

# Symfony

---

*Desarrollo de aplicaciones MVC en el servidor con un  
framework PHP*

## 4. Gestión del modelo de datos con Doctrine

Ignacio Iborra Baeza  
Julio Martínez Lucas

# Índice de contenidos

<b>Symfony.....</b>	<b>1</b>
1.Introducción a Doctrine.....	3
1.1.¿Qué es un ORM?.....	3
1.2.Configuración básica de Doctrine.....	3
2.Creación de entidades.....	5
2.1.Generación del esquema.....	8
2.2.Editar entidades.....	8
2.3.Establecer otras claves primarias.....	9
3.Operaciones contra la base de datos.....	10
3.1.Insertar objetos.....	10
3.1.1.Detectando errores en la inserción.....	11
3.2.Obtener objetos.....	11
3.2.1.Consultas más avanzadas.....	12
3.3.Actualizar y borrar objetos.....	14
3.3.1.Actualizar objetos.....	14
3.3.2.Borrar objetos.....	14
4.Relaciones entre entidades.....	15
4.1.Trabajar con entidades relacionadas.....	17
4.1.1.Inserción de entidades relacionadas.....	18
4.1.2.Búsqueda de entidades relacionadas.....	18
5.Ejercicios.....	20
5.1.Ejercicio 1.....	20
5.2.Ejercicio 2.....	20
5.3.Ejercicio 3 (opcional).....	21
5.4.Ejercicio 4.....	21

# 1. Introducción a Doctrine

---

## 1.1. ¿Qué es un ORM?

---

Un ORM (*Object Relational Mapping*) es un framework encargado de tratar con una base de datos relacional (conectar con ella, realizar operaciones de consulta, inserción, etc.), de forma que, de cara a la aplicación, se convierten a objetos todos los elementos que se extraigan de la base de datos, y viceversa (los objetos de la aplicación se transforman en registros de la base de datos, llegado el caso). De esta forma, el ORM se encargará de realizar esta conversión o mapeo automáticamente por nosotros. Definiendo una serie de reglas, indicaremos qué tablas de la base de datos relacional se corresponden con qué clases de nuestro modelo, y qué campos de cada tabla se corresponden con qué atributos de cada clase. A partir de ahí, el ORM se encargará de extraer la información de la base de datos y crear los objetos correspondientes, o de convertir los objetos con sus atributos en registros de la base de datos, con sus correspondientes columnas.

La principal ventaja de utilizar un ORM como Doctrine es aislar la aplicación del gestor de base de datos que hayamos elegido (MySQL, Oracle, PostgreSQL...) ya que a nivel de aplicación trabajaremos con objetos, y será Doctrine quien se encargue de conectar con la base de datos elegida, y transformar los objetos para adaptarlos a la misma.

## 1.2. Configuración básica de Doctrine

---

Para poder utilizar Doctrine, tenemos que indicar cómo conectar al servidor de base de datos que vayamos a utilizar. Estos parámetros de conexión se pueden configurar en el archivo `.env` de nuestro proyecto. Este es un archivo donde se definen ciertas variables propias de entorno, que luego se procesan y se convierten en variables reales. En nuestro caso, definimos una llamada `DATABASE_URL`, con una URL donde se especifican tanto la dirección y puerto de conexión a la base de datos, como el *login* y *password* necesarios para acceder, y el nombre de la base de datos 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
```

Teniendo en cuenta el usuario y contraseña por defecto de *phpMyAdmin* para XAMPP (usuario *root* y contraseña vacía), éste podría ser un valor válido para conectar a nuestra base de datos de contactos:

```
DATABASE_URL=mysql://root@127.0.0.1:3306/contactos
```

Este archivo `.env` puede utilizarse tanto en desarrollo como en producción, aunque en este último caso se recomienda definir variables de entorno reales en el sistema, para evitar pérdida de rendimiento al tener que traducir este archivo para cada petición.

**NOTA:** a la hora de compartir el código en sistemas *git* o similares, debemos tener cuidado de no compartir este archivo `.env`, ya que puede contener información vulnerable para nuestro sistema, como rutas absolutas a ciertos recursos, usuarios, contraseñas, etc.

En el caso de que la base de datos aún no exista, Doctrine puede crearla por nosotros. Para ello, basta con escribir el siguiente comando:

