# **Projet 8 : Découverte du Framework PHP Symfony 5**

## Ressources :

DEFINITION :

**Symfony** est un ensemble de composants PHP, c’est un [Framework](https://fr.wikipedia.org/wiki/Framework) [MVC](https://fr.wikipedia.org/wiki/Modèle-vue-contrôleur) [libre](https://fr.wikipedia.org/wiki/Logiciel_libre) écrit en [PHP](https://fr.wikipedia.org/wiki/PHP). Il fournit des fonctionnalités modulables et adaptables qui permettent de faciliter et d’accélérer le développement d'un [site web](https://fr.wikipedia.org/wiki/Site_web).

# Objectif :

Le but de ce projet est de procéder à une refonte de votre projet 7 : petite annonces MVC, avec le Framework Symfony.

# Présentation :

Comme dans le projet précédent, il est important de repartir les possibilités d’accès aux données en fonction des rôles des utilisateurs.

## 3 Rôles :

* SIMPLE VISITEUR
* ROLE\_USER
* ROLE\_ADMIN
* ROLE\_SUPER\_ADMIN

1 – Le visiteur :

* Consulter des annonces
* Rechercher des annonces
* Contacter les vendeurs
* Acheter des biens

2 – Le vendeur : ROLE\_USER

* Inscription et connexion
* Accès sécurisé au tableau de bord (BACKEND)
* Dépôt d’annonces et CRUD

3 – Administrateur : ROLE\_ADMIN

* Accès au Bundle EasyAdmin 3 pour manager la plateforme

4 – SUPER ADMIN : ROLE\_SUPERADMIN

* Créer des administrateurs, les édites et les supprimes

A - Avant-Propos :

1. Composer
2. Installer Symfony
3. Structure de Symfony
4. Le routage
5. Le moteur de Template Twig
6. Le profiler Symfony
7. Symfony Flex
8. La couche Modèle Doctrine
9. Les formulaires (+ WEBPACK Encore Bundle)
10. La sécurité
11. Les pages d’erreurs
12. Le multi langue Internationalisation
13. Les Services
14. Swift Mailer (email)
15. Déployer son Site

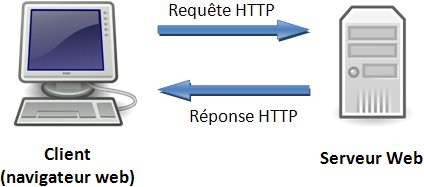
# A - Avant-Propos

1. Être patient
2. Ne pas vouloir tous maitriser (Symfony = assemblage de module complexe)
3. Savoir trouver l’information sur internet : (Taper les mots clés, communauté Symfony, vérifié la configuration du projet, version du Framework, date de la solution, etc…)

Symfony.com, php.net, stackoverflow, tuto, coach, Symfony Cast, Packagist, etc…

1. Être organisé : MCD, Diagramme de séquence, etc…
2. Ne pas chercher à tous développer ☹ Ne pas réinventer la roue, un bon IDE, étape par étape)

# B – Rappel protocole http



Exemple :

1. Une url : [https://exemple.com](https://exemple.com/) = nom de domaine
2. Cette adresse permet de retrouver l’adresse IP du serveur qui l’héberge.
3. En local : IP : 127.0.0.1 = numéro de téléphone
4. Le téléphone = Navigateur Web (Chrome, Firefox etc…)
5. Navigateur -> Nom de domaine -> connexion au serveur = (envoi requête http du navigateur au serveur)
6. La requête est traitée par le serveur et envoie une réponse

Les Codes réponses http (HyperText transfert Protocol)

* 200, la page a été retournée sans erreur du serveur
* 404, le code HTTP pour une ressource qui n'a pas été trouvée sur le serveur ;
* Les codes 3XX, qui signalent les redirections de ressources et qu’il existe plusieurs réponses possibles
* Les codes 4XX, qui signalent une erreur de requête côté utilisateur/client ;
* Les codes 5XX, qui signalent une erreur côté serveur.

# Requêtes et réponses en Symfony

Le Framework Symfony et notamment son composant [**HttpFoundation**](https://symfony.com/doc/current/components/http_foundation.html), apporte une couche d'abstraction pour les requêtes et les réponses, il est simple à utiliser et à manipuler.

* 2 Classe : Request et Response issue de HttpFoundation

# SYMFONY C’EST QUOI ?:

Symfony est un ensemble de composant lié par un noyau (le Kernel).

Le Framework Symfony est construit autour du **paradigme fondamental du web :** un utilisateur fait une requête et le serveur doit retourner une réponse.

* Le composant **HttpFoundation** fournit une couche d’abstraction PHP objet pour la requête et la réponse.
* Le composant **HttpKernel** a la responsabilité de récupérer la requête de l'utilisateur et de renvoyer une réponse.

Un **contrôleur Symfony** est une simple fonction (méthode) d'une classe PHP d’où il est possible de configurer le **routing à l'aide d'annotations PHP** même si d'autres formats de déclaration sont possibles.

Le Framework Symfony non seulement à un composant pour gérer le routing, mais fournit aussi un **contrôleur frontal** en charge de recevoir toutes les requêtes de l'utilisateur et de trouver la bonne action (fonction) du contrôleur à exécuter.

Routes = Annotation @Route(‘’ /accueil ‘’, name= ‘’page\_accueil ‘’)

# C – Un Framework pour quoi faire ?

# AVANTAGES :

1. Gain de temps (Classe et objet préconstruite, Autowiring de Service)
2. Un code déjà structuré
3. Un code standardisé pour un travail en groupe
4. Intégration et reprise de code d’un même Framework
5. Grosse communauté
6. Code, module, composant réutilisable
7. Les Framework évoluent

## INCONVENIENT

1. Bibliothèques lourdes pas utilisées à 100% (Symfony version full)
2. Fin du code personnalisé, tendances à trouver des solutions à l’aide du Framework ou de la communauté
3. Une période d’apprentissage en plus de la connaissance de PHP
4. Les Framework évoluent certaine version change beaucoup (ex : Symfony 2 à 3 avec la disparition des bundles)

# D- Pourquoi Symfony ?

1. Grande communauté
2. Adaptabilité
3. Longévité (LTS LongTimeSupport)
4. Open Source

[https://symfony.com](https://symfony.com/) -> Documentation

# 1 - Composer

Composer est un logiciel gestionnaire de dépendances libre écrit en PHP.

Il permet à ses utilisateurs de déclarer et d'installer les bibliothèques dont le projet principal a besoin. Le développement a débuté en avril 2011 et a donné lieu à une première version sortie le 1ᵉʳ mars 2012.

Toutes les instructions de font en ligne de commande.

Composer require nom du paquet

<https://getcomposer.org/>

# 2 -Installer Symfony 5 :

1. La Version minimal de PHP : 7.2.5
2. <https://symfony.com/download setup.exe>
3. Installer scoop
4. Puis symfony-cli : scoop intall symfony-cli
5. Vérifié sa version : cmd -> **symfony -v**
6. Pour une version Full : **symfony new --webapp my\_project**
7. Pour une version Light (microservice, application console et API): **symfony new my\_project**
8. **Ou via composer : composer create-project symfony/website-skeleton nomApplication**
9. Lancer le server local symfony : cmd -> **symfony server :start ou symfony serve (symfony server :stop pour l’arrêter)**
10. URL (uniform resources locator) = **localhost :8000 ou 127.0.0.1 :8000**
11. Changer de port : **symfony server :start –port 1234**

# 3 - Structure d’un projet de base

##### Le dossier "bin"

Ce dossier contient les exécutables disponibles dans le projet, que ce soit ceux fournis avec le Framework (la console Symfony) ou ceux des dépendances (phpunit, simple-phpunit, php-cs-fixer, phpstan).

##### Le dossier "config"

Il contient toute la configuration de votre application, que ce soit le Framework, les dépendances (Doctrine, Twig, Monolog) ou encore les routes (au format  .yaml).

Ne pas oublier qu'il est possible d'adapter la configuration du Framework en fonction de l'environnement, et qu'une partie de la configuration se trouve aussi dans le fichier .env du projet.

##### Le dossier "public"

Par défaut, il ne contient que le contrôleur frontal de votre application, le fichier dont la responsabilité est de recevoir toutes les requêtes des utilisateurs.

Seul ce dossier doit être accessible de l'extérieur.

C’est le seul dossier accessible par la requête client, il contient tous les fichiers pouvant être chargé par le navigateur (css, js, img, pdf, etc…)

##### Le dossier "migrations"

Dans ce dossier et si vous manipulez une base de données, alors vous trouverez les migrations (requête DQL) de votre projet généré à chaque changement que vous effectuerez sur votre base de données à l'aide de l'ORM Doctrine.

Nous reviendrons sur ce dossier dans le chapitre "Gérez votre base de données avec Doctrine ORM".

##### Le dossier "src"

C'est ici que se trouve le coeur votre application ! Contrôleurs, formulaires, écouteurs d'événements, modèles et tous vos services doivent se trouver dans ce dossier. C'est également dans ce dossier que se trouve le "moteur" de votre application, le kernel.

##### Le dossier "tests"

Dans ce dossier se trouvent les tests unitaires, d'intégration et d'interfaces.

Par défaut, l'espace de nom du dossier **tests** est App\Tests et celui du dossier **src** est App.

##### Le dossier "templates"

Ce dossier contient les gabarits (les vues) qui sont utilisés dans votre projet, ce sont de fichier au format Twig (twig est un générateur de Template HTML)

##### Le dossier "translations"

Symfony fournit un composant appelé [Translation](https://symfony.com/doc/current/components/translation.htm) capable de gérer de nombreux formats de traductions, dont les formats yaml, xliff, po, mo... Ces fichiers seront situés dans ce dossier.

##### Le dossier "var"

Ce dossier contient trois choses principalement :

* Les fichiers de cache dans le dossier **cache**;
* Les fichiers de log dans le dossier **log**;
* Et parfois, si le Framework est configuré pour gérer les sessions PHP dans le système de fichiers, on trouve le dossier **sessions.**

##### Le dossier "vendor"

Ce dossier contient votre chargeur de dépendances (ou "autoloader") et l'ensemble des dépendances de votre projet PHP installées à l'aide de Composer. Une autre façon de découvrir vos dépendances est d'utiliser la commande "composer show".

# Introduction à Symfony Flex

D’un point de vue technique, [Symfony Flex](https://symfony.com/doc/current/setup/flex.html) est juste un plugin Composer. Flex est capable d'écouter les événements Composer, que ce soit l'installation, la mise à jour ou encore la suppression d'une dépendance.

Parmi les tâches qu'il est capable de réaliser :

* Appliquer une configuration par défaut pour un plugin Symfony .
* Créer des fichiers/dossiers .
* Mettre à jour de fichiers (par exemple le fichier config/bundles.php).

Pour cela, Symfony Flex fonctionne à l'aide d'un système de "recettes" qui sont disponibles dans deux dépôts : un dépôt officiel maintenu par l'équipe Symfony et un dépôt communautaire ouvert à tous les mainteneurs de bundles, librairies et projets.

# SYMFONY UTILISE :

Le composant [Dependency Injection](https://symfony.com/doc/current/components/dependency_injection.html)  et notamment comment construire des objets et les récupérer à l'aide du [container de services](https://symfony.com/doc/current/service_container.html) :

**php bin/console debug :container**

**= (ensemble des services de l’application)**

Le container de service est un objet qui est utilisé dans votre projet et auquel on a besoin d'accéder.

Ce service est enregistré dans un container, il est une "recette de cuisine", les étapes nécessaires à sa construction sont les suivantes : dépendances, méthodes et arguments à appeler.

Puisque les services sont présents dans le container de services, on peut les injecter sans crainte dans nos classes grâce à l’autowiring

# L'autowiring de services

Cette fonctionnalité est activée par défaut dans tout projet Symfony 5.

Dans le fichier de configuration : [*services.yaml*](https://github.com/symfony/demo/blob/v1.6.4/config/services.yaml).

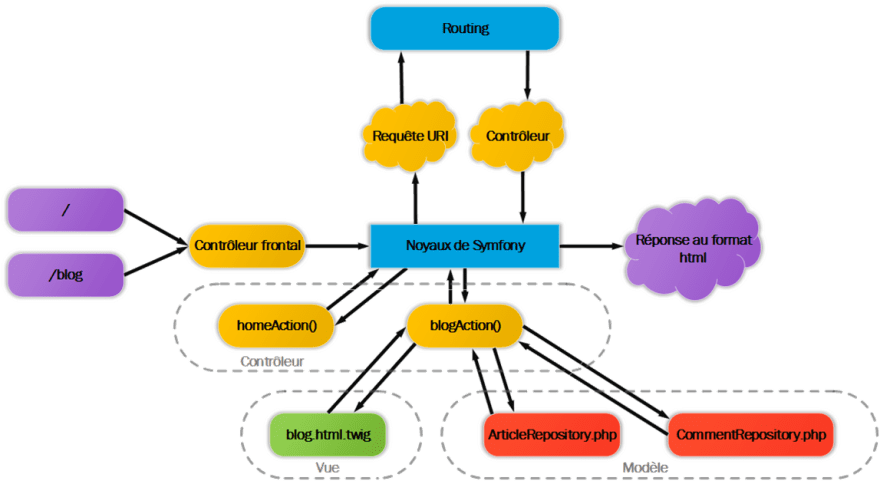
Doc Yaml :

<https://symfony.com/doc/current/components/yaml/yaml_format.html>

## Autowiring liste :

**php bin/console debug:autowiring**

(Ensemble des classes automatiquement charger par le Framework) = pas besoin de créer de service, c’est élément sont a injecté dans les paramètres des méthodes des contrôleurs.

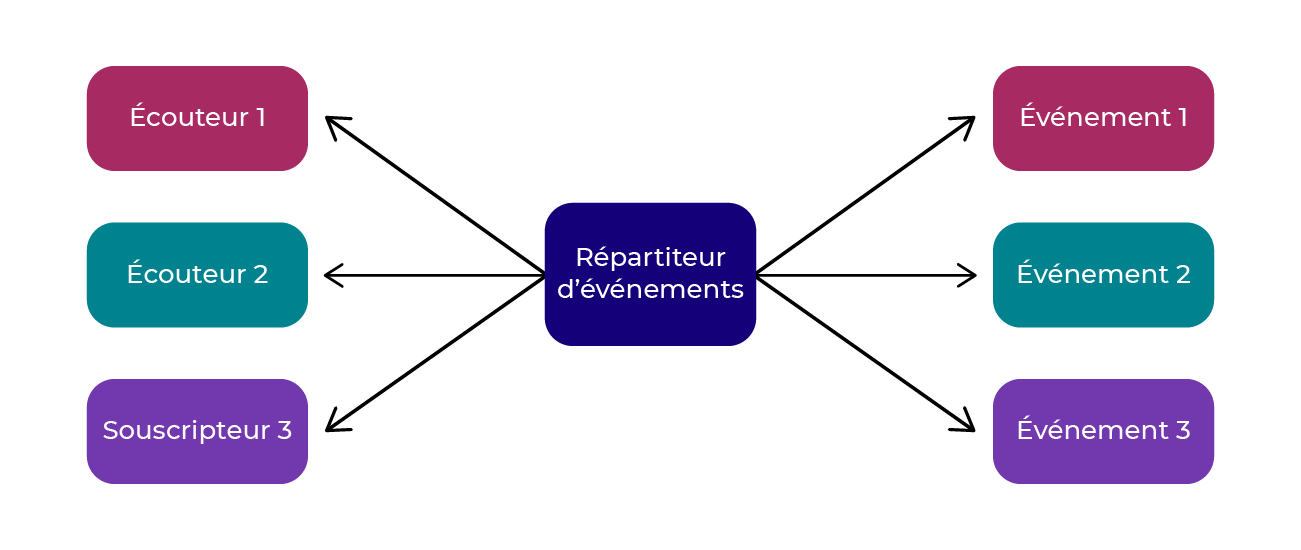


* Grâce à l'autowiring, l'essentiel du temps, nous n'avons rien de spécial à faire pour que nos objets soient automatiquement retrouvés par le container et accessibles dans nos services et nos contrôleurs.
* L'autoconfiguration permet d'ajouter des tags à nos services s'ils implémentent une interface spécifique et les services tagués sont traités différemment par le framework.

Principe de Symfony :

# Le composant EventDispatcher en bref

Une application Symfony dispose d'un**répartiteur d'événements** qui va envoyer une série d'**événements** natifs et métiers. Ensuite, des objets, qui peuvent être des **écouteurs** ou encore des **souscripteurs d'événements,** peuvent écouter ces événements et exécuter des fonctions à partir de données qui sont transmises par l'événement.



