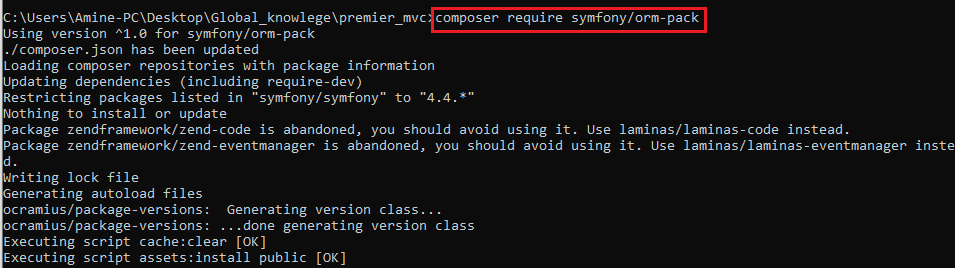
**Première Partie : l’ORM Doctrine**

Symfony dispose d’un outil puissant appelé ORM (Object Relational Mapper) pour la création de base de données ainsi que la création/mise à jour de schéma (tables) et la génération des opérations CRUD de façon automatique et plein d’autres chôses…

* **Installation du package doctine**

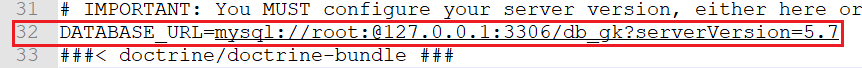
Commençons tout d’abord par installer doctirne via la commande :

composer require symfony/orm-pack



* **Configuration de la base de données**

Afin de configurer votre chaine de connexion, éditer le fichier **.env** se trouvant dans l’arborscence du projet et mettez à jour la chaine de connexion selon vos paramètres de connexion (user\_name = root, password = aucun, nom de la base de données = db\_gk) pour une connexion à mysql en local :

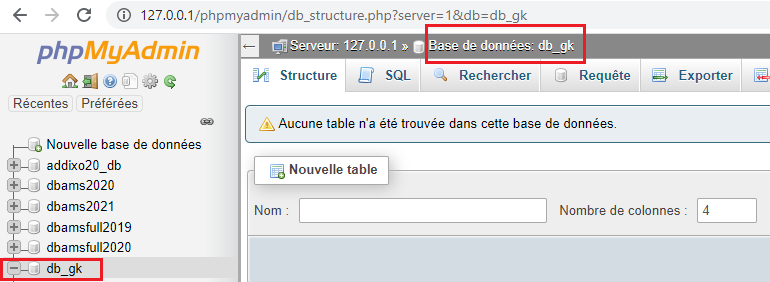


Lancer votre serveur de base de données puis exécuter a commande suivante :

php bin/console doctrine:database:create



Ouvrez votre phpMyAdmin et remarquer la présence de la base de données « **db\_gk »**



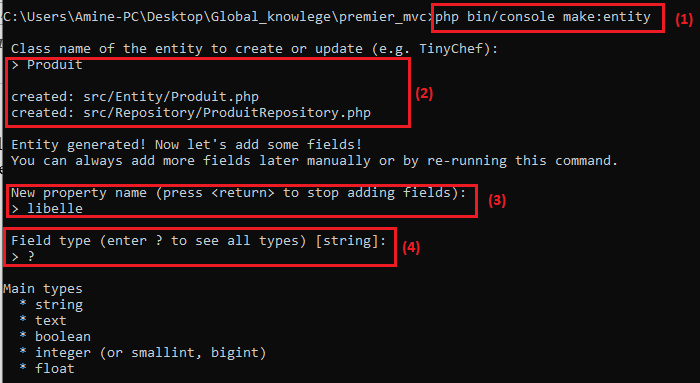
Pour découvrir toutes les commandes possibles avec doctrine tapez :

php bin/console list doctrine

Il est également possible d’apporter autres personnalisations sur la version de votre serveur MySQL dans le fichier « **config/package/doctrine.yaml »**

* **Création des entités via Doctrine**

Nous allons commencer par créer une première entité « **Produit** » contenant quelques champs (libelle, prix). Nous utilisons le bundle maker\_bundle pour la génération automatique du code.