```
php bin/console doctrine:database:create
```

Automáticamente, se tomará el nombre de la base de datos de la variable de entorno anterior, se conectará al servidor y se creará (sin tablas, de momento).

## 2. Creación de entidades

---

Las entidades son las clases que van a componer el modelo de datos de nuestra aplicación. Por ejemplo, para nuestra aplicación de contactos, necesitaremos una entidad/clase llamada *Contacto* que almacene los datos concretos de cada contacto (código, nombre, teléfono y e-mail).

Para crear una entidad, empleamos el siguiente comando desde el terminal (dentro de la carpeta principal de nuestro proyecto Symfony):

```
php bin/console make:entity
```

Se iniciará un asistente que nos irá pidiendo información para construir la entidad:

- Nombre de la clase o entidad
- Propiedades o atributos de la clase, para cada uno, pedirá el nombre (si directamente pulsamos *Intro* dejará de pedirnos más datos), el tipo de dato, la longitud o tamaño del campo, si admite nulos...

Por ejemplo, para el caso de nuestra clase *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!

Como resultado, se generará una clase *Contacto* dentro de la carpeta *src/Entity*. El código queda como sigue:

...

```
use Doctrine\ORM\Mapping as ORM;
```

```
/**
```

```
 * @ORM\Entity(repositoryClass="App\Repository\ContactoRepository")
```

```
 */
```

```
class Contacto
```

```
{
```

```
    /**
```

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

```
     * @ORM\GeneratedValue()
```

```
     * @ORM\Column(type="integer")
```

```
     */
```

```
    private $id;
```

```
    /**
```

```
     * @ORM\Column(type="string", length=255)
```

```
     */
```

```
    private $nombre;
```

```
    /**
```

```
     * @ORM\Column(type="string", length=15)
```

```
     */
```

```
    private $telefono;
```

```

/**
 * @ORM\Column(type="string", length=255)
 */
private $email;

public function getId()
{
    return $this->id;
}

public function getNombre(): ?string
{
    return $this->nombre;
}

public function setNombre(string $nombre): self
{
    $this->nombre = $nombre;
    return $this;
}

public function getTelefono(): ?string
{
    return $this->telefono;
}

public function setTelefono(string $telefono): self
{
    $this->telefono = $telefono;
    return $this;
}

public function getEmail(): ?string
{
    return $this->email;
}

public function setEmail(string $email): self
{
    $this->email = $email;
    return $this;
}
}

```

Como podemos observar, el campo *codigo* que usábamos en nuestra base de datos de prueba lo hemos reemplazado por un *id* autonumérico que se genera automáticamente como clave principal de la clase. Por lo tanto, sólo hemos tenido que especificar el nombre, teléfono y e-mail, de tipo *string*.

En cuanto a los tipos de datos que podemos especificar, si pulsamos ? e *Intro* cuando vayamos a especificar el tipo de dato, veremos un listado completo de los tipos disponibles (también lo podéis consultar [aquí](#)). Lo habitual será trabajar con cadenas de texto de una longitud determinada (*string*), textos ilimitados (*text*), enteros (*integer*), booleanos (*boolean*), reales (*float*), fechas (*date*, *time* o *datetime*, dependiendo de lo que queramos almacenar)...

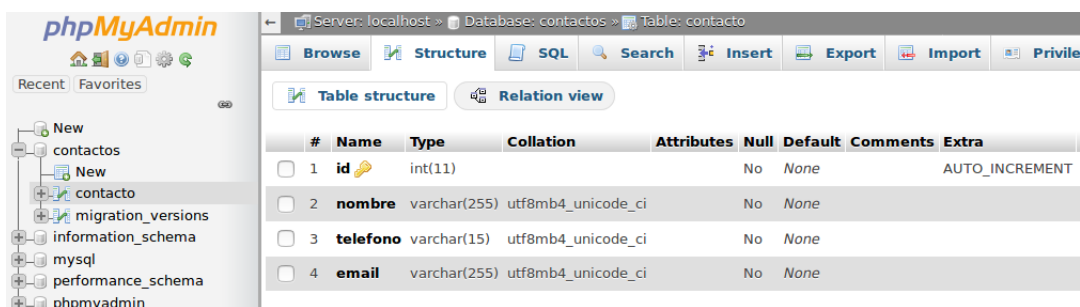
## 2.1. Generación del esquema

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 base de datos, y generar un archivo PHP que se encargará de volcar esos cambios a la base de datos. Por consola se nos informará de dónde está este archivo para que lo comprobemos (estará en la carpeta *src/Migrations*), y si todo es correcto, ejecutando este otro comando se reflejarán los cambios en la base de datos:

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



#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra
1	id	int(11)			No	None		AUTO_INCREMENT
2	nombre	varchar(255)	utf8mb4_unicode_ci		No	None		
3	telefono	varchar(15)	utf8mb4_unicode_ci		No	None		
4	email	varchar(255)	utf8mb4_unicode_ci		No	None		

Como podemos comprobar, cada atributo de nuestra entidad *Contacto* se corresponde con un campo del mismo nombre en la tabla *contacto*.

Conviene mencionar también que este último comando ejecutará todos los archivos de migración que aún no se hayan migrado efectivamente a la base de datos, por lo que si hay alguno que no queramos migrar (porque se haya generado equivocadamente, por ejemplo), deberemos eliminarlo antes.

## 2.2. Editar entidades

¿Qué pasa si, tras crear una entidad, queremos modificar su estructura? Podemos editar la clase de la entidad manualmente para añadir, modificar o borrar campos, pero también podemos volver a ejecutar el comando *make:entity*, indicar el mismo nombre de clase que queremos modificar, y especificar los nuevos campos que queramos añadir (en el caso de que lo que queramos sea añadir campos).

Después de definir los cambios en la(s) entidad(es) deseada(s), deberemos generar una nueva migración con los comandos vistos en el subapartado anterior.



## 2.3. Establecer otras claves primarias

---

Por defecto, hemos visto que Doctrine agrega un campo *id* a las entidades, que es autonumérico y actúa como clave primaria. En el caso de que no queramos que sea así, y prefiramos elegir otro campo no autonumérico como clave primaria, debemos seguir estos pasos:

- Eliminar el atributo *id* y su *getter* correspondiente de la entidad
- Añadir la siguiente anotación al atributo que hayamos elegido como clave primaria:

