

## Projet 7 : Découverte du Framework PHP Symfony 5

Ressources :

<https://github.com/michelonlineformapro/Projet-7-Refonte-Annonces-Symfony-5>

Les + :

<https://github.com/michelonlineformapro/Projet-7-B-Sf5-EasyAdmin-Ajax>

DEFINITION :

**Symfony** est un ensemble de composants PHP c'est un Framework MVC libre écrit en PHP. Il fournit des fonctionnalités modulables et adaptables qui permettent de faciliter et d'accélérer le développement d'un site web.

### Objectif :

Le but de ce projet est de procéder à une refonte de votre projet 6 : petite annonce 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 éditer et les supprimer

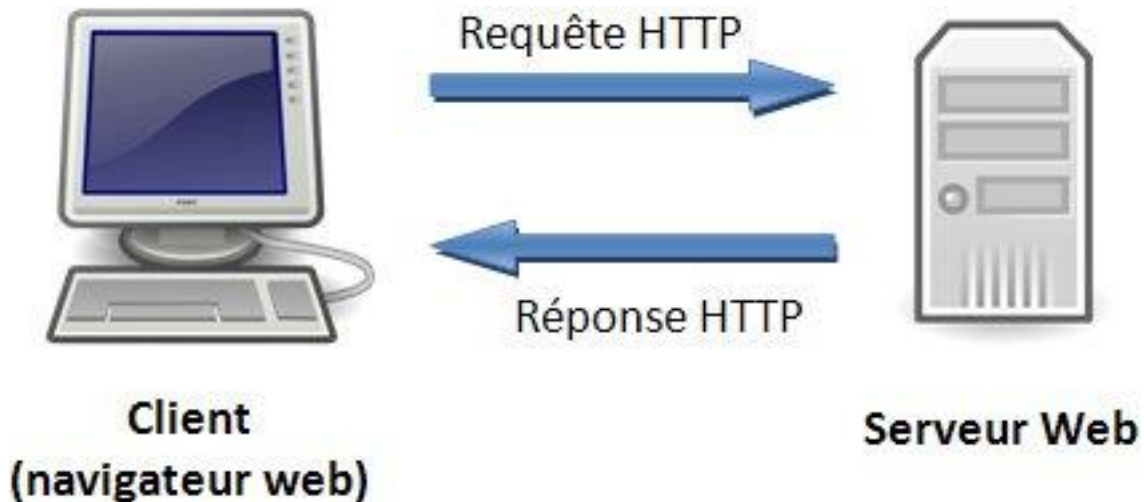
#### A - Avant-Propos :

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 tout maîtriser (Symfony = assemblage de module complexe)
- 3- Savoir trouver l'information sur internet : (Taper les mots clés, communauté Symfony, vérifier la configuration du projet, version du Framework, date de la solution, etc...)  
Symfony.com, php.net, stackoverflow, tuto, coach, Symfony Cast, Packagist, etc...
- 4- Être organisé : MCD, Diagramme de séquence, etc...
- 5- Ne pas chercher à tout développer 😞 Ne pas réinventer la roue, un bon IDE, étape par étape)

## B – Rappel protocole http



Exemple :

- 1) Une url : <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**, 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

### C QUOI ?:

Symfony est un ensemble de composant lié par un noyau (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 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 ?

- 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> -> 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 1<sup>er</sup> mars 2012.

Toutes les instructions de font en ligne de commande.

Composer require quelquechose

<https://getcomposer.org/>

## 2 - Installer Symfony 5 :

- a- La Version minimal de PHP : 7.2.5
- b- <https://symfony.com/download> setup.exe
- c- Vérifié sa version : cmd -> `symfony -v`
- d- Pour une version Full : `symfony new --full my_project`
- e- Pour une version Light (microservice, application console et API):  
`symfony new my_project`
- f- Ou via composer : `composer create-project symfony/website-skeleton nomApplication`
- g- Lancer le server local symfony : cmd -> `symfony server :start` ou `symfony serve` (`symfony server :stop` pour l'arrêter)
- h- URL (uniform resources locator) = `localhost :8000` ou `127.0.0.1 :8000`
- i- 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.

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 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 qui sont utilisés dans votre projet, ce sont de fichier au format Twig (générateur de Template)

- Le dossier "translations"

Symfony fournit un composant appelé Translation capable de gérer de nombreux formats de traductions, dont les formats yaml, xlf, 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 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.

<https://flex.symfony.com/>

## SYMFONY UTILISE :

Le composant Dependency Injection et notamment comment construire des objets et les récupérer à l'aide du container de services :

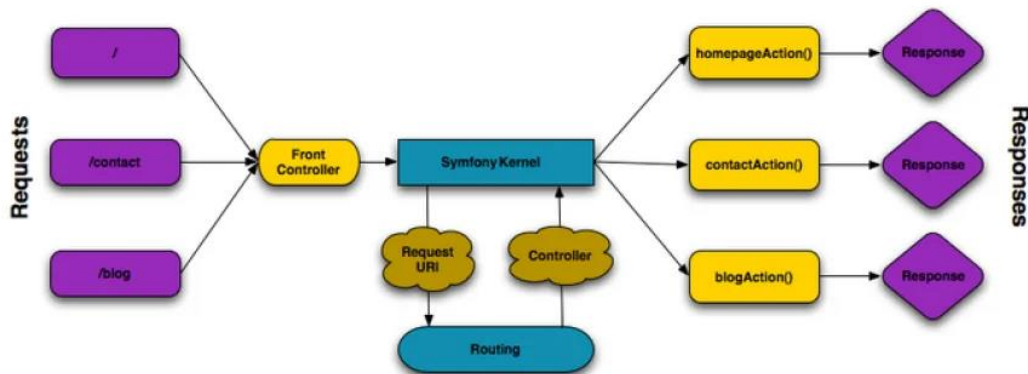
PHP bin/console debug :container = (ensemble des services de l'application)

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.

## An introduction to Symfony



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](#).

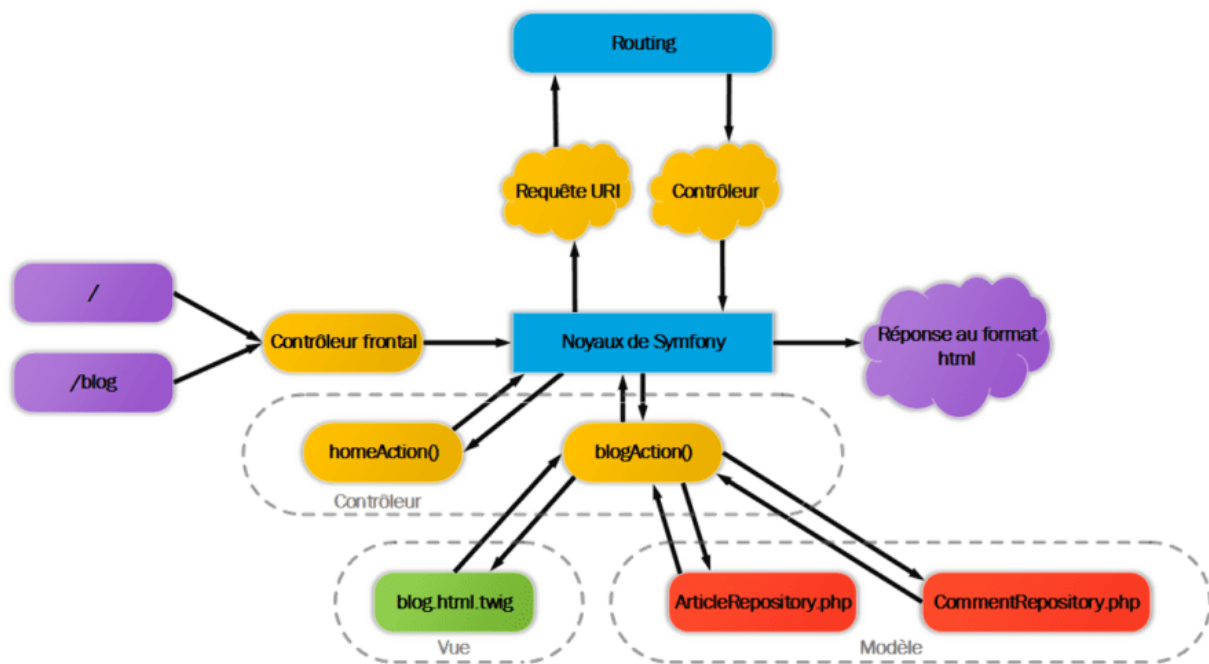
Doc Yaml :

[https://symfony.com/doc/current/components/yaml/yaml\\_format.html](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.

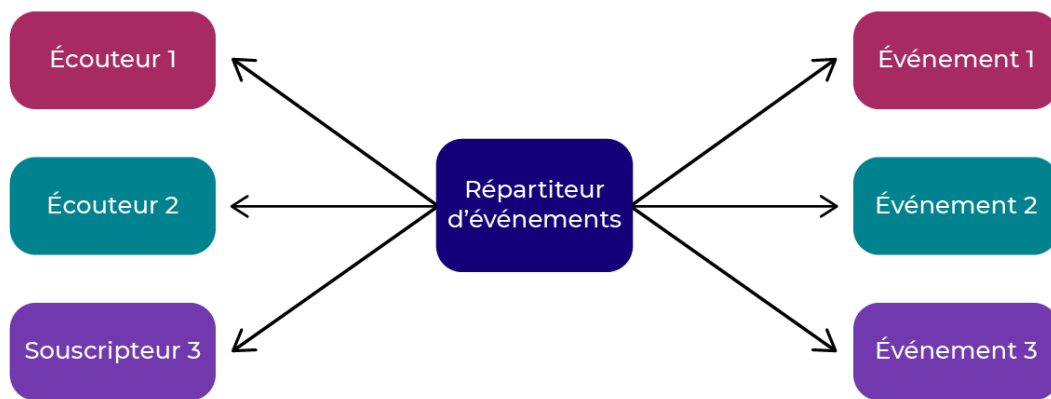


- Grâce à l'*autowiring*, l'essentiel du temps, nous n'avons rien de spécial à faire pour que nos objets soient automatiquement retrouvés par le container et accessibles dans nos services et nos contrôleurs.
- L'*autoconfiguration* permet d'ajouter des tags à nos services s'ils implémentent une interface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

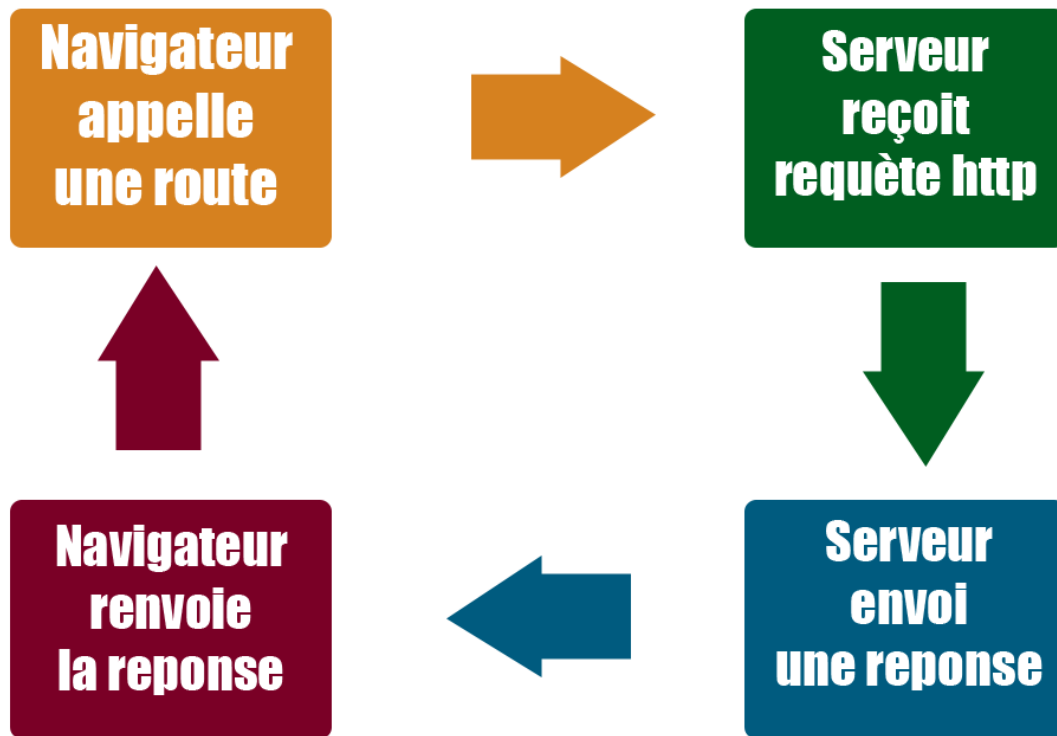
## A – Configurer une application

- a- Dans PHPStorm `file -> setting -> plugins -> Symfony Supports`
- b- 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
- c- 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 à retourner au client, partie visuelle regroupée dans le dossier Template avec extension .html.twig
- 4- Controller = Programme PHP qui coordonne l'application, ils appellent les modèles et les vues à retourner (\$request + \$response)

## LA LOGIQUE DU 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 makeBundle  
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 dispo depuis maker-bundle :

```
php bin/console list make
```

a- Créer un Controller en ligne de commande :

```
php bin/console make :controller
```

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

## Produit Controller

```
.env x ProduitsController.php x
1 <?php
2
3 namespace App\Controller;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
6 use Symfony\Component\HttpFoundation\Response;
7 use Symfony\Component\Routing\Annotation\Route;
8
9 class ProduitsController extends AbstractController
10 {
11     /**
12      * @Route("/produits", name="produits")
13      */
14     public function index(): Response
15     {
16         return $this->render( view: 'produits/index', [
17             'controller_name' => 'ProduitsController',
18         ]);
19     }
20 }
```

