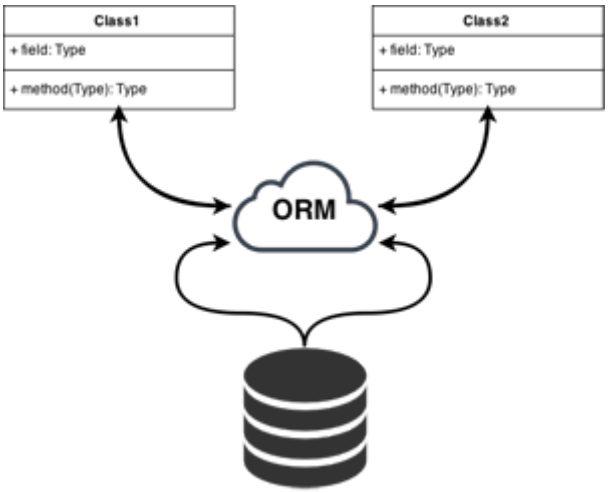
# Unidad 8 - Entidades

## Concepto de ORM

Un ORM (Object-Relational Mapping) es un modelo de programación que permite mapear las estructuras de una base de datos relacional (SQL Server, Oracle, MySQL, etc.), sobre una estructura lógica de objetos (llamados entidades), para simplificar y acelerar el desarrollo de nuestras aplicaciones.




Las estructuras de la base de datos relacional quedan vinculadas con las entidades lógicas o base de datos virtual definida en el ORM, de tal modo que las acciones CRUD (Create, Read, Update, Delete) a ejecutar sobre la base de datos física se realizan de forma indirecta por medio del ORM.

La consecuencia más directa es que, además de “mapear”, los ORMs tienden a “liberarnos” de la escritura o generación manual de código SQL (Structured Query Language) necesario para realizar las queries o consultas y gestionar la persistencia de datos en el RDBMS.

Así, los objetos o entidades de la base de datos virtual creada en nuestro ORM podrán ser manipulados por medio de algún lenguaje de nuestro interés según el tipo de ORM utilizado. La interacción con el RDBMS quedará delegada en los métodos de actualización correspondientes proporcionados por el ORM. Los ORMs más completos ofrecen servicios para persistir todos los cambios en los estados de las entidades, previo seguimiento o tracking automático, sin escribir una sola línea de SQL.

Symfony proporciona todas las herramientas necesarias para usar bases de datos en las aplicaciones gracias a Doctrine, un conjunto de bibliotecas PHP para trabajar con bases de datos. Estas herramientas admiten bases de datos relacionales como MySQL y PostgreSQL y también bases de datos NoSQL como MongoDB.

### Configuración de Doctrine

La información de conexión de la base de datos se almacena como una variable de entorno denominada DATABASE_URL. Para el desarrollo, puede encontrar y personalizar esta variable dentro del archivo .env. Para configurar nuestra base en mysql debemos definir:

```
DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=5.7"
```

Donde:

- db_user: Es el usuario de la base de datos que se utilizará
- db_password: Es la clave del usuario de la base de datos
- 127.0.0.1: Es el nombre del servidor de base de datos, si trabajamos en forma local, no se debe cambiar
- db_name: Es el nombre de la base de datos
- serverVersion: Es la versión de msyql que utilizaremos

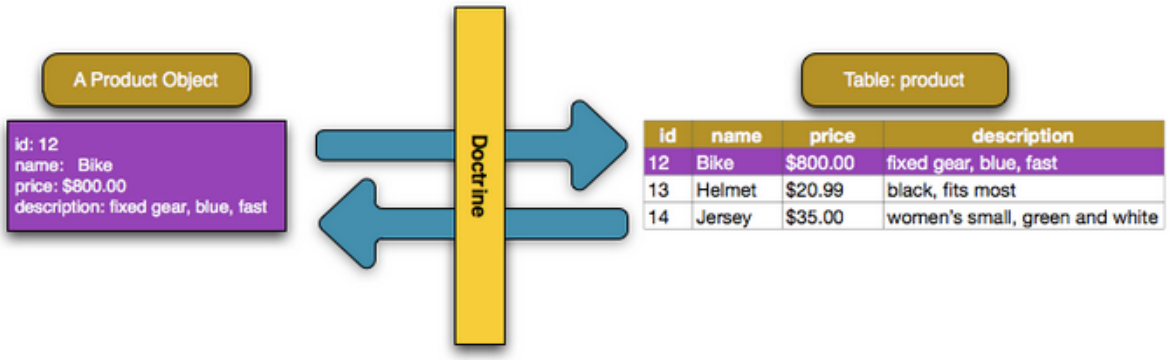
Una vez que se han establecido los parámetros de conexión, Doctrine puede crear la base de datos por nosotros:

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

## Entidades

Se denomina entidades a los objetos PHP utilizados para manipular la información de la base de datos. Generalmente cada tabla de la base de datos se representa mediante una entidad. No obstante, en ocasiones Doctrine crea tablas adicionales para representar la relación entre dos entidades.

