACER EL BORRADO CON DQL. Debe hacerse con `remove` y `flush`.
- Posteriormente, modifica la plantilla **lista\_libros.html.twig** para que, al lado de cada uno, haya un enlace [ eliminar ] que llame al controlador anterior, borrando el libro.
- Añade un nuevo controlador a **LibroController** llamado **filtrarPaginas**, asociado a la URI **/libros/paginas/{paginas}** y buscará los libros que tengan como máximo el número de páginas recibido por parámetro (esta lista se mostrará en **lista\_libros\_paginas.html.twig**).
- Para esto, tendrás que ampliar el repositorio, usando DQL:

```
public function nPaginas($paginas):array
```



- Crea una nueva entidad llamada **Editorial** en la aplicación de libros. Como único campo, tendrá un **nombre** (string de tamaño 255), además del **id** a\_i.
- Ahora, edita la entidad Libro y añade una nueva propiedad llamada editorial, que será una relación muchos a uno (ManyToOne) con la entidad Editorial creada antes. Esta propiedad podrá tener nulos, y no necesitamos que se genere la propiedad en la otra entidad para comunicación bidireccional.
- A continuación, define en **LibroController** un controlador llamado **insertarConEditorial**, que responda a la URI **/contacto/insertarConEditorial** e inserte una editorial llamada *Alfaguara*, y luego el siguiente libro asociado a esa editorial:
  - isbn = "2222BBBB"
  - titulo = "Libro de prueba con editorial"
  - autor = "Autor de prueba con editorial"
  - paginas = 200



Se entrega la carpeta del proyecto **libros**  
comprimida, pero sin la carpeta vendor

Podéis sacarla y meterla arrastrando. O eliminarla y  
luego recuperarla desde el terminal, ubicándote en la  
carpeta del proyecto y ejecutando `composer install`