Explication :

- 1- Le namespace src/produits/ProduitsController
- 2- Import (use = héritage multiple) Classes AbstractController + 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`

9- 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` ces composant `$request` et `$response` sont contenus dans la bibliothèque `HttpFoundation`

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.

```

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class ProduitsController extends AbstractController
{
    /**
     * @Route("/produits", name="produits")
     * @param Request $request
     * @return Response
     */
    public function index(Request $request): Response
    {
        echo $request->getPathInfo();
        return $this->render( view: 'produits/index', [
            'controller_name' => 'ProduitsController',
        ]);
    }
}

```

Ici echo retourne le chemin défini dans @Route soit /Produits

DE MANIERE GENERALE ON PEUT DIRE QUE :

Les classes :

- Request = \$\_POST en PHP (soumission des formulaires)
- Query = \$\_GET en PHP (paramètre transmis dans l'adresse de la requête)
- Cookies = \$\_COOKIE en PHP
- Files = \$\_FILES en PHP relative au fichier transmis
- 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 requête&statut=message

```
/**
 * @Route("/produits", name="produits")
 * @param Request $request
 * @return Response
 */
public function index(Request $request): Response
{
    echo $request->query->get( key: 'info');
    return $this->render( view: 'produits/index', [
        'controller_name' => 'ProduitsController',
    ]);
}
```

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 :

```
public function index(Request $request): Response
```

Où :



```

class ProduitsController extends AbstractController
{
    /**
     * @Route("/produits", name="produits")
     * @param Request $request
     * @return Response
     */
    public function index(Request $request)
    {
        $response = new Response( content: "Salut c michael OLFP");
        return $response;
    }
}

```

Tout à 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 :

```

class ProduitsController extends AbstractController
{
    /**
     * @Route("/produits", name="produits")
     * @param Request $request
     * @return Response
     */
    public function index(Request $request)
    {
        $response = new Response();
        $response->setContent( content: "Salut c MIC");
        $response->headers->set( key: 'Content-Type', values: 'application/json');
        $response->setStatusCode( code: Response::HTTP_OK);
        $response->setCharset( charset: 'utf-8');
        return $response;
    }
}

```

Un exemple de Session et de redirection :

```

<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class ProduitsController extends AbstractController
{
    /**
     * @Route("/produits", name="produits", methods={"GET","POST"})
     * @param Request $request
     * @return Response
     */
    public function index(Request $request):Response
    {
        $session = $request->getSession(); // ceci = session_start()
        $session->set('nom_utilisateur', 'Michael Michel');
        return $this->redirectToRoute( route: 'redirection');
    }

    /**
     * @Route("/produits/redirection", name="redirection")
     * @param Request $request
     * @return Response
     */
    public function redirection(Request $request){
        //Recup de la session
        $session = $request->getSession();
        $nomUtilisateur = $session->get( name: 'nom_utilisateur');
        return new Response( content: "Ici la redirection + nom de session utilisateur $nomUtilisateur");
    }
}

```

## 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.

```

class ProduitsController extends AbstractController
{
  /**
   * @Route("/produits", name="produits", methods={"GET","POST"})
   * @param Request $request
   * @return Response
   */
  public function index(Request $request):Response
  {
    $session = $request->getSession(); // ceci = session_start()
    $session->getFlashBag()->add( type: 'info', message: 'message info');
    $session->getFlashBag()->add( type: 'info', message: 'un autre message');
    $session->set('nom_utilisateur', 'Michael Michel');
    return $this->redirectToRoute( route: 'redirection');
  }

  /**
   * @Route("/produits/redirection", name="redirection")
   * @param Request $request
   * @return Response
   */
  public function redirection(Request $request){
    //Recup de la session
    $session = $request->getSession();
    $info = $session->getFlashBag()->get( type: 'info');
    $affiche = '';
    foreach ($info as $message) {
      $affiche .= $message.'<br />';
    }
    $nomUtilisateur = $session->get( name: 'nom_utilisateur');
    return new Response( content: "message : $affiche $nomUtilisateur");
  }
}

```

#### 4 - LE SYSTEME DE ROUTE :

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

```

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

a- 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>

```
/**  
 * @Route("/produits/{nom}/{prenom}", name="produits", methods={"GET","POST"})  
 * @param Request $request  
 * @return Response  
 */  
public function index(Request $request, $nom, $prenom):Response  
{  
    $nom = "mic";  
    $prenom = "michel";  
}
```

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

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

```
/**  
 * @Route("/produits/{nom}/{prenom}", name="produits", methods={"GET","POST"})  
 * @param Request $request  
 * @return Response  
 */  
public function index(Request $request, $nom = 'LAGADEK', $prenom = 'BOB'):Response  
{  
    return new Response( content: "Bonjour $nom $prenom");  
}
```

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 Contrôleur 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

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

{% block title %}Hello ProduitsController!{% endblock %}

{% block body %}
<style>
  .example-wrapper { margin: 1em auto; max-width: 800px; width: 95%; font: 18px/1.5 sans-serif; }
  .example-wrapper code { background: #F5F5F5; padding: 2px 6px; }
</style>

<div class="example-wrapper">
  <h1>Hello {{ controller_name }}! </h1>

  This friendly message is coming from:
  <ul>
    <li>Your controller at <code><a href="{{ 'C:/wamp64/www/symfony5/src/Controller/ProduitsController.php'|file_link(0) }}">src/Controller/ProduitsController.php</a></code></li>
    <li>Your template at <code><a href="{{ 'C:/wamp64/www/symfony5/templates/produits/index.html.twig'|file_link(0) }}">templates/produits/index.html.twig</a></code></li>
  </ul>
</div>
{% endblock %}
```

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 héritées de `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 (utilisable sans Symfony)

La syntaxe :

`{{...}}` = Afficher le contenu d'une variable ou le résultat d'une expression = Interpolation (issue de mustache, handleBar, Js, etc...)

{% ... %} = Exécuter une structure de contrôle (if, foreach, etc...)

{# ... #} = 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>
    {% endif %}
{% endblock %}
```

```
{% 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>`

Lister vos routes : `php bin/console debug :router`

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'apparaît qu'en mode dev (définie dans .env)



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 recettes (recipes) qui sont des dépôts du site :

<https://flex.symfony.com/> (comme <https://packagist.org/> )

Toutes les dépendances Symfony se situent 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 ):

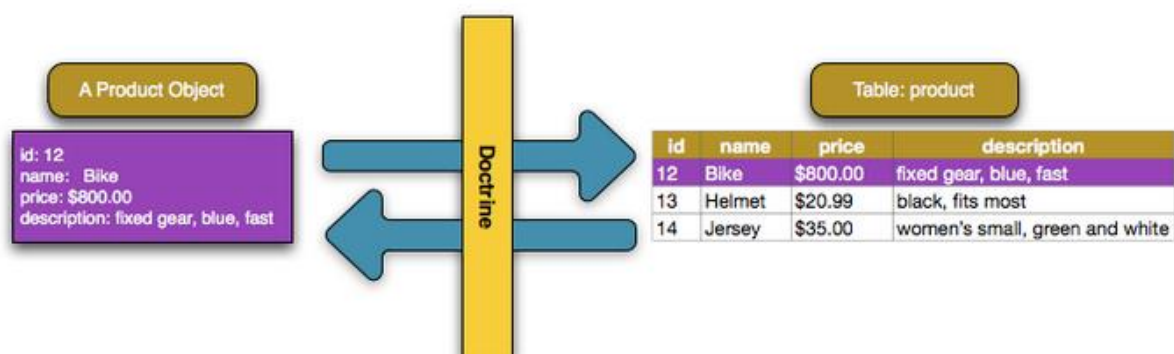
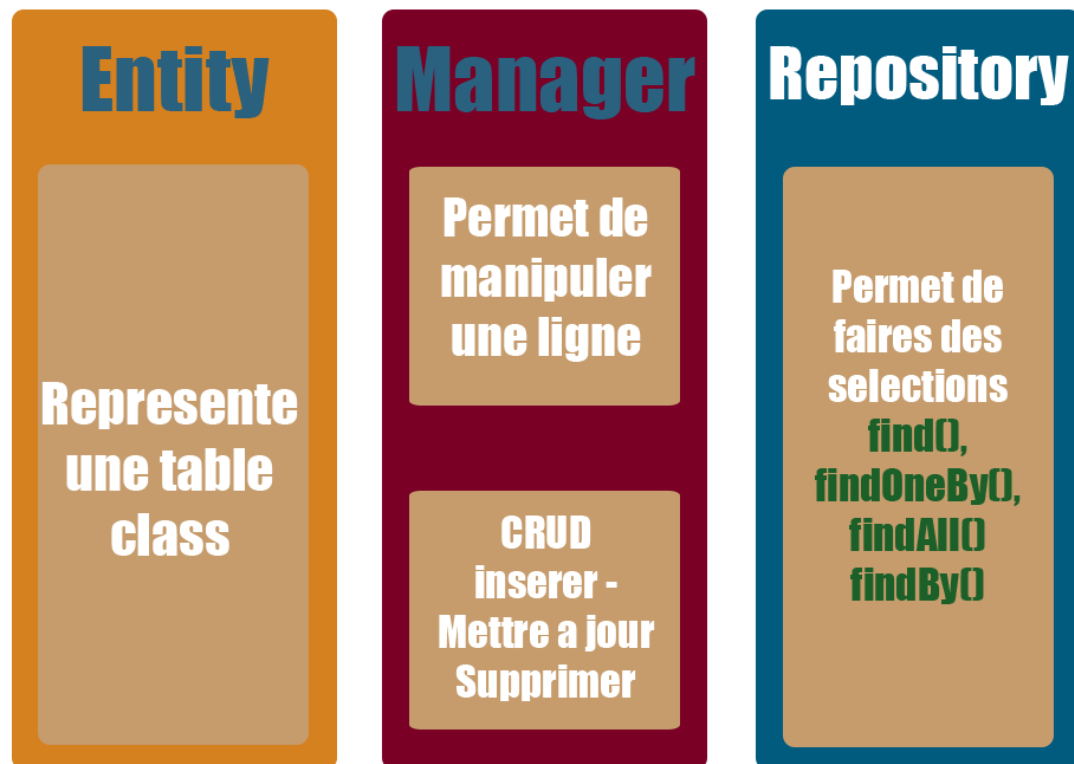
CRUD = Create Read Update Delete

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 :

### DOCTRINE ORM SYMFONY Object Relational Mapping



## B – Utiliser MySQL :

Dans le dossier .env décommenter la ligne suivante :

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

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)

## 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)

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)
 */
```

Décrit le chemin vers le fichier Produit Repository qui fera les requête 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 = chaîne de caractère 255 octets

Etc...

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

## B – Les migrations

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

Ex :

```
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 du SQL classique, les méthodes down et up permettent 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 1<sup>er</sup> version
- prev = Migrez avant la version précédente
- next = Migrez vers la prochaine version
- latest = Migrez à la dernière version

## 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`

Les champs : `nomArticle` 255 string + `contenuArticle` text + `imageArticle` 255 string + `auteurArticle` 255 string + `dateArticle` 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/ArticlesFixture.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

## METHODE 2 : FICHIER EXTERNE

Nous allons charger nos fixtures depuis un fichier externe : Créer un dossier src/Data/ListeProduit.php

Exemple de jeux de données :

```
class ListeProduits
{
    static $mesProduits = [
        ["nomProduit" => "Imprimantes Canon", "prixProduit" => 700.25,
"quantiteProduit" => 15, "rupture" => false],
        ["nomProduit" => "Mario Bros", "prixProduit" => 80.55, "quantiteProduit"
=> 11, "rupture" => false],
        ["nomProduit" => "I-phone 5", "prixProduit" => 1200.75, "quantiteProduit"
=> 5, "rupture" => false],
        ["nomProduit" => "Table bois", "prixProduit" => 40.95, "quantiteProduit" =>
1, "rupture" => false],
        ["nomProduit" => "Souris Logitech", "prixProduit" => 20.25,
"quantiteProduit" => 7, "rupture" => false],
    ];
}
```

Il faut insérer ce code dans ProduitFixtures.php

- 1 - Instance de l'entité (classe produit)
- 2 – Insertion des propriétés avec les setters setNomProduit()
- 3 – Grace a Doctrine et son ObjectManager on va persister les données
- 4 – Ensuite on enregistre avec la méthode flush()
- 5 – Pour prendre en compte toutes les données on réalise une boucle foreach()

```
<?php
namespace App\DataFixtures;

use App\Data>ListeProduits;
use App\Entity\Produit;
```



```

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

class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        //Boucle de parcours des valeur du fichier src/Data/ListeProduit.php
        foreach (ListeProduits::$mesProduits as $monProduit){
            //Instance de entité (classe produit)
            $produit = new Produit();
            //Ajout des données depuis le fichier src/Data/ListeProduit.php
            $produit->setNomProduit($monProduit['nomProduit']);
            $produit->setPrixProduit($monProduit['prixProduit']);
            $produit->setQuantiteProduit($monProduit['quantiteProduit']);
            $produit->setRupture($monProduit['rupture']);
            //Pesistence des donnée grace a Doctrine ObjectManager
            $manager->persist($produit);
        }
        //Enregistrement des données
        $manager->flush();
    }
}

```

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

## 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 ListeProduit`

Pour récupérer les données on utilise Doctrine EntityManager et le ProduitRepository

```

<?php

namespace App\Controller;

use App\Entity\Produit;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