Las clases que utiliza Symfony son clases PHP que no sólo no utilizan un formato especial para su nombre sino que ni siquiera heredan de ninguna otra clase. Por eso estas clases se pueden crear a mano, aunque Symfony incluye comandos para generarlas automáticamente.




### Creando las entidades

Suponga que está creando una aplicación en la que se deben mostrar los productos. Sin siquiera pensar en Doctrine o bases de datos, ya sabemos que necesitaremos un objeto Producto para representar esos productos.

Podemos usar el comando make:entity para crear esta clase y cualquier campo que necesite. El comando le hará algunas preguntas, como se muestra a continuación:

```
$ php bin/console make:entity

Class name of the entity to create or update:
> Product

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

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

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

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

Can this field be null in the database (nullable) (yes/no) [no]:
> no

New property name (press <return> to stop adding fields):
>
(press enter again to finish)
```

Al terminar se genera el archivo src/Entity/Product.php

```php
// src/Entity/Product.php
namespace App\Entity;

use App\Repository\ProductRepository;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass=ProductRepository::class)
 */
class Product
{
    /**
     * @ORM\Id()
     * @ORM\GeneratedValue()
     * @ORM\Column(type="integer")
     */
    private $id;

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

    /**
     * @ORM\Column(type="integer")
     */
    private $price;

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

    // ... getter and setter methods
}
```

Es necesario tener en cuenta que los atributos de la clase deben ser privados y es necesario definir los getters y los setters correspondientes para poder acceder a ellos.

Doctrine soporta una gran variedad de tipos de campos. Los más comunes son:

- string
- text
- boolean
- integer
- smallint
- bigint
- float
- date
- time
- datetime

## Migrations

La clase Producto está completamente configurada y lista para guardarse en una tabla de productos, pero en la base de datos, todavía no está definida la tabla productos. Existen herramientas en Symfony que nos permite llevar el modelo de clases al modelo de tablas. Esto se hace con las migraciones. Como primera medida debemos ejecutar el comando:

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

Este comando genera un archivo php en la carpeta migrations con todas las instrucciones sql para poder actualizar la base de datos. Luego de creado el archivo php corremos el comando:

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

Este comando ejecuta todos los archivos de migración que aún no se han ejecutado en su base de datos. Siguiendo con nuestro ejemplo, si consultamos la base de datos vamos a ver que se ha creado la tabla product.

## Modificación de las entidades

Es posible que necesite agregar o modificar atributos en las entidades. Por ejemplo vamos a agregar el campo descripción a nuestra entidad. Para eso volvemos a ejecutar el comando make:entity y continuamos agregando atributos:

```
$ php bin/console make:entity

Class name of the entity to create or update
> Product

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

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

Can this field be null in the database (nullable) (yes/no) [no]:
> no

New property name (press <return> to stop adding fields):
>
(press enter again to finish)
```

Vemos que se ha agregado el atributo description a nuestra entidad php.

```
// src/Entity/Product.php
  // ...

  class Product
  {
      // ...

     /**
      * @ORM\Column(type="text")
      */
     private $description;

     // getDescription() & setDescription() were also added
  }
```

La nueva propiedad está en nuestra entidad, pero todavía no está en nuestra tabla. Para eso generamos una nueva migración y obtenemos un nuevo archivo php con la sentencia sql correspondiente para agregar el campo:

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

Luego impactamos la migración sobre la base de datos:

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

Esto solo ejecutará el nuevo archivo de migración, porque Symfony sabe que la primera migración ya se ejecutó antes. Detrás de escena, administra una tabla migration_versions para realizar un seguimiento de esto. Cada vez que se realice un cambio en las entidades, se deben ejecutar estos dos comandos para generar la migración y luego ejecutarla.

## Persistir objetos en la base de datos

Vamos a ver ahora cómo trabajar con los objetos que hemos definido y cómo se impactan los cambios en la base de datos.

