

# SYMFONY formation débutant

## Sommaire

Introduction.....	2
Symfony c'est quoi ?.....	2
Documentation officielle.....	2
Installation des outils nécessaires pour notre projet Symfony.....	3
Création de notre premier projet.....	7
Structure de symfony.....	8
La base de donnée.....	12
Les contrôleurs.....	18
Les pages html.twig.....	20
Bonus contrôleurs.....	24
Les paramètres TWIG.....	26
Les conditions.....	28
Les entités / repository.....	30
Les formulaires.....	45
Les request et response.....	56
Gestion de l'authentification.....	57
Le sérialisation.....	58
Le formulaire.....	59
Les fixtures.....	63
Les flash.....	68
Formulaire d'inscription.....	70
Le modèle MVC.....	73
La vue :.....	73
Le contrôleur :.....	73
Le modèle :.....	73
Service Container.....	75
Le projet concret : Le gestionnaire de mot de passe.....	77
Installer Symfony.....	78
Installer les différents paquets nécessaires au projet.....	80
Création de la base de données.....	80
Many To Many.....	80
One To Many.....	80
Many To One.....	81
One To One.....	81
Gestion de l'inscription / Connexion.....	84
L'inscription.....	84
Les contraintes avec ASSERT.....	85
Bonus affichage erreurs.....	86
Connexion.....	87
Ajout de compte au gestionnaire.....	88
La suppression :.....	91
La modification :.....	92

# Introduction

Avant de commencer à vous aventurer dans le framework symfony, il vous faudra certaines bases en PHP natif ainsi qu'en HTML / CSS / JavaScript. **Symfony** est un outil de développement, il ne remplacera pas votre tête.

Il vous aidera à réussir vos projets avec plus de simplicité, mais vous devrez avoir une base solide en algorithmique pour vous lancer dans des projets concrets et aboutis.

Voici quelques liens pour apprendre les bases avant de commencer :

OpenClassRoom (gratuit) :

<https://openclassrooms.com/fr/courses/1603881-creez-votre-site-web-avec-html5-et-css3>  
<https://openclassrooms.com/fr/courses/918836-concevez-votre-site-web-avec-php-et-mysql>  
<https://openclassrooms.com/fr/courses/7696886-apprenez-a-programmer-avec-javascript>

Et si vous êtes prêt à déboursier quelques euros je vous conseil Udemmy :

<https://www.udemy.com/course/developpeur-php-html-css-la-formation-complete-2023/>  
<https://www.udemy.com/course/php-et-mysql-la-formation-ultime/>  
<https://www.udemy.com/course/javascript-la-formation-ultime/>  
<https://www.udemy.com/course/le-javascript-de-a-a-z/>

---

## Symfony c'est quoi ?

Le framework\* PHP Symfony est aujourd'hui l'un des frameworks PHP les plus utilisé dans le monde professionnel. Sponsorisé par l'éditeur **SensioLabs**, **Symfony** vous fournit une palette d'outils pour construire des applications web sur mesure.

La première version du **framework** a été publié en 2005, mais **Symfony 2** a réellement fait un pas-de-géant lors de la sortie de sa version 2 en 2011. **Symfony** vous permet de réaliser des projets d'applications web professionnel, du développement sur mesure, de gérer votre base de données, etc..

Avec Symfony, vous développez des services réutilisables.

*\* un framework est un ensemble cohérent de composants logiciels structurels qui sert à créer les fondations ainsi que les grandes lignes de tout ou partie d'un logiciel, c'est-à-dire une architecture.*

---

## Documentation officielle

Je vous invite à vous rendre également sur la doc officielle de Symfony pour avoir des informations complémentaires :

<https://symfony.com/doc/current/index.html>

# Installation des outils nécessaires pour notre projet Symfony

Avant toute chose, vous devez installer Visual Studio Code (VSC) :

<https://code.visualstudio.com>

C'est un éditeur pour commencer à développer vos projets et interagir avec les différentes lignes de commande au bon fonctionnement de Symfony, il s'appuiera sur l'invite de commande GIT.

<https://git-scm.com>

Git est un CMD (invite de commande) nécessaire pour l'exécution des futures commandes d'installation. Basé sur la structure linux, Git est un indispensable dans le monde du développement.

Soutenu par Node.js pour le côté serveur. Node.js nous permettra de lancer notre application Symfony depuis l'invite de commande.

<https://nodejs.org/en>

Je vous recommande de prendre la version LTS (*c'est une version stable, mais qui sera remplacé par la « Current » quand celle-ci sera dépassée.*).

*La version LTS de Node.js est une version de la plateforme qui est activement maintenue et prise en charge par la communauté Node.js. Cela signifie que les versions LTS de Node.js reçoivent régulièrement des mises à jour et des corrections de bugs, et qu'elles fournissent également une assistance pendant au moins 18 mois.*

*La version stable de Node.js est la dernière version de la plateforme et inclut les dernières fonctionnalités et améliorations. Bien que la version stable soit également considérée comme prête pour la production, elle peut contenir des fonctionnalités nouvelles ou non testées et, en tant que telle, peut être moins stable que la version LTS.*

Nous allons également installer Composer, c'est l'outil indispensable pour installer différentes librairies (*package*) que nous utiliserons pour notre projet Symfony.

<https://getcomposer.org>

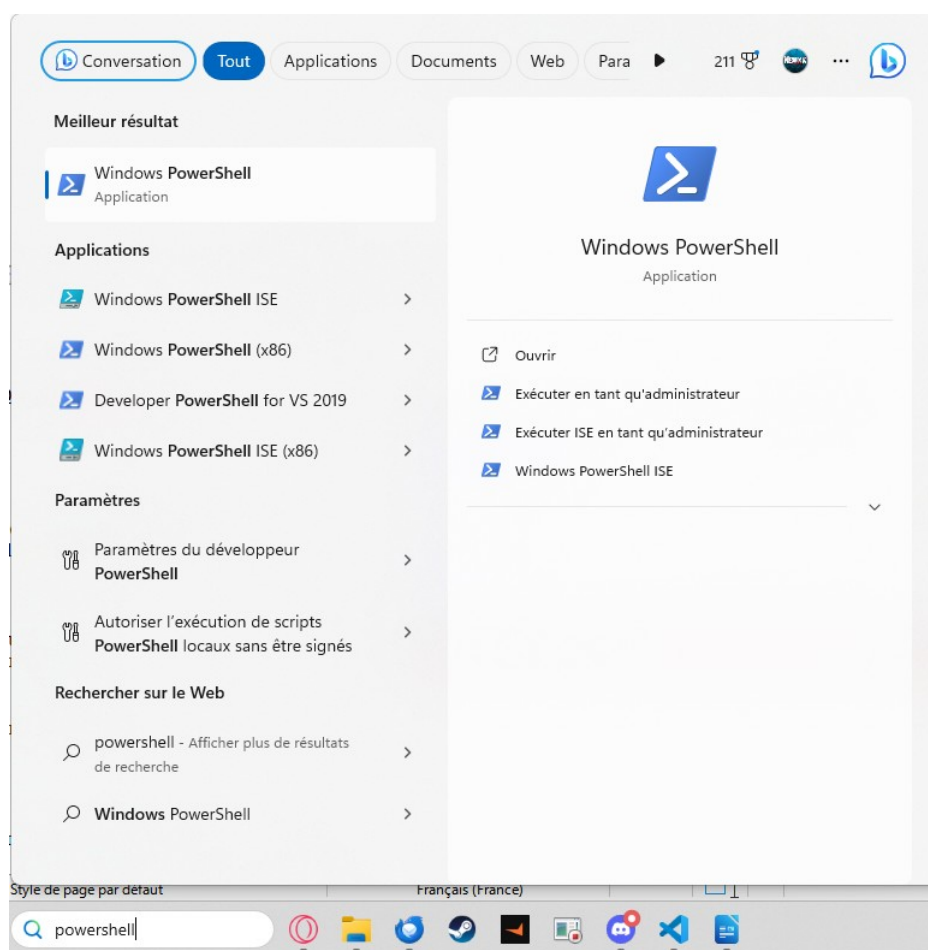
WampServer, c'est l'outil qui nous permettra de gérer notre base de données :

<https://www.wampserver.com>

Et pour terminer dans l'installation des différents modules pour créer notre projet, il nous faudra scoop :

<https://scoop.sh>

Commande à exécuter dans un CMD powershell (windows) :



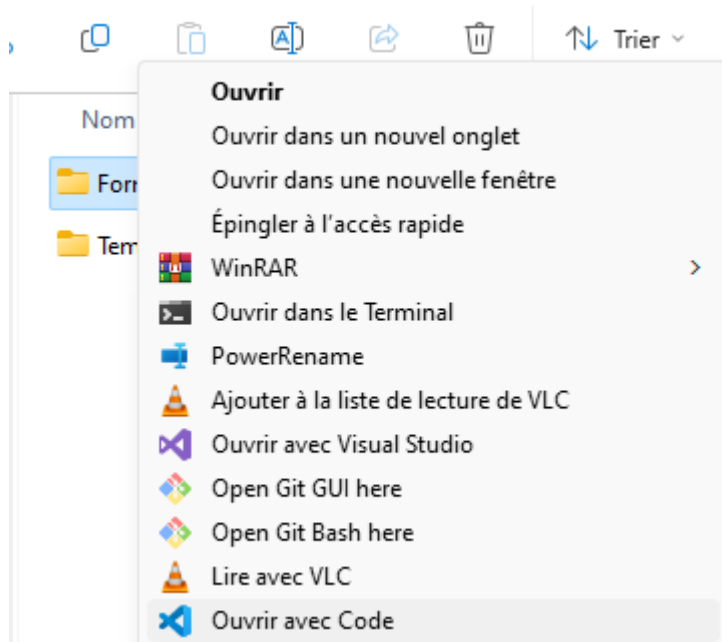
Pour éviter des problèmes de droits, lancer le powershell en mode admin (*Clique droit > Exécuter en mode administrateur*)

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

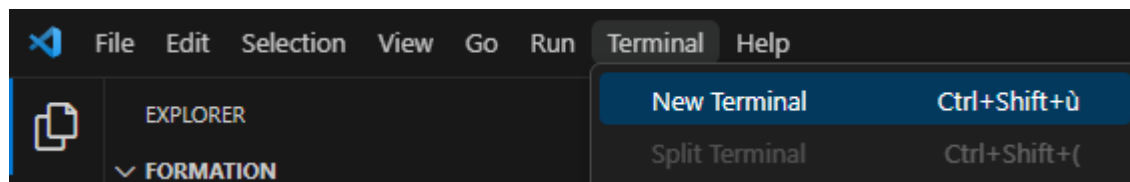
```
Invoke-RestMethod -Uri https://get.scoop.sh | Invoke-Expression
```

Dès que tout est installé correctement, vous êtes prêt à créer votre premier projet Symfony. Il nous faudra encore utiliser l'invite de commande, mais cette fois depuis VSC.

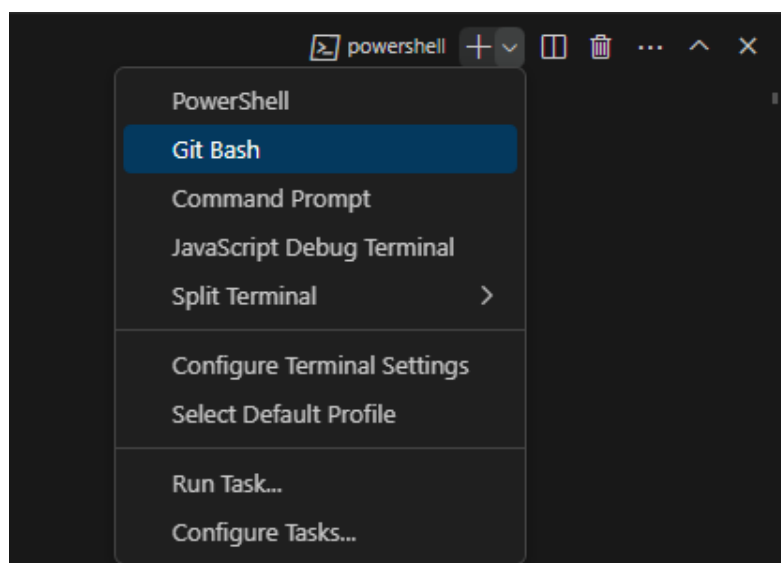
Créer un dossier (de préférence dans le dossier WWW de wamp) puis ouvrez le avec VSC (cliquez droit sur le dossier > Ouvrir avec visual studio code).



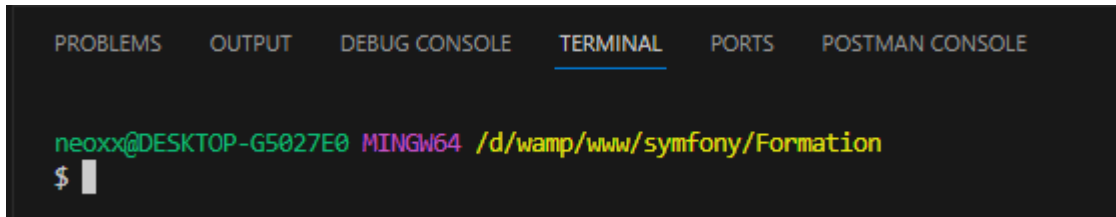
Dès que VSC est ouvert, ouvrez un terminal :



Dans le terminal à droite choisissez Git Bash :



Si c'est bon vous devriez avoir ceci :



```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation
$
```

La zone verte correspond à l'utilisateur connecté sur l'ordinateur.

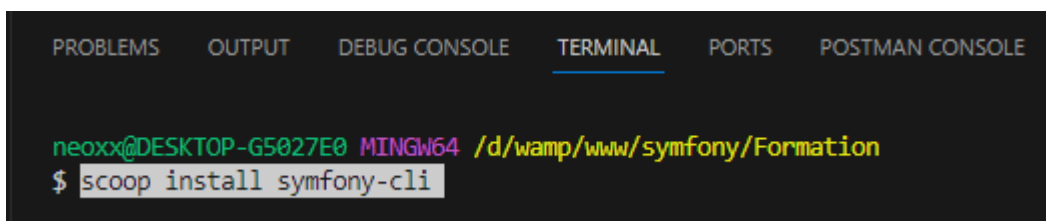
La zone Rose définit votre système d'exploitation.

Et la zone jaune définit votre chemin, là où se trouvent vos différents dossiers.

Pour notre cas seul la zone jaune nous est important. Il faudra garder un œil attentif à celui-ci pour éviter diverses bourdes.

Dès que vous êtes bien dans le dossier qui contiendra notre projet, vous pouvez lancer cette ligne de commande :

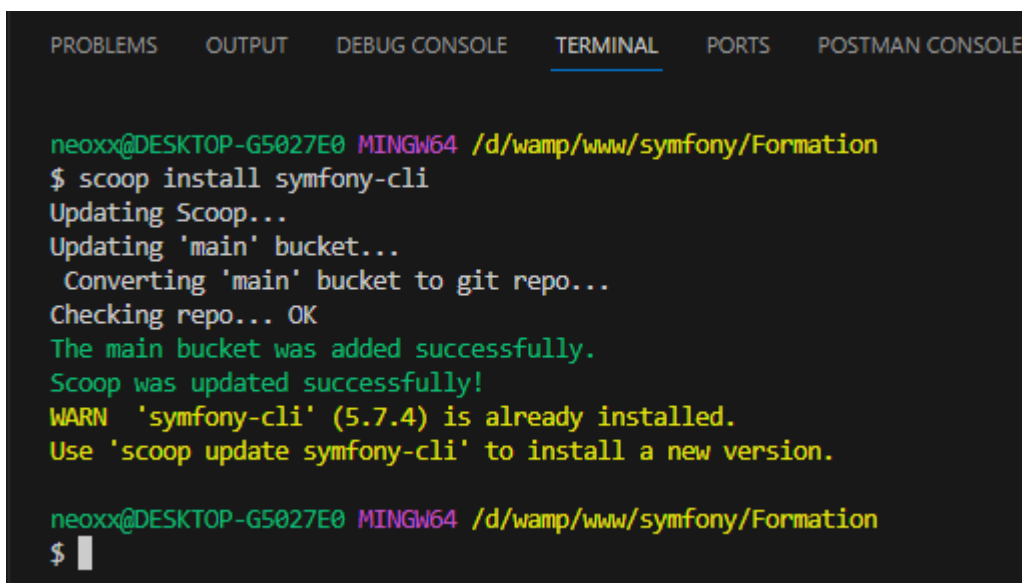
```
scoop install symfony-cli
```



```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation
$ scoop install symfony-cli
```

Vous pouvez copier la ligne de commande et faire clique droit pour la coller dans le terminal.