```

```

use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class ListeProduitsController extends AbstractController
{
    /**
     * @Route("/liste", name="liste_produits")
     */
    public function index(): Response
    {
        //Appel de Doctrine EntityManager et ses methodes
        $entityManager = $this->getDoctrine()->getManager();
        //Stock et appel du Repository
        //Depuis entityManager on appelle la methode get et on passe l'entité en
paramètres
        $produitRepository = $entityManager->getRepository(Produit::class);
        //Creation de liste (Dans produit Repo on accède au methode find()
findOneBy() findAll() et findBy()
        $listeProduit = $produitRepository->findAll();
        //Appel de la vue et passage des paramètres dans un tableau associatif cle +
valeur accessible depuis la vue grace a Twig
        return $this->render('liste_produits/index.html.twig', [
            //Rappel du controleur (optionel)
            'controller_name' => 'ListeProduitsController',
            //Ici la cle (listeProduit) sera accessible dans la vue index.html.twig grace
a l'interpolation {{listeProduit}}
            'listeProduit' => $listeProduit
        ]);
    }
}

```

Ici twig fait de l'interpolation et récupère les Getters

```
{{produit.nomProduit}}
```

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

```

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

{% block title %}Hello ListeProduitsController!{% endblock %}

{% block body %}
<div class="row">

```

```

    {% for produit in listeProduits %}
    <div class="col-md-4 col-sm-12 mt-3">
      <div class="card" style="width: 18rem;">
        
        <div class="card-body">
          <h5 class="card-title">{{ produit.nomProduit }}</h5>
          <p class="card-text"><b>PRIX : {{ produit.prixProduit }} €</b></p>
          <p class="card-text"><b>QUANTITE : {{ produit.quantiteProduit
        }}</b></p>
          {% if produit.rupture %}
          <p class="card-text"><b>Produit en rupture de stock</b></p>
          {% endif %}
          <a href="#" class="btn btn-primary">Go somewhere</a>
        </div>
      </div>
    </div>
    {% endfor %}

  </div>

{% endblock %}

```

On utilise donc une instruction Twig avec une boucle for et le paramètres 'liste Produit' du tableau associatif du contrôleur `{% for produit in listeProduit%} {% endfor %}`

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

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

POUR LES ARTICLES :

```

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

{% block title %}SF5 -ARTICLES-{% endblock %}

{% block body %}
  {% for article in articles %}
    <ul class="list-group mt-3">
      <li class="list-group-item active">{{ article.nomArticle }} du
: {{ article.dateArticle | date('s/m/Y')}} à {{ article.dateArticle |

```

```

date('H:i') }}</li>
        <li class="list-group-item">Auteur : <b>{{
article.auteurArticle }}</b></li>
        <li class="list-group-item">Posté le : {{ article.dateArticle |
date('s/m/Y')}} à {{ article.dateArticle | date('H:i') }}<em></em></li>
        <li class="list-group-item"><a href="{{
path('lire_article',{'slug': article.nomArticle , 'id': article.id}) }}"
class="btn btn-outline-warning">Lire l'article</a></li>
    </ul>
    {% endfor %}
{% endblock %}

```

## 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/twitter_bootstrap_v4_pagination.html.twig' #
            ici le template bootstrap 4
            sortable: '@KnpPaginator/Pagination/sortable_link.html.twig' # sort
            link template
            filtration: '@KnpPaginator/Pagination/filtration.html.twig' # filters
            template

```

Dans le controleur, methode de liste de articles

```

/**
 * @Route("/articles", name="articles")
 */
public function index(ArticlesRepository $articlesRepository,
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
        $articlesRepository->findAll(),
        //On liste par entier (knp_paginator.yaml) on definit la cle dans
        url, par défaut 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' => $articlesRepository->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 passer le Framework en fr :

Config/packages/translation.yaml :

```
framework:  
    default_locale: fr  
    translator:  
        default_path: '%kernel.project_dir%/translations'  
        fallbacks:  
            - fr
```

Effacer le cache au cas :

La commande : php bin/console cache:clear

## PARAM CONVERTER : {id} = find(\$id) = getByid

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

On passe dans l'URL un {slug} et un {id}

Pour le Slug : <https://github.com/cocur/slugify>

Le slug sera égal au titre de l'article.

Puis le code de la méthode lireArticle de ArticlesController

```
/**  
 * @param Articles $articles  
 * @return Response  
 * @Route("/lire_article/{slug}/{id}", name="lire_article")  
 */  
  
public function lireArticle(Articles $articles): Response{  
    return $this->render('articles/detailsArticle.html.twig', [  
        'article' => $articles  
    ]);  
}
```

Focus sur la route :

```
* @Route("/lire_article/{slug}/{id}", name="lire_article")
```

## E – Mettre à jour l'entité Produits :

Ajouter un champ photoProduit a notre entité de nouveau entrée :

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

Puis le même nom de l'entité : Produit

Ajouter le champ photoProduit string 255 nullable = no

De nouveau :

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

Puis : 

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

Le champ a été ajouté mais il faut hydrater (rafraichir) l'entité et ajouté les getters et setters si ça n'est pas fait automatiquement.

Sur PHP Storm : click droit + generate + Getters and Setters...

Mettre à jour les fixtures :

```
Dans AppFixtures.php :  
$produit->setPhotoProduit($monProduit['photoProduit']);
```

Et dans src/Data/ListeProduit :

```
<?php  
  
namespace App\Data;  
  
class ListeProduits  
{  
    static $mesProduits = [  
        ["nomProduit" => "Imprimantes Canon", "prixProduit" => 700.25,  
        "quantiteProduit" => 15, "rupture" => false, "photoProduit" => "imp.jpg"],  
        ["nomProduit" => "Mario Bros", "prixProduit" => 80.55, "quantiteProduit"  
=> 11, "rupture" => false, "photoProduit" => "mario.jpg"],  
        ["nomProduit" => "I-phone 5", "prixProduit" => 1200.75, "quantiteProduit"  
=> 5, "rupture" => false, "photoProduit" => "phone.jpg"],  
        ["nomProduit" => "Table bois", "prixProduit" => 40.95, "quantiteProduit" =>  
1, "rupture" => false, "photoProduit" => "table.jpg"],  
        ["nomProduit" => "Souris Logitech", "prixProduit" => 20.25,
```

```
"quantiteProduit" => 7, "rupture" => false, "photoProduit" => "souris.jpg"],  
];  
}
```

Pour finaliser l'opération : `php bin/console doctrine :fixture :load`

Dans la vue index.html.twig : Afficher les photos

```

```

## F – 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 à partir de la méthode `createQuery()`.

```
$syntaxe = $entityManager->createQuery('Requête DQL');
```

```
$resultats = $syntaxe->getResult();
```

La méthode `getResult()` récupère les résultats de la requête : il existe différentes méthodes

- `getSingleResult()` = retourne un seul objet (erreur si pas d'objet ou plusieurs)
- `getOneOrNullResult()` = récupère un objet ou une valeur null (erreur si plusieurs objets)
- `GetArrayResult()` = retourne les résultats sous forme de tableaux imbriqués et renvoie `ArrayCollection` (ce dernier est différent de `Array` PHP, c'est une classe qui inclut 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 `ProduitRepository` :

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

Dans le fichier `ProduitRepository.php` créer :

```

/**
 * @return mixed
 * @throws \Doctrine\ORM\NonUniqueResultException
 */

public function getDernierProduit()
{
    //Ici on creer une variable qui appelle la methode createQueryBuilder de
    Doctrine et prend un alias en paramètre
    //De cette manière pas besoin d'ecrire de SQL en DUR
    //A noter que Symfony permet de chainer les methodes
    $dernierProduit = $this->createQueryBuilder('p')
        //On utilise l'alias p pour recuperer id et trier par ordre decroissant
        ->orderBy('p.id', 'DESC')
        //Un seul element
        ->setMaxResults(1)
        //Parcours des resultats
        ->getQuery()
        //getOneOrNullResult() = recupère un objet ou une valeur null (erreur si
        plusieurs objet)
        ->getOneOrNullResult();

    //retourne le resultat de ma requête
    return $$dernierProduit;
}

```

Il faut dans le contrôleur, ajouter cette méthode créée :

```

$listeProduit = $produitRepository->findAll();
$dernierProduit = $produitRepository->getDernierProduit();

```

Puis dans le tableau de résultat

```

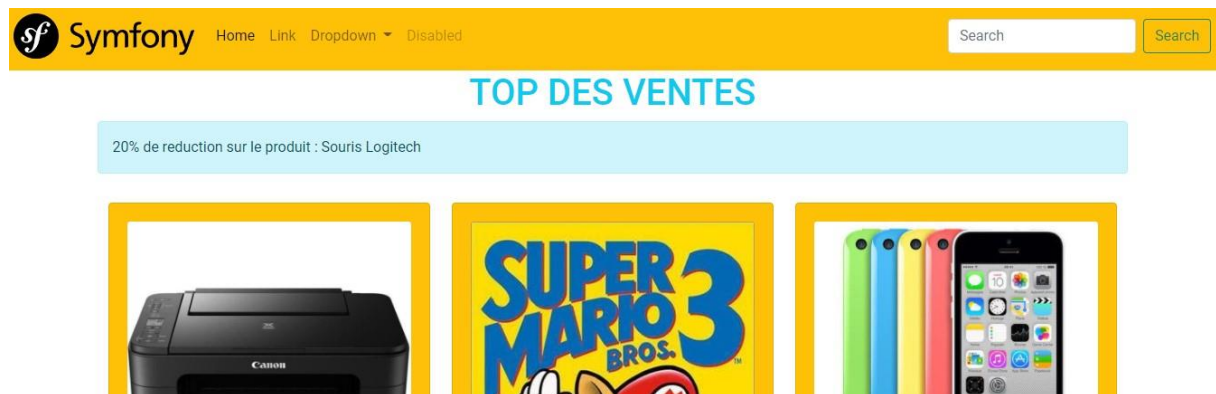
return $this->render('liste_produits/index.html.twig', [
    //Rappel du controleur (optionel)
    'controller_name' => 'ListeProduitsController',
    //Ici la cle (listeProduit) sera accessible dans la vue index.html.twig grace a
    l'interpolation {{listeProduit}} c'est un ArrayCollection
    'listeProduits' => $listeProduit,
    'dernierProduit' => $dernierProduit
]);

```

Dans la vue on récupère le dernier produit ajouter :



```
<div class="alert alert-info">20% de reduction sur le produit : {{  
dernierProduit.nomProduit }}</div>
```



## G – Les relation entre les entités

Avec le 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'à 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)

H – Une nouvelle entité : Références

Créer une nouvelle entité : `php bin/console make :entity`

Reference + champ numero + integer + nullable : no

`php bin/console make:migration`

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

Mise en place de la relation OneToOne : Dans l'entité Produit.php sous \$photoProduit

```
/**
 * @ORM\OneToOne(targetEntity="App\Entity\Reference", cascade={"persist"})
 * @ORM\JoinColumn(nullable=true)
 * @return int|null
 */
private $reference;
```

targetEntity cible la table (entité) inverse à l'aide de son namespace (ici : reference)

Cascade indique qu'il n'est pas nécessaire de persister les objets de l'entité référence qui seront joint à l'entité principale

Rappel dans phpMyAdmin :

@ORM\JoinColumn (au singulier) à remplacé @ORM\Column est true des le depart pour ne pas avoir erreur avec les données déjà présente dans la table Produit (rappel int = 0 pour le champs reference\_id dans entité produits)

Generer les getters et les setters : `php bin/console make :entity --regenerate` App (ou phpStorm click droit + generate + getter and setters...

De nouveau : `php bin/console make:migration` et `php bin/console doctrine:migrations:migrate`

Vérifié dans phpMyAdmin le résultat entre produit et référence :

Contraintes de clé étrangère

Actions	Propriétés de la contrainte	Colonne	Contrainte de clé étrangère (INNODB)		
			Base de données	Table	Colonne
Supprimer	FK_29A5EC271645DEA9 ON DELETE RESTRICT ON UPDATE RESTRICT	reference_id	olfp_sf5	reference	id
+ Ajouter une colonne					
Nom de la contrainte ON DELETE RESTRICT ON UPDATE RESTRICT + Ajouter une colonne					

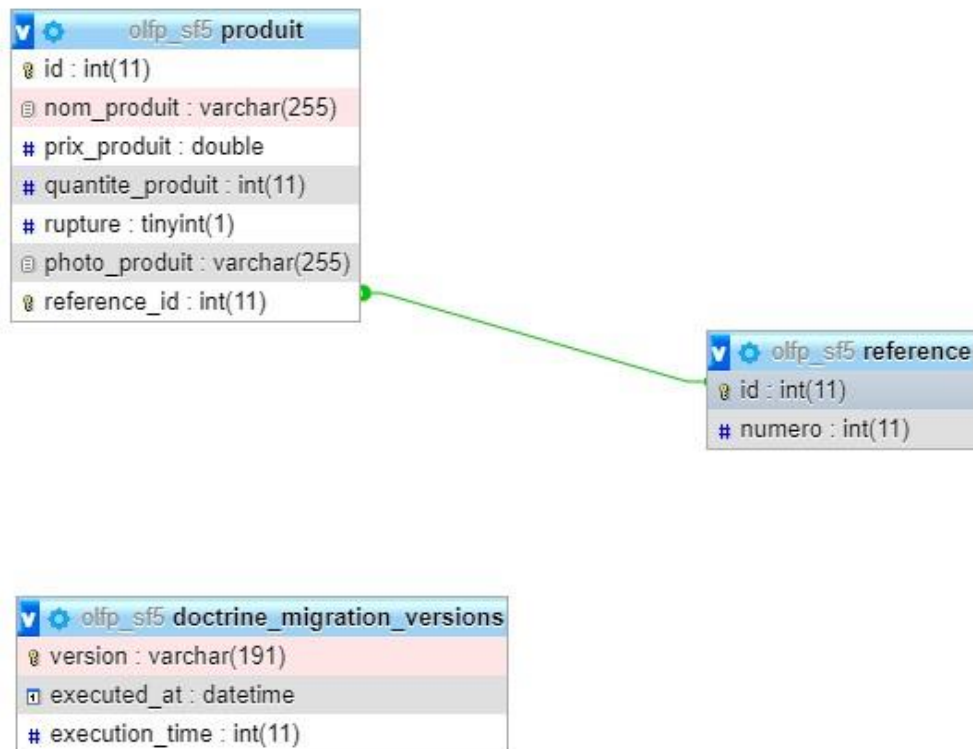
+ Ajouter une contrainte

Index

Action	Nom de l'index	Type	Unique	Compressé	Colonne	Cardinalité	Interclassement	Null	Commentaire
Éditer Supprimer	PRIMARY	BTREE	Oui	Non	id	5	A	Non	
Éditer Supprimer	UNI_29A5EC271645DEA9	BTREE	Oui	Non	reference_id	1	A	Oui	

