Doctrine con

Symfony

Doctrine ORM

Symfony

¿Qué es?

Como ya se vio en el tema de Doctrine, es un ORM y su función es convertir datos a un sistema de tipos de un lenguaje de programación orientado a objetos y una base de datos relacional.

- Las clases → son las tablas de la base de datos.
- Los objetos → son los registros de la base de datos.

ORM de Symfony

Instalación Doctrine en Symfony

Para instalarlo tenemos que implementar el paquete de orm de Symfony mediante composer.

composer require symfony/orm-pack

Y si aún no has instalado el MakerBundle también es recomendable para poder generar de forma automática el código.

composer require –dev symfony/maker-bundle

https://symfony.com/doc/current/doctrine.html#installing-doctrine

ORM de Symfony

Configurar acceso a la BB.DD.

La configuración de acceso a la BB.DD. la realizaremos mediante el fichero *.env* de configuración mediante la línea *DATABASE_URL*

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

- *db_user → usuario de acceso a la BB.DD.*
- *db_password → contraseña de acceso a la BB.DD.*
- *127.0.0.1:3306 → dirección de acceso y puerto.*
- *db_name → nombre de la BB.DD.*

ORM de Symfony

Configurar acceso a la BB.DD. II

Según el tipo de BB.DD. Utilizaremos una u otra sintanxis

 # to use mysql:
DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=5.7“

 # to use mariadb:
DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=mariadb-10.5.8"

 # to use sqlite:
DATABASE_URL="sqlite:///%kernel.project_dir%/var/app.db"

 # to use postgresql:
DATABASE_URL="postgresql://db_user:db_password@127.0.0.1:5432/db_name?serverVersion=11&charset=utf8"

 # to use oracle:
DATABASE_URL="oci8://db_user:db_password@127.0.0.1:1521/db_name“

https://symfony.com/doc/current/doctrine.html#configuring-the-database

ORM de Symfony

Modelado y traslado a la BB.DD.

En el tema de Doctrine vimos como crear entidades que existían ya en una BB.DD. relacional y como usarlas.

Ahora podemos mediante el uso de los comandos predefinidos para Doctrine generar una BB.DD., entidades o más acciones directamente en Doctrine y que luego se migren a la BB.DD.

Para ver todos los comandos de los que disponemos puede usar el comando:

php bin/console list doctrine

Por ejemplo crear una nueva BB.DD:

php bin/console doctrine:database:create

ORM de Symfony

Crear Entidades

Para crear una nueva entidad:

php bin/console make:entity

Sin necesidad de evaluar como va a ser la BB.DD. Sabemos que vamos a necesitar un objeto con el nombre de la entidad. Si vamos a mostrar muchos productos en nuestra aplicación necesitaremos un objeto: Product.

Tras indicar el nombre de la entidad el mismo comando se encargará de ir pidiéndonos los datos para definir cada campo:

- Nombre
- Tipo
- Longitud si corresponde
- ¿Nulo?

ORM de Symfony

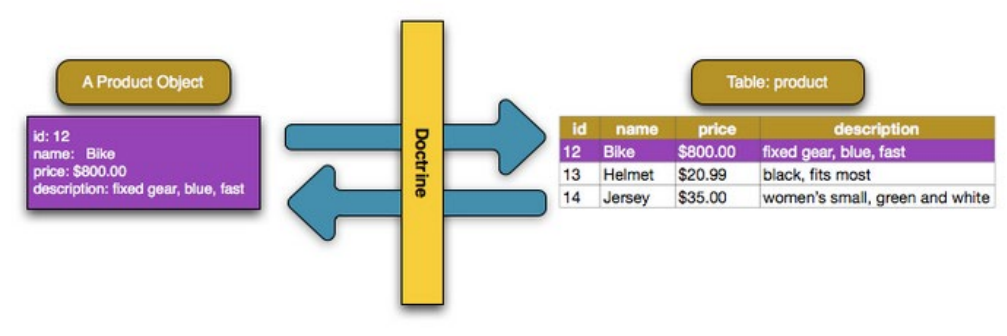
Crear Entidades II

Las entidades que creemos se habrán creado en la carpeta → *src/Entity*

Las entidades creadas usarán atributos de PHP únicamente a partir de la versión de doctrine 1.44.0

Podemos agregar más campos a una entidad volviendo a ejecutar el comando *make:entity*

O como tenemos la completa libertad de modificar el código según consideremos que sea necesario y nos sea más útil.



ORM de Symfony

Crear Entidades III

Si decidimos modificar el código de la entidad agregando nuevas propiedades, podemos volver a generar los métodos getter y setter de forma automática con el comando:

php bin/console make:entity - -regenerate

Si queremos que regenere todos los métodos getter y setter debemos usar también:

php bin/console make:entity - -overwrite

https://symfony.com/doc/current/doctrine.html#creating-an-entity-class

ORM de Symfony

Migrar desde Doctrine a la BB.DD.

Una vez configuradas completamente las entidades desde el lado de Doctrine tendremos que guardarlo en la tabla correspondiente en la base de datos, porque aún no existirá en ella.

Para ello, vamos a usar DoctrineMigrationsBundle que ya tenemos instalado.

De forma que ahora ejecutamos el comando:

php bin/console make:migration

Que nos informará si todo ha salido correctamente y veremos que ha creado un archivo PHP dentro de la carpeta migrations.