Appuyez sur entrée.



```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation
$ scoop install symfony-cli
Updating Scoop...
Updating 'main' bucket...
  Converting 'main' bucket to git repo...
Checking repo... OK
The main bucket was added successfully.
Scoop was updated successfully!
WARN 'symfony-cli' (5.7.4) is already installed.
Use 'scoop update symfony-cli' to install a new version.

neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation
$
```

Si l'installation est réussit vous aurez le message ci-dessus affiché.

Cette ligne de commande permet d'utiliser les lignes de commande intégrée à Symfony pour vous faciliter la gestion de celui-ci.

Exemple :

Commande sans le **Symfony-CLI** :

- 1/ `composer create-project symfony/skeleton:"7.0.*@dev" my_project_directory`
- 2/ `cd my_project_directory`
- 3/ `composer require webapp`

Commande avec **Symfony-CLI** :

- 1/ `symfony new my_project_directory --version="7.0.*@dev" --webapp`

(Si vous utilisez une version inférieure à la 8.2 de PHP utilisez cette commande :  
`symfony new --webapp my_project`)

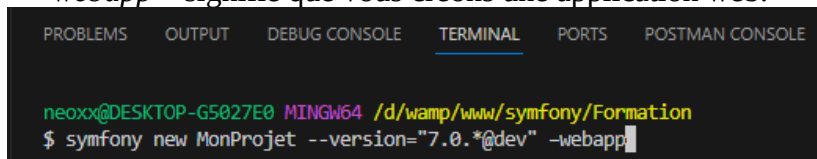
Comme on peut le voir, ça raccourcit légèrement notre ligne de commande et au lieu de devoir chercher quoi appeler pour lancer une commande, nous utiliserons « *Symfony* » pour lancer une commande.

## Création de notre premier projet

Maintenant, que **Symfony-CLI** est installé nous allons créer notre projet.

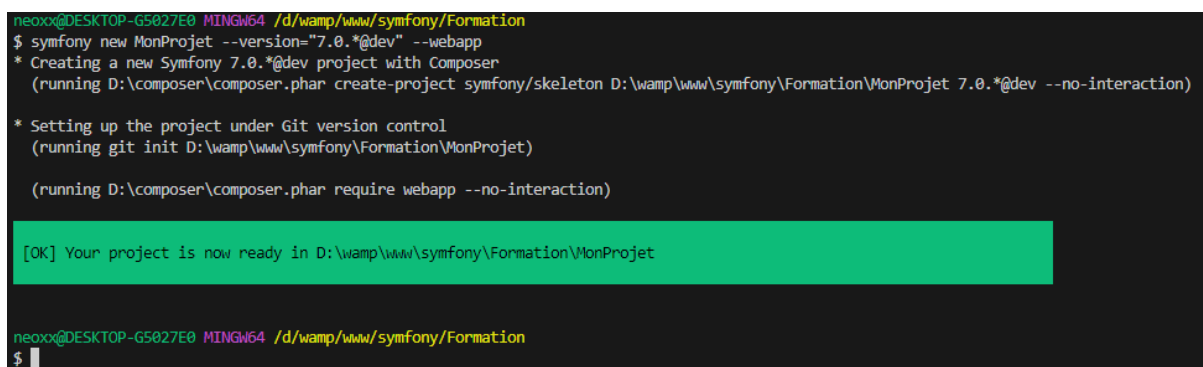
Utilisez la commande **Symfony** ci-dessus en remplaçant **my\_project\_directory** par le nom de votre projet (*exemple : Twitter*).

« *--webapp* » signifie que vous créons une application web.



```
neoux@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation
$ symfony new MonProjet --version="7.0.*@dev" --webapp
```

NB : Nous utilisons la version 7 de Symfony à la création de ce tutoriel, adaptez la version selon la version actuelle de Symfony. Si vous êtes sur la 8, remplacez le 7 par 8.



```
neoux@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation
$ symfony new MonProjet --version="7.0.*@dev" --webapp
* Creating a new Symfony 7.0.*@dev project with Composer
(running D:\composer\composer.phar create-project symfony/skeleton D:\wamp\www\symfony\FORMATION\MonProjet 7.0.*@dev --no-interaction)

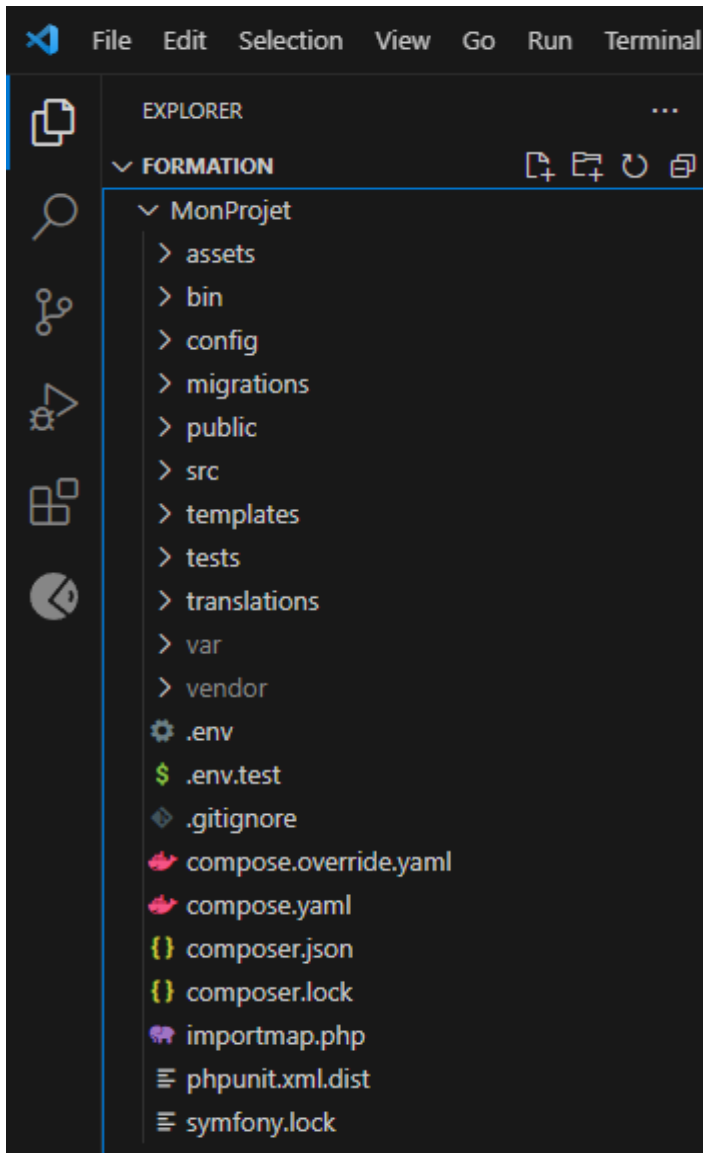
* Setting up the project under Git version control
(running git init D:\wamp\www\symfony\FORMATION\MonProjet)

(running D:\composer\composer.phar require webapp --no-interaction)

[OK] Your project is now ready in D:\wamp\www\symfony\FORMATION\MonProjet

neoux@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation
$
```

# Structure de symfony



On peut constater que dans le nom de notre projet plein de dossier se sont créés.

Nous allons les voir ensemble :

**Assets** : Ce dossier contient les différents style.css ou script JavaScript au bon fonctionnement de notre projet.

**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*).

**Config** : Ce dossier contient toute la configuration de votre application, que ce soit le framework, les dépendances (*Doctrine, Twig, Monolog*) ou encore les routes.

**Migrations** : Dans ce dossier et si vous manipulez une base de données, alors vous trouverez les migrations de votre projet générées à chaque changement que vous effectuerez sur votre base de données à l'aide de l'ORM Doctrine.

**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

**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.

**Templates** : Ce dossier contient les gabarits qui sont utilisés dans votre projet (*c'est ce qu'on verra côté client*).

**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.

**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.

**Var** : Ce dossier contient trois choses principalement :

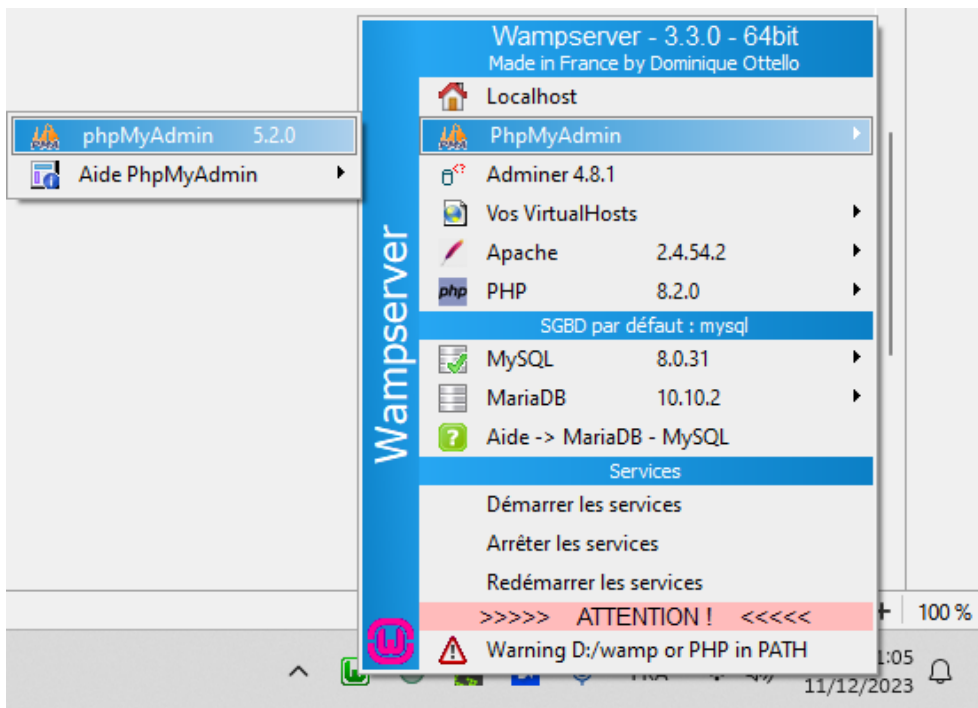
- les fichiers de cache dans le dossier cache ;
- les fichiers de log dans le dossier log ;
- et parfois, si le framework est configuré pour gérer les sessions PHP dans le système de fichiers, on trouve le dossier sessions.

**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**".



Maintenant, que nous avons créé notre projet, allumons WampServer pour accéder à notre base de données.

Faite un clique gauche sur le logo wamp puis allez dans PHPMyAdmin :



Les identifiants de PHPMyAdmin sont souvent :

Et si ça ne fonctionne pas, c'est que vous avez mis « root » en mot de passe également.

NB : Passez sur la version 8.2 de PHP pour être sur la dernière version (*pour ça, clique droit sur le logo wamp>PHP>versions>8.2.0*)

Rendez-vous dans le dossier de notre projet, faire une copie de fichier .env :

Nom	Modifié le	Type	Taille
src	11/12/2023 20:38	Dossier de fichiers	
templates	11/12/2023 20:38	Dossier de fichiers	
tests	11/12/2023 20:38	Dossier de fichiers	
translations	11/12/2023 20:38	Dossier de fichiers	
var	11/12/2023 20:38	Dossier de fichiers	
vendor	11/12/2023 20:38	Dossier de fichiers	
.env	11/12/2023 20:38	Fichier ENV	2 Ko

Et renommez le « .env.local » ça nous permet de saisir les informations de connexion à la base de données sans impacter le fichier .env qui sera à configurer pour la mise en ligne du projet.

.env	28/12/2023 17:36	Fichier ENV	2 Ko
.env.local	28/12/2023 17:36	Fichier LOCAL	2 Ko

Dans VSC, allez dans le fichier .env.local pour y modifier les accès notre future base de données.

```
$ .env.local x
MonProjet > $ .env.local
14 # Run "composer dump-env prod" to compile .env files for production use (requires symfony/flex >=1.2).
15 # https://symfony.com/doc/current/best_practices.html#use-environment-variables-for-infrastructure-configuration
16
17 ###> symfony/framework-bundle ###
18 APP_ENV=dev
19 APP_SECRET=7376d64dc108e7096441aca4c76f6cf1
20 ###< symfony/framework-bundle ###
21
22 ###> doctrine/doctrine-bundle ###
23 # Format described at https://www.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/configuration.html#connecting-using-a-url
24 # IMPORTANT: You MUST configure your server version, either here or in config/packages/doctrine.yaml
25 #
26 # DATABASE_URL="sqlite://%kernel.project_dir%/var/data.db"
27 # DATABASE_URL="mysql://app:!ChangeMe!@127.0.0.1:3306/app?serverVersion=8.0.32&charset=utf8mb4"
28 # DATABASE_URL="mysql://app:!ChangeMe!@127.0.0.1:3306/app?serverVersion=10.11.2-MariaDB&charset=utf8mb4"
29 DATABASE_URL="postgresql://app:!ChangeMe!@127.0.0.1:5432/app?serverVersion=15&charset=utf8"
30 ###< doctrine/doctrine-bundle ###
31
32 ###> symfony/messenger ###
33 # Choose one of the transports below
34 # MESSENGER_TRANSPORT_DSN=amqp://guest:guest@localhost:5672/%2f/messages
35 # MESSENGER_TRANSPORT_DSN=redis://localhost:6379/messages
36 MESSENGER_TRANSPORT_DSN=doctrine://default?auto_setup=0
37 ###< symfony/messenger ###
38
39 ###> symfony/mailer ###
40 # MAILER_DSN=null://null
41 ###< symfony/mailer ###
42
```

Cherchez DATABASE\_URL, nous allons détailler sa structure et aussi savoir lequel choisir.

**DATABASE\_URL="postgresql://app:!ChangeMe!@127.0.0.1:5432/app?serverVersion=15&charset=utf8"**

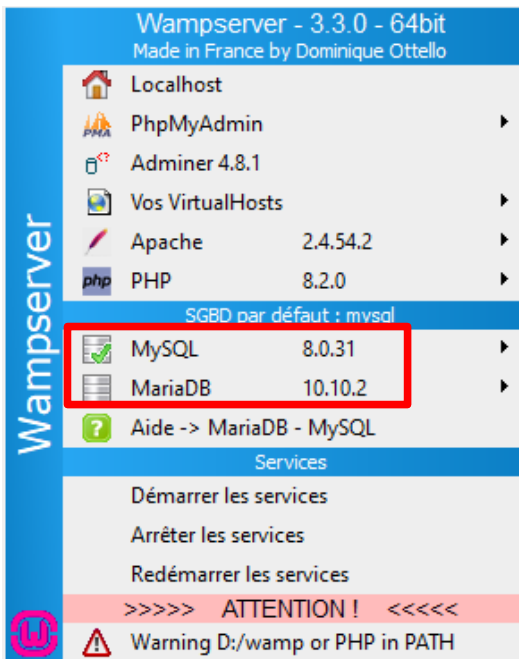
Le **postgresql** sera le moyen SQL que l'on utilisera dans notre cas, on utilisera mysql

Le **app** est notre nom de connexion soit « root »

Le **!ChangeMe!** sera le mot de passe pour accéder à vos bases de données (Si vous n'en avez pas, enlever le ainsi que les : juste avant)

Le **app** sera le nom que l'on donnera à notre future base de données.

Et pour finir, le **15** est la version de notre gestionnaire de bases de données pour **mysql** référez-vous à la version de Wampserver.



Dans le ce tutoriel nous utiliserons **mysql** pour ça ajoutez un # devant le **DATABASE\_URL** non commenté et enlever celui contenant **mysql** avec **MariaDB** (entrer la version de votre MariaDB).

Remplissez les champs selon votre configuration.

```
# DATABASE_URL="sqlite:///kernel.project_dir%/var/data.db"
# DATABASE_URL="mysql://app:!ChangeMe!@127.0.0.1:3306/app?serverVersion=8.0.32&charset=utf8mb4"
DATABASE_URL="mysql://root:root@127.0.0.1:3306/monProjet?serverVersion=10.10.2-MariaDB&charset=utf8mb4"
# DATABASE_URL="postgresql://app:!ChangeMe!@127.0.0.1:5432/app?serverVersion=15&charset=utf8"
###< doctrine/doctrine-bundle ###
```

Pour la version, laissez celle déjà présente, si un conflit survient, changez le pour la version de votre mysql Wampserver (cf. voir plus haut).