```
/**
 * @ORM\Id()
 * ...
 */
private $nombreCampo;
```

En el caso de que sea una clave primaria compuesta por más de un campo, deberemos añadir esta anotación en cada campo que forme parte de la clave primaria.

En este punto, puedes realizar el [Ejercicio 1](#) de los propuestos al final de la sesión.

## 3. Operaciones contra la base de datos

Ahora que ya hemos visto cómo definir entidades simples, veamos cómo realizar operaciones con ellas, tales como inserciones, borrados, modificaciones y consultas. Para realizar estas operaciones, nos valdremos de un objeto muy importante en Doctrine, su *entity manager*, a través del cual haremos las inserciones, borrados, etc. También utilizaremos el repositorio de la entidad correspondiente, para realizar las búsquedas.

### 3.1. Insertar objetos

Si queremos añadir objetos nuevos a nuestra base de datos, basta con que creamos un objeto de la entidad correspondiente en el método oportuno, y llamemos al método *persist* y *flush* del *entity manager* de Doctrine.

Por ejemplo, para probar, vamos a crear un controlador en nuestra clase *ContactoController* asociado a una ruta */contacto/insertar*, que de momento será de pruebas hasta que hagamos un formulario de inserción. Dentro de este método, creamos un objeto *Contacto* con datos prefijados, obtenemos el *entity manager* de Doctrine y persistimos el objeto:

```
/**
 * @Route("/contacto/insertar", name="insertar_contacto")
 */
public function insertar()
{
    $entityManager = $this->getDoctrine()->getManager();

    $contacto = new Contacto();
    $contacto->setNombre("Inserción de prueba");
    $contacto->setTelefono("900110011");
    $contacto->setEmail("insercion.de.prueba@contacto.es");

    $entityManager->persist($contacto);
    $entityManager->flush();

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

**NOTA:** es importante que ubiquemos este controlador antes del controlador de *buscar*, ya que de lo contrario se disparará este último al escribir la URL */contacto/insertar*.

Si accedemos desde el navegador a la ruta *symfony.contactos/contacto/insertar*, podremos ver el resultado en la tabla *contacto* de nuestra base de datos:

+ Options				
	id	nombre	telefono	email
<input type="checkbox"/> Edit <input type="checkbox"/> Copy <input type="checkbox"/> Delete	1	Inserción de prueba	900110011	insercion.de.prueba@contacto.es

Es importante recalcar que la llamada a *persist* por sí sola no actualiza la base de datos, sino que indica que se quiere persistir el objeto indicado. Es la llamada a *flush* la que hace efectiva esa persistencia.