Voici le contenu de notre première entité « Produit »

<?php

namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

/\*\*

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

 \*/

class Produit

{

    /\*\*

     \* @ORM\Id()

     \* @ORM\GeneratedValue()

     \* @ORM\Column(type="integer")

     \*/

    private $id;

    /\*\*

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

     \*/

    private $libelle;

    /\*\*

     \* @ORM\Column(type="float")

     \*/

    private $prix;

    public function getId(): ?int

    {

        return $this->id;

    }

    public function getLibelle(): ?string

    {

        return $this->libelle;

    }

    public function setLibelle(string $libelle): self

    {

        $this->libelle = $libelle;

        return $this;

    }

    public function getPrix(): ?float

    {

        return $this->prix;

    }

    public function setPrix(float $prix): self

    {

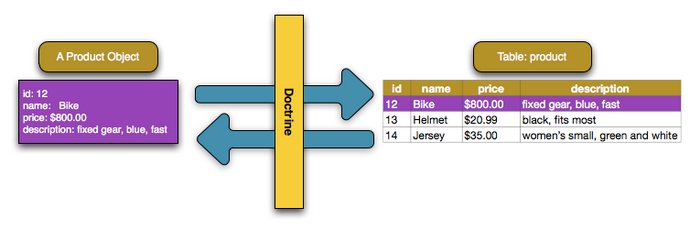
        $this->prix = $prix;

        return $this;

    }

}

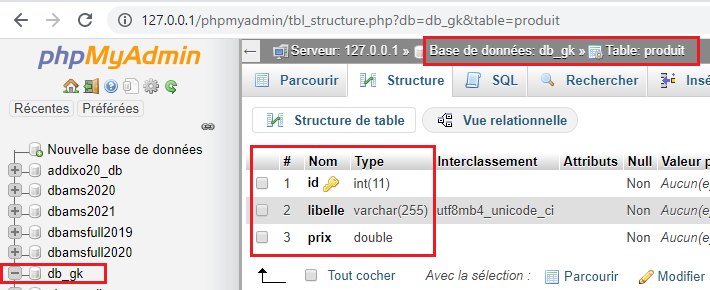
Doctrine va assurer le mapping entre Objet et table. Aussi bien entre attribut et colonne.



Afin de créer le schema (ici une seule table : Produit) dans la base de données, tapez les deux commandes suivantes :

php bin/console make:migration : Cette commande va générer du sql permettant la mise à jour du schéma de la base de données.

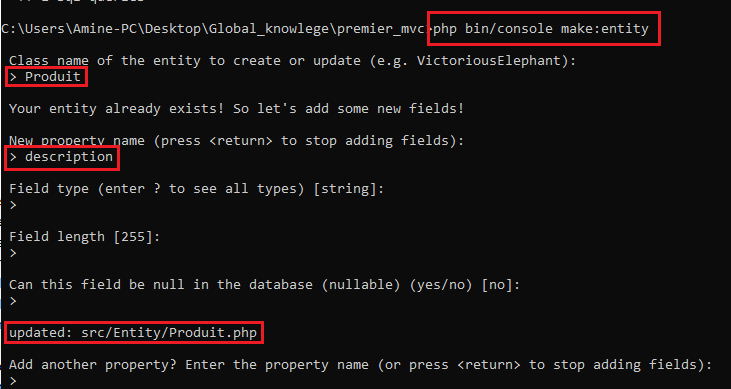
Ensuite tapez la commande suivante : php bin/console doctrine:migrations:migrate pour exécuter le code sql et apporter les modifs à votre BD. Une fois c’est fait consulter votre phpMyAdmin



Parfois, il vous arrive d’apporter des modifications à votre entité, par exemple l’ajout de nouveaux champs (description), pour ce faire, il y a deux façons :