* Écouteur 1 écoute l'événement 1, écouteur 2 l'événement 2, et le souscripteur l'événement 3.
* Les 3 "écouteurs" (2 écouteurs, 1 souscripteur) ont été ajoutés au répartiteur d'événements (ou encore "EventDispatcher").
* Quand le répartiteur envoie les événements, il donne l'information aux écouteurs qui peuvent donc réaliser des actions au bon moment sans pour autant avoir connaissance des autres écouteurs.

# A – Configurer une application

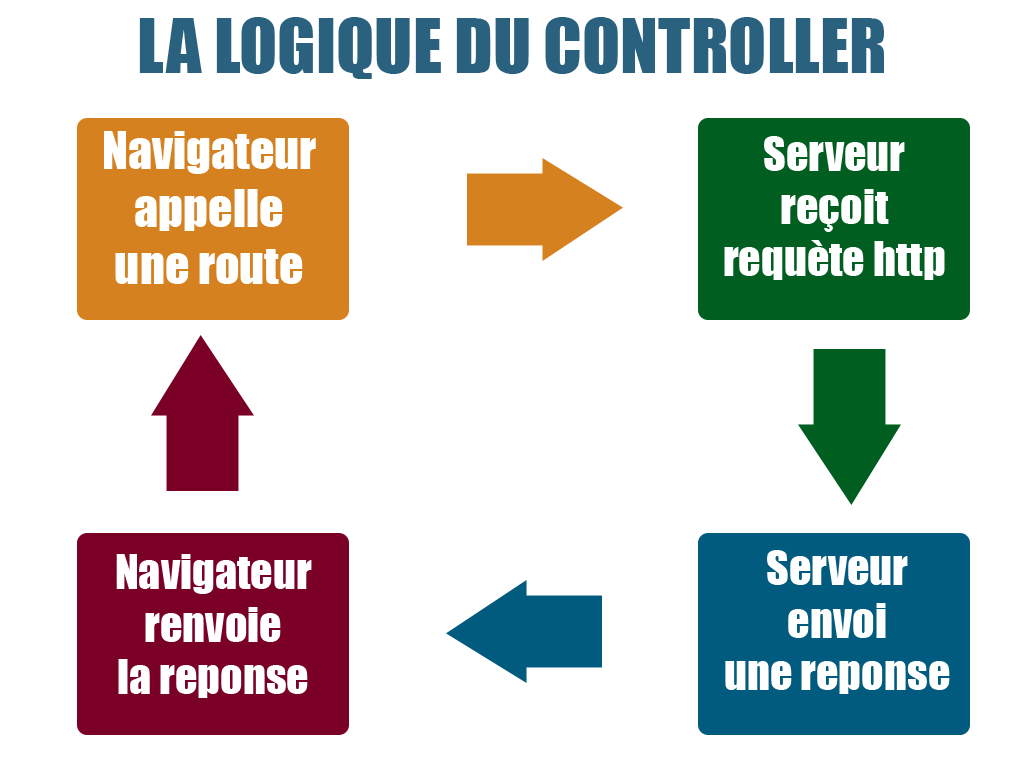
1. Dans PHPStorm **file -> setting -> plugins -> Symfony Supports**
2. Le fichier. env (et test) regroupe toutes les variables d’environnement, elles sont utilisables grâce à la fonction helper\_env() et sont utilisable n’importe où dans le code (variable super gloable Symfony)
3. Lisent des variables d’environnement

**php bin/console debug:container --env-vars**

# B – Symfony et MVC (Models Views Controllers)

1. Models = toutes les données logique métier BDD + Json + Calcul Mathématique + .txt, csv etc…
2. C’est le cœur de métier (utilisé avec ORM (Object relational mapping) DOCTRINE)
3. Views = Génère des pages HTML5 a retournée au client, partie visuelle regroupée dans le dossier Template avec extension .html.twig
4. Controller = Programme PHP qui coordonnes l’application, ils appellent les modèles et les vues a retourné (Classe Request + Response)

# C – Les Contrôleurs : Dossier src / Controller



# LES CONTROLLERS

Dans le dossier : src/Controller

1. Un fichier PHP qui contient une classe
2. Ils doivent se terminer par Controller (ex : AccueilController)
3. On peut les créer à la main ou utiliser makerBundle

Maker-bundle est déjà présent dans l’installation de base de symfony5

Sinon :

composer require symfony/maker-bundle --dev

Pour voir la liste des éléments disponible depuis maker-bundle :

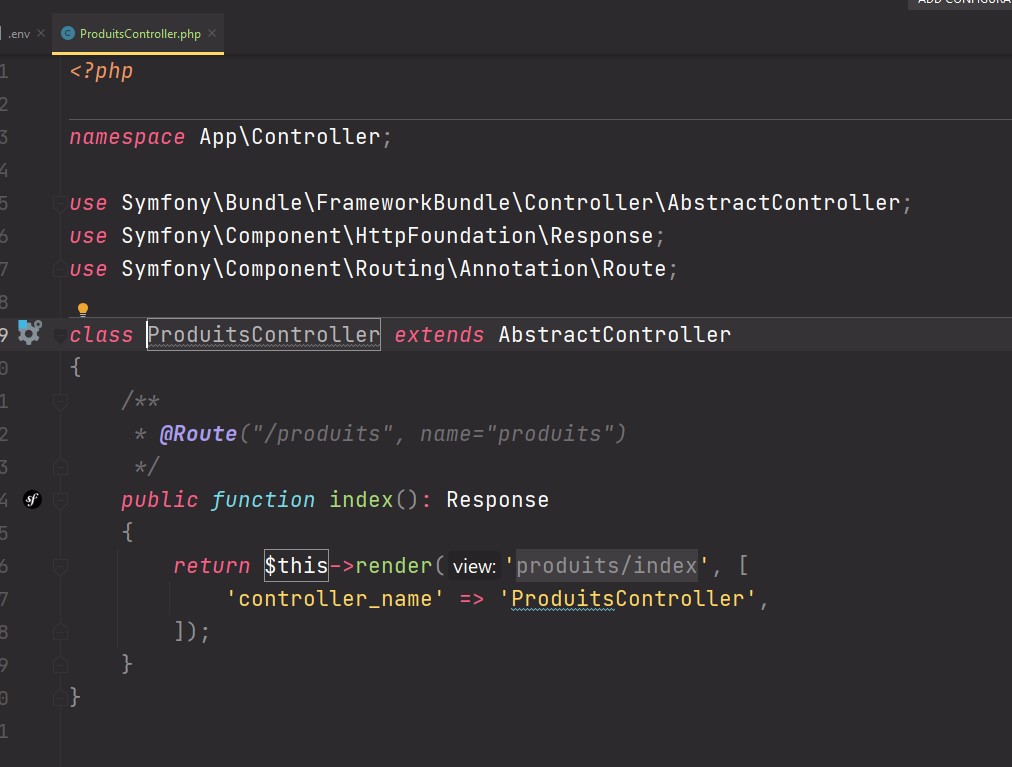
**php bin/console list make**

* 1. Créer un Controller en ligne de commande :

**php bin/console make :controller**

Cette commande a pour effet de générer une classe Controller et un dossier + fichier index.html.twig dans le dossier Templates

Produit Controller



Explication :

1. Le namespace pour le polymorphisme : src/produits/ProduitsController
2. Import (use = appel de classe depuis autoload.php => vendor) Classes AbstactController + Response + Route
3. La classe Produits Controller hérite d’Abstract Controller, ceci donne accès à des méthodes communes à tous mes Controller
4. **Les annotations /\*\* \*/ sont des paramètres de la classe Route**
5. C’est un lien entre une requête envoyée par l’utilisateur et le nom de la méthode a exécuté dans un contrôleur.
6. **CE NE SONT PAS DES COMMENTAIRES**, ici là requête /produits exécutera la méthode index ()
7. Les annotations sont toujours écrites avant la méthode
8. Le paramètre name= ‘nom de la route’

**php bin/console debug :router**

1. La méthode index () exécute la méthode render () (issue d’Abstract Controller), elle permet de faire la jonction avec la vue index.html.Twig dans le dossier Template, elle transmet à la vue une variable controller\_name qui contient la valeur ProduitsController

Pour voir cette vue : **localhost :8000/produits**

# D-Les composants HttpFoundation :

**Tous accès à une application web se fait via une requête http, composée d’une en tête (header)**

* Nom de domaine
* Type de contenus
* Statut

Et est composé d’un Body dans lequel sont passé les paramètres à transmettre

La réponse du serveur à la même syntaxe sauf que le body renvoi de HTML interprétée par le navigateur

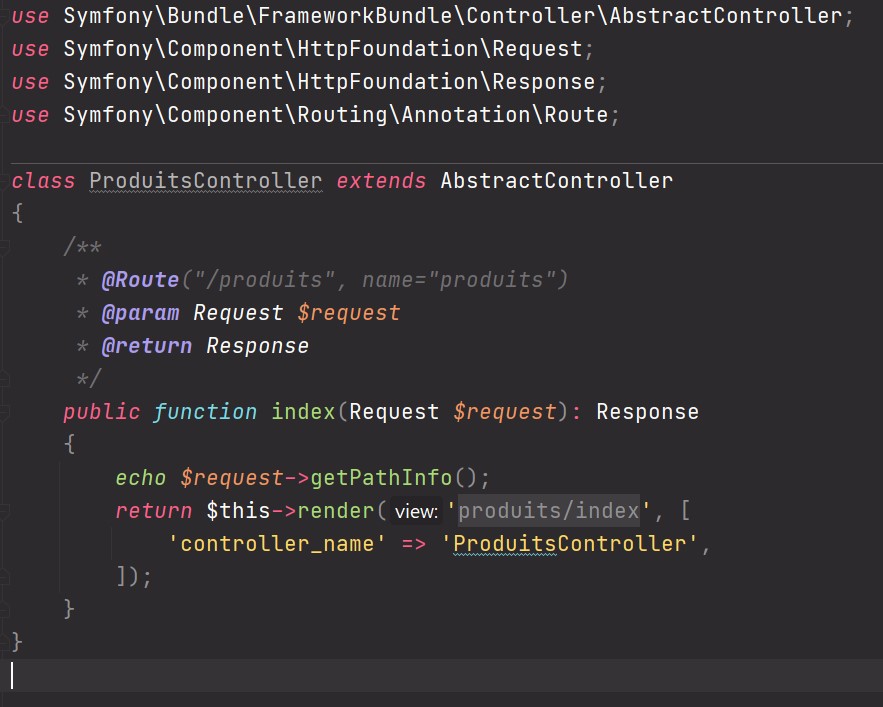
Dans Symfony les composant Request et Response sont contenus dans la bibliothèque HttpFoundation at ajouter grace a l’autowiring

# E-Objet Request :

C’est un objet instancié à partir de la classe Request, cette classe est appelée directement dans les paramètres d’une méthode d’un contrôleur

Grace à autowiring (il est donc directement instancié une seule fois)

Il faut donc appeler la classe en paramètre qui se charge d’instancier l’objet.



DE MANIERE GENERALE ON PEUT DIRE QUE :

Les classes :

* Request = $\_POST en PHP (soumission des formulaires)
* Query = $\_GET en PHP (paramètre transmit dans l’adresse de la requête)
* Cookies = $\_COOKIE en PHP
* Files = $\_FILES en PHP relative au fichier transmit
* Server = $\_SERVER en PHP info serveur
* Headers retourne les entêtes des données des requêtes
* Toutes ces propriétés citées ci-dessus renvoient un objet de la ParameterBag
* On utilise ensuite les méthodes des classes pour récupérer des informations
* All () retourne tous
* keys () retourne le nom des variables,
* get (‘nom de la variable) retourne le nom d’une seule variable
* Has () retourne un booléen

Exemple : localhost :8000 /produits?info=premier requète&statut=message



Pour afficher un tableau de valeur

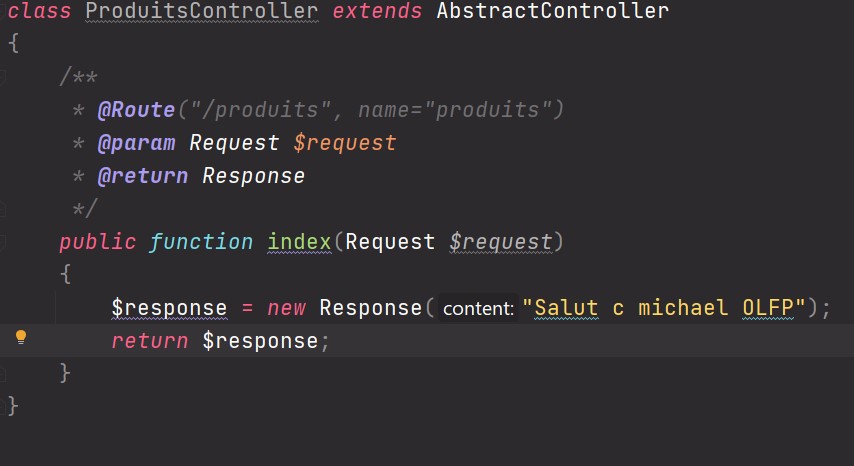
print\_r($request->query-all()) ;

# Objet Response :

Il s’agit également d’un objet instancié à partir de la classe Response, cet objet permet de définir la réponse a envoyé au navigateur

Contrairement à Request on doit l’instancier à l’intérieur de l’action du contrôleur ou après la définition des paramètres de la méthodes ex :



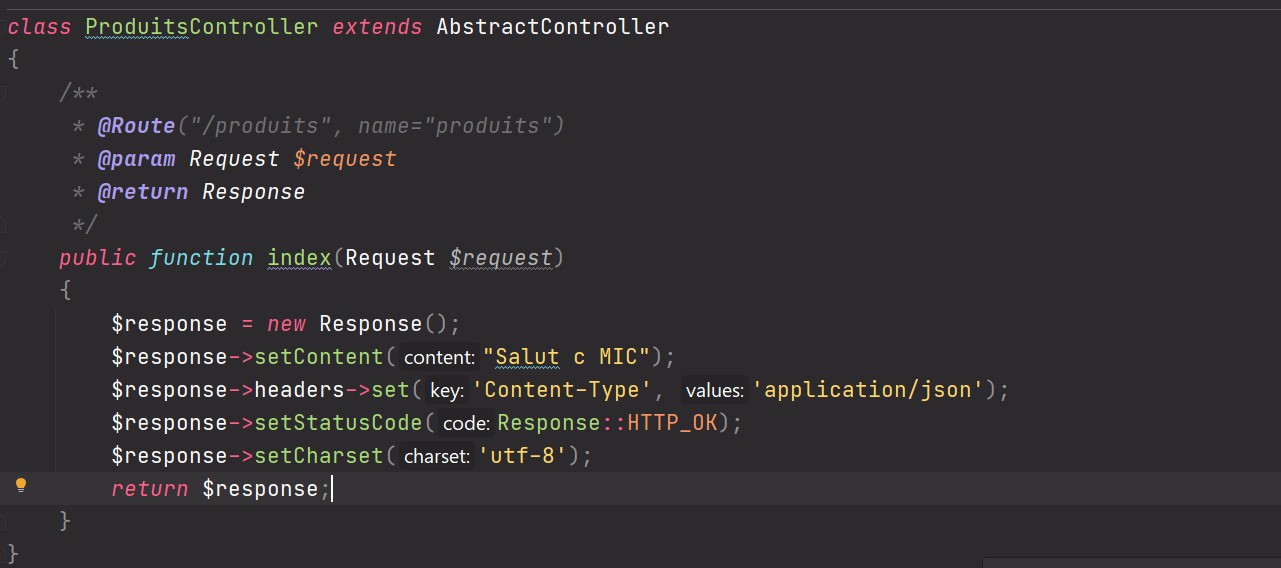


Tout a l’heure la méthode $this->render (page twig) génère implicitement l’objet Response

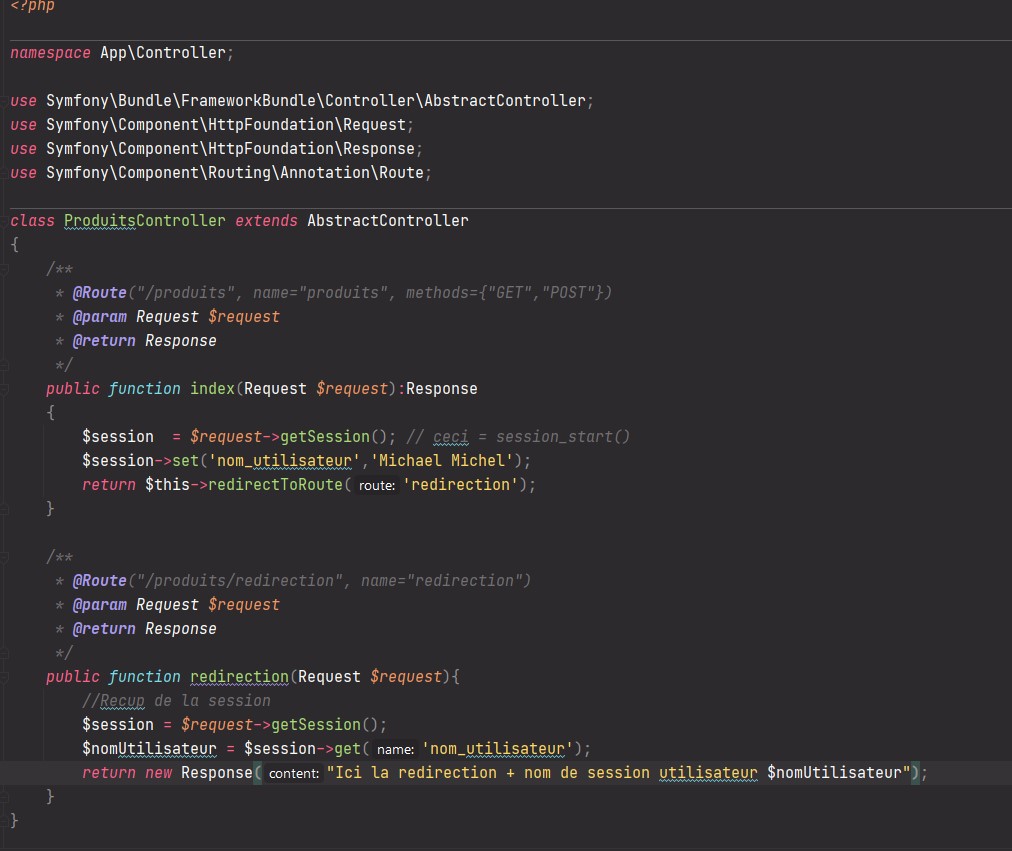
Chaque méthode d’un Controller retourne obligatoirement un objet Response

$response permet d’appeler les méthodes json(), redirect(), generateUrl() et d’autres.