Avant la création de notre base de données, on va ajouter 2 paquets à Symfony, toujours dans le dossier de votre projet, lancer les commandes suivante :

**cd MonProjet**

**composer require symfony/orm-pack**

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE

neorx@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)
$ composer require symfony/orm-pack
```

**composer require --dev symfony/maker-bundle**

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE

neorx@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)
$ composer require --dev symfony/maker-bundle
```

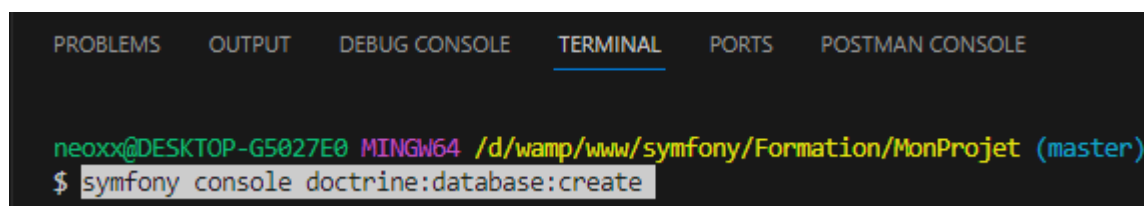
**ORM-pack** servira à la création et gestion des requêtes SQL, ce paquet est requis pour toute demande SQL via doctrine.

**Maker-bundle** sert à créer des entités pour la création des tables SQL.

## La base de donnée

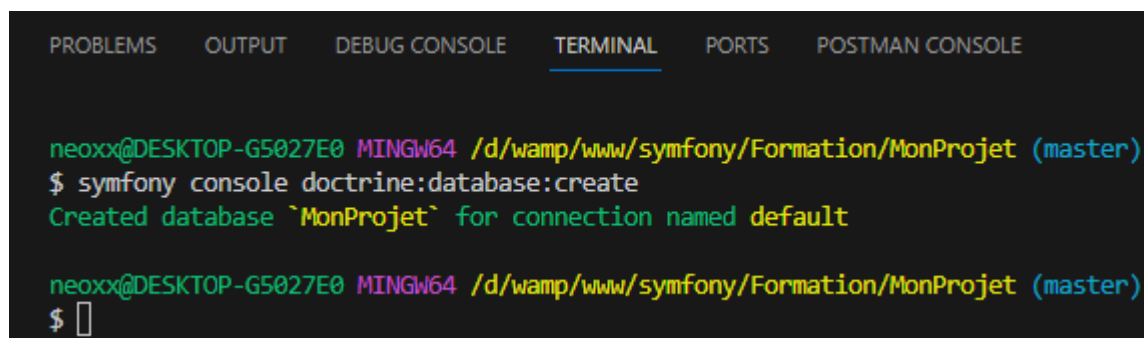
Dès que ses paquets sont installés, nous allons créer notre base de données via la commande :

```
symfony console doctrine:database:create
```



```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)
$ symfony console doctrine:database:create
```

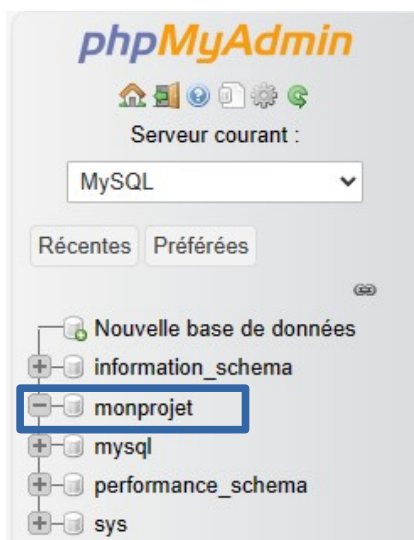
Si ça a bien fonctionné, vous devrez avoir ceci :



```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)
$ symfony console doctrine:database:create
Created database `MonProjet` for connection named default

neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)
$
```

Pour vérifier encore une fois si tout est dans l'ordre, allez sur PHPMyAdmin et regardez que la base de données à bien été créé :



Notre base a bien été créée ! Félicitations !

Sur notre projet, nous allons créer une table «identifiant» qui contiendra :

l'email, mot de passe, rowguid (*clé personnelle*) et compte\_block pour définir les erreurs de l'utilisateur lors de multiples erreurs lors d'une action spécifique pour bloquer le compte et ainsi sécuriser les utilisateurs.

Nous pourrions ajouter des tables dans le futur si le projet évolue sans impacter les données déjà entrées.

Vous souvenez-vous, du paquet **maker-bundle** ? Si, non, je vous invite à retourner le voir plus haut. Si oui, nous allons utiliser ce paquet pour créer notre première table.

Nous allons utiliser :

```
symfony console make:user
```

Pour définir les champs importants et qui devront être sécurisé via le fichier security.yaml qui se trouve dans le dossier « config/packages/security.yaml ».

Dans le premier champs, on spécifie le nom de la table que l'on créera.

```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)
$ symfony console make:user

The name of the security user class (e.g. User) [User]:
> identifiant
```

Sur le second champ Symfony, nous demande si on veut stocker les informations de l'utilisateur dans notre base de données. Ici oui.

```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)
$ symfony console make:user

The name of the security user class (e.g. User) [User]:
> identifiant

Do you want to store user data in the database (via Doctrine)? (yes/no) [yes]:
> yes
```

Lors du troisième champ, Symfony nous demande d'entrée un nom de colonne qui sera unique lors de sa création (exemple : Nous avons l'utilisateur [aze@gmail.com](mailto:aze@gmail.com), personne ne pourra créer de compte s'il utilise l'email [aze@gmail.com](mailto:aze@gmail.com)). Ici, nous allons garder la configuration par défaut (email) :

```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)
$ symfony console make:user

The name of the security user class (e.g. User) [User]:
> identifiant

Do you want to store user data in the database (via Doctrine)? (yes/no) [yes]:
> yes

Enter a property name that will be the unique "display" name for the user (e.g. email, username, uid) [email]:
> 
```

Sur le quatrième champ, on vous demande si on doit hasher les mots de passe, pour des raisons de sécurité oui !

```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)
$ symfony console make:user

The name of the security user class (e.g. User) [User]:
> identifiant

Do you want to store user data in the database (via Doctrine)? (yes/no) [yes]:
> yes

Enter a property name that will be the unique "display" name for the user (e.g. email, username, uuid) [email]:
>

Will this app need to hash/check user passwords? Choose No if passwords are not needed or will be checked/hashed by some other system (e.g. a single sign-on server).
Does this app need to hash/check user passwords? (yes/no) [yes]:
> yes
```

Vous avez créé votre première entité utilisateur :

```
created: src/Entity/Identifiant.php
created: src/Repository/IdentifiantRepository.php
updated: src/Entity/Identifiant.php
updated: config/packages/security.yaml

Success!

Next Steps:
- Review your new App\Entity\Identifiant class.
- Use make:entity to add more fields to your Identifiant entity and then run make:migration.
- Create a way to authenticate! See https://symfony.com/doc/current/security.html

neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)
$
```

Maintenant, ajoutons des colonnes supplémentaire à notre table **identifiant** en utilisant la commande :

```
symfony console make:entity
```

Comme la commande précédente, celle-ci nous permet soit de créer une nouvelle table, mais hors de la sécurité (*de préférence on ne crée pas de table qui stocke des mots de passe*) ou d'ajouter des colonnes à une table déjà existante.

```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)
$ symfony console make:entity

Class name of the entity to create or update (e.g. BravePopsicle):
> identifiant
```

En entrant **identifiant** Symfony, nous dira que l'entité existe et nous permettra donc d'y ajouter des colonnes.

```

neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)
$ symfony console make:entity

Class name of the entity to create or update (e.g. BravePopsicle):
> identifiant
identifiant

Your entity already exists! So let's add some new fields!

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

```

Ajoutons « rowguid » en tant que **string 255** et il ne peut pas être **null** :

```

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

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

Field length [255]:
> 255

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

updated: src/Entity/Identifiant.php

```

De la même manière ajoutons `compte_block`, il sera `bigint` et ne pourra pas être `null` :

```

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

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

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

updated: src/Entity/Identifiant.php

```

Pour terminer l'édition de notre table, faite entrer sans rien mettre dans le champ :

```

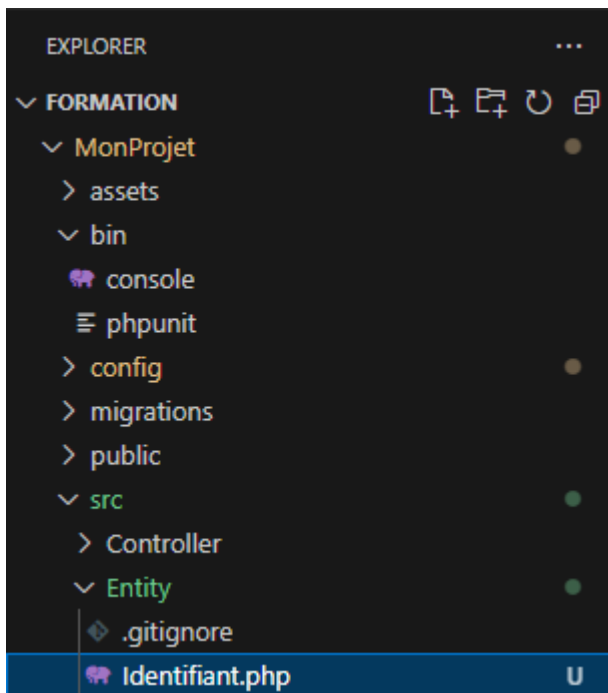
Add another property? Enter the property name (or press <return> to stop adding fields):
>

Success!

Next: When you're ready, create a migration with symfony.exe console make:migration

```

Nous avons donc notre base sur notre table SQL dans le fichier  
« *src>entity>Identifiant.php* »



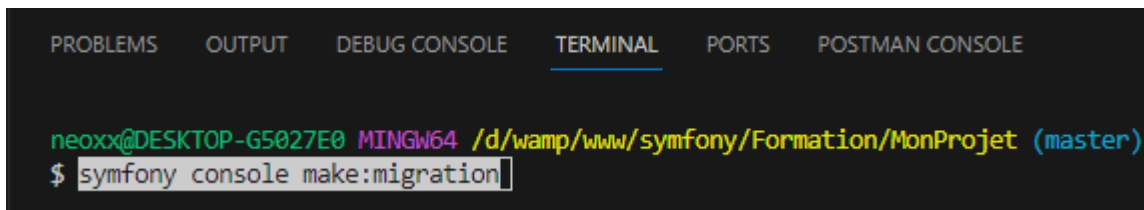
Si on regarde un peu le fichier Identifiant.php, on peut constater que l'on utilise ORM (notre paquet).

```
$.env.local  Identifiant.php U x
MonProjet > src > Entity > Identifiant.php > Identifiant
1  <?php
2
3  namespace App\Entity;
4
5  use App\Repository\IdentifiantRepository;
6  use Doctrine\DBAL\Types\Types;
7  use Doctrine\ORM\Mapping as ORM;
8  use Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface;
9  use Symfony\Component\Security\Core\User\UserInterface;
10
11  #[ORM\Entity(repositoryClass: IdentifiantRepository::class)]
12  class Identifiant implements UserInterface, PasswordAuthenticatedUserInterface
13  {
14      #[ORM\Id]
15      #[ORM\GeneratedValue]
16      #[ORM\Column]
17      private ?int $id = null;
18
19      #[ORM\Column(length: 180, unique: true)]
20      private ?string $email = null;
21
22      #[ORM\Column]
23      private array $roles = [];
24
25      /**
26       * @var string The hashed password
27       */
28      #[ORM\Column]
29      private ?string $password = null;
30
31      #[ORM\Column(length: 255)]
32      private ?string $rowguid = null;
33
34      #[ORM\Column(type: Types::BIGINT)]
35      private ?string $compte_block = null;
36
37      public function getId(): ?int
```



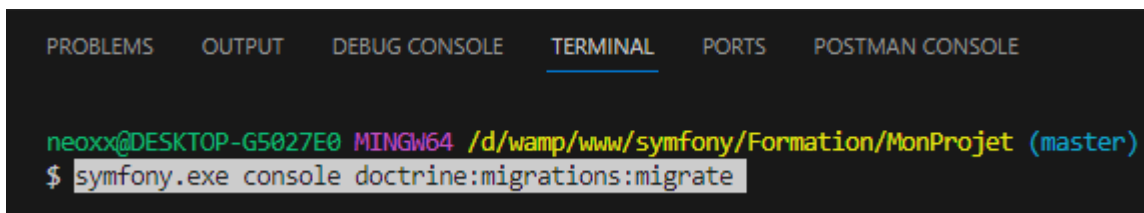
Maintenant, ajoutons notre table Identifiant à notre base de données créée préalablement en utilisant les commandes :

```
symfony console make:migration
```



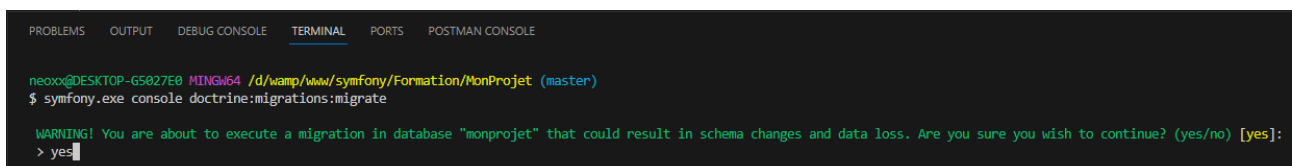
```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)
$ symfony console make:migration
```

```
symfony.exe console doctrine:migrations:migrate
```



```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)
$ symfony.exe console doctrine:migrations:migrate
```

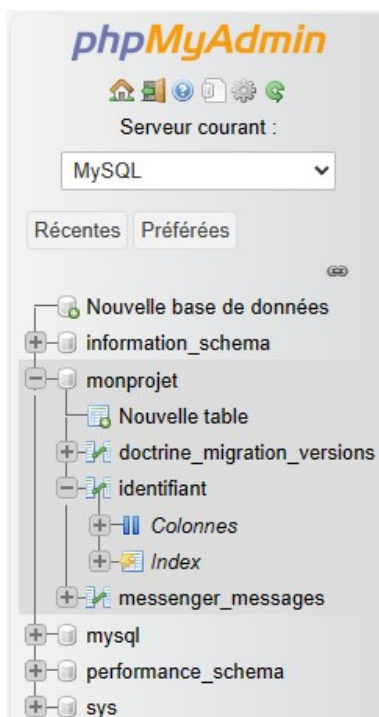
Vous aurez un warning, faites YES puis entrée.



```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)
$ symfony.exe console doctrine:migrations:migrate

WARNING! You are about to execute a migration in database "monprojet" that could result in schema changes and data loss. Are you sure you wish to continue? (yes/no) [yes]:
> yes
```

Voilà, notre table a été push dans notre base de données, pour bien vérifier ça, rendez-vous dans PHPMYAdmin et allons voir notre base.



# Les contrôleurs

Histoire de changer d'air, parlons des contrôleurs sous symfony.

Un contrôleur est une fonction PHP que vous créez et qui lit les informations de l'objet **Request** et crée et renvoie un objet Response. La réponse peut être une page **HTML**, **JSON**, **XML**, un téléchargement de fichier, une redirection, une erreur 404 ou autre. Le contrôleur exécute toute logique arbitraire dont votre application a besoin pour rendre le contenu d'une page.

Avant de commencer, ajoutons un certificat SSL à notre projet web pour éviter tout problème lors de notre déploiement dans le future.

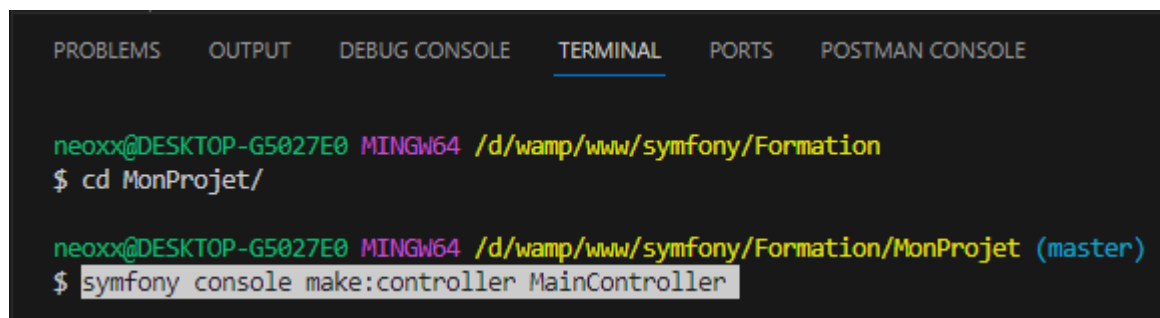
**Symfony server:ca:install**

Le contrôleur sera là pour exécuter du code avant la mise en affichage d'élément.