1-Soit éditer l’entité manuellement et ne pas oublier les getters/setters

2-Soit via la commande make :entity

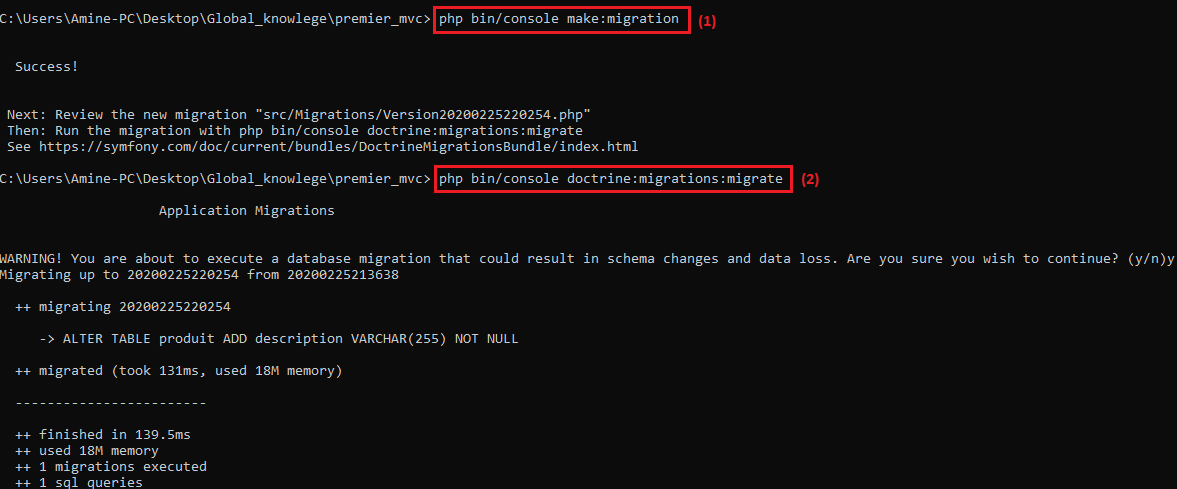


Cela va générer automatiquement l’ajout du champs description ainsi que les deux méthode getDescription() et setDescription(). On peut également remarquer aussi la présence d’un champs id repérsentant la clé primaire. Taper de nouveau la commade :

php bin/console make:migration

Maintenant exécuter ces modifs via la commande :

php bin/console doctrine:migrations:migrate



Si vous opter pour l’ajout des champs de façons manuelle, vous pouvez ajouter les getters/setters via la commande : php bin/console make:entity –regenerate

**Deuxième Partie : Opérations CRUD des entités**

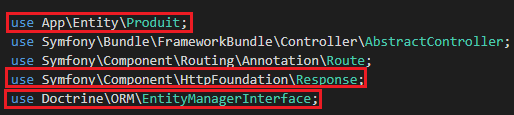
Pour commener, créons tout d’abord un controlleur dans lequel nous plaçons nos actions crud.

php bin/console make:controller ProductController

1. **L’opération d’ajout**

Ensuite ajoutons notre méthode « createProduit ()»

Ne pas oublier d’ajouter les imports nécéssaires



EntityManagerInterface : C’est une interface qui va nous permettre de créer notre entity Manger, Object responsable de l’interaction avec la base de données.

Le code source de notre méthode est comme suit :

<?php

namespace App\Controller;

use App\Entity\Produit;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

use Symfony\Component\Routing\Annotation\Route;

use Symfony\Component\HttpFoundation\Response;

use Doctrine\ORM\EntityManagerInterface;

class ProductController extends AbstractController

{

    /\*\*

     \* @Route("/listproduit", name="listproduit")

     \*/

    public function index()

    {

        return $this->render('product/index.html.twig', [

            'controller\_name' => 'ProductController',

        ]);

    }

    /\*\*

     \* @Route("/addproduit", name="addproduit")

     \*/

    public function createProduit(): Response

    {

        // you can fetch the EntityManager via $this->getDoctrine()