Créer un index sur 1 colonnes Exécuter



Mettre à jour les fixtures :

```

public function load(ObjectManager $manager)
{
    //Boucle de parcours des valeur du fichier src/Data/ListeProduit.php
    foreach (ListeProduits::$mesProduits as $monProduit){
        //Instance de entité (classe produit)
        $produit = new Produit();
        $reference = new Reference();
    }
}
  
```

```

$reference->setNumero(rand());
//Ajout des données depuis le fichier src/Data/ListeProduit.php
$produit->setNomProduit($monProduit['nomProduit']);
$produit->setPrixProduit($monProduit['prixProduit']);
$produit->setQuantiteProduit($monProduit['quantiteProduit']);
$produit->setRupture($monProduit['rupture']);
$produit->setPhotoProduit($monProduit['photoProduit']);

$produit->setReference($reference);
//Persistence des donnée grace a Doctrine ObjectManager
$manager->persist($produit);
}
//Enregistrement des données
$manager->flush();
}

```

Dans src/Data/ListeProduits :

```

["nomProduit" => "Imprimantes Canon", "prixProduit" => 700.25,
"quantiteProduit" => 15, "rupture" => false, "photoProduit" => "imp.jpg",
"referenceProduit" => "124586"],

```

Puis : `php bin/console doctrine :fixture :load`

Afficher les références et leurs numéro dans la vue index.html.twig :

```
{{produit.reference.numero}}
```

```

<p class="card-text"><b>REFERENCE N° : {{ produit.reference.numero }}
</b></p>

```

SECONDE METHODES :

Créer un fichier JoinReferenceFixture.php dans le dossier src/DataFixtures

Pour chaque produit on y a joute une référence unique, aucune obligation de persister les donnée de l'entité référence, c'est automatique grâce a @ORM cascade {persist}

```

<?php

namespace App\DataFixtures;

```

```

use App\Entity\Produit;
use App\Entity\Reference;
use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Persistence\ObjectManager;
use Symfony\Component\DependencyInjection\ContainerAwareInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;

class JoinReferenceFixtures extends Fixture implements
ContainerAwareInterface
{
    private $container;

    public function setContainer(ContainerInterface $container = null)
    {
        $this->container = $container;
    }
    public function load(ObjectManager $manager)
    {
        // Appel d'entityManager
        $entityManager = $this->container-
>get('doctrine.orm.entity_manager');
        $repositoryProduit = $entityManager->getRepository(Produit::class);
        $listeProduits = $repositoryProduit->findAll();
        foreach ($listeProduits as $monProduit) {
            $reference = new Reference();
            $reference->setNumero(rand());
            $monProduit->setReference($reference);
            $manager->persist($monProduit);
            //Pas besoin de persister $reference @ORM cascade={"persist"}
        }
        $manager->flush();
    }
}

```

Afin d'exécuter seulement cette fixture on ajoute des drapeaux (flags) pour éviter que la commande ne supprime tous les enregistrements des entités :

`php bin/console doctrine :fixture :load --group=JoinReferenceFixtures --append`

Il y a maintenant un jeu de données pour la jointure reference aléatoire.

I – Les relations ManyToMany :

Créer une entité Distributeur : `php bin/console make :entity`

Distributeur : nomDistributeur + string + 255 + no

Comme d'habitude : `php bin/console make :migration` puis : `php bin/console doctrine:migrations:migrate`

Dans l'entité Produit.php (sous \$reference) :

```
/**
 * @ORM\ManyToOne(targetEntity="App\Entity\Distributeur",
 cascade={"persist"})
 * @ORM\JoinColumn(nullable=true)
 */
private $distributeurs;
```

Mettre à jour les Getters et les Setters :

`php bin/console make:entity --regenerate App`

Grace a la relation ManyToMany, les Getters et Setters ont changé :

```
public function addDistributeur(Distributeur $distributeur): self
{
    if (!$this->distributeurs->contains($distributeur)) {
        $this->distributeurs[] = $distributeur;
    }

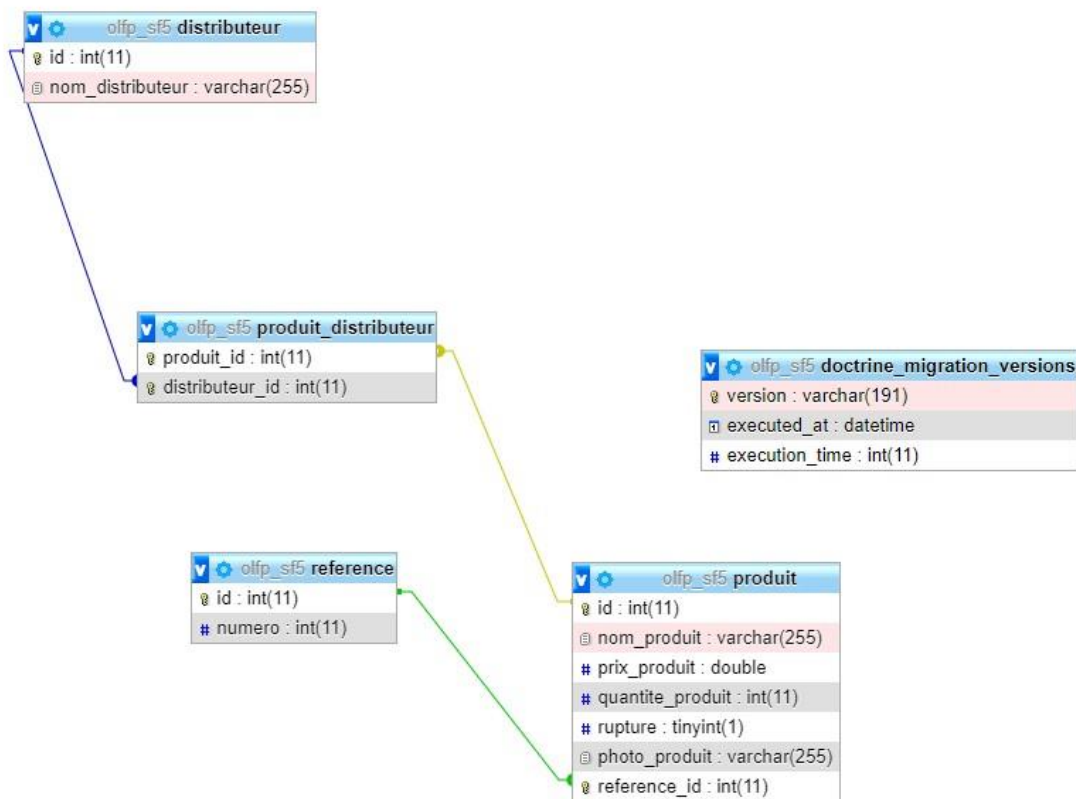
    return $this;
}

public function removeDistributeur(Distributeur $distributeur): self
{
    $this->distributeurs->removeElement($distributeur);

    return $this;
}
```

On peut ajouter et supprimer des distributeurs (dans un formulaire)

Le nouveau MCD :



Ajouter des distributeurs avec phpMyAdmin ainsi que la liaison dans produit\_distributeur

AJOUTER DES FIXTURES MULTIPLES AVEC LA TABLE PRODUITS :

Comme pour les fixtures reference on créer un fichier  
JoinDistributeurFixtures.php

De nouveau on herite de Fixture et on impleemnte 2 interfaces (FixtureInterface + ContainerAwareInterface)

```

<?php

namespace App\DataFixtures;

use App\Entity\Distributeur;
use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Persistence\ObjectManager;
use Symfony\Component\DependencyInjection\ContainerAwareInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;
  
```

```

class JoinDistributeurFixtures extends Fixture implements FixtureInterface,
ContainerAwareInterface
{
    private $container;

    public function setContainer(ContainerInterface $container = null)
    {
        $this->container = $container;
    }

    public function load(ObjectManager $manager)
    {
        $entityManager = $this->container-
>get('doctrine.orm.entity_manager');
        $repositoryProduit = $entityManager->getRepository(Produit::class);

        //Ajout de distributeur
        $intermarche = new Distributeur();
        $intermarche->setNomDistributeur('Intermarche');

        $superu = new Distributeur();
        $superu->setNomDistributeur('Super U');

        $ldlc = new Distributeur();
        $ldlc->setNomDistributeur('LDLC');

        $thomann = new Distributeur();
        $thomann->setNomDistributeur('Thomann');

        //Les jointures : 1er produit
        $produit = $repositoryProduit->findOneBy([
            'nomProduit' => 'Imprimantes Canon'
        ]);
        $produit->addDistributeur($superu);
        $produit->addDistributeur($ldlc);
        $manager->persist($produit);

        //On enregistre les données
        $manager->flush() ;
    }
}

```

Ici un produit possède plusieurs distributeurs.

De nouveau : `php bin/console doctrine :fixture :load --group=JoinDistributeurFixtures --append`

Pour l'affichage dans le vue : `index.html.twig`

```

{% if produit.distributeurs is not empty %}
    {% for distributeur in produit.distributeurs %}
        <p class="card-text"><b>DISTRIBUTEURS : {{ distributeur.nomDistributeur
    }}</b></p>
    {% endfor %}

```



```
{% else %}
    <p class="alert alert-warning">Aucun distributeur</p>
{% endif %}
```

J – Les relations Bidirectionnelles :

Pour assuré une réciprocité afficher les produits en fonction du distributeur, on ajoute un paramètre annotation dans l'entité distributeur (sous \$id) :

```
/**
 * @ORM\ManyToMany (targetEntity="App\Entity\Produit",
 mappedBy="distributeurs")
 * @ORM\JoinColumn (nullable=true)
 */
private $produit;
```

Ici on ajoute le paramètre mappedBy(lui-même)

Il faut également ajouter un paramètre dans l'entité produit :

```
/**
 *
 * @ORM\ManyToMany(targetEntity="App\Entity\Distributeur",inversedBy="produit" cascade={"persist"})
 * @ORM\JoinColumn (nullable=true)
 */
private $distributeurs;
```

Ici on ajoute le paramètre inversedBy

On régénère les Getters et Setters : `php bin/console make :entity --regenerate App`

On ajoute la migration : `bin/console make:migration` et `bin/console doctrine :migrations :migrate`

Pour tester la jointure Bidirectionnel on va ajouter une nouvelle action au contrôleur `ListeProduitController.php`

```
/**
 * @Route("/liste_distributeurs", name="liste_distributeurs")
 */
public function listeDistributeur():Response{
    //Appel de Doctrine Entity Manager
    $entityManager = $this->getDoctrine()->getManager();
    //Utilisation du Distributeur Repository
```

```

    $distributeurRepository = $entityManager->getRepository(Distributeur::class);
    //Lister tous les distributeur repo->findAll()
    $distributeur = $distributeurRepository->findAll();
    //Appel de la vue
    return $this->render('liste_produits/distributeur.html.twig',[
        'distributeurs' => $distributeur
    ]);
}

```

Et dans le vue distributeur.html.twig :

```

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

{% block title %}Sf5 -Liste des distributeurs{% endblock %}

{% block body %}
    <h1 class="text-center text-info">LISTE DES DISTRIBUTEURS</h1>
    <table class="table">
        <thead>
            <tr>
                <th scope="col">DISTRIBUTEURS</th>
                <th scope="col">PRODUIT</th>
                <th scope="col">REFERENCE</th>
            </tr>
        </thead>
        <tbody>
            {% for distributeur in distributeurs %}
            <tr>
                <td scope="row">{{ distributeur.nomDistributeur }}</td>
                {% if distributeur.produit is not empty %}
                {% for produit in distributeur.produit %}
                <td>{{ produit.nomProduit }}</td>
                <td>{{ produit.reference.numero }}</td>
                {% endfor %}
                {% endif %}
            </tr>
            {% endfor %}
        </tbody>
    </table>
{% endblock %}

```

## K – Le Reverse Engineering :

Si vous utilisiez 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 entités : `php bin/console make :entity --regenerate App`

## AJOUTER DES CATEGORIES :

La commande : `php bin/console make :entity Categories`

nomCategorie = field type relation associé a Produit

OneToOne = chaque catégorie peut avoir plusieurs Produits

Dans produit -> champs categorie

`Php bin/console make:migration puis : php bin/console doctrine:migrations:migrate`

Dans votre page Twig :

```
<p class="card-text"><b>CATEGORIE : {{ produit.categories.nomCategorie }}</b></p>
```

## AJOUTER UNE PAGE DE DETAILS :

Ajouter le plugin : Slugify

<https://github.com/cocur/slugify>

Dans ListeProduitsController.php :

```

/**
 * @Route("produit/{slug}/{id}", name="details_produit")
 */
public function detailsProduit(Produit $produit):Response{
    return $this->render('liste_produits/detailsProduit.html.twig',[
        'produit' => $produit
    ]);
}