Prenons par exemple l'affichage de notre première page web html (twig).

Créons un contrôleur qui prendra la commande suivante :

**symfony console make:controller MainController**

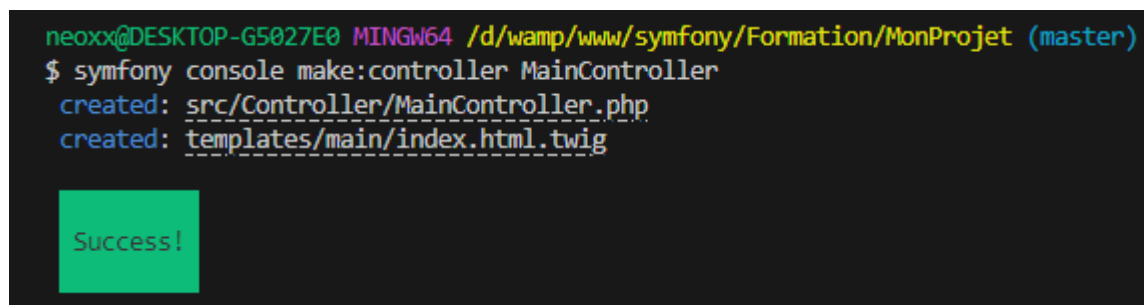


```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  POSTMAN CONSOLE

neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation
$ cd MonProjet/

neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)
$ symfony console make:controller MainController
```

Comme on peut le voir dans notre invite de commande **GIT**, la commande précédente nous à créé une page index.html.twig dans le dossier **templates**, ainsi qu'un fichier MainController.php dans le dossier « *src>controller>MainController.php* »



```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)
$ symfony console make:controller MainController
created: src/Controller/MainController.php
created: templates/main/index.html.twig

Success!
```

Le fichier index.html.twig sera notre page d'accueil, nous reverrons cela plus tard.

Allons faire un tour dans notre fichier contrôleur.

```
MonProjet > src > Controller > MainController.php > ...
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 MainController extends AbstractController
10 {
11     #[Route('/main', name: 'app_main')]
12     public function index(): Response
13     {
14         return $this->render('main/index.html.twig', [
15             'controller_name' => 'MainController',
16         ]);
17     }
18 }
```

On peut voir que notre page PHP n'est pas très grande, car, actuellement, elle ne sert qu'à la redirection du projet sur l'affichage index.html.twig (*page de garde*).

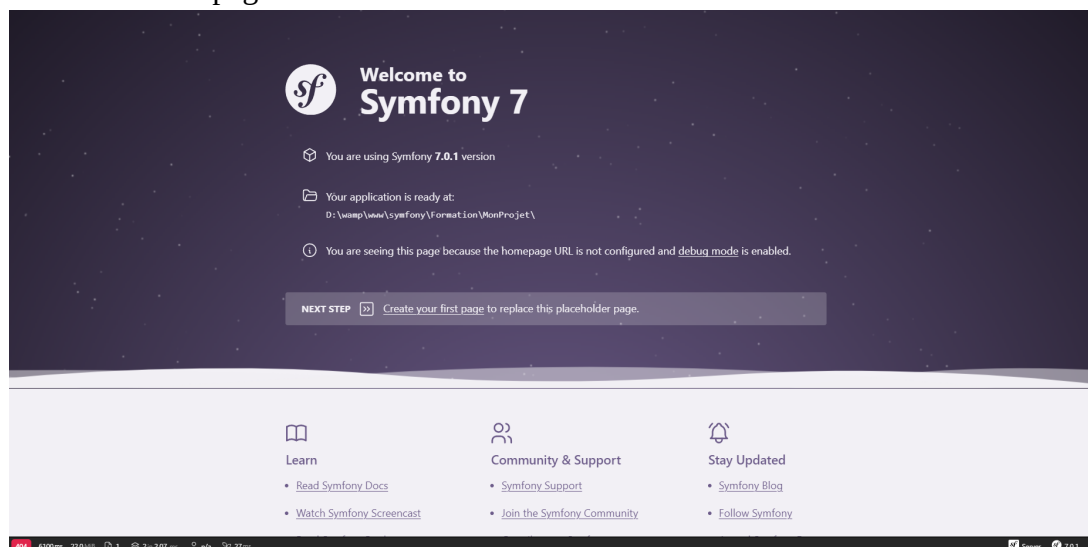
Vérifions cela plus en détail :

### **symfony serve**

Cette commande permet de lancer notre serveur web local pour voir notre projet enfin prendre vie.

Pour voir l'affichage de votre page rendez-vous sur : <https://127.0.0.1:8000>.

Vous aurez une page comme ceci :

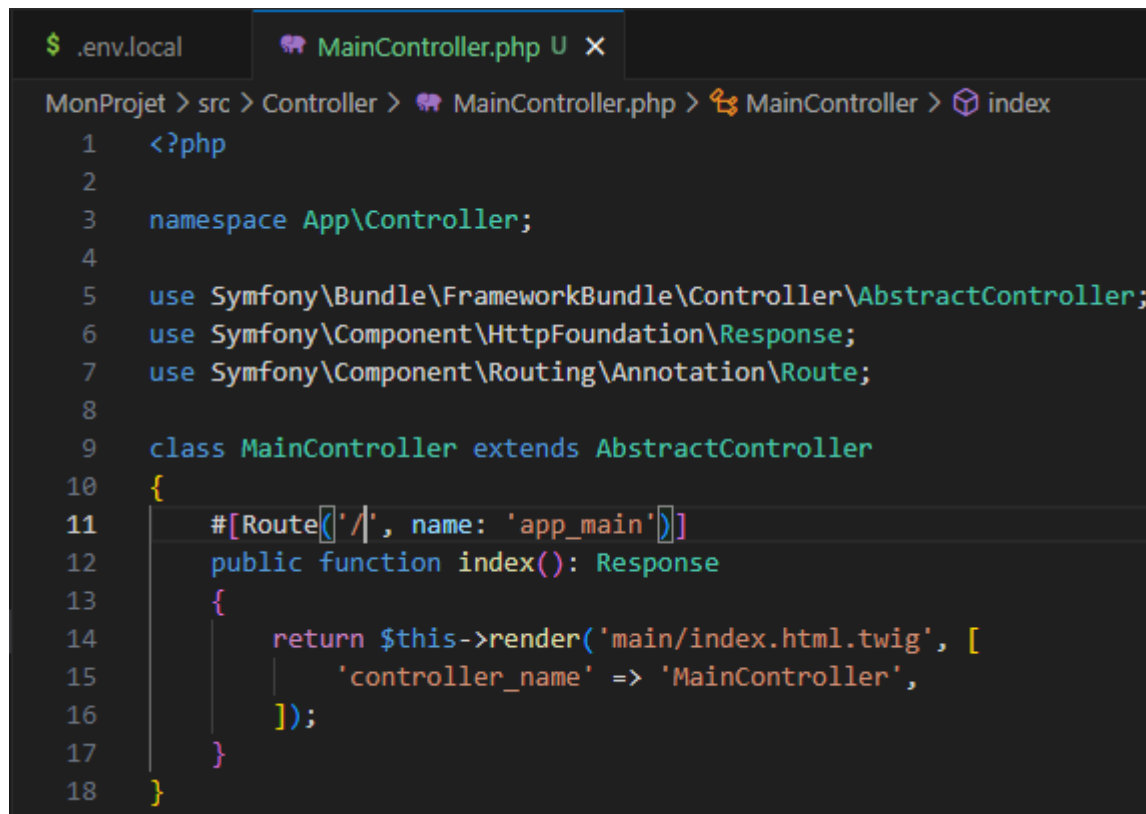


Et utilisons **CTRL+C** ou **symfony serve:stop** pour arrêter notre serveur web.

La page affichée nous dit que notre projet est prêt à l'emploi (affichage seulement pour le moment), mais nous n'avons pas encore accès à notre index.html.twig.

Pour ça, allez dans notre MainController.php puis changez la valeur « `#[Route('/', name: 'app_main')]` » en « `#[Route('/', name: 'app_main')]` ».

Nous verrons les Routes plus en détails dans un chapitre dédié !



```
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 MainController extends AbstractController
10 {
11     #[Route('/', name: 'app_main')]
12     public function index(): Response
13     {
14         return $this->render('main/index.html.twig', [
15             'controller_name' => 'MainController',
16         ]);
17     }
18 }
```

Actualiser votre navigateur et vous verrez que notre page principale a changé !

# Hello MainController!

This friendly message is coming from:

- Your controller at <src/Controller/MainController.php>
- Your template at <templates/main/index.html.twig>

Maintenant, vous avez affiché votre première page HTML.twig Symfony !

## Les pages html.twig

Explorons légèrement plus loin les pages TWIG.

Allez dans le dossier templates>main et sélectionner index.html.twig.

```
MonProjet > templates > main > ./ index.html.twig
1  {% extends 'base.html.twig' %}
2
3  {% block title %}Hello MainController!{% endblock %}
4
5  {% block body %}
6  <style>
7      .example-wrapper { margin: 1em auto; max-width: 800px; width: 95%; font: 18px/1.5 sans-serif; }
8      .example-wrapper code { background: #f5f5f5; padding: 2px 6px; }
9  </style>
10
11  <div class="example-wrapper">
12      <h1>Hello {{ controller_name }}! </h1>
13
14      This friendly message is coming from:
15      <ul>
16          <li>Your controller at <code><a href="{{ 'D:/wamp/www/symfony/Formation/MonProjet/src/Controller/MainController.php'|file_link(0) }}">src/Controller/MainController.php</a></code></li>
17          <li>Your template at <code><a href="{{ 'D:/wamp/www/symfony/Formation/MonProjet/templates/main/index.html.twig'|file_link(0) }}">templates/main/index.html.twig</a></code></li>
18      </ul>
19  </div>
20  {% endblock %}
```

On peut constater trois choses :

1. Nous avons des balises HTML classique (<style></style>).
2. Des balises TWIG ({% block body %} {% endblock %})
3. Et un **extends** de base.html.twig situé en toute première ligne du fichier html.

Commençons par l'**extends**. Toutes nos pages créées seront indirectement reliées à une page parent (dans notre cas base.html.twig), ceci permet d'avoir une certaine uniformité et de pouvoir appeler des éléments implémentés dans le fichier parent.

Exemple :

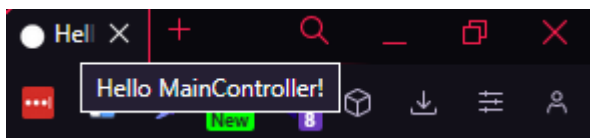
Dans notre fichier index.html.twig nous avons cette ligne :

```
{% block title %}Hello MainController!{% endblock %}
```

On peut constater qu'il contient des balises TWIG et qu'il y a du texte dedans. Si nous allons sur notre fichier base.html.twig nous aurons la même balise, mais avec un texte différent :

```
{% block title %}Welcome!{% endblock %}
```

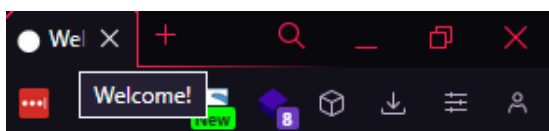
Donc quand on regarde le titre de nos pages web, nous avons bien ce qu'y est entrée dans nos balises TWIG.



Maintenant, faisons un test, remplaçons le texte du fichier index.html.twig par :

```
{% block title %}{{parent()}}{% endblock %}
```

Actualisez la page web, et vous pourrez constater que notre titre de page à changer !



Le rôle parent d'une page est primordiale pour garder des éléments intacts qui ne bougeront pas.

On aurait par exemple notre barre de navigation, le footer de notre page, etc..

À savoir, que si l'on crée une page `html.twig`, nous pouvons lui attribuer n'importe quelle page parent. Pour cela remplacer seulement le nom du fichier `html.twig` dans le « `{% extends 'mon_fichier.html.twig' %}` » en y ajoutant éventuellement son chemin dans vos dossiers.

« `{% extends 'mon_chemin/mon_fichier.html.twig' %}` »

Nous avons également, des balises TWIG comme vue précédemment, regardons à quoi ils servent. Dans nos pages `html.twig` on peut y constater ce formatage :

```
{% block body %}mon code html{% endblock %}
```

Ces balises-là font référence aux balises HTML préalablement créées.

```
MonProjet > templates > | base.html.twig
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset="UTF-8">
5          <title>{% block title %}Welcome!{% endblock %}</title>
6          <link rel="icon" href="data:image/svg+xml,<svg xmlns=%2
7              {% block stylesheets %}
8              {{ ux_controller_link_tags() }}
9              {% endblock %}
10
11          {% block javascripts %}
12      {% block importmap %}{{ importmap('app') }}{% endblock %}
13      {% endblock %}
14  </head>
15  <body>
16      {% block body %}{% endblock %}
17  </body>
18  </html>
```

On peut voir ici, que nos balises TWIG `body` sont incluses dans nos balises `<body></body>`.

Elles prendront alors par défaut le statut des balises parents.

Donc si vous voulions créer une section en TWIG, nous aurions dû faire comme ceci :

```
<section>
    {% block section %}{% endblock %}
</section>
```

Nous l'aurions ajouté à notre `base.html.twig` puis appelé dans `index.html.twig` (vous pouvez également créer des balises TWIG depuis le fichier enfant).

Et pour terminer, on peut y voir des balises HTML classique, en effet, Symfony reste une aide au développement de projet web. Vous pouvez donc utiliser des balises HTML classiques pour faire votre structure de projet.

Voyons une autre façon d'appeler du contenu **HTML** avec Symfony. En effet, il existe encore une façon d'introduire une page entière dans une page déjà existante, c'est ce qu'on appelle « inclure ».

Dans notre exemple, nous avons créé une page **TWIG** : *contact.html.twig* dans le dossier « *templates>main* ».

```
symfony > Formation > MonProjet > templates > main > contact.html.twig
1 <section>
2   <form method="post">
3     <input type="text" name="objet"/>
4     <input type="email" name="email"/>
5     <textarea placeholder="Mon texte"></textarea>
6     <button type="submit" name="envoyer">Envoyer</button>
7   </form>
8 </section>
9
```

Nous y avons ajouté un formulaire basique pour une demande de contact.

Pour le moment, cette page n'est pas incluse dans notre *index.html.twig*, donc si on se rend dessus la page sera affichée sans le formulaire.

Pour ça, il faut ajouter un « *include* » :

```
{{ include('main/contact.html.twig') }}
```

```
symfony > Formation > MonProjet > templates > main > index.html.twig
1 {% extends 'base.html.twig' %}
2
3 {% block title %}{{ parent() }}{% endblock %}
4
5 {% block body %}
6 <style>
7   .example-wrapper { margin: 1em auto; max-width: 800px; width: 95%; font: 18px/1.5 sans-serif; }
8   .example-wrapper code { background: #F5F5F5; padding: 2px 6px; }
9 </style>
10
11 <div class="example-wrapper">
12   <h1>Hello {{ controller_name }}! ✓</h1>
13
14   This friendly message is coming from:
15   <ul>
16     <li>Your controller at <code><a href="{{ 'D:/wamp/www/symfony/Formation/MonProjet/src/Contr'>
17     <li>Your template at <code><a href="{{ 'D:/wamp/www/symfony/Formation/MonProjet/templates/ma'>
18   </ul>
19 </div>
20
21 {{ include('main/contact.html.twig') }}
22 {% endblock %}
```

Et maintenant si on voit notre site web, nous aurons bien le formulaire d'affiché sur notre *index.html.twig*.

## Hello MainController! ✓

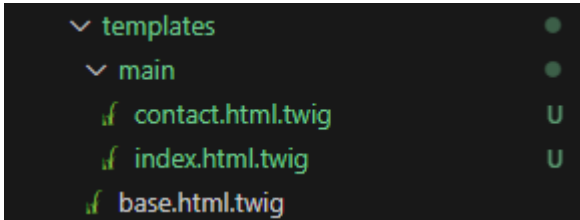
This friendly message is coming from:

- Your controller at [src/Controller/MainController.php](#)
- Your template at [templates/main/index.html.twig](#)

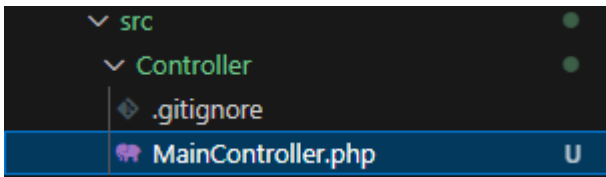
## Bonus contrôleurs

Il faut savoir qu'un fichier contrôleur peut contenir plusieurs routes pour différentes pages.