Exemple :



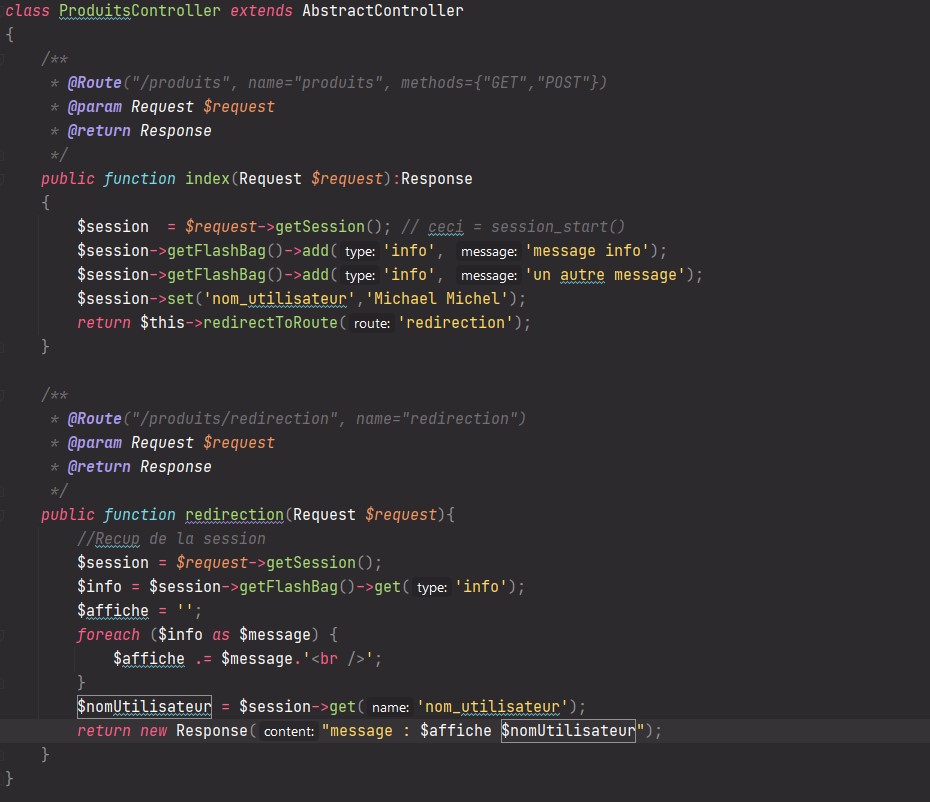
Un exemple de Session et de redirection :



# Les FlashBags :

Ce sont des variables de session qui se supprime elle-même ce sont donc des éléments similaires au notification.

Ex : Sur la page produit, on voit les flashbags qui disparaissent lors du refresh sur la page redirection.



# 4 - LE SYSTEME DE ROUTE :

Dans un contrôleur : Annotation @Route(‘’/ma\_route/ ‘‘) depuis le fichier config/routes/annotation.yaml (Utilise des doubles quottes)

controllers:  
 resource: ../../src/Controller/  
 type: annotation

Les routes sans annotations dans config/routes.yaml

*#index:  
# path: /  
# controller: App\Controller\DefaultController::index*

Annotation @Route(‘’, etc…) peut prendre des paramètres ex :

*\** ***@Route****("/produits", name="produits", methods={"GET","POST"})*

GET pour les routes et POST pour les formulaires

1. Passer des paramètres dans les routes elles même :

*\** ***@Route****("/produits/{nom}/{prenom}", name="produits", methods={"GET","POST"})*

Qui donne url : <http://localhost:8000/produits/mic/michel>



Passer les valeurs des variables dans des paramètre de la méthode :

Pour url : <http://localhost:8000/produits>



Retourne : Bonjour LAGADEK BOB

Passer des paramètres conditionnels = requirements

Ici un exemple avec des Regex (Regular Expression) Expression régulière

*\** ***@Route****("/produits/{nom}/{prenom}", name="produits", methods={"GET","POST"}, requirements={"nom"="[a-z] {2-50}"})*

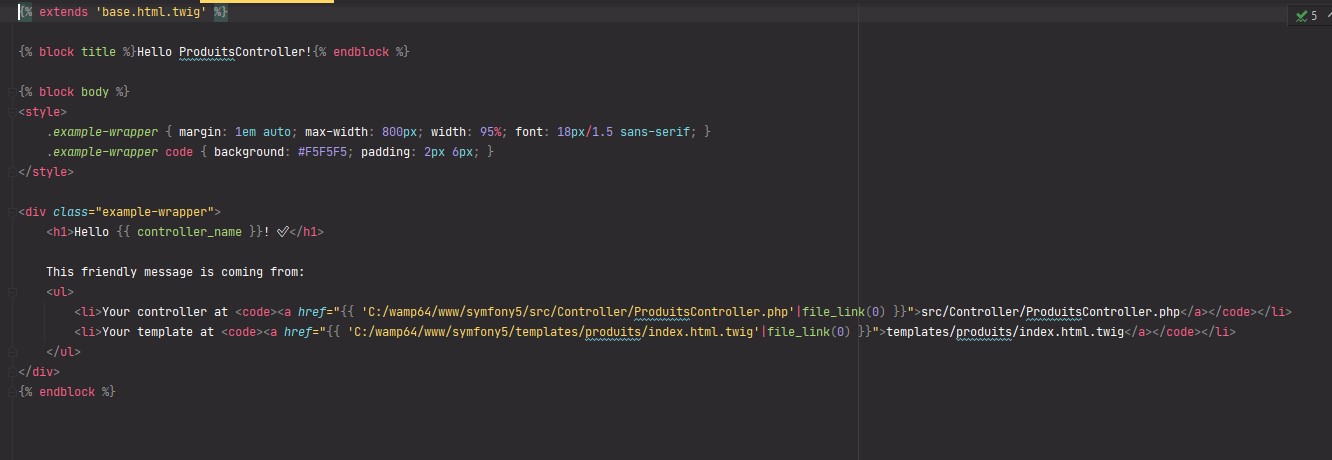
Ici la variable $nom doit être de type alphabétique et comprendre entre 2 et 50 caractères sinon la page retourne une erreur.

Les routes à appeler corresponde au paramètre name= ‘ma\_route’ (ici : produits), c’est ce paramètre qu’il faut appeler dans une ancre par exemple

Pour lister vos routes : php bin/console debug :router

# 5 - A Les Vues : Dossier src / Template

1. La vue est un état final qui génère de HTML5 destinée à l’utilisateur
2. Elle possède l’extension .html.twig, Twig est un langage qui permet de faire des traitements dans la vue comme le ferai une page PHP, Twig permet d’utiliser des variables, faire des conditions, des boucles, des filtres. Ainsi les vues ne contiennent pas de PHP.
3. Lorsque l’on génère un Controller une vue est automatiquement générée
4. Dans le dossier Template on peut y voir un fichier parent nommé base.html. Twig qui est layout, c’est le squelette principal de nos vues



A - Ce fichier hérite du gabarit layout base.html.twig

B - Twig est composé de bloc avec la syntaxe {{}} {%%} {##}

C - Les blocs sont ouvrant et fermant {% block body %} {% endblock %}

D - Le code HTML doit être à l’intérieur d’un block

E – Toutes les vues hérites du template de base : base.html.twig avec :

{% extends 'base.html.twig' %}

# 5 -B TWIG un moteur de Template

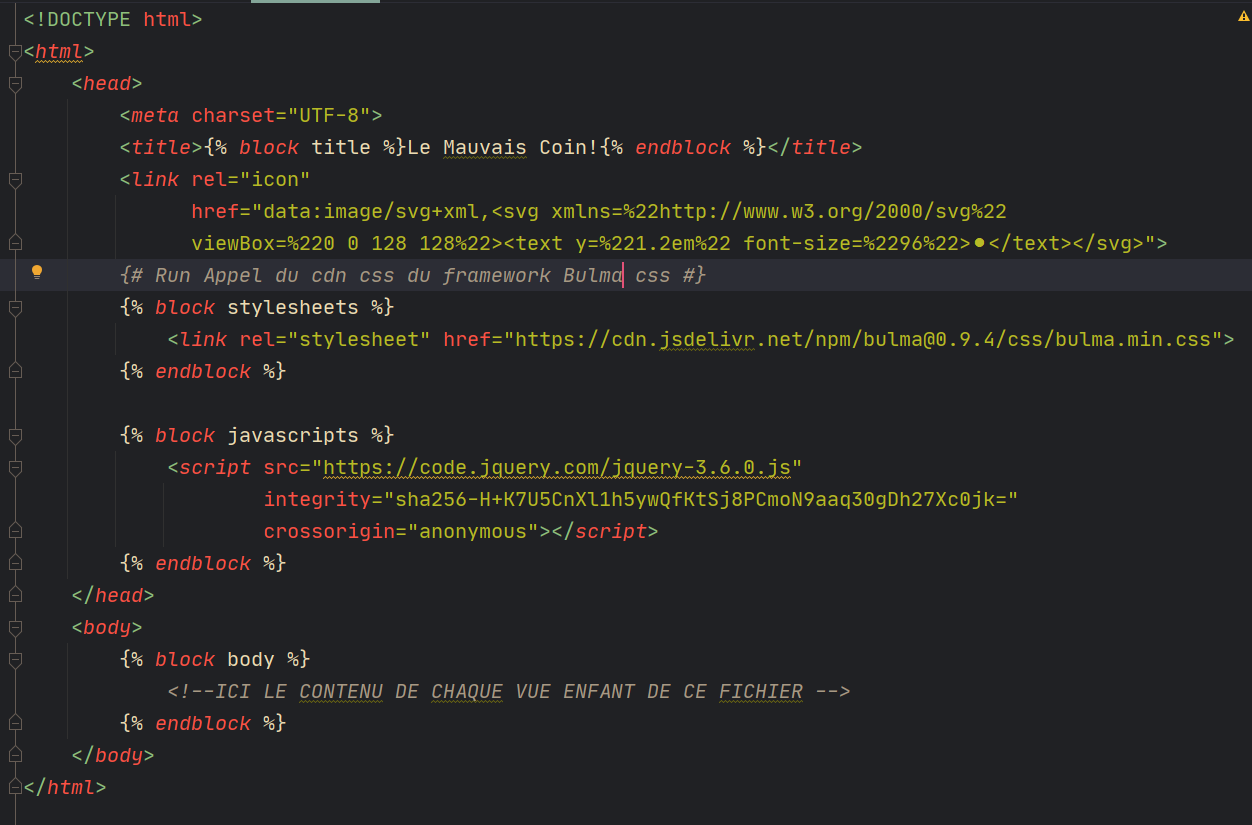
<https://twig.symfony.com/doc/3.x/>

[**Pas de PHP dans vos vues.html.twig (twig est utilisable sans Symfony)**](https://twig.symfony.com/doc/3.x/)

Chaque vue herite du fichier template/base.html.twig :

On peu donc ajouter des appels CSS et JS dans les blocks {%stylesheets%} et JS comme ceci :

Ici j’appel le CDN du Framework Bulma CSS

Chaque vue herite donc de Bulma css + jquery

## La syntaxe :

**{{ des valeurs }}** = Afficher le contenu d’une variable ou le résultat d’une expression = Interpolation (issue de mustache, handleBar, Js, etc…)

**{% instruction %}** = Exécuter une structure de contrôle (if, foreach, etc…)

**{# des commentaires #}** = Commentaire Twig

Afficher des variables :

*public function* index(Request *$request*):Response  
{  
 $nom = 'Michael';  
 $prenom = 'Michel';  
 $age = 20;  
 *return* $this->render('produits/index.html.twig',[  
 'nom' => $nom,  
 'prenom' => $prenom,  
 'age' => $age  
 ]);  
}

Ici les paramètres de la méthode index () sont transmis via un tableau associatif (clé/valeur)

Pour afficher les variables dans index.html.twig entre {% block body %} et {% endblock %}

{% extends 'base.html.twig' %}  
  
{% block title %}Hello ProduitsController!{% endblock %}  
  
{% block body %}  
 <h1>Bienvenue {{ nom }} {{ prenom }} tu as : {{ age }}</h1>  
{% endblock %}

Exemple de condition : {% if age > 15 %} {% else %} {% end if%}

{% extends 'base.html.twig' %}  
  
{% block title %}Hello ProduitsController!{% endblock %}  
  
{% block body %}  
 <h1>Bienvenue {{ nom }} {{ prenom }} tu as : {{ age }}</h1>  
 {% if age > 50 %}  
 <p>Tu as plus de 50 ans</p>  
 {% else %}  
 <p>Tu as moin de 50 ans</p>  
 {% endif %}  
{% endblock %}

Il est possible comme en PHP d’inclure des fichiers

**{% include ‘nom\_de\_la\_vue.html.twig %}** (par exemple dans base.html.twig include de menu.html.twig pour chaque page ou fun footer)

Ajouter une variable d’environnement :

Dans le fichier .env APP\_AUTHOR=MICHEL Michaël

Puis dans le fichier config/package/twig.yaml :

twig:  
 default\_path: '%kernel.project\_dir%/templates'  
 globals:  
 auteur: '%env(APP\_AUTHOR)%'

Attention à l’indentation 4 espace et pas d’espace entre les %%

Enfin dans index.html.twig :

<h3>{{ auteur }}</h3>

Les variables de sessions :

Elles sont appelées via app.

{{ app.session.get('nom\_de\_la\_session') }}

Créer à la racine du dossier Template un fichier alert.html.twig :

<div *class*="container mt-5">  
 {% for message in app.session.flashBag.get('message') %}  
 <span *class*="alert alert-{{ app.session.get('statut') }}">  
 {{ message }}  
 </span>  
 {% endfor %}  
</div>

Puis dans la base base.html.twig  (avant la fermeture du body):

{% include 'alert.html.twig' %}

Et le Controller pour tester :

*public function* index(Request *$request*):Response  
{  
 $session = *$request*->getSession();  
 $session->getFlashBag()->add('message', 'test des alerts');  
 $session->getFlashBag()->add('message', 'SECOND TEST ALERT');  
 $session->set('statut', 'success');  
 *return* $this->render('produits/index.html.twig');  
}

Du CSS et JS dans base.html.twig :

Dans le bloc {% block stylesheet %} et {% block javascript %}

Pour accéder à un élément du dossier public Twig utilise :

{{ asset(‘dossier/fichier’)}} = public/css/fichier.css

<link rel=’stylesheet’ href=’{{ asset(css/styles.css’) }}’ /> et

<script src=’{{ asset(‘js/app.js’) }}’><script>

C’est appel sont dupliquer sur toutes les pages qui hérite de base.html.twig

Des liens <a href=’’></a>

Puis dans twig <a href=’{{ path(‘nom\_de\_la\_route’}}’>Liens</a>

Rappel pour lister les routes : **php bin/console debug :router**

Les filtres :

**{{ expression | filtre | filtre }}**

Des majuscules UPPER :

<h1>Salut a {**{nom | upper}}**</h1>

Des dates :

**{{ Date\_jour | date(‘d-m-Y à H :i :s}}**

Depuis votre contrôleur :

*return* $this->render('produits/index.html.twig',[  
 "Html5" => '<h3 class="text-danger">TEST HTML RAW</h3>'  
]);

Puis dans Twig :

{{ Html5 | raw }}

On récupère la clé du tableau associatif render + raw qui interprète les balises et classes HTML5

**Les fonctions Twig :**

dump() = le debug

{{ dump(Html5) }}

Resultat : "**<h3 class="text-danger">TEST HTML RAW</h3>**"

# 6 - La barre de debug Profiler

C’est une web tool bar qui n’apparait qu’en mode dev (définie dans .env)

**composer require --dev symfony/profiler-pack**



Statut + Route + Temps de chargement + Mémoire + Cache + Profile + Twig

# 7 - Symfony Flex

Installer par défaut lors de l’installation –full :

**symfony new --full my\_project**

Il permet d’installer des dépendances comme composer (composer. json), il utilise des recette (recipes) qui sont des dépôts du site : <https://flex.symfony.com/> (comme https://packagist.org/ )

Toutes les dépendances Symfony se situe dans le fichier composer.json et tous les bundles du noyau Symfony (Kernel) sont dans le fichier config/bundles.php

Pour résoudre des problèmes de norme de codage :

**composer require cs-fixer**

# 8 -La couche modèle avec Doctrine (ORM)

Les base de données = fichier structurée de données destiné à être extrait pour avoir accès à une information de manière sécurisée

Les données sont structurées sous forme de table appelée entité.