### Agregando un objeto

Vamos a trabajar con un ejemplo para ver cómo agregar un producto a la base de datos. Creamos un controlador que al ser invocado agrega el producto en la tabla.

```php
// src/Controller/ProductController.php
namespace App\Controller;

// ...
use App\Entity\Product;
use Doctrine\Persistence\ManagerRegistry;
use Symfony\Component\HttpFoundation\Response;

class ProductController extends AbstractController
{
    /**
      * @Route("/product", name="create_product")
     */
    public function createProduct(ManagerRegistry $doctrine): Response
    {
        $entityManager = $doctrine->getManager();

        $product = new Product();
        $product->setName('Keyboard');
        $product->setPrice(1999);
        $product->setDescription('Ergonomic and stylish!');

        // tell Doctrine you want to save the Product (no queries yet)
        $entityManager->persist($product);

        // actually executes the queries (i.e. the INSERT query)
        $entityManager->flush();

        return new Response('Saved new product with id '.$product->getId());
    }
}
```

Veamos este ejemplo con mayor detalle:

- **línea 14:** El argumento ManagerRegistry $doctrine le dice a Symfony que inyecte el servicio Doctrine en el método del controlador.
- **línea 16:** El método $doctrine->getManager() obtiene el objeto manager de Doctrine, que es el objeto más importante de Doctrine. Es responsable de guardar y recuperar objetos de la base de datos.
- **líneas 18-21:** En esta sección, se crea una instancia y se trabaja con el objeto $product como cualquier otro objeto PHP normal.
- **línea 24:** La invocación a persist($product) le dice a Doctrine que "administre" el objeto $product. Esto no provoca que se realice una consulta a la base de datos.
- **línea 27:** Cuando se llama al método flush(), Doctrine revisa todos los objetos que está administrando para ver si se necesitan persistir en la base de datos. En este ejemplo, los datos del objeto $product no existen en la base de datos, por lo que el administrador de entidades ejecuta una consulta INSERT, creando una nueva fila en la tabla de product.

## Recuperando objetos

Recuperar un objeto de la base de datos es aún más fácil. Supongamos que queremos ver los datos del producto 1.

```
// src/Controller/ProductController.php
namespace App\Controller;

use App\Entity\Product;
use Symfony\Component\HttpFoundation\Response;
// ...

class ProductController extends AbstractController
{
    /**
     * @Route("/product/{id}", name="product_show")
     */
    public function show(ManagerRegistry $doctrine, int $id): Response
    {
        $product = $doctrine->getRepository(Product::class)->find($id);

        if (!$product) {
            throw $this->createNotFoundException(
                'No product found for id '.$id
            );
        }

        return new Response('Check product: '.$product->getName());

        // or render a template
        // in the template, print things with {{ product.name }}
        // return $this->render('show.html.twig', ['product' => $product]);
    }
}
```

Cuando se solicita un tipo particular de objeto, se emplea lo que es conocido como repositorio. Un repositorio es una clase PHP cuyo único trabajo es ayudar a extraer entidades de una clase determinada. Una vez que tenemos el repositorio, tenemos acceso a varios métodos:

```
$repository = $doctrine->getRepository(Product::class);

// look for a single Product by its primary key (usually "id")
$product = $repository->find($id);

// look for a single Product by name
$product = $repository->findOneBy(['name' => 'Keyboard']);
// or find by name and price
$product = $repository->findOneBy([
    'name' => 'Keyboard',
    'price' => 1999,
]);

// look for multiple Product objects matching the name, ordered by price
$products = $repository->findBy(
    ['name' => 'Keyboard'],
    ['price' => 'ASC']
);

// look for *all* Product objects
$products = $repository->findAll();
```

El repositorio es el lugar donde se agrupan las consultas relativas a una entidad, la mayoría relacionadas con la recuperación de datos (**SELECT**). El repositorio puede contener operaciones de creación, modificación o eliminación.

Cada entidad tiene su propio repository. Se puede materializar por una clase definida por el desarrollador, si se indica la propiedad **repositoryClass** de la anotación **@Entity** o si es necesario, por una clase interna de Doctrine. Dentro del repositorio podemos crear métodos personalizados para la manipulación del objeto.

Cuando se generó la entidad Product con make:entity, el comando también generó una clase ProductRepository.