```

Pour la route : {slug} recupère le nom du produit et {id} spécifie l'id

En paramètre : on passe l'entité Produit

On appelle une nouvelle vue : detailsProduit.html.twig et les données seront affichées à l'aide Twig {{produit.Quelquechose}} passé dans le tableau associatif

La vue detailsProduit.html.twig :

```

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

{% block title %}Sf5 -Liste des produits-{% endblock %}

{% block body %}
    <h1 class="text-center text-info">GESTION DES PRODUITS</h1>
    <div class="row">

        <div class="col-md-4 col-sm-12 mt-3">
            <div class="card">
                
                <div class="card-body">
                    <h5 class="card-title">{{ produit.nomProduit
}}</h5>
                    {% if produit.reference is not null %}
                        <p class="card-text"><b>REFERENCE N° : {{
produit.reference.numero }} </b></p>
                    {% endif %}
                    <p class="card-text">DISTRIBUTEURS :</p>
                    {% if produit.distributeurs is not empty %}
                        {% for distributeur in produit.distributeurs %}
                            <p class="card-text alert alert-success mt-
3"><b class="">{{ distributeur.nomDistributeur }}</b></p>
                        {% endfor %}
                    {% else %}
                        <p class="alert alert-warning">Aucun
distributeur</p>
                    {% endif %}

                    <p class="card-text"><b>CATEGORIE : {{
produit.categories.nomCategorie }}</b></p>
                    <p class="card-text"><b>PRIX : {{
produit.prixProduit }} €</b></p>
                    <p class="card-text"><b>QUANTITE : {{
produit.quantiteProduit }} unité(s)</b></p>
                    <p class="card-text"><b>EN STOCK : {{

```

```

produit.rupture ? 'OUI' : 'NON' }} </b></p>
        <a href="#" class="btn btn-outline-success">Ajouter
au panier</a>

        <a href="{{ path('liste_produits') }}" class="btn
btn-outline-warning">Retour</a>

        {% if app.user %}
            <!-- Editer -->
            <a href="{{ path('editer',{'id': produit.id})
}}" class="btn btn-outline-info">Editer</a>
            <!-- Supprimer -->
            <a href="{{ path('supprimer',{'id':
produit.id}) }}" class="btn btn-outline-danger mt-3">Supprimer</a>
            {% endif %}

        </div>
    </div>
</div>
{% endblock %}

```

Dans la vues ListeProduit.html.twig : ajouté le bouton :

```

<a href="{{ path('details_produit',{'slug': produit.nomProduit , 'id':
produit.id}) }}" class="btn btn-outline-warning">Détails</a>

```

Ici le slug est passé en 1<sup>er</sup> paramtres puis l'id

## 9 – Les formulaires :

UNE REALATION AVEC LES ARTICLES :

Ajouter un table (entité) Categories\_article qui aura une relation OneToMany :

Une catégorie peu appartenir a plusieurs Articles et plusieurs Articles on une catégorie

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

Le nom : `Categorie_article`

- 1<sup>er</sup> champ : `nomCategorie + string + 255 null = no`
- `Article_id` : relation associé à Articles +
- New field name inside Articles [categoriesArticles]
- Is the Articles.categoriesArticles property allowed to be null (nullable)?  
(yes/no) [yes]: = no

Do you want to automatically delete orphaned App\Entity\Articles objects (orphanRemoval)? (yes/no) [no]: no

PUIS : `php bin/console make:migration` et `php bin/console make:migrations:migrate`

Ajouter dans les vues :

```
<li class="list-group-item">{{ article.categoriesArticles.nomCategorie
}}</li>
```

On appelle les 2 entités et on donne à la seconde le champ de la clé étrangère à afficher.

## CREER UN FORMULAIRE POUR LES ARTICLES :

Pour générer un CRUD automatiquement : `php bin/console make:crud`

Nom de l'entité : Articles

Symfony Génère un CRUD automatiquement :

## UN CRUD A LA MAIN :

2 Choix :

- Le formulaire dans votre contrôleur
- Ou dans un fichier externe

Créer un formulaire : `php bin/console make:form`

ArticleType -> associé à l'entité Articles

LE CONTRÔLEUR ARTICLES : méthode `ajouterArticle()`

Créer 3 méthodes dans votre `ArticlesContrôleur` :

- `ajouterArticle()`
- `editerArticle()`

- supprimerArticle()

## 1- LA METHODE AJOUTER :

```
/**
 * @param Request $request
 * @return Response
 * @Route("/ajouter_article", name="ajouter_article")
 */
public function ajouterArticle(Request $request):Response{
    //Instance de entité (class) Articles
    $article = new Articles();
    //creation du formulaire
    $formArticle = $this->createForm(ArticlesType::class, $article);

    //Ajout du bouton se soumission
    $formArticle->add('ajouter_article', SubmitType::class, [
        'label' => 'Ajouter l\'article',
    ]);

    //Recup des données du formulaire
    $formArticle->handleRequest($request);

    //Cette condition verifie la methode et creer un __token qui lutte
    contre les injection SQL
    //Faible CSRF Cross Site Request Forgery
    if($request->isMethod("post") && $formArticle->isValid()){
        //Appel de doctrine entityManager
        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->persist($article);
        $entityManager->flush();

        //Si ca marche
        return $this->redirectToRoute('articles');
    }
    //Afficher le formulaire d'ajout dans ca vue
    return $this->render('articles/ajouterArticles.html.twig', [
        'form_article' => $formArticle->createView()
    ]);
}
```

## 2-TYPE LES DONNEES DU FORMULAIRE :

ArticlesType.php (Cette opération peu etre faite dans le controlleur)

```
<?php

namespace App\Form;

use App\Entity\Articles;
use App\Entity\CATEGORIESArticles;
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
use Symfony\Component\Form\AbstractType;
```

```

use Symfony\Component\Form\Extension\Core\Type\DateTimeType;
use Symfony\Component\Form\Extension\Core\Type\FileType;
use Symfony\Component\Form\Extension\Core\Type\TextareaType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class ArticlesType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('nomArticle', TextType::class)
            ->add('contenuArticle', TextareaType::class)
            ->add('imageArticle', FileType::class, [
                'required' => false,
                'data_class' => null,
                'empty_data' => 'Aucune image pour cet article',
            ])
            ->add('auteurArticle', TextType::class)
            ->add('dateArticle', DateTimeType::class)
            ->add('categoriesArticles', EntityType::class, [
                'class' => CategoriesArticles::class,
                'choice_label' => 'nomCategorie',
                'label' => 'Catégorie de l\'article',
            ])
        ;
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => Articles::class,
        ]);
    }
}

```

## LA VUES AJOUTER ARTICLES :

Pour le rendu :

2 Choix :

- Dans ArticlesType (Voir Admin)
- Ou dans Twig

## LE RENDU DU FORMULAIRE DANS TWIG

2 Methodes :

- 1- Simplement {{ form(form\_article) }} (valeur tableau associatif contrôleur)



2- {{ form\_start(form\_article) }} {{ form(form\_end) }}

Exemple :

## ADMINISTRATION :

La partie Admininstartion : `php bin/console make :controller`

AdminController avec 2 route et 3 methodes (ajouter, editer et supprimer)

```
<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class AdminController extends AbstractController
{
    /**
     * @Route("/ajouter", name="ajouter")
     */
    public function ajouter(Request $request):Response{
        return $this->render('admin/ajouter.html.twig');
    }

    /**
     * @Route("/editer/{id}", name="editer")
     */
    public function editer(Request $request, $id):Response{
        return $this->render('admin/editer.html.twig');
    }

    /**
     * @Route("/supprimer/{id}", name="supprimer")
     */
    public function supprimer(Request $request):Response{
```

```
}  
}
```

## A – Form Builder :

La methode createFormBuilder() permet de generer des formulaire et d'ajouter des champs gracie a add()

Ici nous allons utiliser des formulaire externalisé (Hors du controlleur)

La ligne de commande : `php bin/console make :form`

ProduitType

Le nom de l'entité a utilisé pour ce formulaire = Produit

Cette commande va créer un dossier src/Form avec le le fichier ProduitType.php

Ce fichier est une classe qui herite de AbstractType, il faudra donc préciser le type de chaque champ

Exemple :

```
<?php  
  
namespace App\Form;  
  
use App\Entity\Produit;  
use Symfony\Component\Form\AbstractType;  
use Symfony\Component\Form\Extension\Core\Type\CheckboxType;  
use Symfony\Component\Form\Extension\Core\Type\FileType;  
use Symfony\Component\Form\Extension\Core\Type\NumberType;  
use Symfony\Component\Form\Extension\Core\Type\TextType;  
use Symfony\Component\Form\FormBuilderInterface;  
use Symfony\Component\OptionsResolver\OptionsResolver;  
  
class ProduitType extends AbstractType  
{  
    public function buildForm(FormBuilderInterface $builder, array  
$options)  
    {  
        $builder  
            ->add('nomProduit', TextType::class, [  
                'label' => 'Nom du produit :',  
                'attr' => [  
                    'placeholder' => 'ex: Table basse',  
                    'class' => 'mt-3 mb-3'  
                ]  
            ])  
            ->add('prixProduit', NumberType::class, [  
                'label' => 'Prix du produit en € :',  
                'attr' => [  
                    'placeholder' => 'ex: 124.25 ',  
                    'class' => 'mt-3 mb-3'  
                ]  
            ])  
    }  
}
```

```

        ->add('quantiteProduit', NumberType::class, [
            'label' => 'Quantité en stock :',
            'attr' => [
                'placeholder' => "ex: 250",
                'class' => 'mt-3 mb-3'
            ]
        ])

        ->add('photoProduit', FileType::class,[
            'label' => 'Image du produit :',
            'required' => false,
            'data_class' => null,
            'empty_data' => 'Aucune image',
            'attr' => [
                'class' => 'form-control mt-3 mb-3'
            ]
        ])
        ->add('rupture', CheckboxType::class,[
            'label' => 'En stock ?',
            'required' => false,
            'attr' => [
                'class' => 'form-check-input'
            ]
        ])
        //->add('reference')
        //->add('distributeurs')
    };

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => Produit::class,
        ]);
    }
}

```

Les champs rupture et photoImage possède un paramètre required => false

Le navigateur autorise donc que ses 2 champs soit vide (pas besoin de cocher ou changer la photo pour la mise à jour.

Data\_class => null permet de ne pas préciser par défaut cette entité au cas où l'image n'est pas mise à jour, à la place on affiche 'Aucune image' ou une image par défaut.

Pas bouton SubmitType, il sera ajouté directement dans le contrôleur.

LE CONTRÔLEUR ADMIN :

Pour tester le formulaire on retourne du Json

```

public function ajouter(Request $request):Response{
    //Instance du produit
    $produit = new Produit();
    //Appel de la methode createForm de AbstractController
    //En paramètre on passe le formulaire + entité
    $formProduit = $this->createForm(ProduitType::class, $produit);
}

```

```

//Ajout d'un bouton de soumission
$formProduit->add('ajouter', SubmitType::class,[
    'label' => 'Ajouter le produit',
    'attr' => [
        'class' => 'btn btn-outline-success mt-3'
    ]
]);

//Condition de sécurité et de methode de formInterface (booléen verifie
l'intégrité des champs)
if($request->isMethod('post')){

    return new JsonResponse($request->request->findAll());

    /*
    //Appel d'entityManager de Doctrine
    $entityManager = $this->getDoctrine()->getManager();
    //Persistance des données entrées dans le formulaire
    $entityManager->persist($produit);
    //Enregistrement en base de données
    $entityManager->flush();
    return $this->redirectToRoute('liste_produits');
    */

}
return $this->render('admin/ajouter.html.twig',[
    //Ici la methode formView genere de html
    'formulaire_produit' => $formProduit->createView()
]);
}

```

Dans la vue Twig ajouter.html.twig :

```

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

{% block title %}Sf5 -Ajouter un produits-{% endblock %}

{% block body %}
    <div class="mt-3">
        <h1 class="text-success text-center">AJOUTER UN PRODUITS</h1>
        {{ form(formulaire_produit) }}
    </div>
{% endblock %}