### 3.1.1. Detectando errores en la inserción

Si ocurre algún error en la inserción (por ejemplo, porque algún campo sea nulo y no pueda serlo, o porque se duplique la clave primaria), se provocará una excepción al llamar al método *flush*. Podemos tratar este error simplemente capturando la excepción y generando la respuesta apropiada:

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

## 3.2. Obtener objetos

---

A la hora de obtener objetos de una tabla, existen diferentes métodos que podemos emplear. Por ejemplo:

- El método *find* localiza el objeto por la clave primaria (normalmente el *id*) que se le pasa como parámetro. Así buscaríamos el contacto con *id* 1:

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

- El método *findOneBy* localiza un objeto que cumpla los criterios de búsqueda pasados como parámetro. Así buscaríamos el contacto cuyo teléfono sea "900110011":

```
$contacto = $repositorio->findOneBy(["telefono" => "900110011"]);
```

En el caso de querer definir más criterios de búsqueda, se pasarían uno tras otro en el array, separados por comas.

- El método *findBy* localiza todos los objetos que cumplan los criterios de búsqueda pasados como parámetro. Esta instrucción es como la anterior, pero devuelve un array de contactos con todos los resultados coincidentes (en el caso de que haya un solo resultado, devuelve un array de un elemento):

```
$contactos = $repositorio->findBy(["telefono" => "900110011"]);
```

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

```
$contactos = $repositorio->findAll();
```

Todos estos métodos se obtienen a partir de un repositorio de la clase, que viene a ser algo así como un asistente que nos ayuda a obtener objetos que pertenezcan a esa clase.

Veamos un ejemplo con nuestra clase *ContactoController*: vamos a modificar nuestro método *ficha* para que, en lugar de buscar en la base de datos de prueba que hemos venido empleando en sesiones anteriores, busque por *id* en la base de datos real. Para

ello, obtenemos el repositorio de nuestra clase *Contacto* y buscamos (*find*) el contacto con el *id* que hemos recibido como parámetro:

```
/**
 * @Route("/contacto/{codigo}", name="ficha_contacto", requirements={"codigo"="\d+"})
 */
public function ficha($codigo)
{
    $repositorio =
        $this->getDoctrine()->getRepository(Contacto::class);
    $contacto = $repositorio->find($codigo);

    if ($contacto)
        return $this->render('ficha_contacto.html.twig',
            array('contacto' => $contacto));
    else
        return $this->render('ficha_contacto.html.twig',
            array('contacto' => NULL));
}
```

En este punto, puedes realizar el [Ejercicio 2](#) de los propuestos al final de la sesión.

### 3.2.1. Consultas más avanzadas

Con los métodos de consulta anteriores podemos realizar consultas que se limitan a comprobar si uno o varios campos de un objeto son iguales a unos criterios de búsqueda determinados. Pero, ¿cómo podríamos, por ejemplo, buscar los contactos cuyo nombre contenga un cierto texto, o los libros de más de 100 páginas? Para este tipo de consultas, necesitamos ampliar el repositorio de nuestra entidad.

Por ejemplo, para nuestra entidad *Contacto*, imaginemos que queremos buscar los contactos cuyo nombre concierne un cierto texto. Para conseguir esto, necesitamos editar el repositorio de la entidad, que está en *src/Repository/ContactoRepository.php*. Este archivo contiene comentados un par de métodos de prueba que podríamos definir para ampliar las capacidades de la entidad.

En nuestro caso, vamos a añadir un método que se encargará de obtener los contactos cuyo nombre contenga un texto determinado que le pasemos como parámetro:

```
public function findByName($text): array
{
    $qb = $this->createQueryBuilder('c')
        ->andWhere('c.nombre LIKE :text')
        ->setParameter('text', '%' . $text . '%')
        ->getQuery();

    return $qb->execute();
}
```

Empleamos el *query builder* de Doctrine para construir la consulta con esa sintaxis específica. En primer lugar, definimos un elemento que hemos llamado *c* (de *Contacto*) que usaremos para referenciar las propiedades de los contactos, por ejemplo, en la

cláusula *where*. Lo que viene a hacer este código es buscar aquellos contactos *c* cuyo nombre sea como el parámetro *text*, y a continuación especifica que dicho parámetro *text* es igual al parámetro que recibimos en el método, encerrado entre símbolos '%', para indicar que da igual lo que haya delante o detrás del texto.