Symfony propose de base :

* SQLite
* MySQL
* PostgreSQL

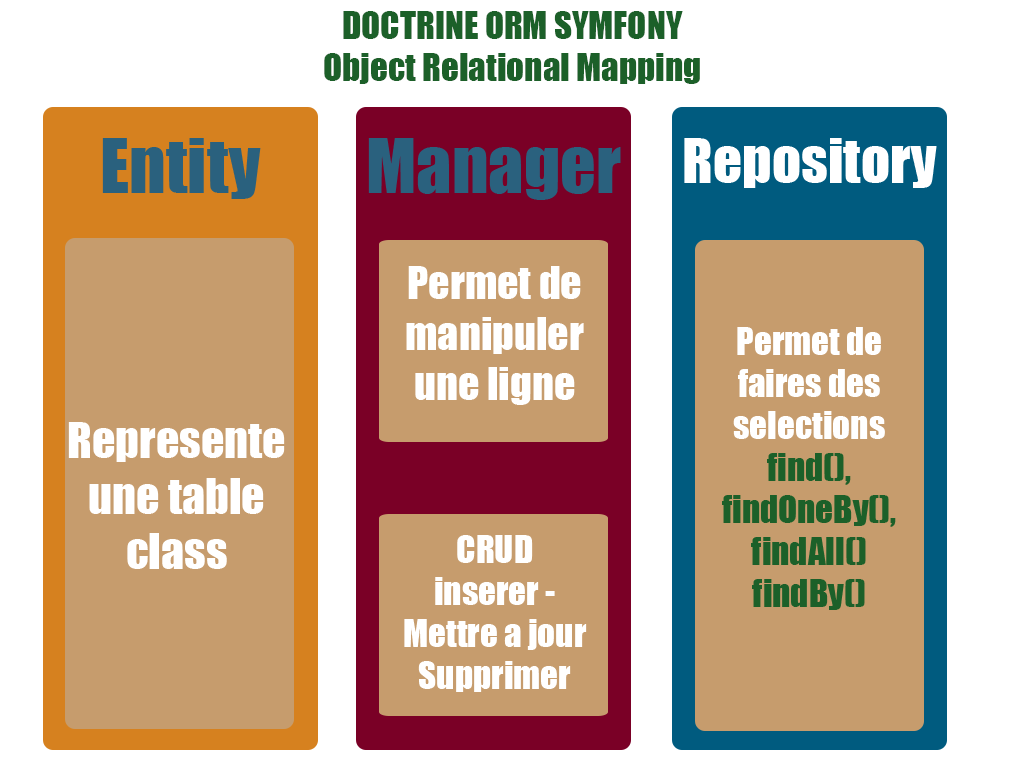
SQL (Services Query Language ):

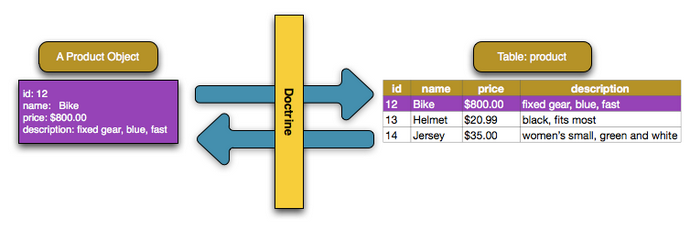
CRUD = Create Read Update Delete

Symfony dispose de Doctrine (ORM = Object Relational Mapping)

Il faut donc simplement utiliser des classes de Doctrine au lieu d’écrire du SQL

# A - DOCTRINE :





# B – Utiliser MySQL :

Dans le dossier .env décommenter la ligne suivante :

Ce paramètre implique de disposer de WampServer (Combo PHP + Apache + MySQL)

Ligne par defaut dans .env :

*DATABASE\_URL="mysql://db\_user:db\_password@127.0.0.1:3306/db\_name?serverVersion=5.7&charset=utf8mb4"*

Vous donc quelquechose de similaire a une connexion PDO :

$dbh = new PDO('mysql:host=localhost;dbname=test', $user, $pass);

Pour Symfony :

mysql:host utilisateur PhpMyAdmin + Mot de passe PhpMyAdmin + IP Locale + port MySQL + Nom de base de données + version du serveur

**DATABASE\_URL=mysql://root@127.0.0.1:3306/symfony5?serverVersion=5.7**

# C – Créer une base de données :

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

Par défaut la base de données crée prend le nom du paramètre décrite dans le fichier .env (ici : symfony5)

Verifié la creation de la base de données a l’adresse :

http://localhost/phpmyadmin

# D – Créer une entité (Table et / ou Classe) et des migrations (fichier script) :

Une migration fait passer votre base de données d’un état A => B :

Les migrations exécutes des requêtes DQL (Doctrine Query Language)

**php bin/console make :entity**

Répondre aux questions posées :

Une fois terminé : **php bin/console make:migration**

Puis : le Flush : **php bin/console doctrine:migrations:migrate**

Ceci a pour effet de créer un dossier src/Entity/produits.php et Repository/ProduitRepository

# A – Entity/Produit.php

Ce fichier définit l’entité produit toutes ses propriétés et ses accesseur et Mutateur (Getter & Setter) similaire au langage Java

Ligne de commandes

**php bin/console make:entity**

Entrer le nom de votre entité (C’est une classe donc une majuscule)

ex : Produits

Ajouter votre premier champs :

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

ex : nom\_produit

A CE STADE ENTRER : ?

Cette etape permet de connaître tous les types de champ proposé par symfony

**PAS BESOIN DE CREER DE CHAMP CLE PRIMAIRE SYMFONY LE FAIT POUR VOUS**

Ce champ est de type :

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

Longueur du champ :

Field length [255]:

Le champ peux est null :

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

On vous demande si vous voulez ajouter un autres champ :

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

Si vous avez terminer : valider avec la touche entrée du clavier

**EN CAS D’ERREUR : PAS DE PANIQUE VOTRE ENTITE PEUX ETRE MODIFIER A TOUS MOMENT**

# B – Les migrations

Pour créer des entité (table) Symfony passe par un stade de migration (classe qui décrit comment faire l’opération)

Pour executer une migration

**php bin/console make:migration**

**PUIS :**

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

Ceci a pour effet de generer un fichier de migration dans le dossier migrations

Ex : Dossier + fichier : migrations/Version21458251.php

*public function* up(Schema *$schema*) : void  
{  
 *// this up() migration is auto-generated, please modify it to your needs* $this->addSql('CREATE TABLE produit (id INT AUTO\_INCREMENT NOT NULL, nom\_produit VARCHAR(255) NOT NULL, prix\_produit DOUBLE PRECISION NOT NULL, quantite\_produit INT NOT NULL, rupture TINYINT(1) NOT NULL, PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8mb4 COLLATE `utf8mb4\_unicode\_ci` ENGINE = InnoDB');  
}  
  
*public function* down(Schema *$schema*) : void  
{  
 *// this down() migration is auto-generated, please modify it to your needs* $this->addSql('DROP TABLE produit');  
}

Ces opérations sont des requètes SQL classique, les méthodes down et up permet d’avancer ou de retourner en arrière

Vérifié la création de votre table dans PhpMyAdmin : <http://localhost/phpmyadmin/>

Pour revenir en arrière (migration précédente) :

**php bin/console doctrine :migration :migrate prev**

4 Alias pour les migrations

* first = Migrez avant la 1er version
* prev = Migrez avant la version précédente
* next = Migrez vers la prochaine version
* latest = Migrez à la dernière version

LES FICHIERS DANS LE DOSSIER src/entity/Produits.php

Chaque entité (table dans PhpMyAdmin) generée par la ligne de commande créer une classe avec des propriétées et des methodes

Les propriétés sont les champs de la tables + types

Les methodes sont des getters (accesseur) et des setters (mutateur)

Chaque propriétés possède donc sa methode pour acceder et modifier un champ de la table

Chaque entité est lié a un Repository, c’est une classe + methode capable d’effectuer des requètes SQL pour acceder et modifier une entité .

**src/Produits.php**

Ce fichier dispose également d’annotation pour chaque propriété, elles précisent les informations utiles de la table produits au sein de la base de données Symfony5.

*/\*\*  
 \** ***@ORM\Entity****(repositoryClass=ProduitRepository::class)  
 \*/*

@ORM : Décrit le chemin vers le fichier ArticlesRepository qui fera les requêtes SQL (DQL = Doctrine Query Langage)

*/\*\*  
 \** ***@ORM\Id*** *\** ***@ORM\GeneratedValue*** *\** ***@ORM\Column****(type="integer")  
 \*/  
private* $id;

@ORM\GeneratedValue () définit id comme clé primaire, auto incrémentée de type entier (Integer)

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

Le champ nom du produit de type string = chaine de caractère 255 octets

Etc…

Puis chaque champ possède ses accesseurs et mutateur (Getter et Setter)

# C – Fixtures ou faux jeux de données

Remplir une table avec un jeu de fausses données est simple avec Symfony, elles sont réalisables dans le Controller ou en dehors grâce à la commande :

Installation du bundle: **composer require --dev orm-fixtures**

Le drapeau (flag) –dev stipule que les fixtures sont ajoutées seulement pour le mode développement (.env = dev)

Toutes les fixtures héritent de la classe Fixtures du Fixture Bundle (classe abstraite) et sont chargée grâce à la méthode abstraite load()

La ligne de commande a pour effet de créer le dossier src/DataFixtures + un fichier ProduitFixture.php

# METHODE 1 : FAKER

Créer une entité Articles : **php bin/console make :entity**

**Articles**

Les champs :

nom\_article 255 string

contenu\_article text

image\_article 255 string

auteur\_article  255 string

date\_depot\_article dateTime

Validation : **php bin/console make:migration**

Puis : **php bin/console doctrine :migrations :migrate**

Insatller Faker : <https://github.com/fzaninotto/Faker>

**composer require fzaninotto/faker**

Puis : **php bin/console make:fixtures**

ArticlesFixtures

Le code dans DataFixtrues/ArticlesFixtures.php

<?php  
  
namespace App\DataFixtures;  
  
use App\Entity\Articles;  
use Doctrine\Bundle\FixturesBundle\Fixture;  
use Doctrine\Persistence\ObjectManager;  
use Faker;  
class ArticlesFixtures extends Fixture  
{  
 public function load(ObjectManager $manager)  
 {  
 //Appel de faker  
 $faker = Faker\Factory::*create*('fr\_FR');  
 //Creation d'un tableau vide  
 $articles = Array();  
 //Boucle soit 20 elements  
 for ($i = 0; $i < 20; $i++) {  
 //Insatnce de entité class  
 $articles[$i] = new Articles();  
 //Jeu de fausse donnée  
 $articles[$i]->setNomArticle($faker->word);  
 $articles[$i]->setContenuArticle($faker->sentence($nbWords = 6, $variableNbWords = true));  
 $articles[$i]->setImageArticle($faker->imageUrl($width = 640, $height = 480));  
 $articles[$i]->setAuteurArticle($faker->lastName);  
 $articles[$i]->setDateArticle($faker->dateTime($max = 'now', $timezone = null));  
 $manager->persist($articles[$i]);  
 }  
  
 $manager->flush();  
 }  
}

Puis : **php bin/console doctrine:fixtures:load**

Pour focus sur un fichier particulier :

**php bin/console doctrine:fixtures:load --group=ArticlesFixtures –append**

Votre entité articles est remplie

Effectuer la meme opération pour votre entité (Table) Produits

En cas d’erreur d’entité :

Modifié les champs de votre entité et mettre a jour le schema

**php bin/console doctrine :schema:update –force**

puis :

**php bin/console doctrine:cache:clear**

# D – Créer un contrôleur et afficher les données de la table produit

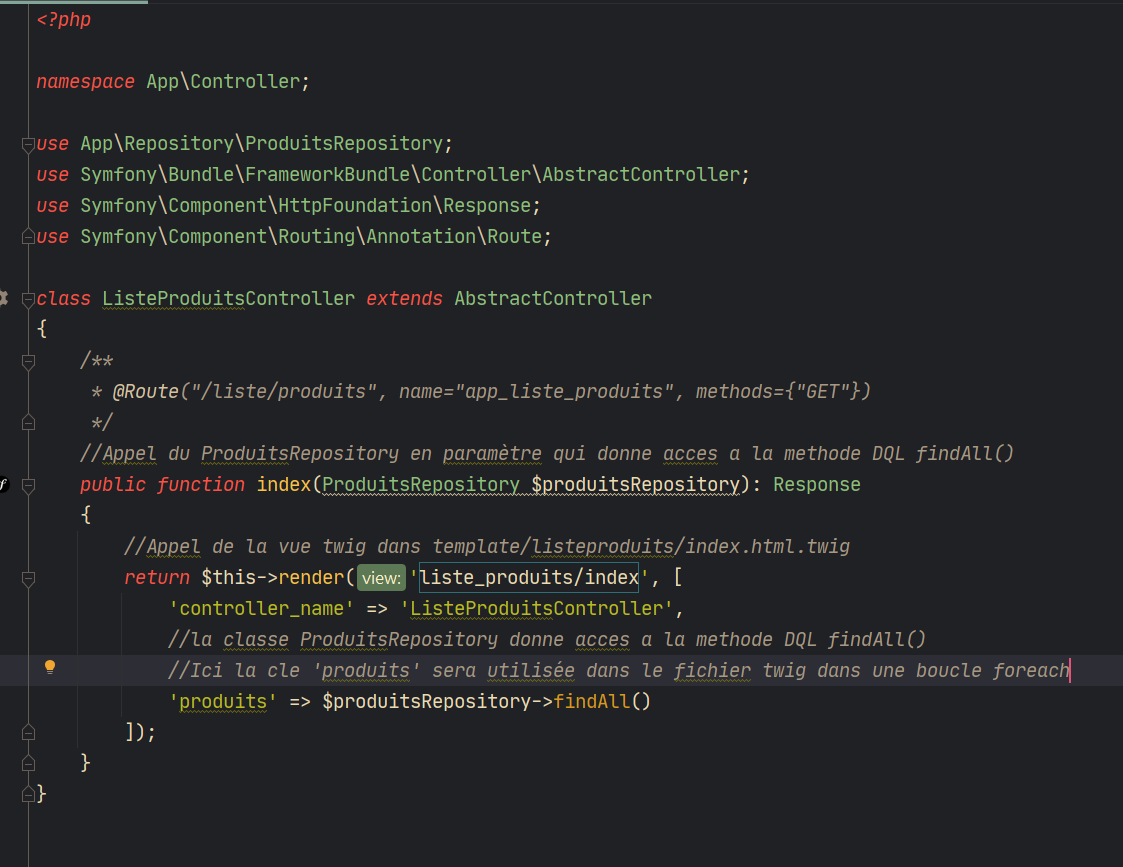
Dans un contrôleur la récupération de données ce fait également avec Doctrine

php bin/console make :controller ListeProduits

Cette ligne de commande a pour effet de créer un dossier src/controller fichier ListeProduitsController et un dossier liste\_produits + un vue dans template/liste\_produits/index.html,twig

Pour récupérer les données on utilise Doctrine EntityManager et le ProduitRepository

**Afficher les produit avec : ListeProduitsController.php**

*LA VUE TWIG : template/liste\_produits/index.html.twig*

Ici twig recupère la cle ‘produits’ du controleur et grace a l’interpolation on récupère les Getters