Reprenons l'exemple de notre page de contact. Nous l'avons créée dans le dossier templates>main soit dans le même dossier qu'index.html.twig.



Rendez-vous dans le contrôleur que nous avons créé précédemment :



Juste après notre route pour notre index.html.twig nous allons y ajouter une autre route pour définir notre page de contact :

```
symfony > Formation > MonProjet > src > Controller > MainController.php > ...
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 MainController extends AbstractController
10 {
11     #[Route('/', name: 'app_main')]
12     public function index(): Response
13     {
14         return $this->render('main/index.html.twig', [
15             'controller_name' => 'MainController',
16         ]);
17     }
18 }
```

1: Contrôleur par défaut



Et ajoutons ce code :

```
#[Route('/contact', name: 'app_contact')]
public function contact(): Response
{
    return $this->render('main/contact.html.twig', [
        'controller_name' => 'MainController',
    ]);
}

9 class MainController extends AbstractController
10 {
11     #[Route('/', name: 'app_main')]
12     public function index(): Response
13     {
14         return $this->render('main/index.html.twig', [
15             'controller_name' => 'MainController',
16         ]);
17     }
18
19     #[Route('/contact', name: 'app_contact')]
20     public function contact(): Response
21     {
22         return $this->render('main/contact.html.twig', [
23             'controller_name' => 'MainController',
24         ]);
25     }
26 }
```


## 2: Contrôleur avec la nouvelle route

Retournons sur notre page index.html et ajoutez ce petit bout de code pour créer un bouton de redirection sur notre page de contact.html.twig.

```
<a href="{{ path('app_contact') }}">Contact</a>
```

```
symfony > Formation > MonProjet > templates > main > index.html.twig
1  {% extends 'base.html.twig' %}
2
3  {% block title %}{{parent()}}{% endblock %}
4
5  {% block body %}
6  <style>
7      .example-wrapper { margin: 1em auto; max-width: 800px; width: 95%; font: 18px/1.5 sans-serif; }
8      .example-wrapper code { background: #F5F5F5; padding: 2px 6px; }
9  </style>
10
11  <div class="example-wrapper">
12      <h1>Hello {{ controller_name }}! ✓</h1>
13
14      This friendly message is coming from:
15      <ul>
16          <li>Your controller at <code><a href="{{ 'D:/wamp/www/symfony/Formation/MonProjet/src/Contr'
17          <li>Your template at <code><a href="{{ 'D:/wamp/www/symfony/Formation/MonProjet/templates/m
18      </ul>
19  </div>
20
21  <a href="{{ path('app_contact') }}">Contact</a>
```

---

Hello MainController! 

This friendly message is coming from:

- Your controller at [src/Controller/MainController.php](#)
- Your template at [templates/main/index.html.twig](#)

[Contact](#)

Et si on clique sur Contact, ça nous redirigera sur notre page contact.html.twig.

---

<input type="text"/>	<input type="text"/>	<input type="text" value="Mon texte"/>	<input type="button" value="Envoyer"/>
----------------------	----------------------	--	--

## Les paramètres TWIG

Symfony intègre des options pour nos pages (exemple : récupérer un nom), nous allons voir comment cela fonctionne :

Avant toute chose, il faut se rendre dans notre page contrôleur.

On va ajouter un paramètre à notre index pour qu'il nous retourne un nom quand celui-ci sera spécifié dans l'URL.

```
#[Route('/{name}', name: 'app_main')]
public function index($name="test"): Response
{
    return $this->render('main/index.html.twig', [
        'controller_name' => 'MainController',
        'name'=>$name
    ]);
}
```

Ici, on peut voir que **\$name** prend la valeur « test » et que dans notre URL par défaut (qui était /) nous avons ajouté **{name}**.

Cette façon-là est une manière de définir la variable \$nom directement depuis l'URL.

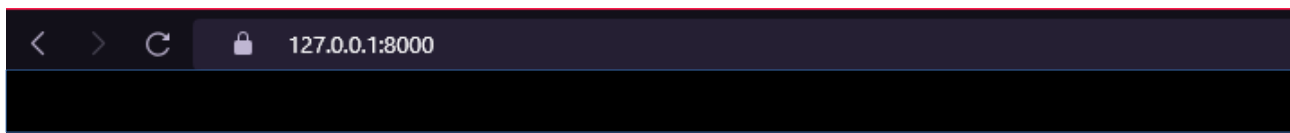
Regardons cette exemple :

Dans notre index.html.twig, j'ai supprimé le contenu pour le remplacer par :

```
<div class="example-wrapper">
    Mon nom est : {{ name }}
    <br>
    <a href="{{ path('app_contact') }}">Contact</a>
</div>
```

Notre variable \$nom est appelé avec l'appellation : **{{name}}**.

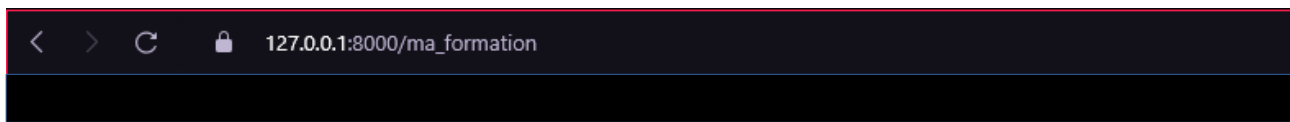
Donc si on se rend sur notre index.html.twig, nous aurons par défaut dans **{{name}}** : test.



Mon nom est : test

[Contact](#)

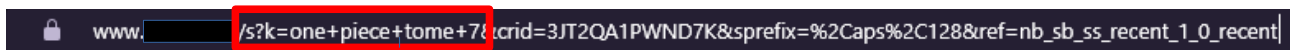
Par contre si on met du texte dans notre URL après « / », test changera pour devenir le texte défini dans l'URL :



Mon nom est :ma\_formation

[Contact](#)

Cette méthode serait utile pour des utilisations \$\_GET\* pour l'affichage d'un produit spécifique par exemple :



On peut constater que dans cet URL, il y a « ?K= » qui définit le nom du \$\_GET['K'], donc tout ce qui se trouve après le = et avant le & sera examiné selon les spécificités données !

C'est exactement de la même manière que fonctionne notre **{name}**, il va définir notre variable **\$name** que l'on appelle avec **{{name}}**.

Avec cette façon-là de procéder, nous pouvons rendre les possibilités presque infinies.

Imaginons ce coup-ci que nous voulions un âge, nous remplaçons **\$name** par **\$age** et **{name}** par **{age}** ainsi que **{{name}}** par **{{age}}**

```
#[Route('/{age}', name: 'app_main')]
public function index($age="1"): Response
{
    return $this->render('main/index.html.twig', [
        'controller_name' => 'MainController',
        'age'=>$age
    ]);
}
```

```
<div class="example-wrapper">
    Mon âge est de {{age}} ans
    <br>
    <a href="{{ path('app_contact') }}">Contact</a>
</div>
```

\*Cliquez dessus pour en savoir plus sur la méthode \$\_GET.

Par défaut, nous avons mis l'âge à 1. Donc si nous n'entrons pas de valeur dans l'URL, notre index retournera 1.



Et également, si on met une valeur après le « / », l'index nous retournera cette valeur. Vous avez compris ce principe.

## Les conditions

Les conditions sont une chose essentielle dans la programmation, c'est ce qui nous permet d'exécuter telle ou telle action selon le plan donné.

Exemple :

Vous voulez aller en boîte de nuit, mais vous n'êtes pas majeur, vous n'aurez donc pas le droit d'entrer, alors qu'une personne majeure, elle, pourra. Les conditions sont pareilles.

Pour créer des conditions, nous utiliserons le balisage TWIG :

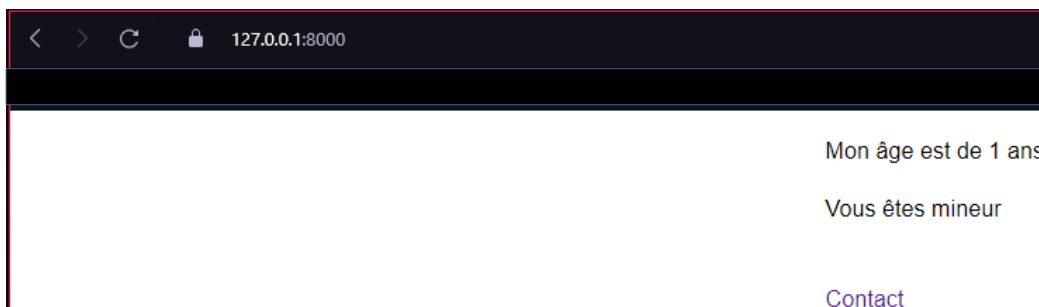
```
<div class="example-wrapper">
  Mon âge est de {{age}} ans
  {% if age >= 18 %}
    <p>Vous êtes majeur</p>
  {% else %}
    <p>Vous êtes mineur</p>
  {% endif %}
  <br>
  <a href="{{ path('app_contact') }}">Contact</a>
</div>
```

Détaillons ce bout de code :

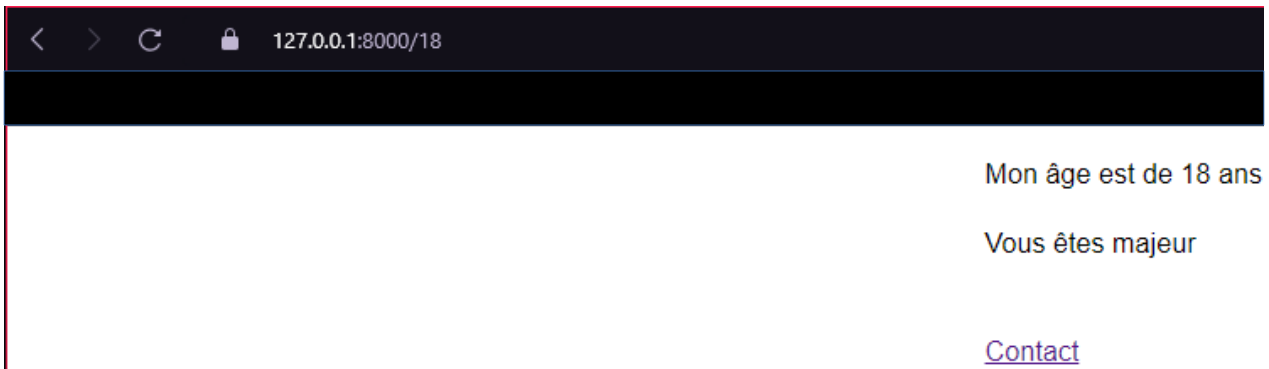
**{% if age ≥ 18 %}** = Si l'âge est supérieur ou égale à 18 alors « Vous êtes majeur »

**{% else %}** = Sinon « Vous êtes mineur »

**{% endif %}** = Fin du bloc de condition, très important !!!



1: Sans paramètre dans l'URL par défaut 1



2: Avec paramètre dans l'URL et supérieur ou égale à 18

Il y a une autre façon de décrire la condition `{% if $variable %}`. C'est ce qu'on appelle « **les conditions ternaires** ».

On reste sur notre exemple d'âge :

```
{{ age >= 18 ? "Vous êtes majeur" : "Vous êtes mineur" }}
```

Détaillons le tout :

**age** = Valeur de notre variable \$age.

**≥ 18** = Supérieur ou égale à 18.

**?** = Si c'est vrai alors « Vous êtes majeur »

**:** = Sinon « Vous êtes mineur »

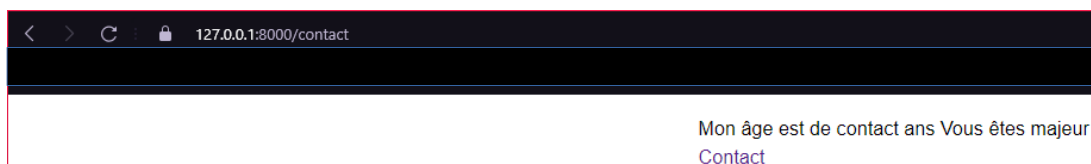
**{{ }}** = Affichage du tout donc dans notre cas, du texte.

Le résultat sera le même que la condition d'avant, nous avons juste réduit les lignes de code.

Pour en savoir plus sur les conditions, je vous invite à vous rendre sur :

<https://openclassrooms.com/>

Si on remarque notre bouton contact, il ne fonctionne plus, pourquoi à votre avis ? Car la route sur notre contrôleur est `{age}` donc quand on clique sur contact, il met dans l'URL `127.0.0.1/contact`, donc la valeur \$age prend « contact » et nous retourne contact :



Pour régler ça, il faut ajouter un chemin supplémentaire dans l'URL (`/c/contact`).

```
#[Route('/c/contact', name: 'app_contact')]
public function contact(): Response
{
    return $this->render('main/contact.html.twig', [
        'controller_name' => 'MainController',
    ]);
}
```

# Les entités / repository

Revenons sur les entités, on les a déjà vues rapidement avec les commandes :

`symfony console make:user` et `symfony console make:entity`, maintenant que nous avons vu les TWIG et comment ils fonctionnent avec les contrôleurs, on va pouvoir pousser légèrement plus les entités.

Pour des facilités d'usage et de compréhension, nous utiliserons le framework tailwind pour tout le côté CSS ainsi que la librairie flowbite.

Je ne vous montrerai pas comment installer tailwind via le [CLI symfony](#), nous utiliserons seulement le [CDN](#) disponible.

Mettez les lignes suivantes dans la partie head de notre fichier base.html.twig :

```
<script src="https://cdn.tailwindcss.com"></script>
<link href="https://cdn.jsdelivr.net/npm/flowbite@2.2.0/dist/flowbite.min.css" rel="stylesheet"/>
<script src="https://cdn.jsdelivr.net/npm/flowbite@2.2.0/dist/flowbite.min.js"></script>
```

```
symfony > Formation > MonProjet > templates > | base.html.twig
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <meta charset="UTF-8">
5      <title>
6        {% block title %}Welcome!{% endblock %}
7      </title>
8      <script src="https://cdn.tailwindcss.com"></script>
9      <link href="https://cdn.jsdelivr.net/npm/flowbite@2.2.0/dist/flowbite.min.css" rel="stylesheet"/>
10     <script src="https://cdn.jsdelivr.net/npm/flowbite@2.2.0/dist/flowbite.min.js"></script>
```

Pour notre exemple, nous allons recréer une table SQL qui portera comme nom « produits » :

```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)
$ symfony console make:entity

Class name of the entity to create or update (e.g. GentlePopsicle):
> produits
produits

Add the ability to broadcast entity updates using Symfony UX Turbo? (yes/no) [no]:
>

created: src/Entity/Produits.php
created: src/Repository/ProduitsRepository.php

Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-running this command.

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

En colonnes nous aurons :

1. **nom** = string en 255 et qui ne peut être null !

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

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

Field length [255]:
>

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

updated: src/Entity/Produits.php
```

2. **description** = text qui peut être null.

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

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

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

updated: src/Entity/Produits.php
```

3. **price** = integer qui ne peut être null.

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

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

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

updated: src/Entity/Produits.php
```

4. **stock** = boolean qui ne peut être null.

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

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

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

updated: src/Entity/Produits.php
```

Faites entrer pour valider la création de notre entité.

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

Success!

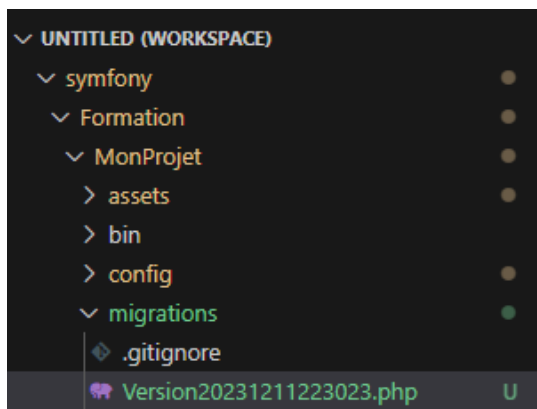
Next: When you're ready, create a migration with symfony.exe console make:migration

neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)
$
```

Comme la console nous le dis, nous allons migrer notre nouvelle entité pour la créer dans notre base de donnée. Avant cela synchronisons nos données avant la migration :

```
symfony console doctrine:migrations:sync-metadata-storage
```

```
symfony console make:migration
```



La commande précédente nous créait un fichier VersionXXXXX.php qui contient toute notre requête sql pour la création et insertion de notre table. Si vous avez fait des modifications après coup sur votre entité, pas de problème Symfony le gère très bien et fera seulement une mise à jour de votre table SQL sur l'ancienne sans toucher aux données présente.