        // or you can add an argument to the action: createProduct(EntityManagerInterface $entityManager)

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

        $product = new Produit();

        $product->setLibelle('Smart Phone');

        $product->setPrix(350);

        $product->setDescription('un samsung J7!');

        // tell Doctrine you want to (eventually) save the Product (no queries yet)

        $entityManager->persist($product);

        // actually executes the queries (i.e. the INSERT query)

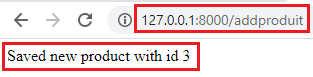
        $entityManager->flush();

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

    }

}

Lancer votre navigateur et taper la route corresponsdant à votre action, vous obtenez  :

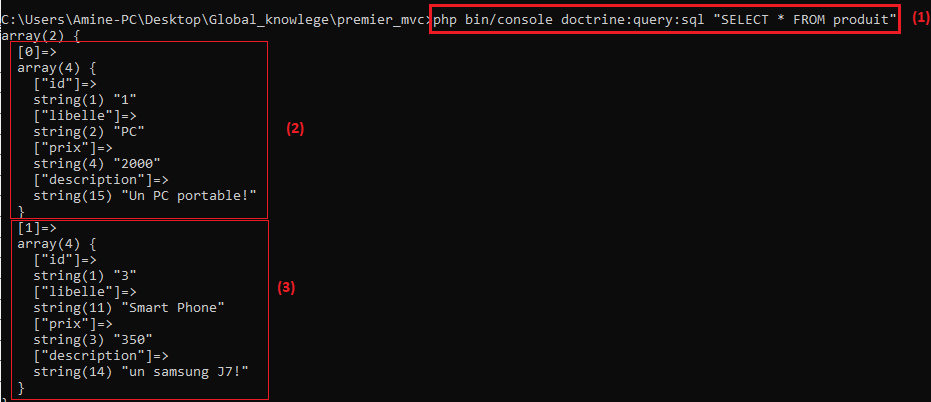


Félicitation, votre entityManger a enregistrer vos produits dans la table « Produit »



Lancer la commande :

php bin/console doctrine:query:sql "SELECT \* FROM produit"

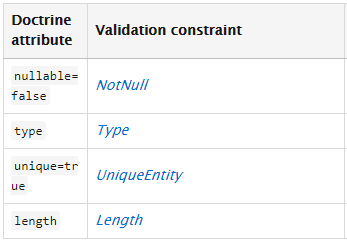


Pour l’ajout ou la modification d’un objet dans la base, les instructions sont les mêmes. Doctrine est capable à lui seul de distinger une opération d’ajout de celle de la mise à jour.

* Validation des objets avant persistance

Nous pouvons s’assurer que les objets que nous cherchons à persister(ajouter ou modifier) respectent certaines regles mentionnées via les annotations (les méta data).

Le tableaux ci-après présente un résumé des annotation et les contraintes associés.



Commencer par apporter les modifications suivantes :

1-Import du validator Inteface

use Symfony\Component\Validator\Validator\ValidatorInterface;

2-Injection de dépence

   public function createProduit(ValidatorInterface $validator): Response

3-Ajouter le code de validation

$errors = $validator->validate($produit);

        if (count($errors) > 0) {

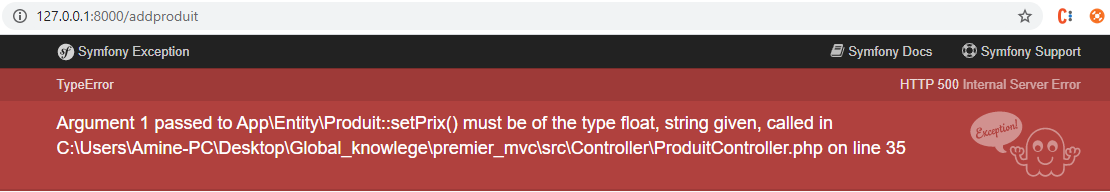
            return new Response((string) $errors, 400);

        }

4-Supposons qu’on va mettre « abc » à la place d’un prix valide :

$product->setPrix("abc");

5-Exécuter votre navigateur !