{{produit.nomProduit}}

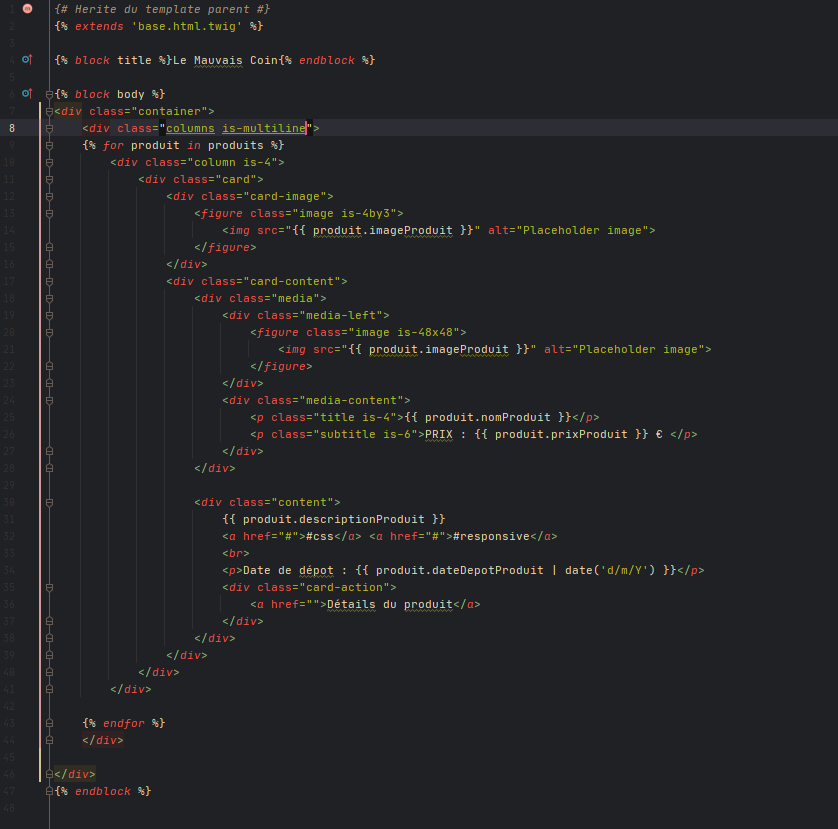
On créer donc une boucle foreach avec un alias de ‘produits’ du controleur

{% for produit in produits %}

Mise en page Bulma card + appel des données avec Twig :

**(AJOUTER LE CDN BULMA A template/base.html.twig)**

La vue twig qui affiche les produits (template/liste\_produits/index.html.twig)



On utilise donc une instruction Twig avec une boucle for et le paramètres ‘liste Produit’ du tableau associatif du contrôleur :

{% for produit in produits %}

{% endfor %}

Pour afficher chaque élément on appelle l’attribut de notre boucle produit.Getter soit :

{{produit.nomProduit }} le getter getNomProduit()

*public function* getNomProduit(): ?*string*  
{  
 *return* $this->nom\_produit;  
}

# AFFICHER UNE PAGINATION :

Utilisé le bundle : <https://github.com/KnpLabs/KnpPaginatorBundle>

Installation : composer require knplabs/knp-paginator-bundle

Créer un fichier config/packages/knp\_paginator.yaml

knp\_paginator:  
 page\_range: 5 *# Nombre de liens montre de la pagination* default\_options:  
 page\_name: page *# Nom de la cle query paramètre* sort\_field\_name: sort *# paramètre query du champs de tri* sort\_direction\_name: direction *# direction du tri* distinct: true *# Utile en cas de group\_by* filter\_field\_name: filterField *# champ de filtre* filter\_value\_name: filterValue *# valeur du filtre* template:  
 pagination: '@KnpPaginator/Pagination/bulma\_pagination.html.twig

' *# ici le template bulma = vendor/knp-paginator-bundle-src-template/* sortable: '@KnpPaginator/Pagination/sortable\_link.html.twig' *# sort link template* filtration: '@KnpPaginator/Pagination/filtration.html.twig' *# filters template*

Dans le controlleur, methode de liste de articles

*/\*\**  
 *\* @Route("/", name="app\_produits\_index", methods={"GET"})*  
 *\* @param ProduitsRepository $produitsRepository*  
 *\* @param PaginatorInterface $paginator*  
 *\* @param Request $request*  
 *\* @return Response*  
 *\*/*  
*public function* index(ProduitsRepository $produitsRepository, PaginatorInterface $paginator, Request $request): Response  
{  
 *//Appel du service PaginatorInterface en paramètre*  
 *//Appel de la methode paginate + paramètres*  
$pagination = $paginator->paginate(  
 *//On recupère tous les articles*  
$produitsRepository->findAll(),  
 *//On liste par entier (knp\_paginator.yaml) on definit la cle dans url, par defaut ma page=1 + nombre d'article a afficher (ici 2)*  
$request->query->getInt('page', 1), 2  
 );  
  
 *return* $this->render('articles/index.html.twig', [  
 'controller\_name' => 'ArticlesController',  
 'articles' => $produitsRepository->findAll(),  
 'pagination' => $pagination  
 ]);  
}

**Traduire les boutons en fr**

Les fichiers de traductions sont déjà fournis :

Vendor/knplabs/knp-paginator-bundle/translation/fr .xliff

Il faut simplement passe le Framework en fr :

Config/packages/translation.yaml :

framework:  
 default\_locale: fr  
 translator:  
 default\_path: '%kernel.project\_dir%/translations'  
 fallbacks:  
 - fr

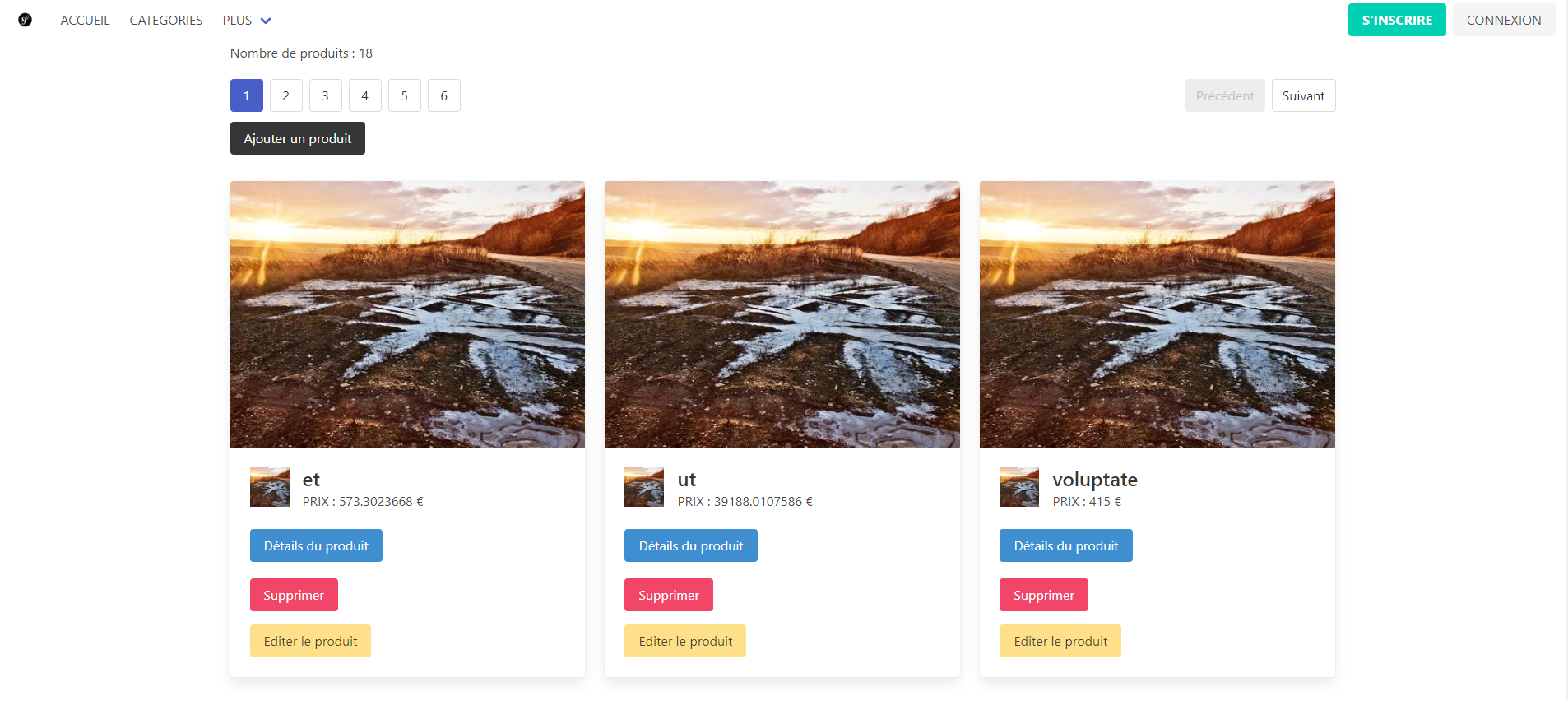
**Effacer le cache :**

La commande : **php bin/console doctrine :cache:clear**

**La vue :**

*<!--LA PAGINATION-->*  
<*div* class="count">  
 Nombre de produits : {{ pagination.getTotalItemCount }}  
</*div*>  
<*div* class="navigation" style="margin-top: 20px">  
 {{ knp\_pagination\_render(pagination) }}  
</*div*>

Resultat :



# E – LES DETAILS D’UN PRODUIT

Pour afficher les détails de l’article, on ajoute une méthode et une route avec un paramConverter capable de récupérer les information de notre entité

On passe dans url un {id}

*@Route("/produits/{id}"*

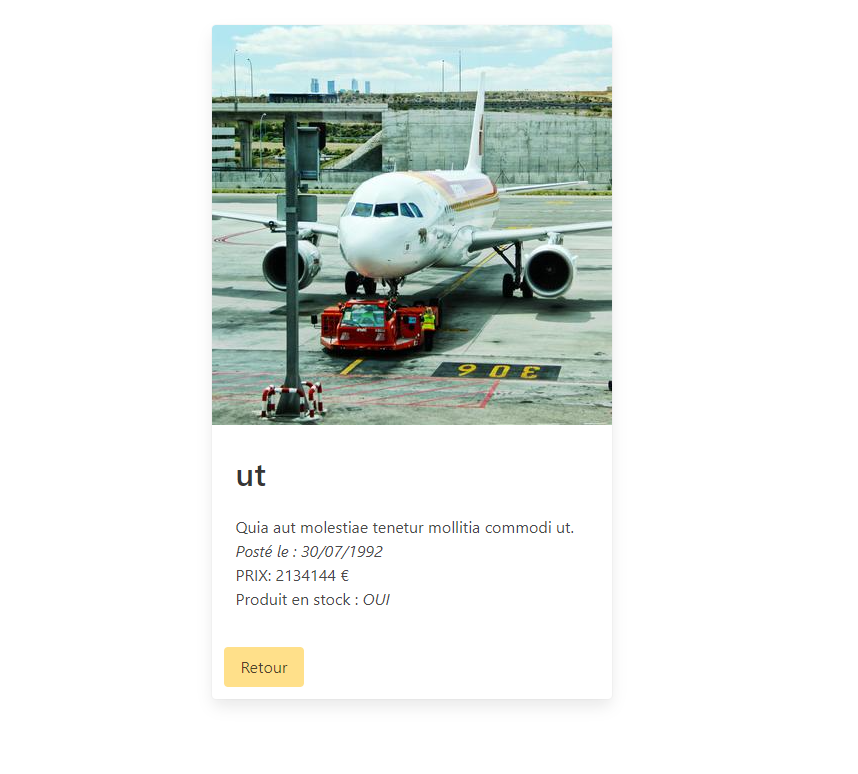
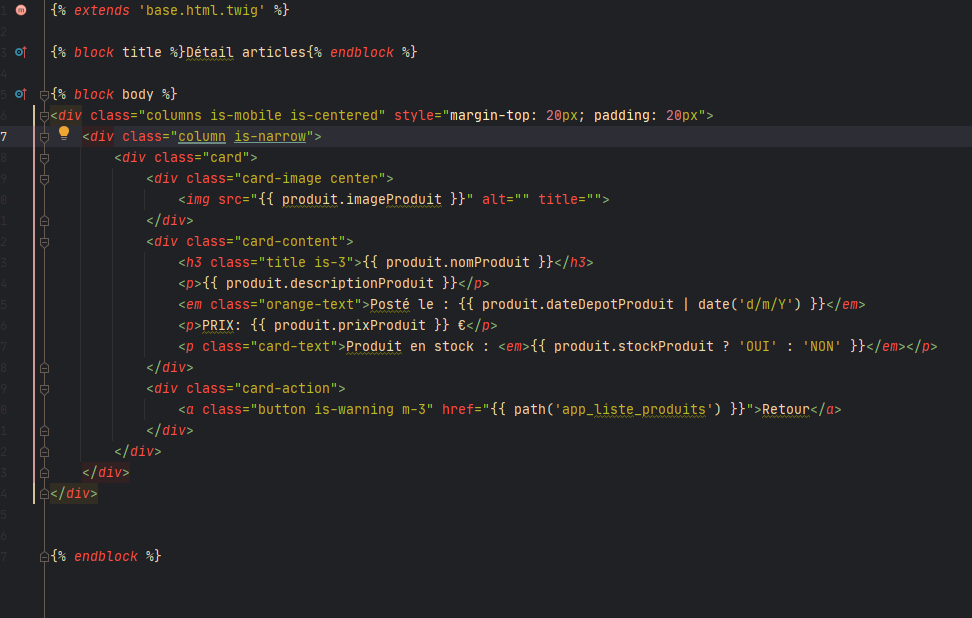
Puis le code de la méthode detailsProduits dans ListeProduitsController

*/\*\**  
 *\* @Route("/produits/{id}", name="app\_details\_produits", methods={"GET"})*  
 *\*/*  
*//En paramètres l'entité Produits*  
*public function* detailsProduits(Produits $produits):Response{  
 *return* $this->render('liste\_produits/detailsProduit.html.twig',[  
 'produit' => $produits  
 ]);  
}

Focus sur la route :

**La vue :**

Ici la cle ‘produit’ sera utilisée dans la vue twig pour afficher les details de chaque produit dans url : exemple http://localhost:8000/produits/21



# F – SUPPRIMER UN PRODUIT

Pour supprimer un produit :

Le controleur a besoin de 4 paramètres

- La classe Request

- L’entité Produits

- Le repository Produits (requète DQL)

- La classe Reponse

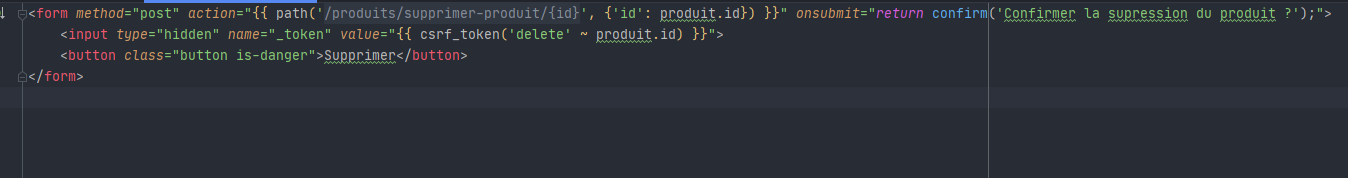
Le controleur va vérifié la validité d’un jeton unique a chaque produits : si ce jeton est valide => on supprime le Produits