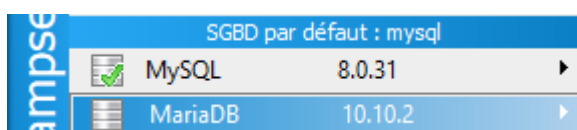
Puis nous allons la migrer :

```
symfony console doctrine:migrations:migrate
```

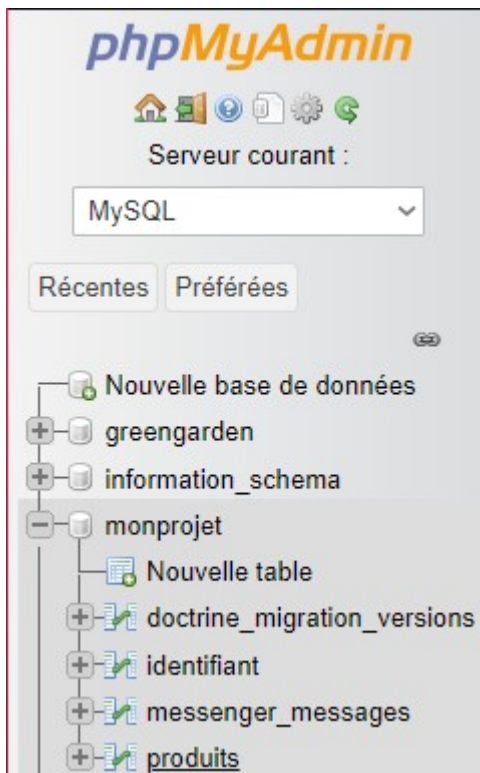
Si vous rencontrez des problèmes lors de l'export de la migration, pensez à mettre à jour la ligne :

```
DATABASE_URL="mysql://root:root@127.0.0.1:3306/monprojet?serverVersion=10.10.2-MariaDB&charset=utf8mb4"
```

En spécifiant la version de votre MariaDB de wamp (*clique gauche sur l'icône*).

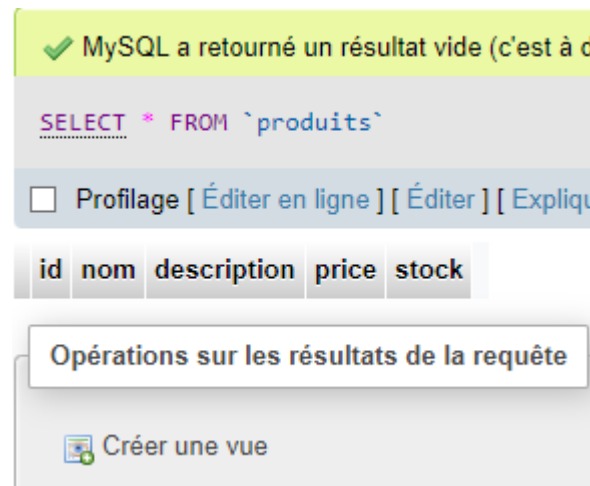




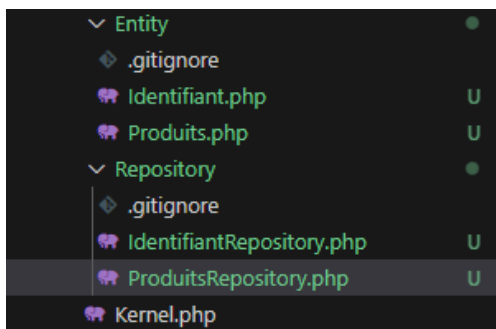


Retournez sur PHPMyAdmin pour bien constater que notre nouvelle table a été créée !

Cliquez dessus pour voir que toutes nos colonnes sont bien présentes.



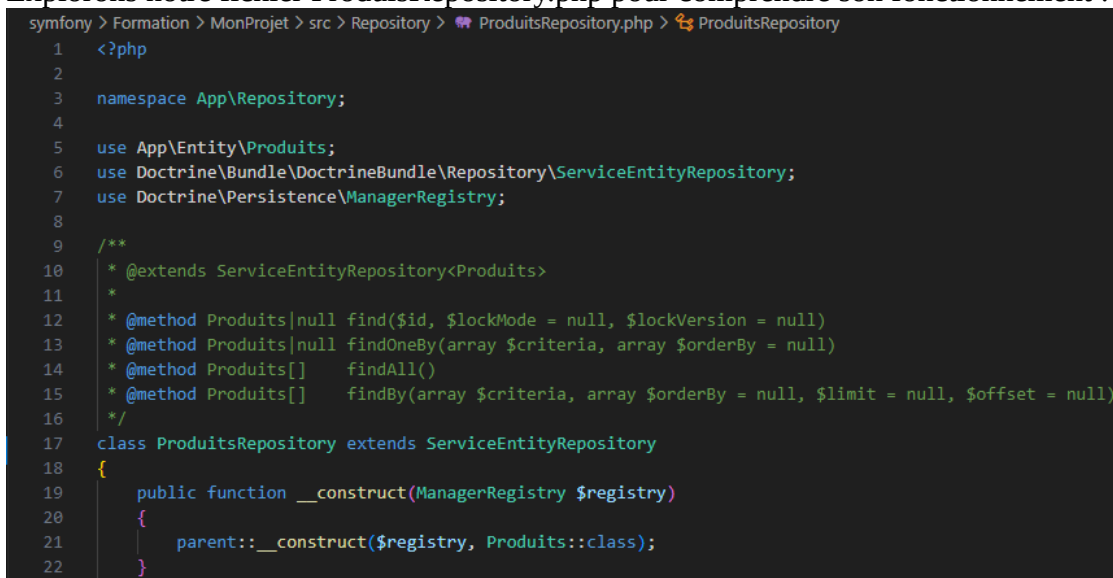
Dans le dossier « src>Repository » nous avons des Repository pour toutes nos tables créée :



Dans ces fichiers, nous auront les fonctions qui feront appel à notre table.

Dans notre exemple pas besoin de créer une fonction, celles, déjà, présente sont suffisante.

Explorons notre fichier ProduitsRepository.php pour comprendre son fonctionnement :



On peut voir que l'on a :

- `namespace App\Repository;`
- `class ProduitsRepository extends ServiceEntityRepository`
- ```
public function __construct(ManagerRegistry $registry)
{
    parent::__construct($registry, Produits::class);
}
```

Le namespace correspond à l'environnement de travail ;

Dans leur définition la plus large, ils représentent un moyen d'encapsuler des éléments. Cela peut être conçu comme un concept abstrait, pour plusieurs raisons.

Par exemple, dans un système de fichiers, les dossiers représentent un groupe de fichiers associés et servent d'espace de noms pour les fichiers qu'ils contiennent.

Un exemple concret est que le fichier « foo.txt » peut exister dans les deux dossiers « /home/greg » et « /home/other », mais que les deux copies de « foo.txt » ne peuvent pas co-exister dans le même dossier.

La `class` qui est étendue au `ServiceEntityRepository` ;

La classe définit son nom et tout ce qui se trouve dedans ne peut être exécuté que par la classe elle-même ou par un de ses enfants !

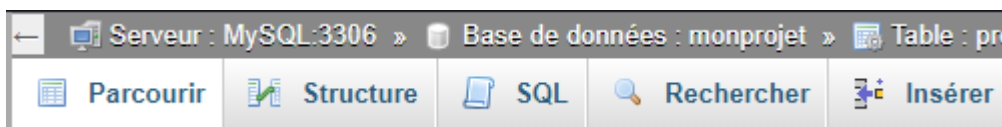
Exemple avec la public function juste en dessous de la classe.

Je vous invite à faire un tour sur la documentation officielle de Symfony pour en savoir plus sur les classes et leur utilité, mais avec notre exemple, vous devriez comprendre.

<https://symfony.com/doc/current/doctrine/events.html#doctrine-entity-listeners>

Ajoutons tout d'abord des informations sur nos produits. Pour cela rendez-vous dans PHPMyAdmin et dans notre table produits.

Dès que vous êtes dedans, aucune donnée n'est présente et c'est normal, pour en ajouter manuellement cliquer sur insérer :



Remplissez les champs comme vous le souhaitez (*seulement la colonne valeur tout en laissant la valeur de **id** vide*).

Serveur : MySQL 3306 » Base de données : monprojet » Table : produits

Parcourir Structure SQL Rechercher Insérer Exporter Importer Privilèges Opérations Déclencheurs

| Colonne | Type         | Fonction | Null | Valeur                   |
|---------|--------------|----------|------|--------------------------|
| id      | int(11)      |          |      |                          |
| nom     | varchar(255) |          |      | Pommes                   |
|         |              |          |      | On aime les pommes ici ! |

Ajouter autant de produits que vous désirez. Personnellement, je vais en ajouter deux.

|                          |        |        |           | id | nom        | description                                          | price | stock |
|--------------------------|--------|--------|-----------|----|------------|------------------------------------------------------|-------|-------|
| <input type="checkbox"/> | Éditer | Copier | Supprimer | 1  | Pommes     | On aime les pommes ici !                             | 3     | 0     |
| <input type="checkbox"/> | Éditer | Copier | Supprimer | 2  | ordinateur | ordinateur gaming prêt à l'emploi pour vos meille... | 1000  | 1     |

Souvenez-vous, nous avons installé les CDN de tailwind et de flowbite, rendez-vous sur flowbite pour prendre une carte pour l'affichage de nos produits.

Supprimons tout notre index.html.twig, pour le remplacer par notre carte flowbite :

```

templates > main > index.html.twig
1  {% extends 'base.html.twig' %}
2
3  {% block title %}
4      {{parent()}}
5  {% endblock %}
6
7  {% block body %}
8      <style>
9          .example-wrapper {
10             margin: 1em auto;
11             max-width: 800px;
12             width: 95%;
13             font: 18px / 1.5 sans-serif;
14         }
15         .example-wrapper code {
16             background: #f5f5f5;
17             padding: 2px 6px;
18         }
19     </style>
20
21     <div class="example-wrapper">
22
23         <a href="#" class="block max-w-sm p-6 bg-white border border-gray-200 rounded-lg shadow hover:bg-gray-100 dark:bg-gray-800 dark:border-gray-700 dark:hover:bg-gray-700">
24
25             <h5 class="mb-2 text-2xl font-bold tracking-tight text-gray-900 dark:text-white">Noteworthy technology acquisitions 2021</h5>
26             <p class="font-normal text-gray-700 dark:text-gray-400">Here are the biggest enterprise technology acquisitions of 2021 so far, in reverse chronological order.</p>
27         </a>
28     </div>
29
30
31  {% endblock %}

```

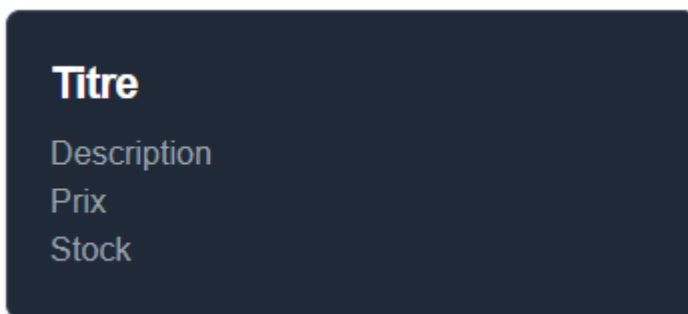
Vous devriez avoir ceci :



On voudrait que notre carte prenne :

- Le nom du produit
- Sa description
- Son prix
- Son stock.

Donc modifier les champs pour arriver à ce résultat-là. Pour information, la carte est cliquable donc pas besoin de bouton pour afficher le produit. On aura juste à cliquer sur la carte.



Vu que nous sommes encore dans l'`index.html.twig`, rendez-vous dans le contrôleur et changer les paramètres suivants :

```
class MainController extends AbstractController
{
    #[Route('/', name: 'app_main')]
    public function index(EntityManagerInterface $entity): Response
    {
        $products = $entity->getRepository(Produits::class)->findAll();
        return $this->render('main/index.html.twig', [
            'controller_name' => 'MainController'
        ]);
    }
}
```

Dans les paramètres d'`index()`; on y ajoute [EntityManagerInterface](#) qui prend comme variable `$entity`, on crée une variable `$products` qui prend la connexion à l'entity, qui actionne la fonction `getRepository()`, qui dans cette même fonction cherche notre classe Produits que nous avons créée et liste toutes les occurrences de notre table SQL.

Pour vérifier nos propos, utilisons la fonction `dd()`; qui prendra comme paramètre `$products`.

PS : `dd` signifie : die and dump, c'est une fonction équivalente à `var_dump` en PHP vanilla.

```
class MainController extends AbstractController
{
    #[Route('/', name: 'app_main')]
    public function index(EntityManagerInterface $entity): Response
    {
        $products = $entity->getRepository(Produits::class)->findAll();

        dd($products);

        return $this->render('main/index.html.twig', [
            'controller_name' => 'MainController'
        ]);
    }
}
```

Comme on peut le constater, `$products` nous retourne un tableau de deux entrées (0 et 1), grâce à cela, nous pourrions créer une boucle (Si vous ne savez ce qu'est une boucle, je vous invite à aller regarder du côté de PHP vanilla) pour ainsi récupérer toutes les données de chaque produit et les insérer dans une seule carte.

```
MainController.php on line 20:
array:2 [▼
  0 => App\Enti_\Produits {#2475 ▼
    -id: 1
    -nom: "Pommes"
    -description: "On aime les pommes ici !"
    -price: 3
    -stock: false
  }
  1 => App\Enti_\Produits {#2559 ▼
    -id: 2
    -nom: "ordinateur"
    -description: "ordinateur gaming prêt à l'emploi pour vos meilleures session gaming"
    -price: 1000
    -stock: true
  }
]
```

Ajoutons un dernier paramètre à notre contrôleur pour récupérer les valeurs de notre variable `$products`. Dans le `return` de notre contrôleur, on devra y ajouter l'objet `products` qui aura comme valeur `$products`, tout en supprimant notre `dd()` :

```
return $this->render('main/index.html.twig', [
    'controller_name' => 'MainController',
    'products' => $products
]);
```

Désormais, nous allons créer notre première boucle pour que symfony nous créer automatiquement le nombre de cartes nécessaire en fonction du nombre de produits que l'on a dans notre base de données.

Pour la boucle, nous allons utiliser un FOR, tout en utilisant les balises TWIG, nous allons créer une boucle sur la variable \$products :

```
{% for product in products %}
    <a href="#" class="block max-w-sm p-6 bg-white border border-gray-200 rounded-lg shadow hover:bg-gray-100">
        <h5 class="mb-2 text-2xl font-bold tracking-tight text-gray-900 dark:text-white">[ ]</h5>
        <p class="font-normal text-gray-700 dark:text-gray-400">[ ]</p>
        <p class="font-normal text-gray-700 dark:text-gray-400">[ ]</p>
        <p class="font-normal text-gray-700 dark:text-gray-400">Quantité : [ ]</p>
    </a>
{% endfor %}
```

Détaillons la boucle :

**{% %}** = Balisage TWIG.

**For** = Notre condition pour boucler, il en existe d'autres, mais nous les verrons sûrement plus tard.

**product** = Les valeurs qui se trouvent dans products

**in** = Dans

**products** = Notre variable définie dans le contrôleur.

Donc pour faire gros, on cycle sur **\$products** et **\$product** sans « S » prend les valeurs qui se trouvent dans le tableau.

Maintenant pour afficher le nom, la description ou encore le prix de nos articles, il faut encore une étape. Avant de vous l'expliquer, je vous invite à lire cette article sur les [API](#), car pour appeler le nom de ce qu'on a besoin, c'est exactement pareil qu'en Javascript lors d'une utilisation d'API.

Maintenant, que vous vous êtes documenté sur les API, revenons sur notre affichage de produit. Pour afficher une valeur spécifique de notre tableau récemment créé avec notre contrôleur et cycler via notre boucle. Il faut taper la variable product puis mettre le nom de la colonne souhaité.

```
<h5 class="mb-2 text-2xl font-bold tracking-tight text-gray-900 dark:text-white">{{product.nom}}</h5>
<p class="font-normal text-gray-700 dark:text-gray-400">{{product.description}}</p>
<p class="font-normal text-gray-700 dark:text-gray-400">{{product.price}}€</p>
<p class="font-normal text-gray-700 dark:text-gray-400">Quantité : {{product.stock}}</p>
```

Pour rappel, product sans « S », prend les valeurs du tableau \$products, donc pour afficher une partie de ce tableau, nous allons récupérer les noms qui figurent dans ce tableau.

```
MainController.php on line 20:
array:2 [▼
  0 => App\Enti...\Produits {#2475 ▼
    -id: 1
    -nom: "Pommes"
    -description: "On aime les pommes ici !"
    -price: 3
    -stock: false
  }
  1 => App\Enti...\Produits {#2559 ▼
    -id: 2
    -nom: "ordinateur"
    -description: "ordinateur gaming prêt à l'emploi pour vos meilleures session gaming"
    -price: 1000
    -stock: true
  }
]
```

Ici, nous pouvons utiliser :

- id
- nom
- description
- price
- stock

Voici le résultat que vous devriez avoir :



Pour plus de style, je vous invite à parcourir la documentation de tailwind ainsi que l'utilisation des flexbox.