## Actualizando objetos

Una vez obtenido el objeto de Doctrine, se puede interactuar con él de la misma manera que con cualquier objeto de PHP:

```
// src/Controller/ProductController.php
namespace App\Controller;

use App\Entity\Product;
use App\Repository\ProductRepository;
use Symfony\Component\HttpFoundation\Response;
// ...

class ProductController extends AbstractController
{
    /**
     * @Route("/product/edit/{id}")
     */
    public function update(ManagerRegistry $doctrine, int $id): Response
    {
        $entityManager = $doctrine->getManager();
        $product = $entityManager->getRepository(Product::class)->find($id);

        if (!$product) {
            throw $this->createNotFoundException(
                'No product found for id '.$id
            );
        }

        $product->setName('New product name!');
        $entityManager->flush();

        return $this->redirectToRoute('product_show', [
            'id' => $product->getId()
        ]);
    }
}
```

El uso de Doctrine para editar un producto existente consta de tres pasos:

1. obtener el objeto de Doctrine;
2. modificar el objeto;
3. llamar a flush().

Se puede llamar a $entityManager->persist($product), pero no es necesario: Doctrine ya está "observando" el objeto en busca de cambios.

## Borrando objetos

Eliminar un objeto es muy similar, pero requiere una llamada al método remove() del entityManager:

```
$entityManager->remove($product);
$entityManager->flush();
```

El método remove() notifica a Doctrine que te gustaría eliminar el objeto dado de la base de datos. La consulta DELETE no se ejecuta realmente hasta que se llama al método flush().

## Cargar datos de prueba

Symfony nos suministra unas fixtures que se pueden utilizar dentro del desarrollo de nuestras aplicaciones para cargar una serie de datos de prueba y, cuando generemos un nuevo entorno o si compartimos nuestro proyecto con otros usuarios o compañeros, estos puedan empezar ya a trabajar con una serie de datos por defecto con los que probar las operaciones o la maquetación del sitio.

Lo primero que debemos hacer es instalar el bundle:

```
composer require --dev orm-fixtures
```

### Crear los datos de prueba

Ahora vamos a crear los datos de prueba (Por ej. para la entidad Product). Para ello ejecutamos el comando:

```
php bin/console make:fixtures ProductFixtures
```

Este comando creará el esqueleto del script para cargar los datos de prueba. En la carpeta DataFixture se creará el archivo ProductFixture.php

```
<?php

namespace App\DataFixtures;

use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Persistence\ObjectManager;

class ProductFixtures extends Fixture
{
    public function load(ObjectManager $manager): void
    {
        // $product = new Product();
        // $manager->persist($product);

        $manager->flush();
    }
}
```

Dentro del método load introducimos la lógica de creación de los datos:

```
// src/DataFixtures/AppFixtures.php
namespace App\DataFixtures;

use App\Entity\Product;
use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Persistence\ObjectManager;

class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        // create 20 products! Bam!
        for ($i = 0; $i < 20; $i++) {
            $product = new Product();
            $product->setName('product '.$i);
            $product->setPrice(mt_rand(10, 100));
            $manager->persist($product);
        }

        $manager->flush();
    }
}
```

## Cargar los datos de prueba

Una vez escrito el fixture lo cargamos a través del siguiente comando:

```
php bin/console doctrine:fixtures:load
```

De forma predeterminada, el comando de carga purga la base de datos y elimina todos los datos de cada tabla. Para agregar los datos del fixture, se debe agregar la opción --append.

## Cargar los datos en orden

Muchas veces es necesario definir un orden de carga, ya que es necesario tener cargados unos datos antes que otros, de este modo podemos mantener la integridad referencial. Por ejemplo, si tenemos una entidad usuario y otra grupo, donde la entidad grupo agrupa a varios usuarios, es necesario cargar primero los usuarios y luego a la entidad grupo indicarle los usuarios que pertenecen al mismo.

Para realizar esto en el fixture de grupo debemos implementar la interfaz DependentFixtureInterface y agregar el método getDependencies(). Esto devolverá una matriz de las clases fixtures que deben cargarse antes de esta:

```
// src/DataFixtures/UserFixtures.php
namespace App\DataFixtures;

// ...
class UserFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        // ...
    }
}

// src/DataFixtures/GroupFixtures.php
namespace App\DataFixtures;
// ...
use App\DataFixtures\UserFixtures;
use Doctrine\Common\DataFixtures\DependentFixtureInterface;

class GroupFixtures extends Fixture implements DependentFixtureInterface
{
    public function load(ObjectManager $manager)
    {
        // ...
    }

    public function getDependencies()
    {
        return [
            UserFixtures::class,
        ];
    }
}
```