Ahora, ya podríamos utilizar este método desde donde lo necesitemos. Por ejemplo, podemos modificar el método *buscar* de *ContactoController* para que busque contactos por nombre empleando este nuevo método:

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

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

Si, por ejemplo, quisiéramos buscar por una propiedad numérica (por ejemplo, personas cuya edad sea mayor que una dada), usaríamos una sintaxis como esta (también muy similar a SQL):

```
$qb = $this->createQueryBuilder('p')
->andWhere('p.edad > :edad')
->setParameter('edad', $edad)
->getQuery();
```

Alternativamente, también podemos emplear un lenguaje llamado DQL (*Doctrine Query Language*) para realizar la consulta anterior:

```
$entityManager = $this->getEntityManager();
$query = $entityManager->createQuery(
    'SELECT c FROM App\Entity>Contacto c WHERE c.nombre LIKE :text'
)->setParameter('text', '%' . $text . '%');

return $query->execute();
```

Y, como tercera vía, también podemos emplear SQL estándar, pero en este caso lo que obtendríamos ya no sería un array de objetos, sino un array de registros, como si empleáramos la librería *mysqli* de PHP para acceder a la base de datos.

Aquí tenéis enlaces para consultar información adicional tanto de [Query Builder](#) como del lenguaje [DQL](#).

En este punto, puedes realizar el [Ejercicio 3](#) del final de la sesión, de carácter optativo.

## 3.3. Actualizar y borrar objetos

---

### 3.3.1. Actualizar objetos

Para actualizar un objeto en una base de datos, debemos seguir tres pasos:

- Obtener el objeto de la base de datos (típicamente haciendo un *find* por su clave primaria)
- Modificar los datos necesarios con los respectivos *setters* del objeto
- Hacer un *flush* para actualizar los cambios en la base de datos.

Si, por ejemplo, quisiéramos actualizar los datos del contacto con *id* = 1, haríamos esto:

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

if ($contacto)
{
    $contacto->setNombre("Contacto modificado");
    $entityManager->flush();
}
```

### 3.3.2. Borrar objetos

El borrado de objetos es similar a la actualización: debemos obtener el objeto también, 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. Debemos tenerlo en cuenta para capturarla y generar la respuesta oportuna.

## 4. Relaciones entre entidades

---

Hasta ahora las operaciones que hemos hecho se han centrado en una única tabla o entidad (la entidad/tabla de contactos en el ejemplo de los apuntes, y la entidad/tabla de libros en los ejercicios propuestos). Veamos ahora cómo podemos trabajar con más de una tabla/entidad que estén relacionadas entre sí.

Existen dos tipos principales de relaciones entre entidades:

- *Muchos a uno*: 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.

Vamos a definir una relación *muchos a uno* en nuestra base de datos de contactos. Para ello, vamos a crear primero una entidad llamada *Provincia*, que sólo contenga 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, creamos la correspondiente tabla en la base de datos a través de la migración.

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

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

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 relación *muchos a uno* (un contacto pertenecerá a 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
ManyToMany	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!

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 indicamos que no para simplificar el código.

Tras estos cambios, realizamos de nuevo la migración, y ya tendremos el nuevo campo añadido en nuestra entidad *Contacto* y a la tabla *contacto* de la base de datos:

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

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

## 4.1. Trabajar con entidades relacionadas

Ahora que ya sabemos relacionar entidades entre sí, ¿cómo podemos insertar una entidad que depende de otra, o acceder a los datos de una entidad desde la otra?

### 4.1.1. Inserción de entidades relacionadas

Por ejemplo, si quisiéramos insertar un contacto asignándole una provincia:

- Si la provincia no existe, creamos un objeto de tipo *Provincia*, y después otro de tipo *Contacto*, estableciendo como provincia el objeto *Provincia* recién creado:

```
$entityManager = $this->getDoctrine()->getManager();

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

$contacto = new Contacto();
$contacto->setNombre("Inserción de prueba con provincia");
$contacto->setTelefono("900220022");
$contacto->setEmail("insercion.de.prueba.provincia@contacto.es");
$contacto->setProvincia($provincia);
```