```

Pour ajouter du css bootstrap au formulaire Symfony a prévu un layout de base

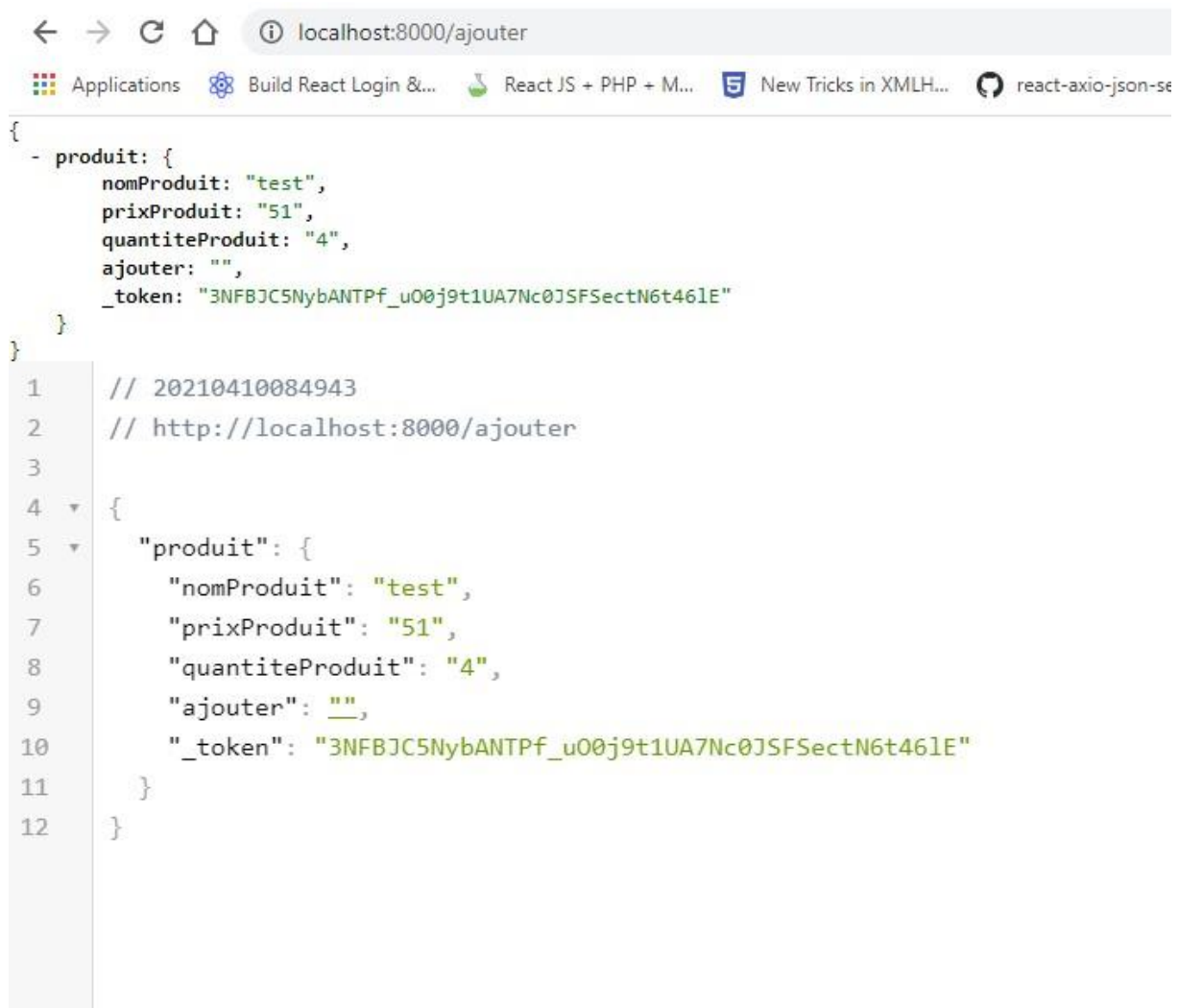
Dans le fichier : config/packages/twig.yaml

```

twig:
    default_path: '%kernel.project_dir%/templates'
    globals:
        auteur: '%env(APP_AUTHOR)%'
    form_themes: ['bootstrap_4_layout.html.twig']

```

A la soumission le formulaire retourne du Json :



```
{
  - produit: {
    nomProduit: "test",
    prixProduit: "51",
    quantiteProduit: "4",
    ajouter: "",
    _token: "3NFBJC5NybANTPf_u00j9t1UA7Nc0JSFSectN6t461E"
  }
}
```

```
1 // 20210410084943
2 // http://localhost:8000/ajouter
3
4 {
5   "produit": {
6     "nomProduit": "test",
7     "prixProduit": "51",
8     "quantiteProduit": "4",
9     "ajouter": "",
10    "_token": "3NFBJC5NybANTPf_u00j9t1UA7Nc0JSFSectN6t461E"
11  }
12 }
```

## AJOUTER DU CSS ET JS AVEC WEBPACK :

Pour inclure des fichiers CSS et JS et utiliser des bibliothèques ou framework externe on utilise : webpack (installer nodejs au préalable)

`composer require symfony/webpack-encore-bundle`

Cette ligne de commande ajoute un fichier package.json

Webpack à créer un dossier assets avec à l'intérieur :

- Controller (pour Stimulus javascript ecosysteme)
- Styles (app.css)
- Un fichier point d'entrée app.js
- Un fichier bootstrap.js
- Et un controller.json

Pour installer SASS

Npm install sass-loader node-sass --dev

Dans webpack.config.js décommenter : .enableSassLoader() et .addEntry('app', './assets/js/app.js')

Dans app.js : ajouter import '../css/app.scss'

INITIALISER BOOTSTRAP :

Npm install bootstrap et npm install jquery popper.js --dev

UTILISER app.css et app.js dans les vues :

Il faut réaliser une transpilation pour générer les préprocesseurs :

Npm run dev ou pour grader le projet sur vos fichiers npm run watch

Dans le fichier base.html.twig : décommenter :

```
{#{ {{ encore_entry_link_tags('app') }} #}}
```

Et :

```
{#{ {{ encore_entry_script_tags('app') }} #}}
```

Pour utiliser .css ou .js :

Écrire dans les fichiers assets/app.js et assets/styles/app.css

Puis transpiler le résultat : ex : app.js

```
// any CSS you import will output into a single css file (app.css in this case)
import './styles/app.css';

// start the Stimulus application
import './bootstrap';
//Jquery
const $ = require('jquery')

alert('test')
```

Ici une alerte se déclenche à chaque rafraîchissement.

AJOUTER LE NOM DU FICHIER DE L'IMAGE CHARGÉE :

Dans app.js : Du jquery :

```
//Afficher le nom de l'image dans le formulaire
$(".custom-file-input").on("change", function () {
    let fileName = $(this).val().split("\\").pop()
    $(this).siblings(".custom-file-label").addClass("selected").html(fileName)
})
```

## TRAITEMENT DES DONNEES :

Avec Symfony tout ce passe en une action :

1. Le controleur appelle la vue du formulaire
2. On récupère les données de l'entité associée lors de la génération du formType avec la méthode `handleRequest()` en passant l'objet `$request` en paramètres
3. On teste si le formulaire est soumis et s'il est valide
4. On appelle Doctrine et son `entityManager()`
5. Les données du formulaire sont déjà dans l'entité grâce à `createForm()` et le paramètre `$produit` (entité)

## POUR UPLOAD DE L'IMAGE :

1. La méthode `getData()`
2. Récupération du nom original de la photo
3. Deviner l'extension
4. Déplacer la photo (`move_uploaded_file` PHP)
5. La destination `public/img` à configurer avec les paramètres `images_directory` dans `config/services.yaml`
6. Insérer l'image grâce au Setter `$produit->setImg($fileName)`
7. Tester si une image a bien été chargée => sinon aucune image
8. Utiliser le flashbag pour afficher le résultat de l'opération d'ajout de produit

Dans `service.yaml` :

```
parameters:
    images_directory: '%kernel.project_dir%/public/img'
```

La vue à ajouter.html.twig

```
/**
 * @Route("/ajouter", name="ajouter")
 */
public function ajouter(Request $request):Response{
    //Instance du produit
    $produit = new Produit();
    //Appel de la méthode createForm de AbstractController
    //En paramètre on passe le formulaire + entité
    $formProduit = $this->createForm(ProduitType::class, $produit);
```

```

//Ajout d'un bouton de soumission
$formProduit->add('ajouter', SubmitType::class,[
    'label' => 'Ajouter le produit',
    'attr' => [
        'class' => 'btn btn-outline-success mt-3'
    ]
]);

//Recupération des données des champs du formulaire
$formProduit->handleRequest($request);

//Condition de sécurité et de methode de formInterface (booléen verifie
l'intégrité des champs)
if($request->isMethod('post') && $formProduit->isValid()){

    //return new JsonResponse($request->request->all());

    //Appel d'entityManager de Doctrine
    $entityManager = $this->getDoctrine()->getManager();

    //Upload de la photo (appel de private $photoProduit; de entité)
    $file = $formProduit['photoProduit']->getData();

    //Une image a t elle ete chargée
    if(!is_string($file)){
        //Nom originale de la photo
        $fileName = $file->getClientOriginalName();
        //php move_uploaded chemin dans config/services.yaml
        $file->move(
            $this->getParameter('images_directory'),
            $fileName
        );
        //Entité produit et Setters photoImage
        $produit->setPhotoProduit($fileName);
    }else{
        //Erreur + flashbag
        $session = $request->getSession();
        $session->getFlashBag()->add('message', 'Merci d\'ajouter une
image au produit');
        $session->set('statut', 'danger');
        //Si ca marche
        return $this->redirect($this->generateUrl('ajouter'));
    }

    //Persistance des données entrées dans le formulaire
    $entityManager->persist($produit);
    //Enregistrement en base de données
    $entityManager->flush();
    return $this->redirectToRoute('liste_produits');

}
return $this->render('admin/ajouter.html.twig',[
    //Ici la methode formView genere de html
    'formulaire_produit' => $formProduit->createView()
]);
}

```

Ajouter le message de reusite FlashBag apres le flush()



```
//Message de reussite
$session = $request->getSession();
$session->getFlashBag()->add('message', 'Le produit à bien été ajouté');
$session->set('statut', 'success');
```

De meme pour la methode de mise a jour `editier()` :

```
/**
 * @Route("/editer/{id}", name="editer")
 */
public function editier(Request $request, $id):Response{

    //Appel d'entityManager de Doctrine
    $entityManager = $this->getDoctrine()->getManager();
    //Appel du repository avec entité en paramètre
    $produitRepository = $entityManager->getRepository(Produit::class);
    //Creation du produit on passe id du produit en paramètre
    $produit = $produitRepository->find($id);

    //On recupère la photo existante
    $img = $produit->getPhotoProduit();

    //Appel de la methode createForm de AbstactController
    //En paramètre on passe le formulaire + entité
    $formProduit = $this->createForm(ProduitType::class, $produit);
    //Ajout d'un bouton de soumission
    $formProduit->add('ajouter', SubmitType::class, [
        'label' => 'Mettre à jour le produit',
        'attr' => [
            'class' => 'btn btn-outline-warning mt-3'
        ]
    ]);

    //Recupération des données des champs du formulaire
    $formProduit->handleRequest($request);

    //Condition de securité et de methode de formInterface (booléen verifie
    l'intégrité des champs)
    if($request->isMethod('post') && $formProduit->isValid()){

        //return new JsonResponse($request->request->all());

        //Upload de la photo (appel de private $photoProduit; de entité)
        $file = $formProduit['photoProduit']->getData();

        //Une image a t elle ete chargée
        if(!is_string($file)){
            //Nom originale de la photo
            $fileName = $file->getClientOriginalName();
            //php move_uploaded chemin dans config/services.yaml
            $file->move(
                $this->getParameter('images_directory'),
                $fileName
            );
            //Entité produit et Setters photoImage
            $produit->setPhotoProduit($fileName);
        }else{
            //Si l'image ne change pas on recupère l'ancienne (Setters $img
            (Getters))
            $produit->setPhotoProduit($img);
        }
    }
}
```

```

    }

    //Persistence des données entrées dans le formulaire
    $entityManager->persist($produit);
    //Enregistrement en base de données
    $entityManager->flush();
    //Message de reussite
    $session = $request->getSession();
    $session->getFlashBag()->add('message', 'Le produit à bien été mis
à jour');
    $session->set('statut', 'success');
    //Si ca marche redirection vers la page de liste des produits
    return $this->redirectToRoute('liste_produits');

}
//Affiche de base le formulaire @Route (ajouter etc...)
return $this->render('admin/editer.html.twig',[
    //Ici la methode formView genere de html
    'formulaire_produit' => $formProduit->createView()
]);
}

```

Ici si la photo n'est pas changée on récupère l'existante

Dans le vue listeProduit ajouter le bouton d'edition et de suppression :

```

<!-- Editer -->
<a href="{{ path('editer',{'id': produit.id}) }}" class="btn btn-outline-
info">Editer</a>
<!-- Supprimer -->
<a href="{{ path('supprimer',{'id': produit.id}) }}" class="btn btn-
outline-danger mt-3">Supprimer</a>

```

La vue editer.html.twig :

```

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

{% block title %}Sf5 -Editer un produits-{% endblock %}

{% block body %}
    <div class="mt-3 d-flex justify-content-center">

        {#form(formulaire_produit) #}

        <div class="form-group">
            <h1 class="text-success text-center">EDITER LE PRODUITS</h1>
            {{ form_start(formulaire_produit) }}
            {{ form_errors(formulaire_produit) }}
            {{ form_end(formulaire_produit) }}
            <a href="{{ path('liste_produits') }}" class="btn btn-outline-
info">Annuler</a>
        </div>

    </div>

{% endblock %}

```

Ici on remarque que les champs sont déjà pré rempli

## LE CONTROLLEUR SUPPRIMER UN PRODUITS :

```
/**
 * @Route("/supprimer/{id}", name="supprimer")
 */
public function supprimer(Request $request, $id):Response{

    //Appel de la methode entityManage de Doctrine
    $entityManager = $this->getDoctrine()->getManager();
    //Recuperation du Repo et de l'entité en paramètre
    $produitRepository = $entityManager->getRepository(Produit::class);
    //Recupération du produit
    $produit = $produitRepository->find($id);
    //Suppression du produit
    $entityManager->remove($produit);
    //Confirmation de la suppression
    $entityManager->flush();

    //Message de succes depuis l'objet request
    $session = $request->getSession();
    $session->getFlashBag()->add('message', 'Le produit à bien été
supprimer !');
    $session->set('statut', 'success');
    //Redirection
    return $this->redirect($this->generateUrl('liste_produits'));
}
```

## LE TRAITEMENT DE LA JOINTURE ONE TO ONE ET LES FORMULAIRES : (\$reference)

Créer un nouveau formulaire qui va être imbriqué dans le formulaire  
ProduitsType

La commande : `php bin/console make :form`

ReferenceType -> associé à l'entité : Reference

Ajoute le type de champs : `numero de type NumberType::class`

```
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('numero', NumberType::class, [
            'label' => 'N° de reference'
        ])
    ;
}
```