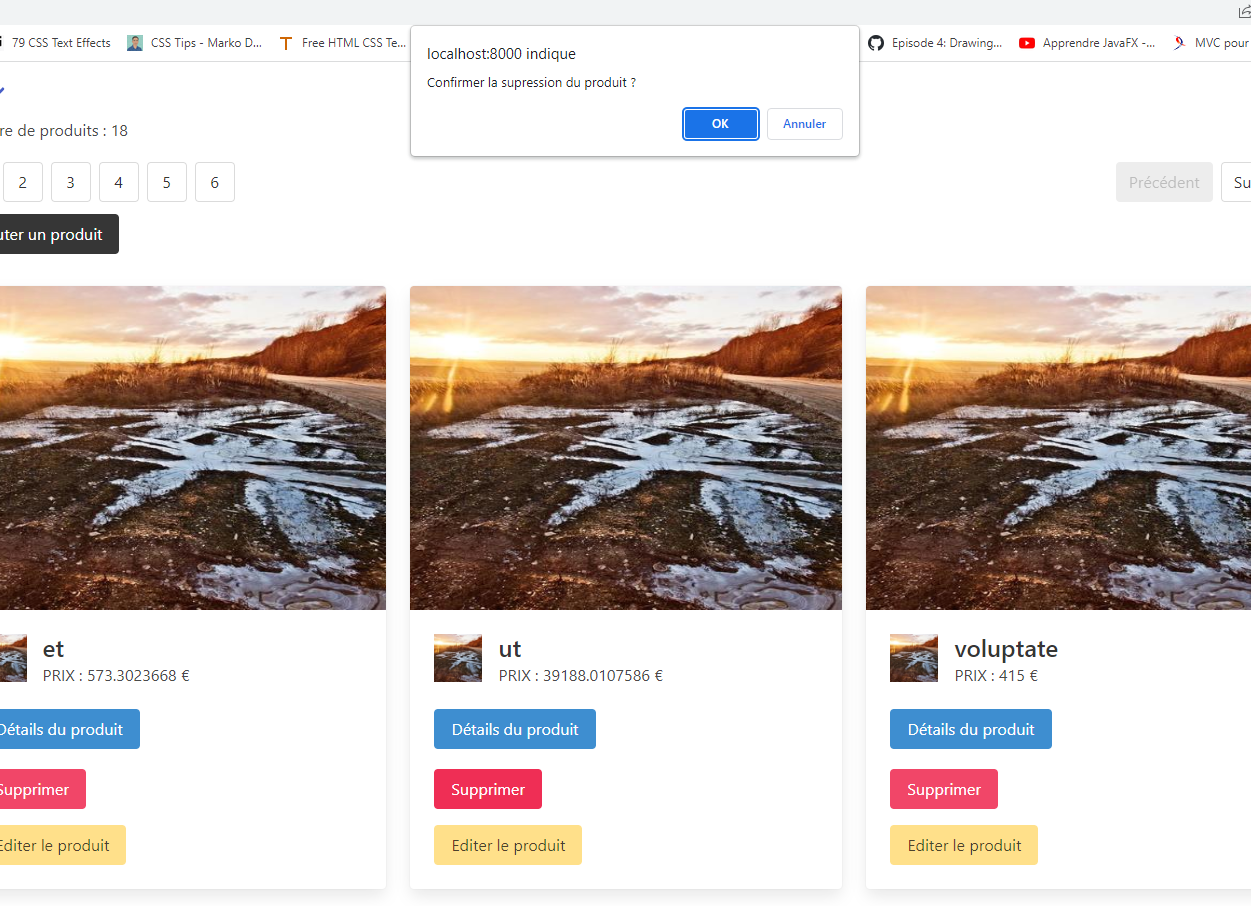
Etapes en 2 temps :

Le controleur check le jeton de la vue \_delete.form.html.twig

Le controleur :

La vue : \_delete.form.html.twig

A le supression une alerte popup est declenchée et demande a l’utilisateur de valider la suppression



# F – AJOUTER UN PRODUIT

Pour appliquer un theme bulma au formulaire :

- Recupere le fichier twig suivant :

- https://raw.githubusercontent.com/dsmink/twig-bulma-form-theme-bundle/master/views/Form/bulma\_0\_3\_x\_layout.html.twig

- Placer le contenu dans un fichier templates/bulma\_form.html.twig

- Ouvrir le fichier : config/packages/twig.yaml

- Ajouter la ligne :

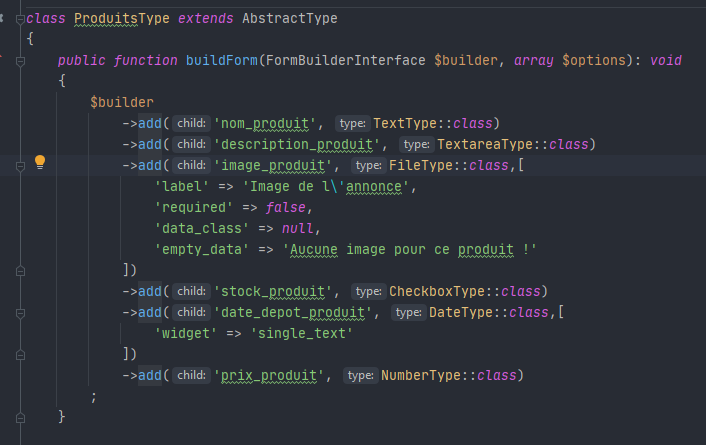
twig:  
 default\_path: '%kernel.project\_dir%/templates'  
 form\_themes: ['bulma\_form.html.twig']

VOS FORMULAIRE ONT TOUS LES CLASSES TWIG

Ajouter a votre fichier src/form/ProduitsType.php ces options pour les dates :

->add('date\_depot\_produit', DateType::*class*,[  
 'widget' => 'single\_text'  
])

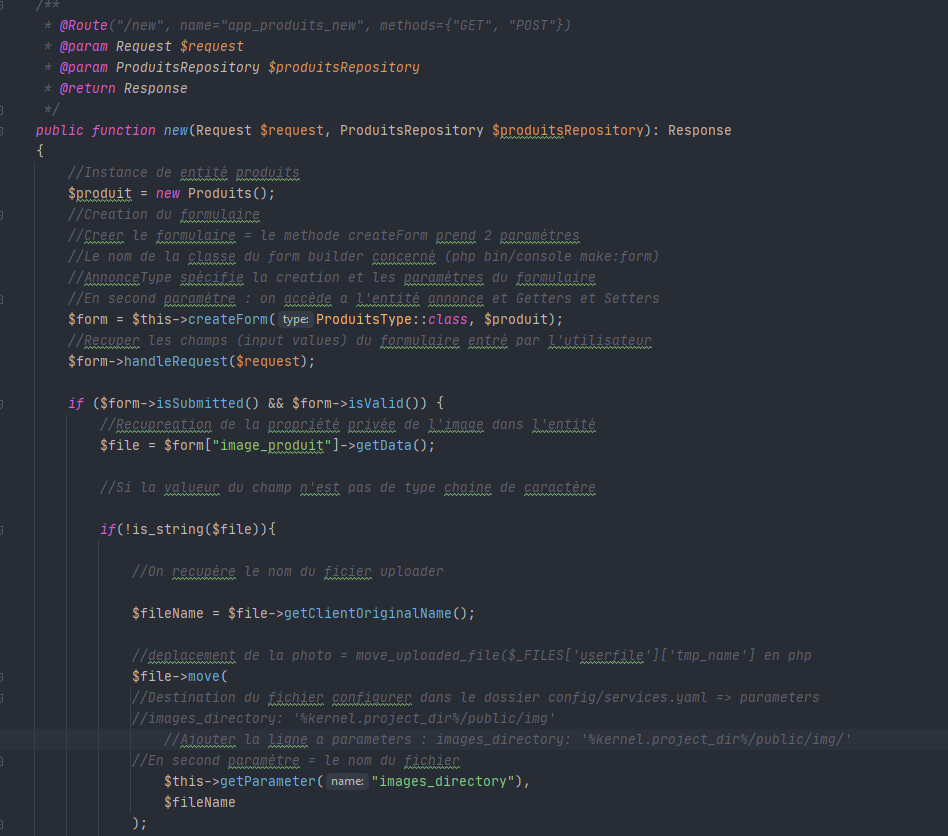
**Ajouter un systeme d’upload de photo au controleur:**

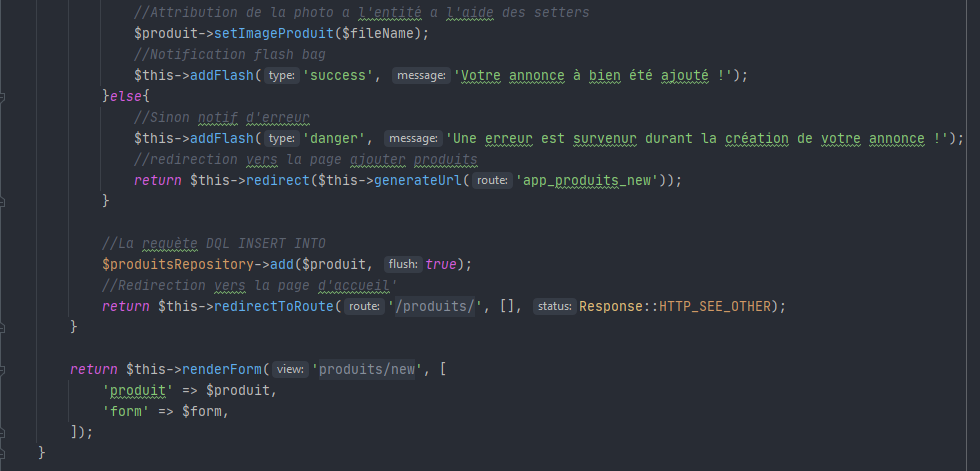
Dans votre fichier src/form/ProduitsType.php

Chaque champs est typé à l’aide des classes core type

*use* Symfony\Component\Form\Extension\Core\Type\

La vue :



Configurer la destination dans votre fichier config/services.yaml

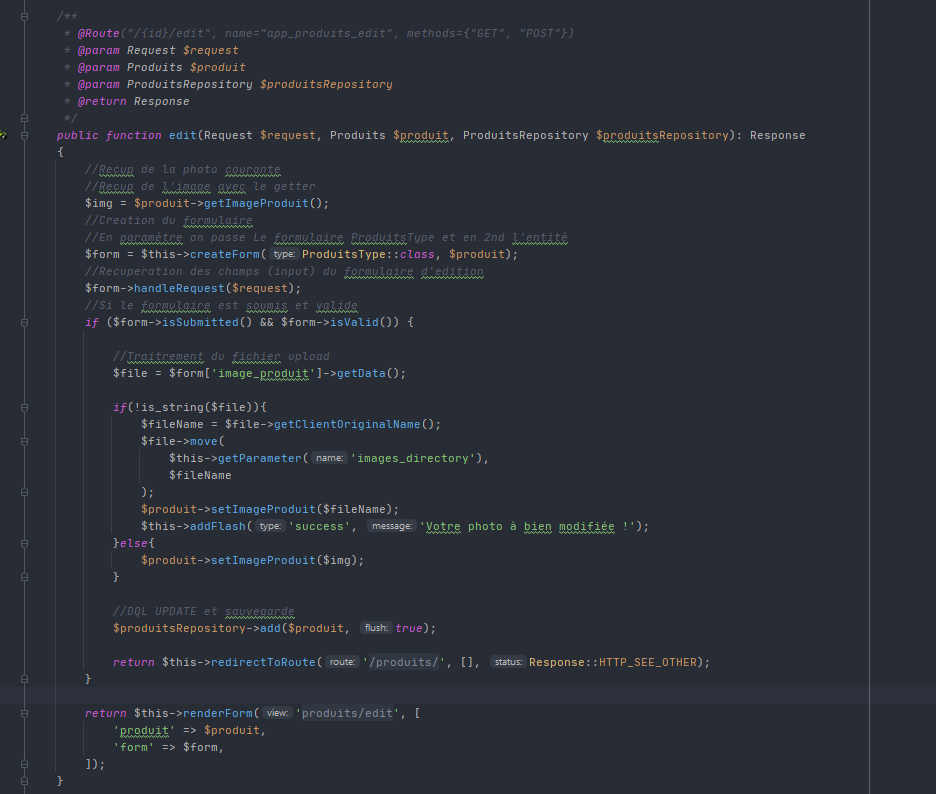
parameters:  
 images\_directory: '%kernel.project\_dir%/public/img/'

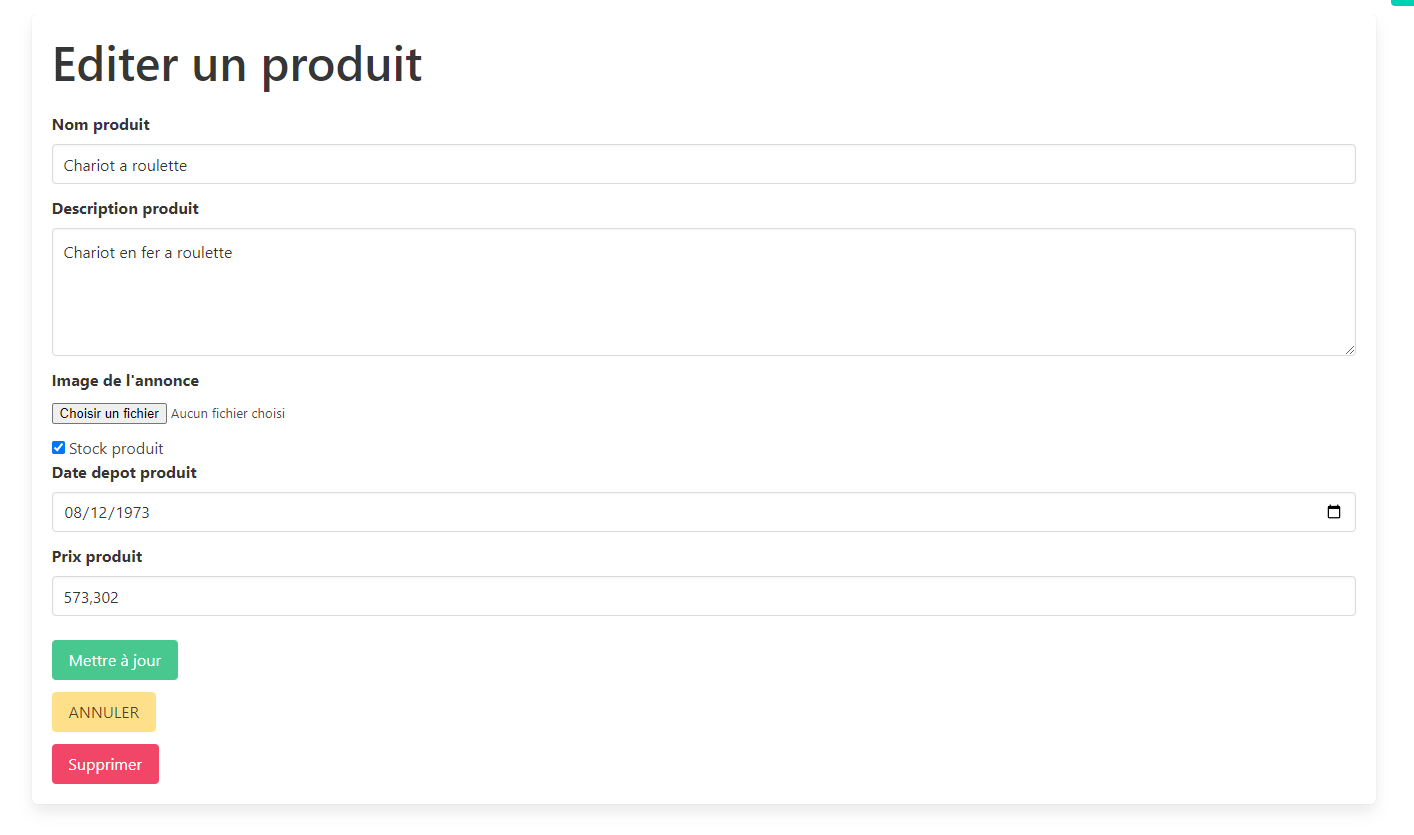
Résultat :



# G – EDITER UN PRODUITS

Mettre a jour le contoleur :





# H – Le langage DQL (Doctrine Query Language)

Similaire à SQL, il s’applique à l’entité et non sur la base de données, l’utilisation se fait via un Repository (generer a la creation de l’entité) à partir de la méthode createQuery ().