```
$entityManager->persist($provincia);
$entityManager->persist($contacto);
$entityManager->flush();
```

- Si la provincia sí existe, la buscamos en la base de datos (con algún método *find* o similar) y después creamos el objeto *Contacto* y le asignamos ese objeto *Provincia*:

```
$entityManager = $this->getDoctrine()->getManager();
$repositorio = $this->getDoctrine()->getRepository(Provincia::class);
```

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

```
$contacto = new Contacto();
$contacto->setNombre("Inserción de prueba con provincia");
$contacto->setTelefono("900220022");
$contacto->setEmail("insercion.de.prueba.provincia@contacto.es");
$contacto->setProvincia($provincia);
```

```
$entityManager->persist($provincia);
$entityManager->persist($contacto);
$entityManager->flush();
```

### 4.1.2. Búsqueda de entidades relacionadas

En el caso de que hagamos una búsqueda de una entidad que está relacionada con otra, el acceso a esa otra entidad es inmediato desde la primera. Por ejemplo, si quisiéramos saber el nombre de la provincia del contacto con código 1, haríamos algo así:

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

En cualquier caso, es importante recalcar que Doctrine no recupera los datos de la entidad relacionada (la provincia, en este caso), hasta que se piden efectivamente (es

decir, hasta que no preguntamos por el nombre de la provincia, Doctrine no trata de obtener el objeto *Provincia* completo).

En este punto, puedes realizar el [Ejercicio 4](#) de los propuestos al final de la sesión.

## 5. Ejercicios

---

### 5.1. Ejercicio 1

---

En la aplicación de libros, crea una nueva entidad llamada *Libro* con los siguientes atributos:

- *isbn* (*string* de tamaño 20). Este atributo deberá ser establecido como clave primaria
- *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` en las variables de entorno, crear la base de datos, y luego migrar estos cambios a la base de datos, empleando los comandos vistos en esta sesión.

### 5.2. Ejercicio 2

---

Sobre la base de datos *libros* y la entidad/tabla *Libro* que hemos creado en el ejercicio anterior, vamos a hacer un par de operaciones, a través de la clase *LibroController* que ya tenemos definida:

- En primer lugar, vamos a añadir un nuevo controlador llamado *insertar*, que responderá a la ruta `/libro/insertar`, e insertará un libro de prueba con estos datos:
  - *isbn* = "1111AAAA"
  - *titulo* = "Libro de prueba"
  - *autor* = "Autor de prueba"
  - *paginas* = 100

Comprueba que, si accedes por primera vez a la URL, el libro se inserta en la base de datos, pero el resto de veces ya no, al estar duplicada la clave primaria (el ISBN). En este caso, deberás capturar la excepción y generar una respuesta apropiada (basta con un *Response* con un mensaje de error).

- Ahora, vamos a modificar el controlador *ficha* para que, en lugar de buscar en la base de datos de prueba, busque por ISBN en la tabla *libro* de la base de datos MySQL. Prueba a cargar la URL `symfony.libros/1111AAAA` para obtener la ficha del libro de prueba insertado antes.

Deberás colocar estos dos controladores en el orden adecuado para que no se solapen.

- Modifica también el controlador *inicio* en la clase *InicioController* para que muestre el listado de la tabla *libro* de MySQL en lugar de la base de datos de pruebas.

### 5.3. Ejercicio 3 (opcional)

---

Añade un nuevo controlador a la clase *LibroController* que se llame *filtrarPaginas* y busque y muestre los libros que no tengan más del número de páginas que se reciba como parámetro. Irá asociado a la URI */libro/paginas/{paginas}*, y renderizará una vista (puede ser la página principal), pasándole como parámetro el listado filtrado.

Emplea para hacer la consulta o bien el *Query Builder* de Doctrine o bien su lenguaje DQL, añadiendo el método correspondiente en el repositorio de la entidad *Libro*.

### 5.4. Ejercicio 4

---

Crea una nueva entidad llamada *Editorial* en la aplicación de libros. Como único campo, tendrá un nombre (*string* de tamaño 255), además del *id* autogenerado. 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 deja como **opcional** el comprobar que ya exista una editorial con ese nombre antes de insertar la nueva. Si existe, se obtendrá dicha editorial y se asignará al libro. Si no existe, se creará y se asignará. Para probar que funciona, deberás modificar al menos el ISBN del nuevo libro a insertar cada vez.