Nous passons maitenant à voir comment doctrine va nous facilter la tache de récupération des objest depuis la base de données

1. **L’opération de lecture (One or All)**

Pour récupérer un objet précis via son id, créons une nouvelle méthode « **show\_produit($id){…} »** et lui ajoutons le code suivant :

 /\*\*

    \* @Route("/show\_produit/{id}", name="show\_produit")

    \*/

    public function show\_produit($id)

    {

        $produit = $this->getDoctrine()

            ->getRepository(Produit::class)

            ->find($id);

        if (!$produit) {

            throw $this->createNotFoundException(

                'No product found for id '.$id

            );

        }

        return new Response('Check out this great product: '.$produit->getLibelle());

        // or render a template

        // in the template, print things with {{ product.name }}

        // return $this->render('product/show.html.twig', ['product' => $product]);

    }

Taper sur votre navigateur : <http://127.0.0.1:8000/show_produit/4>

Vous aurez le libelle du produit recherché.

Maitenant passons à la recherche et l’affichage de touts les produits qui existent dans la table produit

Ajoutons la méthode suivante dans notre controlleur :

/\*\*

     \* @Route("/list\_produits", name="listproduit")

     \*/

    public function index()

    {

        $produits = $this->getDoctrine()

            ->getRepository(Produit::class)

            ->findAll();

        if (!$produits) {

            throw $this->createNotFoundException(

                'No products found in data base'

            );

        }

        return $this->render('produit/list.html.twig', [

            'produits' => $produits

        ]);

    }

Suite à ça développer un nouveau fichier produit/list.html.twig

{% extends 'base.html.twig' %}

{% block body %}

 <table align="center" border="1">

 <tr><th>ID</th><th>Libelle</th><th>Prix</th><th>Description</th></tr>

    {% for p in produits %}

    <tr>

        <td>{{p.id}}</a></td>

        <td>{{p.libelle}}</td>

        <td>{{p.prix}}</td>

        <td>{{p.description}}</td>

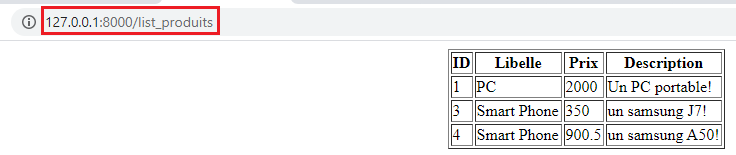
    </tr>

     {% endfor %}

</table>

  {% endblock %}

Et voila le résulat obtenu :



Lorsque vous recherchez un objet particulier, vous utilisez toujours ce que l'on appelle son "repository". Vous pouvez considérer un "repository" comme une classe PHP dont le seul objectif est de vous aider à récupérer les entités d'une certaine classe.

Une fois que vous avez un objet "repository", vous disposez de nombreuses méthodes:

$repository = $this->getDoctrine()->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();

1. **L’opération de mise à jour**

Pour faire la mise à jour d’un produit, il faut tout d’abord chercher cet element, ensuite modifier ses attributs et enfin le sauvegarder. Ajoutons le code suivant à notre controlleur :

    /\*\*

     \* @Route("/produit/edit/{id}")

     \*/

    public function edit\_produit($id)

    {

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

        $produit = $entityManager->getRepository(Produit::class)->find($id);

        if (!$produit) {

            throw $this->createNotFoundException(

                'Pas de produit avec cet ID '.$id

            );

        }

        $produit->setLibelle('Un Iphone!');

        $entityManager->flush();

        return $this->redirectToRoute('show\_produit', [

            'id' => $produit->getId()

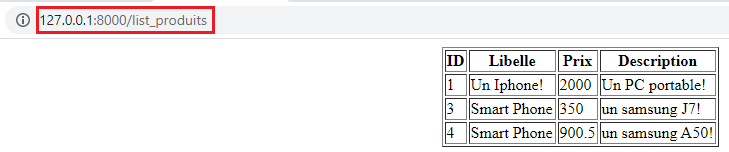
        ]);

    }

Exécuter le code suivant :



Voilà, le produit a été mis à jour !