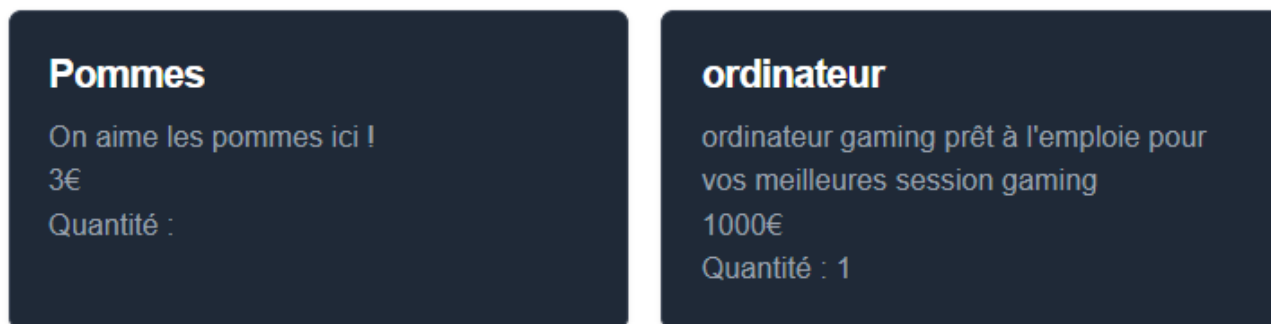
Ajoutons un autre élément pour apprendre d'autres choses sur Symfony.

On voudrait que notre page index.html.twig est un titre qui liste combien de produits, on possède. Toujours en gardant notre boucle FOR, nous allons ajouter une balise H1 au-dessus de celle-ci :

```
<h1 class="w-[30%] mx-auto text-3xl">Mes produits</h1>

<div class="example-wrapper flex gap-5">
{% for product in products %}
    <a href="#" class="w-full block max-w-sm p-6 bg-white border border-gray-200 rounded-lg shadow hover:bg-gray-100">
        <h5 class="mb-2 text-2xl font-bold tracking-tight text-gray-900 dark:text-white">{{product.nom}}</h5>
        <p class="font-normal text-gray-700 dark:text-gray-400">{{product.description}}</p>
        <p class="font-normal text-gray-700 dark:text-gray-400">{{product.price}}€</p>
        <p class="font-normal text-gray-700 dark:text-gray-400">Quantité : {{product.stock}}</p>
    </a>
{% endfor %}
```

## Mes produits



Maintenant, pour afficher le nombre de produits que l'on possède dans notre boucle, il faudra utiliser l'affichage TWIG que nous avons déjà vu dans les chapitres précédents :

```
{{ products }}
```

Cette commande nous permet d'afficher dans la théorie les valeurs de notre tableau. Sauf que, vu que `products` est un tableau, on ne peut pas l'afficher tel quel. Nous devons donc passer par une option supplémentaire que l'on utilise comme suit :

```
{{ products | length }}
```

Observons ça de plus près :

**products** = Notre valeur de tableau

**length** = Compte le nombre de valeurs du tableau, on peut utiliser `length` pour compter le nombre de caractère d'une chaîne de caractère.

On utilise la barre « | » pour séparer notre variable de notre option.

## Mes produits (2)

### Pommes

On aime les pommes ici !

3€

Quantité :

### ordinateur

ordinateur gaming prêt à l'emploi pour vos meilleures session gaming

1000€

Quantité : 1

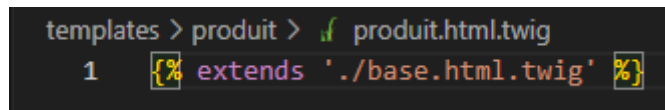
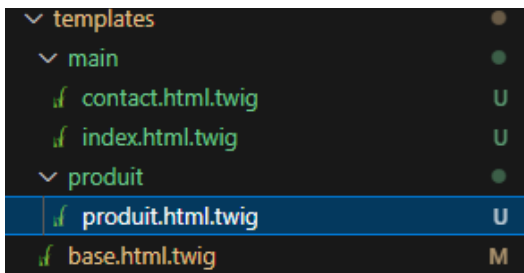
```
<h1 class="w-[30%] mx-auto text-3xl">Mes produits ({{ products | length }})</h1>

<div class="example-wrapper flex gap-5">
{% for product in products %}
  <a href="#" class="w-full block max-w-sm p-6 bg-white border border-gray-200 rounded-lg shadow hover:bg-gray-100">
    <h5 class="mb-2 text-2xl font-bold tracking-tight text-gray-900 dark:text-white">{{ product.nom }}</h5>
    <p class="font-normal text-gray-700 dark:text-gray-400">{{ product.description }}</p>
    <p class="font-normal text-gray-700 dark:text-gray-400">{{ product.price }}€</p>
    <p class="font-normal text-gray-700 dark:text-gray-400">Quantité : {{ product.stock }}</p>
  </a>
{% endfor %}
```

Désormais, on voudrait une page pour afficher les détails de notre produit, pour ensuite éventuellement, l'ajouter à notre panier, puis l'acheter.

Pour cela, créer une autre page `html.twig` dans notre dossier `templates`, pour se familiariser avec le côté MVC (Modèle Vue Contrôleur) intégré à Symfony, nous créerons un sous-dossier dans le dossier `templates` que l'on appellera « produit ». Dans le dossier « produit », créer un fichier `produit.html.twig`, tout en reliant l'extends à notre `base.html.twig`.

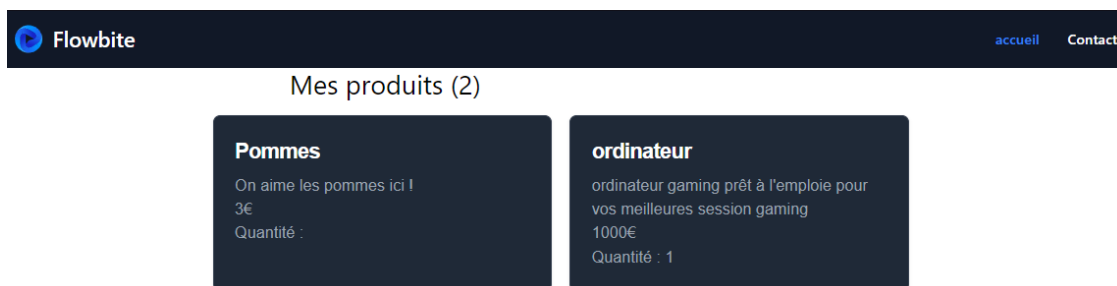




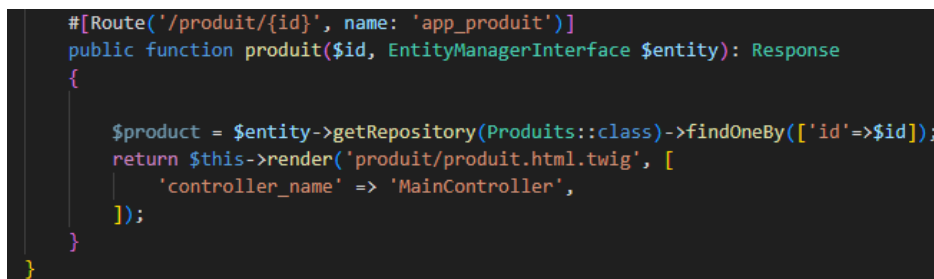
Je vous invite à faire la route nécessaire au bon fonctionnement de cette page.  
Prenez-le comme un exercice !

Pour des questions de facilité de navigation, j'ai ajouté une barre de navigation dans notre `base.html.twig` (avec les composants *flowbite*), tout en y incluant nos chemins vers nos pages (cf. page 20).

Vous devriez avoir un résultat semblable à ceci :



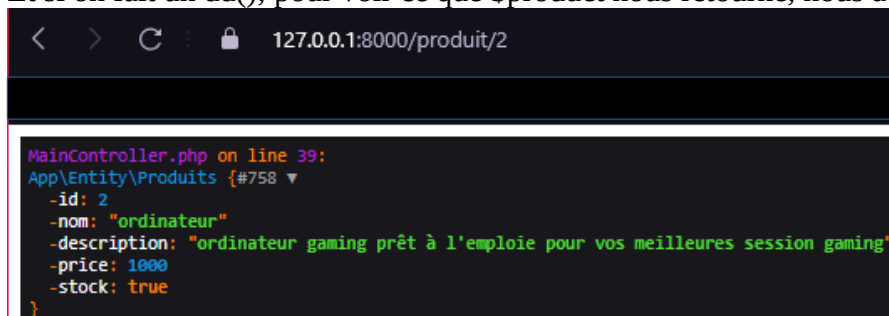
Pour afficher un produit en particulier dans notre page `produit.html.twig`, il faudra configurer le contrôleur pour que celui-ci retourne un `$id` (cf. page 26), tout en gardant `EntityManagerInterface`



On devra créer une variable `$product`, car nous voulons que ça nous retourne seulement la valeur d'un seul produit (*celui qui sera défini par l'id*), et on lui fait un `findOne($id)` (fonction disponible grâce à *Symfony*)

```
$product = $entity->getRepository(Produits::class)->findOneBy(['id'=>$id]);
```

Et si on fait un `dd()`; pour voir ce que `$product` nous retourne, nous aurions ceci :



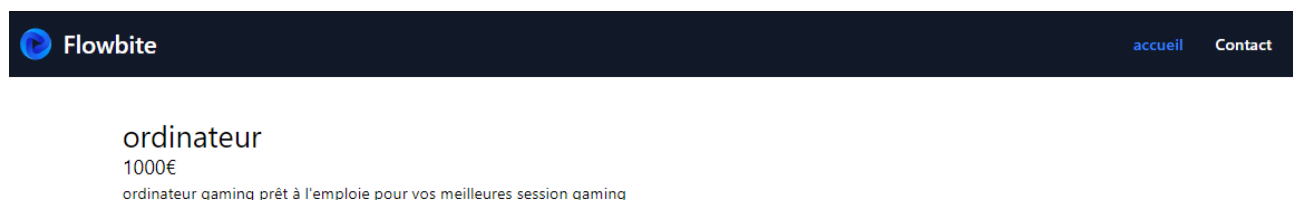
Comme vu précédemment, pour appeler les différentes instances, il faudra définir \$product dans notre contrôleur, puis appeler le nom des colonnes retournées :

```
{{ product.nom }}
```

```
{% block body %}
  <article class="w-[50%] mx-auto mt-10">
    <h1 class="text-3xl">{{ product.nom }}</h1>
    <h2 class="text-xl">{{ product.price }}€</h2>
    <p>{{ product.description }}</p>
  </article>
{% endblock %}
```

```
#[Route('/produit/{id}', name: 'app_produit')]
public function produit($id, EntityManagerInterface $entity): Response
{
    $product = $entity->getRepository(Produits::class)->findOneBy(['id'=>$id]);
    return $this->render('produit/produit.html.twig', [
        'controller_name' => 'MainController',
        'id'=>$id,
        'product'=>$product
    ]);
}
```

Et maintenant, quand on entre un \$id qui est disponible dans notre base de données, nous aurons bien la page de notre produit affichée.



Pour rendre nos cartes cliquables vers le bon produit, nous allons retourner dans notre page index.html.twig, pour modifier le <a></a>.

On lui définit le chemin de notre app\_produit, puis nous lui ajoutons l'id de notre produit via la variable \$product qui a été générée via notre boucle FOR.

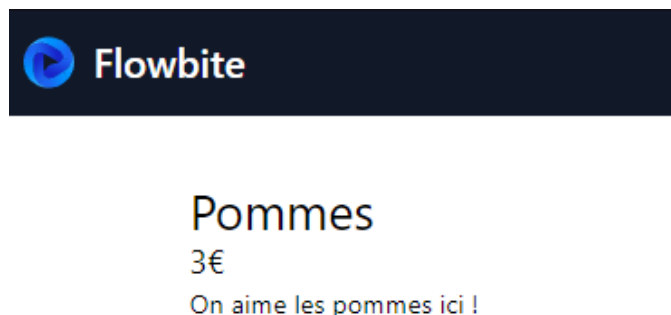
```
{{ path('app_produit', {id: product.id}) }}
```

```
{% for product in products %}
  <a href="{{ path('app_produit', {id: product.id}) }}"
```

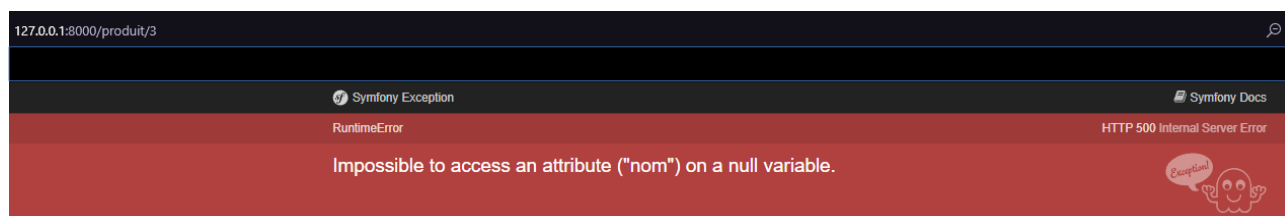
Et maintenant, si on passe la souris sur nos cartes, on peut observer qu'elles contiennent bien le chemin de notre app\_produit ainsi que l'\$id :

`https://127.0.0.1:8000/produit/1`

Puis, quand on clique dessus, ça nous redirige bien vers notre page de produit :



Un inconvénient est présent pour le moment, si dans notre URL, on remplace l'id par un \$id qui n'existe pas dans notre base de données, vous aurez ceci :



Pour palier à ce problème, il faudra retourner dans notre contrôleur, aller dans la route de notre produit puis ajouter ceci :

```
#[Route('/produit/{id}', name: 'app_produit')]
public function produit($id, EntityManagerInterface $entity): Response
{
    $product = $entity->getRepository(Produits::class)->findOneBy(['id'=>$id]);

    if(is_null($product)){
        return $this->redirectToRoute('app_main');
    }

    return $this->render('produit/produit.html.twig', [
        'controller_name' => 'MainController',
        'product'=>$product
    ]);
}
```

Détaillons la condition :

**if(is\_null(\$product))** = Si \$product est vide

**Return \$this → redirectToRoute('app\_main');** = Alors on redirige vers la page de nos produits

Maintenant, Si on met un \$id qui n'existe pas, nous serons redirigées vers l'index.html.twig.

Dernière petite chose avec nos produits, dans l'hypothèse où nous voudrions que seuls les produits en stock soient affichés, il faudrait rajouter une condition dans le contrôleur de notre `index.html.twig`.