$syntaxe = $entityManager->createQuery(‘Requète DQL’) ;

$resultats = $syntaxe->getResult() ;

La méthode getResult () récupère les résultats de la requête : il existe différente méthode

* getSingleResult () = retourne un seul objet (erreur si pas d’objet ou plusieurs)
* getOneOrNullResult() = récupère un objet ou une valeur null (erreur si plusieurs objet)
* GetArrayResult() = retourne les résultats sous forme de tableaux imbriqué et renvoie ArrayCollection (ce dernier est différent de Array PHP, c une classe qui inclus la liste des entités et met à disposition un certain nombre de méthodes)
* GetScalarResult () = retourne des valeurs scalaires qui peuvent contenir des données doubles
* GetOneScalarResult () = retourne une seule valeur scalaire

Exemple de requête DQL tri des produits par ordre décroissant dans Produit Repository :

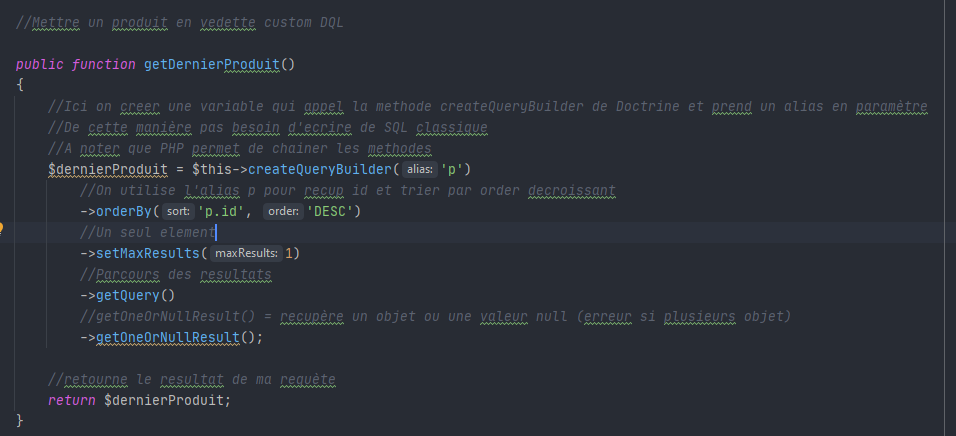
Pour ne pas écrire de SQL en dur on utilise les méthodes de Doctrine Symfony createQueryBuilder ()

Dans le fichier ProduitRepository.php : on ajoute une methode getDernierProduit()

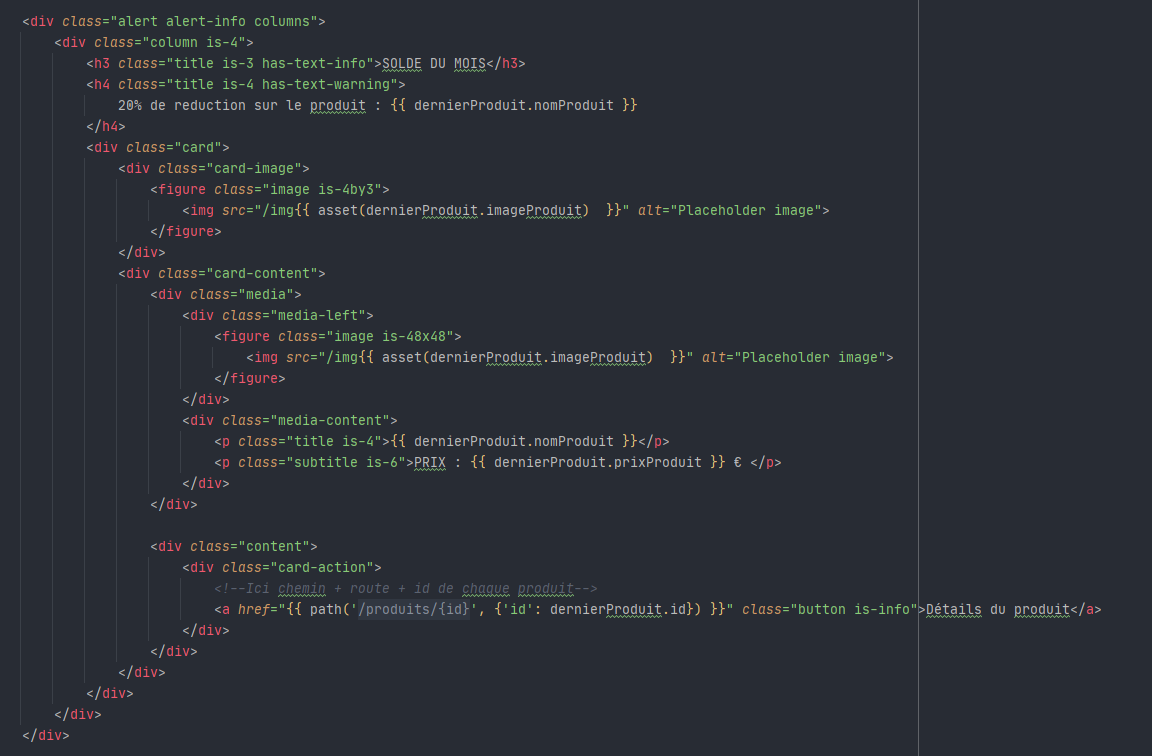
On utilise la methode createQueryBuilder qui prend un alias en paramètre

On effectue un tri decroissant + un nombre de resultat + le query de PDO + retourner un resultat ou rien

Enfin on retourne notre variable $dernierproduit ;

On met a jour le controleur :

Et la vue :



Resultat :



# UN SYSTEME D’INSCRIPTION ET DE CONNEXION SECURISÉ AVEC DES ROLES

3 etapes :

**A – ETAPES 1 : créer la table utilisateurs :**

[https://symfony.com/doc/current/security.html#the-user](https://symfony.com/doc/current/security.html" \l "the-user)

On utilise le security-bundle + symfony Flex

**php bin/console make:user**

Repondre au questions :

- Nom de la table (entité) = User

- Stocker les données en base de données = YES

- Le paramètre de connexion (email, username ou uuid) = email

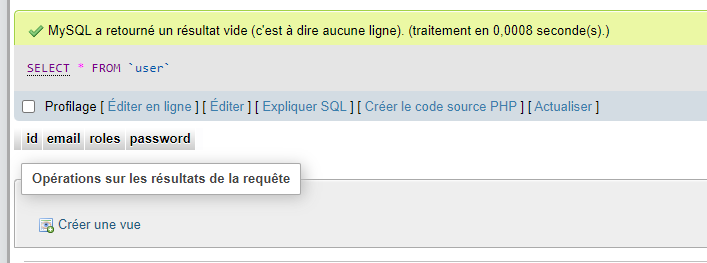
- haché le mot de passe = YES

- Realisé les etapes de migration : php bin/console make:migration

- php bin/console doctrine:migrations:migrate

- Verifié sur phpMyAdmin que votre table User est bien crée

- On constate que la table generée possède un email + mot de passe + rôle



**B- ETAPES 2 : générer un fomulaire et systeme d’inscription**

- A ce stade il est possible de generer un formulaire d’inscription lié a l’entité User

- php bin/console make :registration-form

- Repondre aux questions

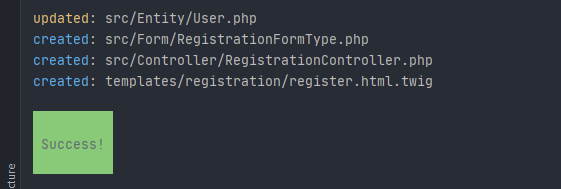
- Ajouter a User un paramètre @UniqueEntity pour eviter la duplication d’email = YES

- Envoyer un email pour valider l’inscription = NO

- Connecter utilisateur apres inscription = NO

-Quelle route pour la redirection apres inscription (page d’accueil) = 0

- La console affiche les fichiers crees et le status



- Entité User a été mise a jour

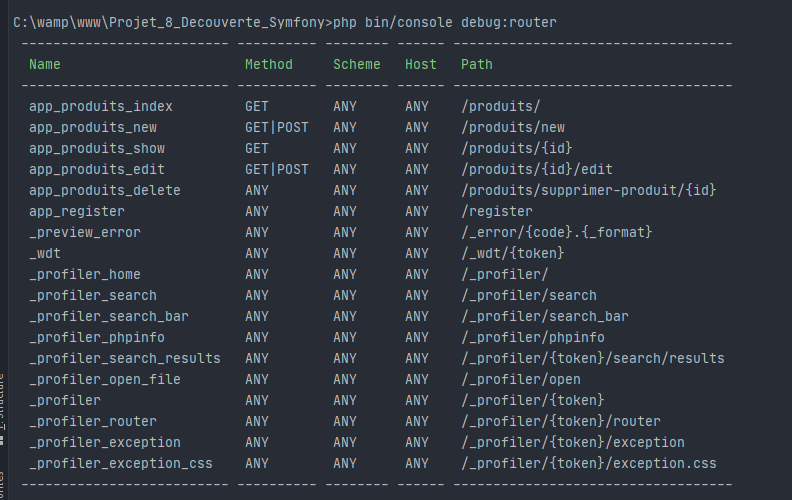
- Un fichier formulaire a été créer

- Un controleur + methode et route a été créer

- Une vue pour afficher le formulaire d’inscription

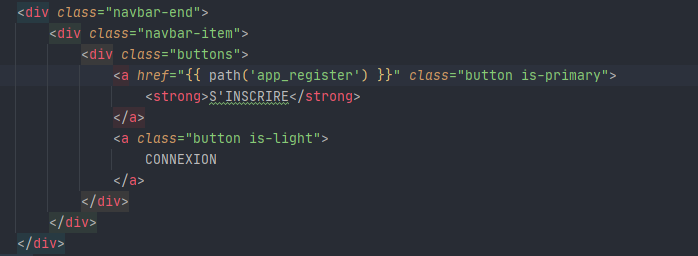
LISTER VOS ROUTES DANS LA CONSOLE

**php bin/console debug:router**

****

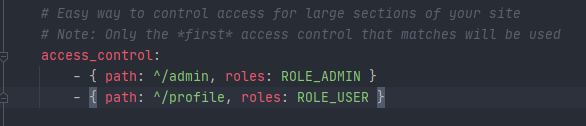
Rendez-vous sur la page : localhost:8000/votre projet /register

Mise a jour de la navbar :

Traduire la vue register.html.twig

Par defaut dans l’entité le rôle de chaque utilisateur est ROLE\_USER

Decommenter les options access\_control dans votre fichier config/packages/secutity/yaml

 **C- ETAPES 3 : généré la connexion securisée**

- https://symfony.com/doc/5.2/security/form\_login\_setup.html

- Suivez les instructions

- **php bin/console make:auth**

- Suivez les instructions et repondez aux questions

- Style de connexion = 1 (Login form Authenticator)

- Le nom de la classe = AppCustomAuthenticator

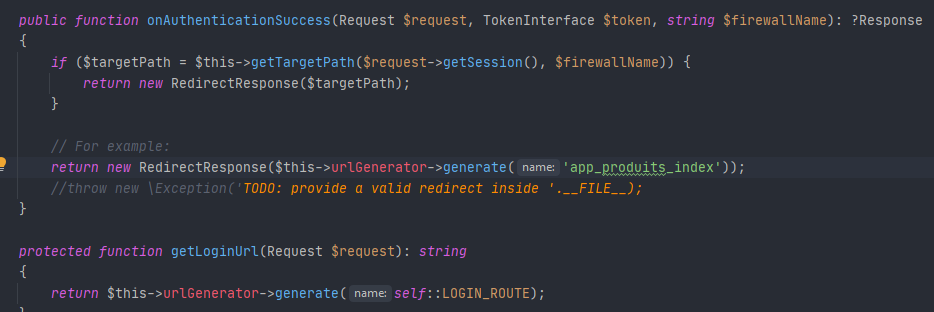
- Le nom du controleur = SecurityController

- Generer un systeme de deconnexion (logout) = yes

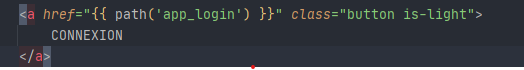
- Il faut maintenant gerer la redirection une fois connecter + le rendu du formulaires

- Dans votre dossier src/security/AppCustomAuthenticator.php

- Modifier la redirection une fois connecter



- Mise a jour du menu :



- Sur la vue : templates/security/login.html.twig



- Tester votre connexion :

- Verifié le status dans le profiler



# G – Les relation entre les entités

Avec SQL - DQL ou QueryBuilder on peut faire des jointures entre les tables, on fait donc appel à des clés étrangères et des table intermédiaires

4 Sortes de jointures :

OneToOne :

* Un enregistrement de la table propriétaire ne peut être lié qu’a un seul enregistrement de la table secondaire (ex : Un produit => une seule référence (n° facture))

OneToMany :

* Un enregistrement de la table propriétaire est lié à plusieurs enregistrements de la table secondaire mais pas réciproquement (ex : Un ordinateur => PC, MAC, Pc portable, etc…)

ManyToOne :

* Autant de relation que l’on souhaite entre la table propriétaire et la table secondaire et réciproquement (ex : Spaghetti => Butonni, Panzani, Barilla, top budget, Casino, etc…)

ManyToMany :

* Les relations sont complètes, autant de relation que l’on souhaite entre les deux tables (ex : Produits => marques)

# K – UNE RELATION ONE TO ONE

- Pour illustrer une relation One to One nous allons créer une table (entité) References

- LES CARDINALITES

- Un produit possède une et une seule reference et une reference appartient a un seul produit

-Dans la console :

**php bin/console make:entity**

- Nom de l’entité = References

- Un champ numero\_reference de type integer

- Realiser la migration : php bin/console make:migration

- Valider la migration : php bin/console doctrine:migrations:migrate

- Verifier sur PhpMyAdmin la creation de la tableaux

METTRE A JOUR LA TABLE (entité) PRODUITS

**php bin/console make:entity**

- Enter le nom de l’entité existante Produits

- Ajouter un champ numero

- Le type est relation

- Cette relation est lié a la classe References

- Quelle type de relation ? = OneToOne

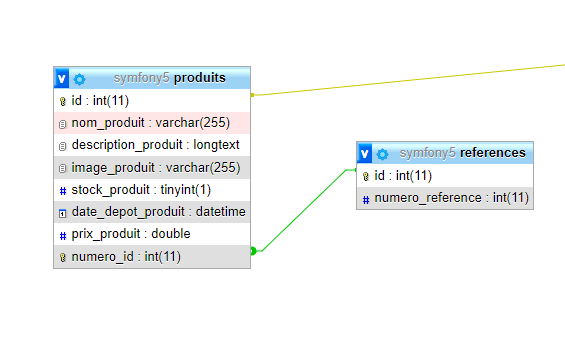
- La jointure Produits.numero peut il etre null = no

- Voulez vous ajouter une propriete a Reference pour acceder au Produits = no

- Executer de nouveau la migration de l’entité Produits : php bin/console make:migration

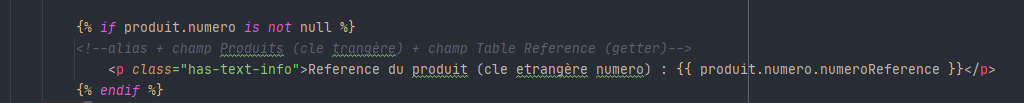
- Valider la migrations : php bin/console doctrine:migrations:migrate

- Verifié l’opration dans PhpMyAdmin :



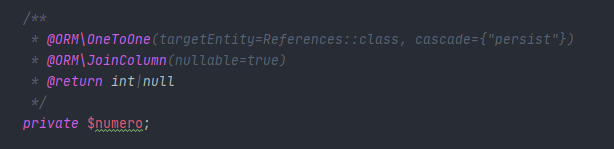
- Afficher la cle etrangere dans la vue

- Cette etape consiste a appeler votre alias de la boucle for + le getter de la cle etrangère + le champ de la table reference



- Dans l’entité Produits mis a jour  on constate des annotation reciproque:

Produits.php

Dans la vue index.html.twig

{% *if* produit.numero *is not* null %}  
*<!--alias + champ Produits (cle trangère) + champ Table Reference (getter)-->*  
<p *class*="has-text-info">Reference du produit (cle etrangère numero) : {{ produit.numero.numeroReference }}</p>  
{% *endif* %}

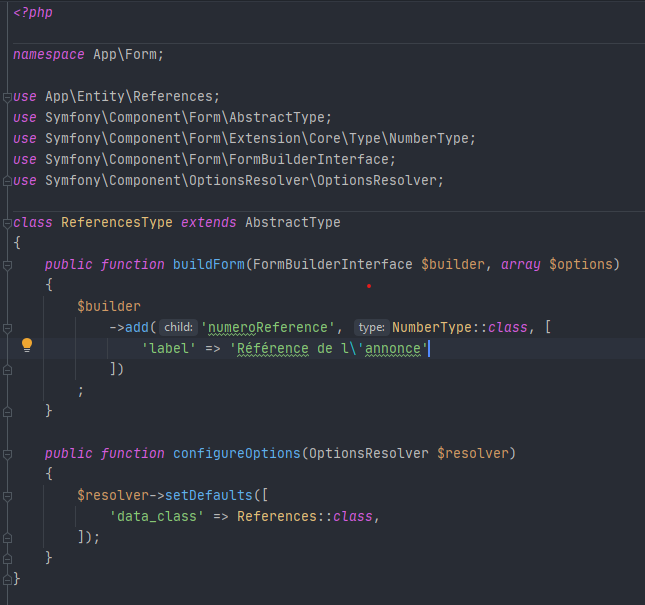
Le resultat :



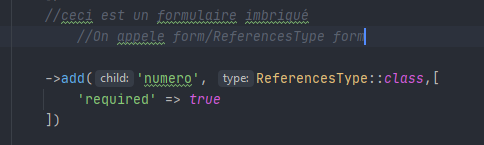
Pour ajouter un produit : il faut mettre a jour notre fichier ProduitsType

On utilise ici des formulaire imbriquer **(c’est a dire un formulaire pour la reference est imbiqué dans le formulaire produits)**

- Dans votre dossier src/form : créer une classe ReferencesType.php



- Ensuite ajouter ce champ a src/form/ProduitsType.php



Lors de l’ajout d’un produits : nous avons un nouveau champs



# L – Une relation Many To Many et les table intermediare

- Ajouter une nouvelle entité Distributeur :