1. **L’opération de suppression**

La suppression est super facile, au fait après avoir récupérer un objet précis via son id, il suffit ensuite de taper les commandes suivantes :

$entityManager->remove($produit);

$entityManager->flush();

**Troixième Partie : Manipultation du Repository**

Nous avons utiliser l’objet « repository » pour exécuter des requetes CRUD très facilement. Mais si vous avez besoin de requets plus complexes ?

Lorsque vous avez céer votre entité avec la commande « make :entity » cette commande a également générer une classe « ProduitRepository » sous le dossier Repository. Pour utiliser une des méthodes de ce repository taper les commandes suivantes :

// from inside a controller

$repository = $this->getDoctrine()->getRepository(Product::class);

$product = $repository->find($id);

On va voir dans cette section , comment on peut enrichir notre « Repository » avec de nouvelles méthodes. Supposons qu’on veut développer un méthode permettant d’afficher les produits ayant un prix supérieur à 900 ?

1. **Doctrine Query Builder (DQL)**

Au niveau de votre repository, ajouter la métho suivante :

/\*\*

     \* @return Produit[]

     \*/

    public function findAllGreaterThanPrice($prix): array

    {

        $entityManager = $this->getEntityManager();

        $query = $entityManager->createQuery(

            'SELECT p

            FROM App\Entity\Produit p

            WHERE p.prix > :prix

            ORDER BY p.prix ASC'

        )->setParameter('prix',$prix);

        // returns an array of Product objects

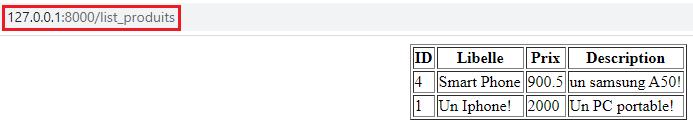
        return $query->getResult();

    }

Ensuite modifier votre méthode index comme suit :

****

Faites un refresh de votre page liste\_produits !



1. **Query Builder**

public function findAllGreaterThanPriceQB($prix): 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.prix > :prix')

            ->setParameter('prix', $prix)

            ->orderBy('p.prix', 'ASC');

        $query = $qb->getQuery();

        return $query->execute();

        // to get just one result:

        // $product = $query->setMaxResults(1)->getOneOrNullResult();

    }

Ensuite, il faut juste appeler cette méthode via votre méthode dans le controlleur.

  $prix = 900;

        $produits = $this->getDoctrine()

            ->getRepository(Produit::class)

           // ->findAll();

           //->findAllGreaterThanPrice($prix);

           ->findAllGreaterThanPriceQB($prix);

1. **SQL**

Enfin, il est possible d’utiliser SQL pour créer des requêtes spécifiques. Elle resemble à la méthode PDO de PHP5.

public function findAllGreaterThanPriceSQL($prix): array

    {

        $conn = $this->getEntityManager()->getConnection();

        $sql = '

            SELECT \* FROM product p

            WHERE p.prix > :prix

            ORDER BY p.prix DES

            ';

        $stmt = $conn->prepare($sql);

        $stmt->execute(['prix' => $prix]);

        // returns an array of arrays (i.e. a raw data set)

        return $stmt->fetchAll();

    }

**Exercice (30 min) :**

1-Créer une entité Categorie ayant comme attribute (libelle : String, nbreProduits : integer)

2-Créer le schéma de cette entité dans la base de données.

3-Créer les Opérations CRUD.

4-Ajouter dans la page list\_Categories deux requêtes permentant de :

1. Afficher le nombre de catégorie dans la base de donnés
2. Afficher le nombre de catégorie ayant un nombre de produits supérieur à 10.
3. Utiliser DQL puis QueryBuilder

***Bon travail !***