Pour l'imbrication dans ProduitType :

```
->add('reference', ReferenceType::class,[
    'label' => 'Référence du produit',
    'required' => false
])
```

Le type de champs est directement le formulaire de l'entité reference

Dans le vue ajouter et editer.html.twig (les 2 formulaires)

```
<div class="form-group">
    <h1 class="text-success text-center">AJOUTER UN PRODUITS</h1>
    {{ form_start(formulaire_produit) }}
    {{ form_errors(formulaire_produit.reference.numero) }}
    <!--Le champs reference formulaire imbriquer -->
    {{ form_label(formulaire_produit.reference.numero) }}
    {{ form_widget(formulaire_produit.reference.numero) }}
    {{ form_end(formulaire_produit) }}
    <a href="{{ path('liste_produits') }}" class="btn btn-outline-
info">Annuler</a>
</div>
```

## TRAITEMENT DE LA JOINTURE MANY TO MANY

Ici il faut imbriquer plusieurs formulaires autant de fois qu'il y a besoin de jointure avec l'entité inverse

On utilise un champ de type CollectionType

La commande : `php bin/console make :form DistributeurType` associé à l'entité Distributeur

Dans le fichier DistributeurType.php

```
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('nomDistributeur', TextType::class,[
            'label' => 'Nom du distributeur'
        ])
        ->add('produit')
    ;
}
```

Puis dans ProduitType.php :

```
->add('distributeurs', CollectionType::class,[
    'entry_type' => Distributeur::class,
    'allow_add' => true,
    'allow_delete' => true
])
```

entry\_type = le nom du formulaire a imbriquer

allow\_add = autorise autant de formulaire imbriqué que désiré

allow\_delete = autorise la suppression des formulaires imbriqués

Dans la vue ajouter.html.twig et éditer

```
{{ form_label(formulaire_produit.distributeurs) }}  
{{ form_widget(formulaire_produit.distributeurs) }}
```

Si vous rencontrez des problèmes ajoutez la méthode magique :

Entité Produits.php :

```
public function __toString()  
{  
    // Le nom du produit  
    return $this->nomProduit;  
}
```

Le résultat est décevant (inspecter le code F12 et trouver la div avec l'attribut data-prototype)

Pour ajouter et supprimer des distributeurs créer un fichier .js => public/js => formdist.js

```
//jQuery  
$(document).ready(function () {  
    //Recup de la div qui contient l'attribut data-prototype  
    let $container = $('#produit_distributeurs');  
    //Ajout d'un lien pour ajouter un distributeur  
    let $addLink = $('<a href="#" id="add_distributeur" class="btn btn-outline-success">Ajouter un distributeur</a>')  
    $container.append($addLink)  
    //On ajoute un nouveau champ à chaque click sur le lien d'ajout  
    $addLink.click(function (event) {  
        //Méthode de l'entité Produits  
        addDistributeur($container);  
        //Supprime le comportement normal HTML  
        event.preventDefault();  
        return false;  
    })  
  
    //On définit un index unique pour nommer les champs qu'on ajoute dynamiquement  
    let index = $container.find(':input').length  
  
    if(index !== 0) {  
        //Pour chaque DistributeurType qui existe on ajoute un lien de suppression  
        $container.children('div').each(function () {  
            addDeleteLink($(this))  
        })  
    }  
}
```

```

//La fonction qui ajoute un formulaire DistributeurType
function addDistributeur($container){
    //Dans le contenu de attribut data-prototype on remplace
    // __name__ label__ = label du champ
    // __name__ = numero du champs
    let $prototype = $($container.attr('data-
prototype')).replace(/__name__label__/g, 'Distributeur N°' + (index +
1)).replace(/__name__/g, index))

    //On ajoute au prototype un lien pour pouvoir supprimer le
DistributeurType
    addDeleteLink($prototype)
    //On ajoute le prototype modifié a la fin de la balise <div>
    $container.append($prototype)

    //Enfin on incremente le compteur pour que le prochain ajout ce
fasse avec un autre numero
    index++;
    //La fonction qui ajoute le lien de suppression d'un distributeur
    function addDeleteLink($prototype){
        //Creation du lien
        $deleteLink = $('<a href="#" class="btn btn-outline-
danger">Supprimer</a>')
        //Ajout du liens
        $prototype.append($deleteLink)

        //Ajout de l'event listener au click du liens
        $deleteLink.click(function (event){
            $prototype.remove()
            event.preventDefault()
            return false
        })
    }
}
})

```

Puis dans base.html.twig : appelé jquery cdn + notre fichier js

```
<script src="{ asset('js/formdist.js') }"></script>
```

## LE TYPE ENTITY TYPE POUR LISTER DES ELEMENTS :

Ce type permet de lister les distributeurs lors de l'ajout d'un produit

Dans ProduitType.php

```

->add('distributeurs', EntityType::class, [
    'class' => Distributeur::class,
    'choice_label' => 'nomDistributeur',
    'label' => 'Selectionnez un ou plusieurs distributeur(s)',
    'multiple' => true,
    'required' => false
])

```

Choice\_label = le nom de la propriété qui sera affiché

Multiple = on peut selectionner plusieurs distributeur

Required = champ nom obligatoire

(Pour choisir plusieurs distributeur on maintient la touche ctrl + click)

## LA VALIDATION DES FORMULAIRES ET LA SÉCURITÉ

Pour éviter l'injection SQL et la faille xsrf (injection de code malicieux)

Symfony utilise des règles de validation dans ses entités appelées assertions dans les annotations (Asserts)

Dans Produit.php ajouté le use (trait + interface)

```
use Symfony\Component\Validator\Constraints as Assert;
```

Puis sur le nom du produit :

```
/**
 * @ORM\Column(type="string", length=255)
 * @Assert\Length(
 *     min=2, max=50, minMessage="Le nom du produit doit avoir au moins {{
limit }} caractères", maxMessage="Le nom du produit doit avoir au maximum {{
limit }} caractères"
 * )
 */
```

On peut ajouter des ReGex (Expression Régulière)

<https://symfony.com/doc/current/reference/constraints/Regex.html>

## SERVICE VALIDATOR

Les entités utilisent un service particulier Validator (c'est une classe utilisable depuis n'importe où sans préciser son namespace)

Dans le contrôleur c'est la méthode `$formProduit->isValid()`

On peut contraindre des éléments depuis un contrôleur

Asserts sur les accesseurs (Getters)

Ex pour les quantités :

```
/**
 * @return bool|null
 * @Assert\IsTrue(message="Erreur: la quantité et le prix ne peuvent pas
être une valeur négative")
 */
public function isPrixQuantiteValid(){
    if(is_float($this->getPrixProduit()) && (is_int($this-
>getQuantiteProduit()) && ($this->getPrixProduit() > 0) && ($this-
>quantiteProduit) > 0){
        return true;
    }
}
```

```

        }else{
            return false;
        }
    }
}

```

## CREER SES PROPRES CONTRAINTES DE VALIDATION SUR UN CALLBACK

Si votre critère ne correspond à aucune assertion, il est possible de créer la votre :

Elle est validée avec l'assertion Callback, c'est une méthode qui reçoit un objet Context issu de l'interface ExecutionContextInterface.

On va interdire la validation au dépôt de produit interdit (médicament, armes, etc...)

Créer une méthode callback pour déclencher une erreur

```

/**
 * @Assert\Callback()
 */
public function isContentValid(ExecutionContextInterface $context){
    //Liste de mot interdit
    $forbiddenWords = array('arme', 'médicament', 'drogue');
    //cette condition qui fait le travail = si $this->getNomProduit
    contient un mot de la liste on déclenche une erreur
    //#i indique que le champ est insensible à la casse (majuscule +
    minuscule)
    if(preg_match('#'.implode('|', $forbiddenWords).'#i', $this-
    >getNomProduit())){
        //Erreur de validation
        $context->buildViolation('Ce produit est interdit à la vente')
            ->atPath('produit')
            ->addViolation();
    }
}

```

## CONTRAÎTE DE VALIDATION SUR UNE CLASSE

Eviter 2 produits avec le même nom :

Dans Produit.php au-dessus de la déclaration de la classe :

```

/**
 * @ORM\Entity(repositoryClass=ProduitRepository::class)
 * @UniqueEntity(fields="nomProduit", message="Erreur : un produit possède
    déjà ce nom dans notre base de données")
 */
class Produit

```



Attention cette assertion possède un inconvenient, si on essaie d'editer un produit ceci va declencher une erreur.

On va arranger avec le système de groupe qui s'applique a certains formulaire :

```
* @UniqueEntity(fields="nomProduit", message="Erreur : un produit possède déjà ce nom dans notre base de données", groups={"produits"})
```

Ensuite on précise les formulaires qui appartiennent à ce groupe

Ça se passe dans le controlleur methode ajouter sur le bouton submit

```
//Ajout d'un bouton de soumission
$formProduit->add('ajouter', SubmitType::class, [
    'label' => 'Ajouter le produit',
    'validation_groups' => array('produits'),
    'attr' => [
        'class' => 'btn btn-outline-success mt-3'
    ]
]);
```

Il faut maintenant test edition d'un produit, mais attention toutes les asserts doivent appartenir au groupes :

Il faut donc créer un nouveau groupe différent de produits pour valider les champs ex : le groupe all

Pour chaque Asserts :

```
* @Assert\Type("float", message="Le prix du produit {{ value }} n'est pas une donnée valide {{type}}"), groups="all"
```

Puis ajouter ce groupe au methode ajouter() et editer() dans le controlleur

Pour ajouter() les 2 groupes :

```
'validation_groups' => array('produits', 'all'),
```

Pour editer un seul groupe :

```
'validation_groups' => array('all'),
```

CREER SES PROPRES CONTRAINTES :

Toutes les contraintes héritent de Constraints, ici on crée un Antispam.php

Src/Validator/Antispam.php (classe)

```
<?php

namespace App\Validator;

use Symfony\Component\Validator\Constraint;
```

```

/**
 * Class Antispam
 * @package App\Validator
 * @Annotation
 */

class Antispam extends Constraint
{
    public $message = "<p class='alert alert-danger'>Le champ est trop
court</p>";
}

```

Le lien entre la containte et le validator est par defaut la methode  
validateBy() (ctrl + click sur Constraint)

Crer une classe AntispamValidator.php

```

<?php

namespace App\Validator;

use Symfony\Component\Validator\Constraint;
use Symfony\Component\Validator\ConstraintValidator;

class AntispamValidator extends ConstraintValidator
{
    public function validate($value, Constraint $constraint)
    {
        // REGEX.
        if(!preg_match('/^[a-zA-Z]+$/', $value, $matches)){

            $this->context->buildViolation($constraint->message)
                ->setParameters(array('%string%' => $value))
                ->addViolation();
        }
    }
}

```

Dans Produit.php

```

* @Antispam(message="Le nom du produit : %string% ne doit contenir que des
caracèteres alphanumeriques", groups="all")

```

Pour tester entrer des chiffres dans le nom du produit

Une erreur ce déclenche

## 10 – La Sécurité

2 partie distinctes :

- L'authentification : authentifier un utilisateur et accéder à une partie sécurisée
- L'autorisation : définir des droits pour un utilisateur pour accéder à certaines ressources (Routes, Contrôleur, Méthodes, Vues, Entités, etc...)

Authentifié un utilisateur :

La sécurité d'une application Symfony se situe dans :

Config/packages/security.yaml :

```
security:
    # https://symfony.com/doc/current/security.html#where-do-users-come-from-user-providers
    providers:
        users_in_memory: { memory: null }
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            anonymous: true
            lazy: true
            provider: users_in_memory

            # activate different ways to authenticate
            # https://symfony.com/doc/current/security.html#firewalls-authentication

            #
            https://symfony.com/doc/current/security/impersonating_user.html
            # switch_user: true

            # Easy way to control access for large sections of your site
            # Note: Only the *first* access control that matches will be used
            access_control:
                # - { path: ^/admin, roles: ROLE_ADMIN }
                # - { path: ^/profile, roles: ROLE_USER }
```

La sécurité de base autorise tous le monde : anonymous : lazy, signifie que si on accède pas à une ressource protégée, la sécurité n'est pas activée

Dans la barre profiler :



3 étiquettes :

- Providers = comment et ou trouvé l'identifiant des utilisateurs identifié, c'est la methode memory qui renseigne directement les utilisateurs dans le fichier securité
- Firewall = c'est le pare-feu qui vont intervenir pour authentifier un utilisateur, on peu en cumuler plusieurs
- Access Control = precise les autorisations exigées pour acceder au routes

2 pare-feux :

- Dev = concerne toute les routes qui commence par /\_profiler, /\_wdt, /css, /img, /js (le dossier public) ont une securité desactivée
- Main = Toutes les autres routes, lazy et anonymous empeche la creation d'un identifiant de sessions s'il n'y a pas besoin d'autorisation. Dans le cas contraire ce sont les utilisateur definit dans users\_in\_memory (le provider) qui auront les autorisations d'y acceder grace a des roles

#### A – INSTALLER LE PACKAGE SECURITE

- composer require security (déjà present par default )
- Créer entité user php bin/console make :user
- Valider [User] + store in database : yes + connexion par email + hash password : yes
- La commande créer : src/Entity/User.php
- Un repository (DQL) : src/Repository/UserRepository.php
- Un fichier de configuration config/packages/security.yaml modifié

Dans security.yaml :

- Une nouvelle etiquettes encoders (hasher les mots de passe)

```
security:
  encoders:
    App\Entity\User:
      algorithm: auto
```