ORM de Symfony

Migrar desde Doctrine a la BB.DD. II

El archivo generado tras la migración contiene los comandos SQL necesarios para actualizar la BB.DD.

Ahora, nos quedará únicamente terminar la migración mediante el comando:

php bin/console doctrine:migrations:migrate

Que se encarga de ejecutar el archivo que hemos generado anteriormente.

https://symfony.com/doc/current/doctrine.html#migrations-creating-the-database-tables-schema

ORM de Symfony

Migrar desde Doctrine a la BB.DD. III

Si tenemos que volver a modificar la entidad tras migrar.

Simplemente podemos volver a hacer *make:entity* o modificar el código de la entidad y posteriormente lanzar los comandos de la migración para volver a sincronizar nuestra aplicación con la BB.DD.

php bin/console make:migration

php bin/console doctrine:migrations:migrate

https://symfony.com/doc/current/doctrine.html#migrations-adding-more-fields

ORM de Symfony

Autowiring con EntityValueResolver

A partir de la versión 6.2 de symfony se introdujo el Entity Value Resolver que nos permite hacer autowiring automático, en Doctrine se introdujo a partir de la versión 2.7.1.

Será el EntityValueResolver el que se encargará de generar la consulta que necesitemos automáticamente por nosotros y el controlador se simplifica notablemente.

https://symfony.com/doc/current/doctrine.html#automatically-fetching-objects-entityvalueresolver

ORM de Symfony

Autowiring con EntityValueResolver II

Será el bundle el que se encarga mediante la etiqueta {id} de la ruta de crear la consulta de Product. Si no la encuentra generará una página 404.

Esta opción la tenemos habilitada por defecto para todos los controladores.

Si la deshabilitamos podemos habilitarla para un determinado controlador usando: *#[MapEntity]*

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

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

class ProductController extends AbstractController
{
    #[Route('/product/{id}')]
    public function show(Product $product): Response
    {
        // use the Product!
        // ...
    }
}
```

ORM de Symfony

Autowiring con EntityValueResolver III

El Autowiring funcionará si:

- Si *{id}* está en la ruta, buscando por la clave principal con el método *find()*
- O intentará realizar un método *findOneBy()* con todos los parámetros de la ruta que sean propiedades de la entidad.

Utilizando el atributo *MapEntity* y sus opciones podemos configurar de una forma más adecuada el autowiring.

https://symfony.com/doc/current/doctrine.html#mapentity-options

ORM de Symfony

Manejo de datos

El manejo de datos con doctrine será tal y como hemos visto en los temas anteriores de Doctrine.

- Leer un dato:

```
// from inside a controller
$repository = $doctrine->getRepository(Product::class);
$product = $repository->find($id);
```

https://symfony.com/doc/current/doctrine.html#querying-for-objects-the-repository

ORM de Symfony

Manejo de datos II

- Actualizar un objeto tras leerlo:

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

https://symfony.com/doc/current/doctrine.html#updating-an-object

ORM de Symfony

Manejo de datos III

- Eliminar un objeto:

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

https://symfony.com/doc/current/doctrine.html#deleting-an-object

ORM de Symfony

Query Builder

Doctrine nos permite generar consultas personalizadas orientadas a objetos mediante QueryBuilder.

https://www.doctrine-project.org/projects/doctrine-orm/en/current/reference/query-builder.html

Se recomienda usarlo cuando las consultas se generan dinámicamente, según las condiciones de PHP.

```php
<?php

use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
// src/Repository/ProductRepository.php

// ...
class ProductRepository extends ServiceEntityRepository
{
    public function findAllGreaterThanPrice(int $price, bool $includeUnavailableProducts = false): array
    {
        // automatically knows to select Products
        // the "p" is an alias you'll use in the rest of the query
        $qb = $this->createQueryBuilder('p')
            ->where('p.price > :price')
            ->setParameter('price', $price)
            ->orderBy('p.price', 'ASC');

        if (!$includeUnavailableProducts) {
            $qb->andWhere('p.available = TRUE');
        }

        $query = $qb->getQuery();

        return $query->execute();

        // to get just one result:
        // $product = $query->setMaxResults(1)->getOneOrNullResult();
    }
}
```

https://symfony.com/doc/current/doctrine.html#querying-with-the-query-builder

ORM de Symfony

Usando SQL

También podemos usar directamente sentencias SQL para obtener datos, pero debemos recordar que los datos que obtenemos no están procesados, no son objetos. Para ello, deberíamos usar NativeQuery.

```php
// src/Repository/ProductRepository.php

// ...
class ProductRepository extends ServiceEntityRepository
{
    public function findAllGreaterThanPrice(int $price): array
    {
        $conn = $this->getEntityManager()->getConnection();

        $sql = '
            SELECT * FROM product p
            WHERE p.price > :price
            ORDER BY p.price ASC
            ';
        $stmt = $conn->prepare($sql);
        $resultSet = $stmt->executeQuery(['price' => $price]);

        // returns an array of arrays (i.e. a raw data set)
        return $resultSet->fetchAllAssociative();
    }
}
```

https://symfony.com/doc/current/doctrine.html#querying-with-sql

ORM de Symfony

Documentación

Disponemos de toda la documentación para trabajar con BB.DD. en la página de Symfony:

https://symfony.com/doc/current/doctrine.html