**php bin/console make:entity**

- Ajouter une propriété nom\_distributeur : string varchar 255

- la migration : **php bin/console make :migration**

- Puis : **php bin/console doctrine:migrations:migrate**

- Verifié le resultat dans phpMyAdmin

- Modifier l’entité Produits (on a joute une cle etrangère distributeur)

**php bin/console make:entity Produits**

- Ajouter le champs distributeur : type relation

- On lie l’entité Produits a l’ entité Distributeurs

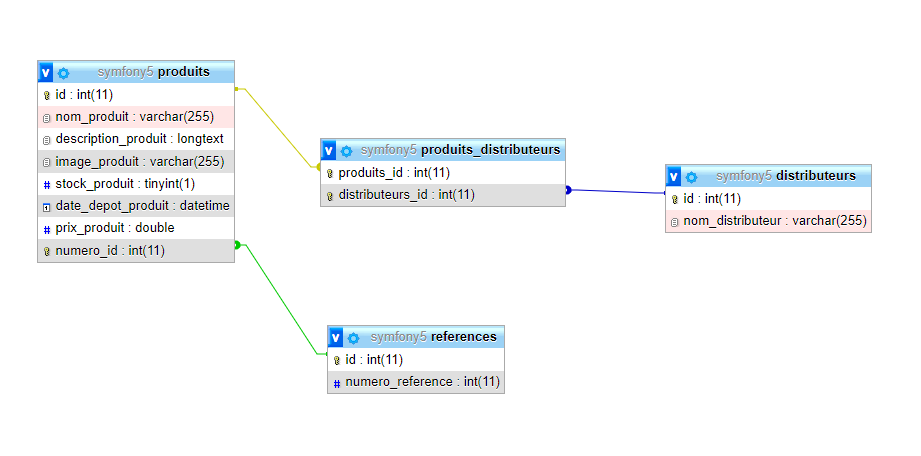
- Le type de relation est ManyToMany

- Ajouter une propriete a Distributeurs pour acceder aux Produits = YES

**php bin/console make :migration**

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

- Ces opérations ont pour effet de créer une table intermédiare entre Produits et Distibuteurs

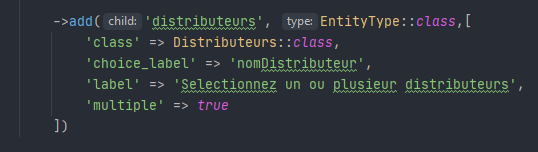


Pour ajouter un produit, il faut evidement mettre a jour votre formulaire ProduitsType :

- Le type de champs est EntityType

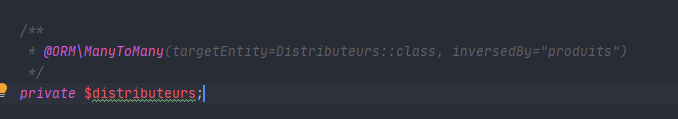
- On ajoute la jointure ManyToMany via la cle ‘class’ => Entité Distributeurs

- Ce champs est un select => option Multiple HTML

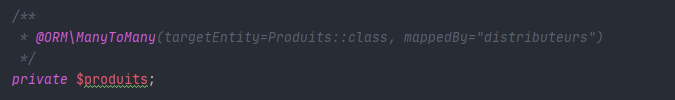


- Au niveau des entités les annontations utilise une relation (jointure) reciproque

Produits.php



- Idem pour l’entité Distributeurs.php

Pour la vue :

Ainsi la table produits\_distributeurs (la table intermediare entre produits et distributeurs) s’occupe de faire les jointures entre les 2 cles primaires

Dans la vue index.html.twig

*<!--Plusieur distributeur donc boucle for + champs table Distributeurs = alias + getter-->*  
{% *for* distributeur *in* produits.distributeurs %}  
 <p *class*="has-text-danger">Distributeur(s) :{{ distributeur.nomDistributeur }}</p>  
{% *endfor* %}

# M– Une relation One To Many et Many To One

- On ajoute a nos produits une categories

- Creer une entité Categories avec les champs suivants :

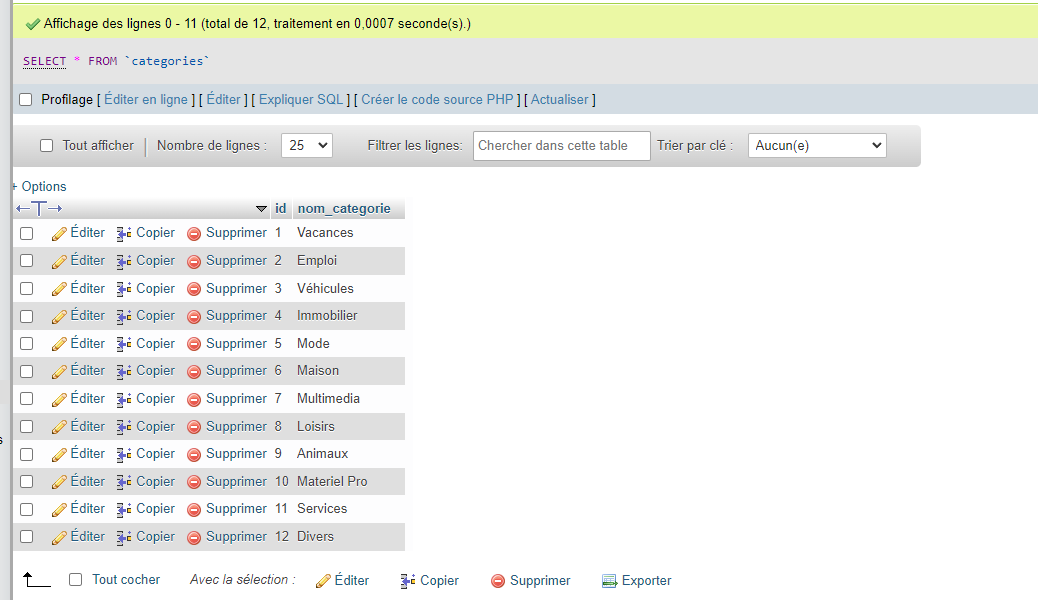
**php bin/console make :entity Categories**

- Ajouter le champ nom\_categorie string 255

**php bin/console make:migration**

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

- Verifié le resultat dans PhpMyAdmin et ajouter quelques catégories



- Ajouter une cle etrangère a l’entité Produits de type relation OneToMany

LES CARDINALITES

Plusieurs produits ont Une et Une seule Catégories (ManyToOne) et reciproquement Une Categorie appartient a UN ou PLUISEUR Produits (OneToMany)

**php bin/console make:entity Produits**

- Ajouter un champ categories

- Type relations

- Lie a l’entité Categories

- Relation ManyToOne (la reciprocité est automatique)

- Produits.categorie peux etre null = no

- Categories peux acceder a produit = yes

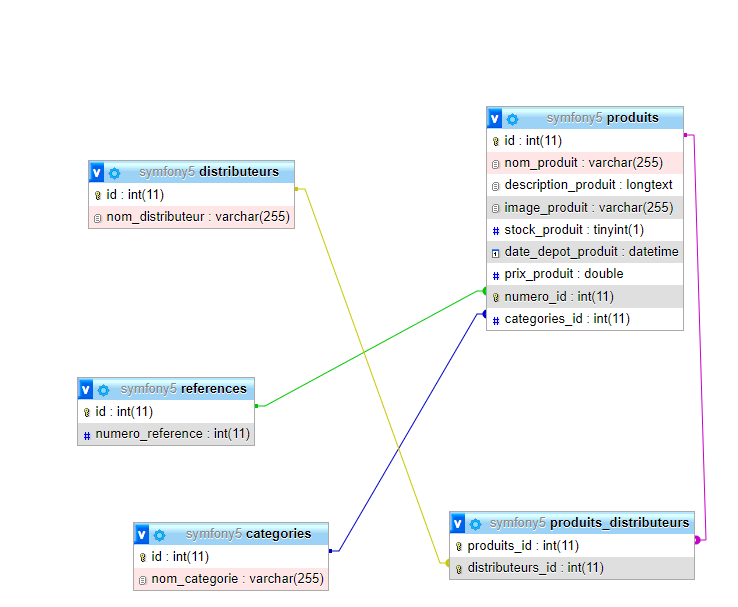
-Categories.produits

- Supprimer automatiquement les orphelins = no

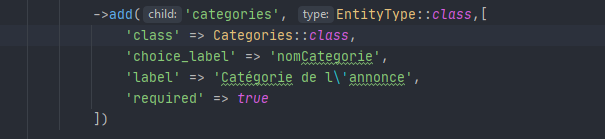
**php bin/console make:migration**

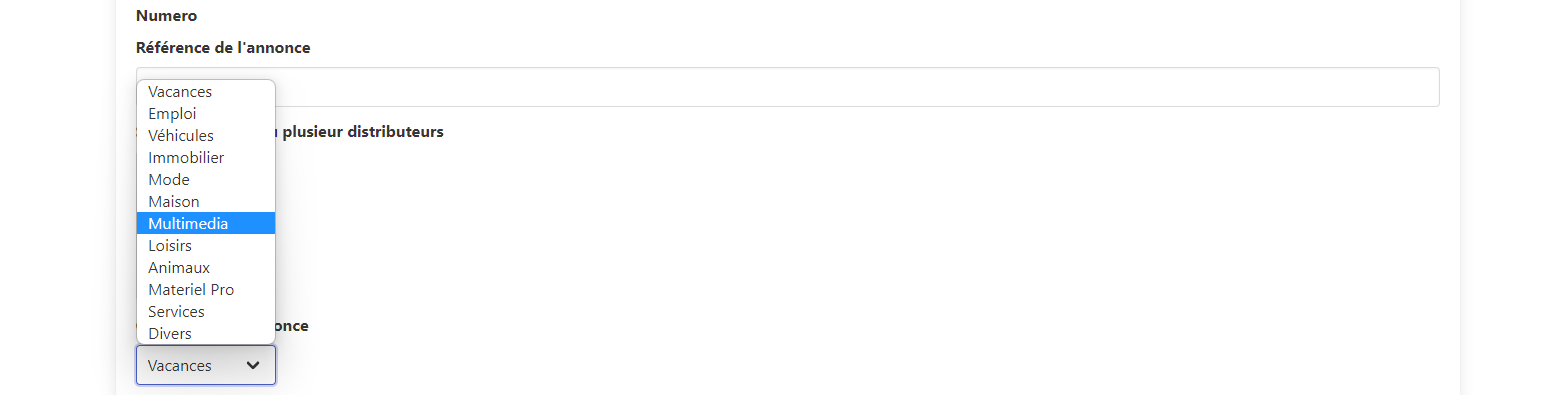
**php bin/console doctrine:migrations:migrate**

- Verifié le resultat sur PhpMyAdmin :



Mettre a jour votre formulaire ProduitsType.php :

La vue :

Dans votre page Twig :

*<!--alias + champ Produits (cle trangère) + champ Table Categories (getter)-->*  
<p *class*="has-text-warning">Categorie : (cle etrangère ManyToOne & OneToMany) : {{ produit.categories.nomCategorie }}</p>

# N – Le Reverse Engineering :

Si vous utilisé une base de données déjà existante il sera fastidieux de créer à la main toutes les entités, pour cela Symfony a prévu un cas :

Créer les entités automatiquement :

La commande suivante : **php bin/console doctrine :mapping :import ‘’App\Entity’’ annotation –path=src/Entity/Reverse**

Dans le dossier src/Entity ce créera un dossier Reverse qui contiendra toutes les entités générées.

Les jointures sont détectées

Il faudra régénérer les entité : php bin/console make :entity –regenerate App

# **ADMINISTRATION : LE BUNDLE EASY ADMIN**

- https://symfony.com/bundles/EasyAdminBundle/current/index.html

- Le bundle EasyAdmin créer des backends d’adminsitration pour votre site web rapidement

- il est important de comprendre que le bundle ne faut tous et que certaines etapes devront etre realise a la maintenant

- Installer le bundle

**composer require easycorp/easyadmin-bundle**

**-** Generer un tableau de bord pour les administrateurs

**php bin/console make:admin:dashboard**

**-** Le nom du controller = DashboardController

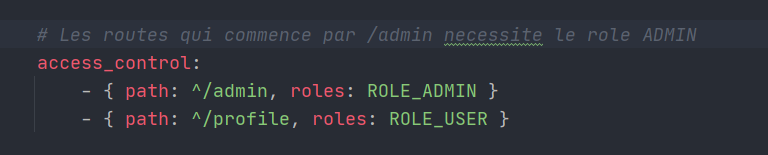
- Le repertoire = src/Controller/Admin

- Creer un utilisateur a l’aide de votre formulaire d’inscription

- Dans PhpMyAdmin : modifier son rôle

- [‘’ ROLE\_ADMIN ‘’] = Attention ici le format est du Json, il faut imperativement mettre des guillemet double

- Dans votre fichier config/packages/security.yaml : les routes /admin demande le rôle Admin pour etre accessibles

- Connecter vous avec compte administrateur et verifier le status



**-** Rendez-vous sur l’URL suivate : localhsot:8000/admin

- Nous allons generer un CRUD Admin pour chaque tableaux

**php bin/console make:admin:crud**

- Choisissez l’entité sur laquel vous souhaité réalisé le CRUD

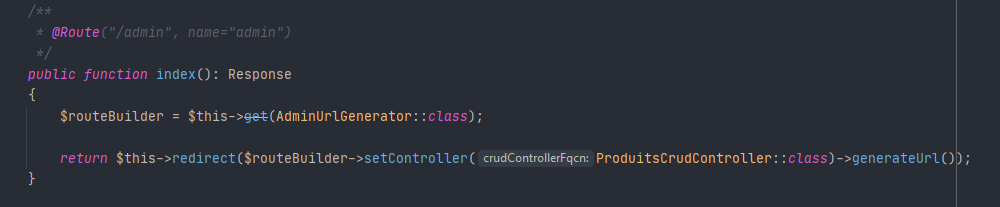
- Le dossier est src/Controller/Admin/ crud de chaque entité

- Idem pour le namespace

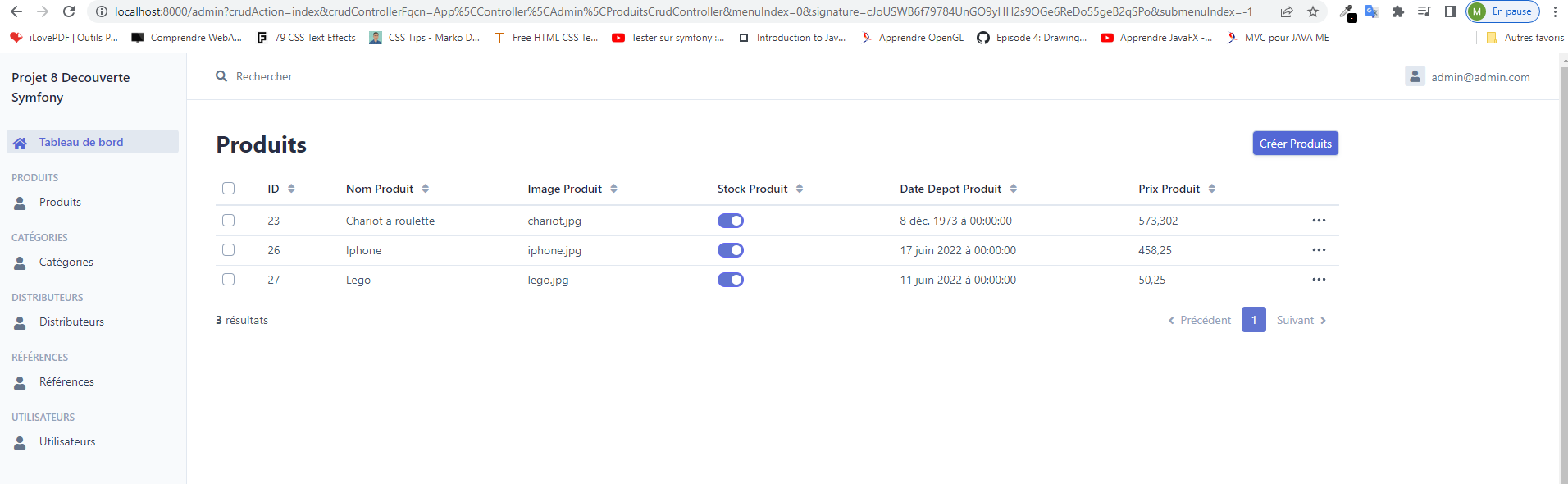
**Rendez-vous dans votre fichier src/Controller/Admin/DashboardController.php**

Dans la methode configureMenuItems() : Creer un tableau et appeler vos entités

Par defaut la page d’accueil doit etre modifier pour laisser aparaitre votre table Produits

On appel la classe AdminUrlGenerator et on lui stipule que la page d’accueil est votre controlleur Produits

Résultat :

CUSTOM EASY ADMIN etc,,,,,,