- Un nouveau provider : app\_user\_provider, il fait appel a l'entité User avec sa propriété email (la connexion se fait donc a l'aide de email + mot de passe)
- Il faut maintenant créer les routes, les controleurs, et les vues
- On utilise la commandes : php bin/console make :auth
- Choisir 1 (créer un formulaire de connexion)
- Nom de Authenticator = CustomAuthenticator
- Créer le SecurityController

- Créer une route logout : yes

Cette ligne de commande met à jour security.yaml, créer des routes (/login et /logout), une classe CustomAuthenticator, un SecurityController et une vue login.html.twig

Modification de la classe src/Security/CustomAuthenticator :

Modifié la méthode suivante :

```
public function onAuthenticationSuccess(Request $request, TokenInterface $token, string $providerKey)
```

Décommenter la redirection en cas de succès et commenter la ligne de détection d'erreur :

```
return new RedirectResponse($this->urlGenerator->generate('liste_produits'));
//throw new \Exception('TODO: provide a valid redirect inside '.__FILE__);
```

Dans menu.html.twig : (rappel pour voir vos routes : php bin/console debug:router)

```
<a class="nav-link" href="{{ path('app_login') }}"><b class="h5">CONNEXION</b></a>
```

Modifier : templates/security/login.html.twig :

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

{% block title %}Sf5 -CONNEXION-{% endblock %}

{% block body %}
<form method="post">
    {% if error %}
        <div class="alert alert-danger">{{
error.messageKey|trans(error.messageData, 'security') }}</div>
    {% endif %}

    {% if app.user %}
        <div class="mb-3">
            Vous êtes connecté en tant que : {{ app.user.username }}, <a
href="{{ path('app_logout') }}">Déconnexion</a>
        </div>
    {% endif %}

    <h1 class="h3 mb-3 font-weight-normal text-info">CONNEXION</h1>

    <div class="form-group">
        <label for="inputEmail">Email</label>
```

```

        <input type="email" placeholder="Votre email" value="{{
last_username }}" name="email" id="inputEmail" class="form-control"
required autofocus>
    </div>

    <div class="form-group">
        <label for="inputPassword">Password</label>
        <input type="password" placeholder="Votre mot de passe"
name="password" id="inputPassword" class="form-control" required>
    </div>

    <input type="hidden" name="_csrf_token"
        value="{{ csrf_token('authenticate') }}"
    >

    {#
        Uncomment this section and add a remember_me option below your
        firewall to activate remember me functionality.
        See https://symfony.com/doc/current/security/remember_me.html
    #}

    <div class="checkbox mb-3">
        <label>
            <input type="checkbox" name="_remember_me"> Remember me
        </label>
    </div>
    #}

    <button class="btn btn-lg btn-outline-primary" type="submit">
        Connexion
    </button>
</form>
{% endblock %}

```

Il faut créer des utilisateurs : `php bin/console make:migration` puis `php bin/console doctrine:migrations:migrate`

AJOUTER UN CONTROLLEUR POUR S'INSCRIRE

La commande : `php bin/console make:controller RegisterController`

Cette commande a pour effet de créer :

`Src/Controller/RegisterController` + `templates/register/index.html.twig`

Ici 2 choix :

- Créer le traitement du formulaire dans le contrôleur
- Générer un formulaire d'inscription : `php bin/console make:form RegisterType`

Externalisé le formulaire d'inscription :

`php bin/console make:form RegisterType`

Form/RegisterType.php :

```
<?php

namespace App\Form;

use App\Entity\User;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
use Symfony\Component\Form\Extension\Core\Type\EmailType;
use Symfony\Component\Form\Extension\Core\Type>PasswordType;
use Symfony\Component\Form\Extension\Core\Type\RepeatedType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class RegisterType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('email', EmailType::class)
            ->add('roles', ChoiceType::class, [
                'choices' => [
                    'ROLE_USER' => 'ROLE_USER',
                    'ROLE_ADMIN' => 'ROLE_ADMIN',
                    'ROLE_SUPER_ADMIN' => 'ROLE_SUPER_ADMIN',
                ],
                'multiple' => true
            ])
            ->add('password', RepeatedType::class, [
                'type' => PasswordType::class,
                'required' => true,
                'first_options' => ['label' => 'Mot de passe'],
                'second_options' => ['label' => "Confirmer mot de passe"]
            ])
        ;
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => User::class,
        ]);
    }
}
```

Puis le RegisterController :

```
<?php

namespace App\Controller;

use App\Entity\User;
use App\Form\RegisterType;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;
```

```

class RegisterController extends AbstractController
{
    /**
     * @Route("/inscription", name="register")
     */
    public function inscription(Request $request,
UserPasswordEncoderInterface $passwordEncoder): Response
    {
        // 1 - Construire le formulaire
        $user = new User();

        //Appel de RegisterType et association a entité User
        $formUser = $this->createForm(RegisterType::class, $user);

        //Ajout du bouton de soumission
        $formUser->add('register', SubmitType::class, [
            'label' => "S'inscrire",
            'attr' => [
                'class' => 'btn btn-outline-success'
            ]
        ]);

        //Request recup les données du formulaire
        $formUser->handleRequest($request);

        //Soumission et condition de valider
        if($request->isMethod('post') && $formUser->isValid()){
            //Accès aux champs du formulaire
            $data = $formUser->getData();
            //On encode le mot de passe
            $password = $passwordEncoder->encodePassword($user, $user-
>getPassword());
            $user->setPassword($password);

            //Les roles Appel de getRoles Array avec au minimul le Role
User
            $user->setRoles($user->getRoles());
            //Appel de Doctrine EntityManager
            $entityManager = $this->getDoctrine()->getManager();
            //Persistance des données
            $entityManager->persist($user);
            //Enregistrement des données
            $entityManager->flush();
            //Si ca marche on redirige vers login
            return $this->redirectToRoute('app_login');
        }

        //Appel de la vue et du formulaire d'inscription
        return $this->render('register/index.html.twig', [
            'controller_name' => 'RegisterController',
            'form_user' => $formUser->createView()
        ]);
    }
}

```

Enfin la vue register/index.html.twig

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



```
{% block title %}Sf5 -INSCRIPTION-{% endblock %}

{% block body %}

    <div class="container">
        <h1 class="text-warning">INSCRIPTION</h1>
        <!--Appel du $form->createView() du controller -->
        {{ form_start(form_user) }}

            {{ form_row(form_user.email) }}
            {{ form_row(form_user.roles) }}
            {{ form_row(form_user.password) }}

            <button class="btn btn-success" type="submit">S'inscrire</button>

        {{ form_end(form_user) }}
    </div>
{% endblock %}
```

Et enfin dans le menu.html.twig (rappel pour les routes : php bin/console debug :router)

```
<a class="nav-link" href="{{ path('register') }}"><b
class="h5">INSCRIPTION</b></a>
```

Un fois inscrits connectez-vous :



Le profile a changé.

LES AUTORISATIONS :

On va protéger les page d'administration (CRUD)

Plusieurs choix

- Access\_control = les autorisations au niveau des routes
- Acces au niveau du controleur
- Acces action = autorisation au niveau des methodes d'un controlleur
- Acces vue

ACCESS\_CONTROL : config/packages/security.yaml

Les 3 methodes ajouter + editer + supprimer de AdminController ne doivent etre utilisable que lorsque l'on connecté en tant qu'admin ou super admin

Dans AdminController au dessus de la declaration de la classe :

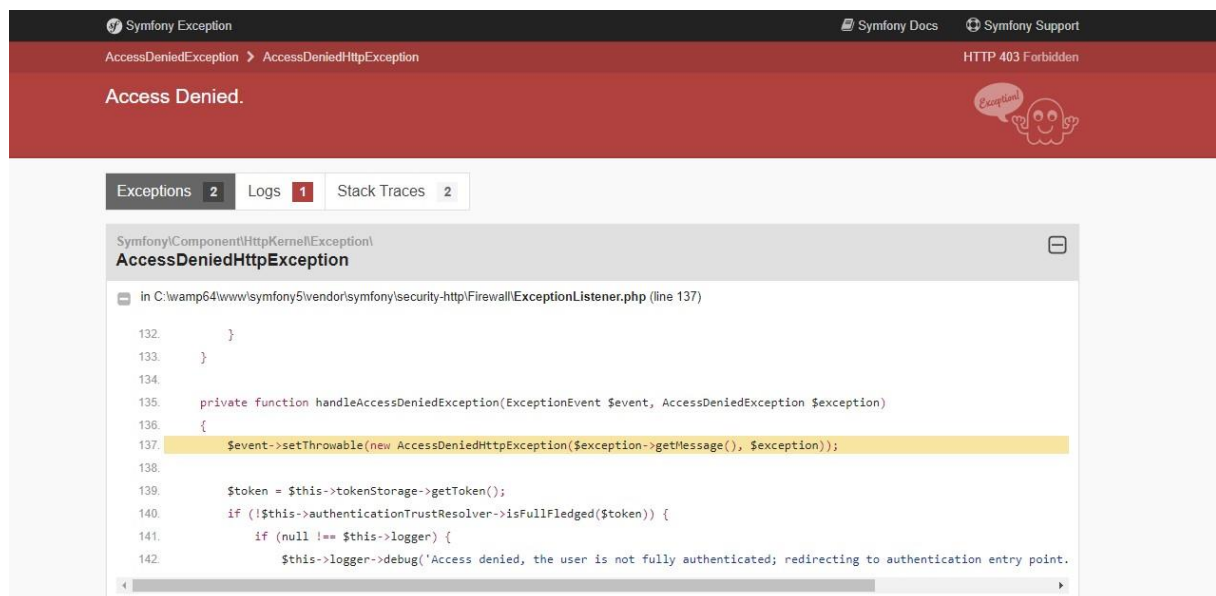
```
/**
 * Class AdminController
 * @package App\Controller
 * @Route ("/admin")
 */
class AdminController extends AbstractController
```

Puis dans security.yaml decommenté l'accès au route admin et son role :

```
access_control:
- { path: ^/admin, roles: ROLE_ADMIN, ROLE_SUPER_ADMIN }
```

Le symbole ^ signifie que toutes les routes qui commence par /admin sont interdite si on n'est pas connecté en tant qu'admin

Si on essaie d'accéder à ajouter :



Faire une redirection : La classe qui génère cette exception est AccessDeniedHandler.php

Créer sa propre classe d'erreur : src/Security/AccessDeniedHandler.php

Cette classe va hérité de AbstractController et implémenter AccessDeniedHandlerInterface

```
class AccessDeniedHandler extends AbstractController implements
AccessDeniedHandlerInterface
```

On ajoute simplement une redirection si le roles n'est pas bon

```
<?php
```

```
namespace App\Security;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Security\Core\Exception\AccessDeniedException;
use
Symfony\Component\Security\Http\Authorization\AccessDeniedHandlerInterface;

class AccessDeniedHandler extends AbstractController implements
AccessDeniedHandlerInterface
{
    public function handle(Request $request, AccessDeniedException
$accessDeniedException)
    {
        return $this->redirectToRoute('app_login');
    }
}
```

Dans security.yaml : (apres logout)

```
logout:
    path: app_logout
    # where to redirect after logout
    target: liste_produits
access_denied_handler: App\Security\AccessDeniedHandler
```

## IDEM DEPUIS UN CONTROLLEUR

```
/**
```

```
*@Security("is_granted('ROLE_ADMIN')")
```

Appelé le use IsGranted

Mettre en commenataire :

```
access_control:
    # - { path: ^/admin, roles: ROLE_SUPER_ADMIN }
    # - { path: ^/profile, roles: ROLE_USER }
```

## ACCES PAR ACTION :

```
/**
```

```
*@Security("is_granted('ROLE_ADMIN')")
```

Idem mais cette fois ci sous la route de la methodes

## ACCES PAR LA VUE :

Avec Twig {% if is\_granted('ROLE\_ADMIN') %}

```
<a href="{{ path('ajouter') }}">Ajouter produit</a>
```

```
{% endif %}
```

## PERSONALISE LES PAGES D'ERREURS :

Les erreurs avec Symfony entraînent des Exceptions qui correspondent à l'erreur générée

En production il est gênant de voir ces pages page 408

## L'INTERNATIONALISATION :

Traduire votre site dans plusieurs langues

3 éléments :

- Les éléments à traduire
- La langue locale
- Les dictionnaires de traduction

Il n'y a pas de système de traduction littérale c'est un transfert de chaîne de caractères d'une langue à l'autre en fonction d'une variable et du dictionnaire

ATTENTION À LA CASSE (Majuscule et Minuscule)

## LA VARIABLE LOCALE :

Config/packages/translation.yaml

## UN FORMULAIRE DE RECHERCHE (par catégorie et prix)

Créer une entité PropertySearch.php

```
<?php

namespace App\Entity;

class PropertySearch
{
    //ICI PAS ORM DOCTRINE -> Recherche par prix et catégorie

    //Liste déroulante
    /**
     * @var string
     */
    private $categorieProduit;
```

```

/**
 * @return string
 */
public function getCategorieProduit(): string
{
    return $this->categorieProduit;
}

/**
 * @param string $categorieProduit
 */
public function setCategorieProduit(string $categorieProduit): void
{
    $this->categorieProduit = $categorieProduit;
}

/**
 * @return int|null
 */
public function getMaxPrix(): ?int
{
    return $this->maxPrix;
}

/**
 * @param int|null $maxPrix
 */
public function setMaxPrix(?int $maxPrix): void
{
    $this->maxPrix = $maxPrix;
}

//Un entier ou null
/**
 * @var int|null
 */
private $maxPrix;
}

```

Generer un formulaire : php bin/console make :form

PropertySearchType

Lors de la creation du formulaire associé le \App\Entity\PropertySeach = le namespace + Nom de l'entité

## LES SERVICES