Avant nous avions :

```
$products = $entity->getRepository(Produits::class)->findAll();
```

Puis nous allons le remplacer par :

```
$products = $entity->getRepository(Produits::class)->findBy(["stock"=>true]);
```

```
#[Route('/', name: 'app_main')]
public function index(EntityManagerInterface $entity): Response
{
    $products = $entity->getRepository(Produits::class)->findBy(["stock"=>true]);

    return $this->render('main/index.html.twig', [
        'controller_name' => 'MainController',
        'products'=>$products
    ]);
}
```

Et vu que dans mon cas, seul l'ordinateur est en stock, alors nous aurons seulement ce produit d'afficher :

			id	nom	description	price	stock				
<input type="checkbox"/>		Éditer		Copier		Supprimer	1	Pommes	On aime les pommes ici !	3	0
<input type="checkbox"/>		Éditer		Copier		Supprimer	2	ordinateur	ordinateur gaming prêt à l'emploi pour vos meille...	1000	1

Flowbite accueil Contact

Mes produits (1)

**ordinateur**  
ordinateur gaming prêt à l'emploi pour  
vos meilleures session gaming  
1000€  
Quantité : 1

Et dans l'hypothèse où notre autre produit venait à de nouveau être en stock :

							id	nom	description	price	stock
<input type="checkbox"/>		Éditer		Copier		Supprimer	1	Pommes	On aime les pommes ici !	3	2
<input type="checkbox"/>		Éditer		Copier		Supprimer	2	ordinateur	ordinateur gaming prêt à l'emploi pour vos meille...	1000	1

Flowbite accueil Contact

Mes produits (2)

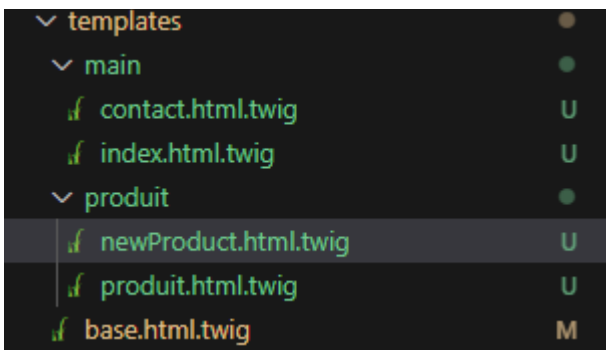
**Pommes**  
On aime les pommes ici !  
3€  
Quantité : 1

**ordinateur**  
ordinateur gaming prêt à l'emploi pour  
vos meilleures session gaming  
1000€  
Quantité : 1

# Les formulaires

Dans cette partie, nous allons voir comment créer un formulaire et le rendre interactif selon nos besoins. Dans notre cas, on va garder l'exemple de nos produits, on créera un formulaire pour l'ajout de produits.

Tout d'abord, créons une nouvelle page (*newProduct.html.twig*) :



Restons dans notre dossier produit pour la création de la nouvelle page.

Comme la page « produit », ajouter l'extends de base et ajouter les base twig pour le block body.

```
templates > produit > if newProduct.html.twig
1  {% extends './base.html.twig' %}
2
3  {% block body %}
4
5  {% endblock %}
```

Je vous invite à créer la route nécessaire dans notre main contrôleur et d'ajouter un lien dans notre navbar pour aller plus rapidement sur notre page d'ajout de produit.

```
#[Route('/produit/new', name: 'app_produit_new')]
public function newProduit(): Response
{
    return $this->render('produit/newProduct.html.twig', [
        'controller_name' => 'MainController'
    ]);
}
```

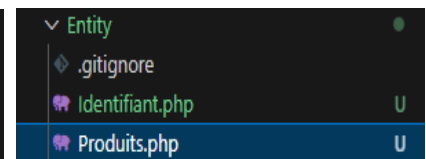
Rendez-vous dans la console GIT bash pour créer notre formulaire, Symfony le fait pour nous et nous facilite la tâche :

```
symfony console make:form
```

Dans le premier champ, il faudra reprendre le nom de notre entité (*dans notre cas Produits.php*), puis y ajouter Type juste après :

```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)
$ symfony console make:form

The name of the form class (e.g. OrangeElephantType):
> ProduitsType
```



Dans le second champ, Symfony, nous demande sur quelle entité il veut se baser, on remet « Produits » :

```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)
$ symfony console make:form

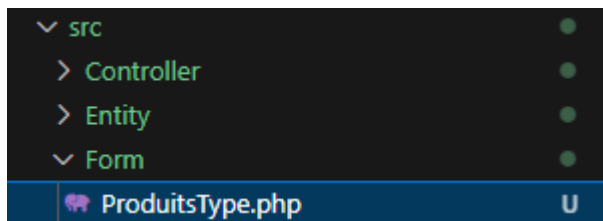
The name of the form class (e.g. OrangeElephantType):
> ProduitsType

The name of Entity or fully qualified model class name that the new form will be bound to (empty for none):
> Produits

created: src/Form/ProduitsType.php

Success!
```

Grâce à cette commande, on peut remarquer que Symfony, nous a ajouté un dossier dans ./src.



```
<?php

namespace App\Form;

use App\Entity\Produits;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class ProduitsType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('nom')
            ->add('description')
            ->add('price')
            ->add('stock')
        ;
    }

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

Si, on va dans le fichier ProduitsType.php :

→ add('nom') = ajout d'un input pour notre form.

Retournons dans notre contrôleur, nous allons lui ajouter différentes options supplémentaires pour la création de notre formulaire ainsi qu'au bon fonctionnement de celui-ci dans notre base de données !

```
#[Route('/produit/p/new', name: 'app_produit_new')]
public function newProduit(Request $request): Response
{
    $product = new Produits();
    $form = $this->createForm(ProduitsType::class, $product);

    return $this->render('produit/newProduct.html.twig', [
        'controller_name' => 'MainController'
    ]);
}
```

Dans un premier temps, il faut ajouter **Request \$request** dans les paramètres de notre fonction **newProduit()**. Cela permet, de préparer la fonction à un lancement de requête SQL.

On ajoutera également, **\$product** qui prendra comme valeur la structure d'un produit vide.

Puis, nous ajouterons **\$form**, qui lui va créer un formulaire en se basant sur notre formulaire type créé précédemment, tout en incluant les valeurs de notre produits.

Maintenant, il faut dire à la variable **\$form** qu'à l'envoi du formulaire, il prépare la requête SQL puis l'intègre à notre produit, pour cela, il faut ajouter un **handleRequest()** et lui spécifier que c'est bien une requête :

```
$form->handleRequest($request);
```

Et comme souvent, si on veut interagir avec une variable dans notre page vue (*newProduct.html.twig*), il faut le déclarer dans le **render()** pour créer la vue du formulaire :

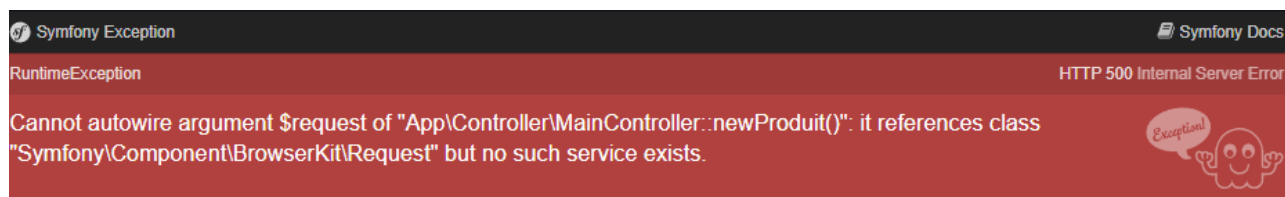
```
#[Route('/produit/p/new', name: 'app_produit_new')]
public function newProduit(Request $request): Response
{
    $product = new Produits();
    $form = $this->createForm(ProduitsType::class, $product);
    $form->handleRequest($request);

    return $this->render('produit/newProduct.html.twig', [
        'controller_name' => 'MainController',
        'form'=>$form->createView()
    ]);
}
```

Dès que c'est fait, on peut se rendre dans notre page newProduct.html.twig, puis ajouter ceci :

```
{{form(form)}}
```

Si jamais, vous avez l'erreur suivante :



Il faudra vous rendre dans le contrôleur puis changer la ligne :

```
use Symfony\Component\BrowserKit\Request;
```

Par la ligne :

```
use Symfony\Component\HttpFoundation\Request;
```

```
namespace App\Controller;

use App\Entity\Produits;
use App\Form\ProduitsType;
use Doctrine\ORM\EntityManager;
use Doctrine\ORM\EntityManagerInterface;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\BrowserKit\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

namespace App\Controller;

use App\Entity\Produits;
use App\Form\ProduitsType;
use Doctrine\ORM\EntityManager;
use Doctrine\ORM\EntityManagerInterface;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
```

Vous devriez avoir un résultat similaire à ceci :

A screenshot of a web form titled 'Créer un produit'. The form has a dark blue header with the 'Flowbite' logo. Below the header, there are four input fields: 'Nom', 'Description', 'Price', and 'Stock'. The 'Stock' field is a checkbox. At the bottom of the form, there is a link 'Retour au produit'.



Pour plus de personnalisation, il y a une autre façon de créer un formulaire depuis notre page vue (*newProduct.html.twig*).

On ouvrira le balisage TWIG comme ceci :

```
{{ form_start(form) }}
```

```
{{ form_end(form) }}
```

Détaillons un peu tout ça :

**{{}}** = balisage twig

**form\_start** = Début du formulaire

**form** = La variable que l'on a définie dans notre contrôleur → `'form'=>$form->createView()`

**form\_end** = Fin du formulaire

Et dans ces balises **form** nous pouvons y ajouter et personnaliser nos inputs :

```
{{ form_widget(form.nom) }}
```

Avant de détailler le widget, je vous invite à lire la documentation Symfony :

[https://symfony.com/doc/current/form/form\\_customization.html#form-form-view-variables](https://symfony.com/doc/current/form/form_customization.html#form-form-view-variables)

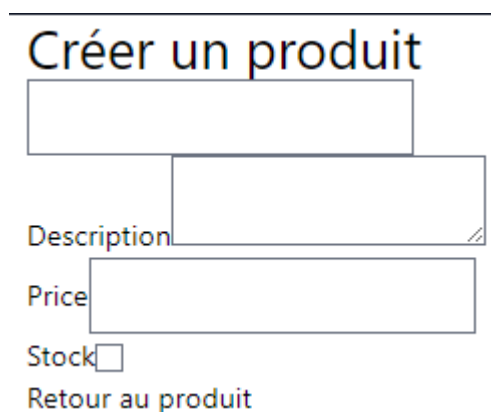
Passons le balisage TWIG, vous commencez à connaître. Passons directement à :

**form\_widget** = On sélectionne l'input sans le label

**form.nom** = On prend l'input « **nom** » de notre formulaire.

Donc pour résumer :

Je sélectionne un input sans son label, et je prends l'input qui est en rapport avec mon nom.



*Nouvelle façon d'inclure un input*


On peut constater que dans la version 2 de notre formulaire, le labelle nom à disparu, car on ne l'appelle plus.

Grâce à notre fichier ProduitsType.php situé dans le dossier « src/form », on pourra ajouter des classes de style directement depuis le → `add()`;



*Première façon de créer un formulaire*

Pour cela, il faudra spécifier le type d'input que c'est :

```
public function buildForm(FormBuilderInterface $builder, array $options): void
{
    $builder
        ->add('nom', TextType)
        ->add('description',  TextType [Doctrine\DBAL\Types]
        ->add('price', NumberType)
        ->add('stock')
    ;
}
```

Attention à choisir le type importé depuis [Symfony] :

<https://symfony.com/doc/current/forms.html#creating-form-classes>

```
use Symfony\Component\Form\Extension\Core\Type\TextType;
```

```
$builder
    ->add('nom', TextType::class)
    ->add('description', TextType::class)
    ->add('price', NumberType::class)
    ->add('stock')
```

Puis nous pouvons, lui ajouter des attributs :

```
$builder
    ->add('nom', TextType::class, [
        'attr'=>[]
    ])
    ->add('description', TextType::class)
    ->add('price', NumberType::class)
    ->add('stock')
```

On ouvre un tableau associatif qui prend comme valeur : « attr » qui lui-même doit retourner un tableau où on pourra y mettre ce qu'on veut

Donc on y ajoute, « class » → « nos classes tailwind » :

```
$builder
    ->add('nom', TextType::class, [
        'attr'=>[]
    ])
    ->add('description', TextareaType::class)
    ->add('price', NumberType::class)
    ->add('stock')
```

Et maintenant, si on retourne sur notre page d'ajout de produit :

## Créer un produit

Description

Price

Stock

[Retour au produit](#)

Notre champ « nom » à pris le style CSS tailwind. Dans l'hypothèse, nous pourrions y ajouter un « id », un « name » ou tout autre valeurs que l'on aurait besoin.

Pour vous le montrer, ajoutons un placeholder :

```
$builder
  ->add('nom', TextInputType::class, [
    'attr'=>["class=>"bg-gray-50 border border-gray-300 text-gray-900 text-sm rounded-lg focus:ring-blue-500 focus:border-blue-500 block w-[50%] p-2.5
      dark:bg-gray-700 dark:border-gray-600 dark:placeholder-gray-400 dark:text-white dark:focus:ring-blue-500 dark:focus:border-blue-500", "placeholder=>"Nom du
      produit"]
  ])
  ->add('description', TextInputType::class, [
    'attr'=>["class=>"bg-gray-50 border border-gray-300 text-gray-900 text-sm rounded-lg focus:ring-blue-500 focus:border-blue-500 block w-[50%] p-2.5
      dark:bg-gray-700 dark:border-gray-600 dark:placeholder-gray-400 dark:text-white dark:focus:ring-blue-500 dark:focus:border-blue-500",
      "placeholder=>"Description"]
  ])
  ->add('price', NumberType::class, [
    'attr'=>["class=>"bg-gray-50 border border-gray-300 text-gray-900 text-sm rounded-lg focus:ring-blue-500 focus:border-blue-500 block w-[50%] p-2.5
      dark:bg-gray-700 dark:border-gray-600 dark:placeholder-gray-400 dark:text-white dark:focus:ring-blue-500 dark:focus:border-blue-500", "placeholder=>"Prix"]
  ])
  ->add('stock')
```

## Créer un produit

Nom du produit

Description

Prix

Stock ☐

[Retour au produit](#)

Le placeholder, permet d'ajouter un texte de remplacement si le champ est vide

NB : J'ai stylisé un peu les input en y ajoutant des « div » avec un mb-5 :

```
<article>
  {{ form_start(form) }}
  <div class="mb-5">
    {{ form_widget(form.nom) }}
  </div>
  <div class="mb-5">
    {{ form_widget(form.description) }}
  </div>
  <div class="mb-5">
    {{ form_widget(form.price) }}
  </div>
  {{ form_end(form) }}
</article>
```

On peut se poser la question de, pourquoi nous avons la checkbox stock ? → Stock ☒

En fait, nous allons devoir la supprimer dans notre contrôleur, car par défaut à l'ajout d'un produit, on désire que celui-ci soit en stock :

```
class ProduitsType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('nom', TextType::class, [
                'attr'=>["class"=>"bg-gray-50 border border-gray-300 text-gray-900"]
            ])
            ->add('description', TextareaType::class, [
                'attr'=>["class"=>"bg-gray-50 border border-gray-300 text-gray-900"]
            ])
            ->add('price', NumberType::class, [
                'attr'=>["class"=>"bg-gray-50 border border-gray-300 text-gray-900"]
            ])
        ;
    }
}
```

## Créer un produit

Vu que l'on a supprimé le  
→ **add('stock')**  
l'input checkbox a été supprimé.

On comprend mieux  
l'utilité des → **add()**

[Retour au produit](#)

Maintenant, il nous manque un bouton pour valider l'envoi du formulaire. Pour cela, nous allons créer un bouton des plus basique qui aura le type « **submit** » :

```
<button type="submit" class="focus:outline-none text-white bg-green-700 hover:bg-green-800 focus:ring-2 dark:bg-green-600 dark:hover:bg-green-700 dark:focus:ring-green-800">Créer le produit</button>
```

```
<button type="submit" class="">Créer le produit</button>
```

## Créer un produit

[Retour au produit](#)

Donc la création de tout ce qu'on a fait jusqu'à présent nous retourne :

```
<form name="produits" method="post">
  <div class="mb-5">
    <input type="text" id="produits_nom" name="produits[nom]" required="required" class="bg-gray-50 border border-gray-300 text-gray-900 text-sm rounded-lg focus:ring-blue-500 focus:border-blue-500 block w-[50%] p-2.5 dark:bg-gray-700 dark:border-gray-600 dark:placeholder-gray-400 dark:text-white dark:focus:ring-blue-500 dark:focus:border-blue-500" placeholder="Nom du produit">
  </div>
  <div class="mb-5">
    <textarea id="produits_description" name="produits[description]" required="required" class="bg-gray-50 border border-gray-300 text-gray-900 text-sm rounded-lg focus:ring-blue-500 focus:border-blue-500 block w-[50%] p-2.5 dark:bg-gray-700 dark:border-gray-600 dark:placeholder-gray-400 dark:text-white dark:focus:ring-blue-500 dark:focus:border-blue-500" placeholder="Description"></textarea>
  </div>
  <div class="mb-5">
    <input type="text" id="produits_price" name="produits[price]" required="required" class="bg-gray-50 border border-gray-300 text-gray-900 text-sm rounded-lg focus:ring-blue-500 focus:border-blue-500 block w-[50%] p-2.5 dark:bg-gray-700 dark:border-gray-600 dark:placeholder-gray-400 dark:text-white dark:focus:ring-blue-500 dark:focus:border-blue-500" placeholder="Prix" inputmode="decimal">
  </div>
  <button type="submit" class="focus:outline-none text-white bg-green-700 hover:bg-green-800 focus:ring-4 focus:ring-green-300 font-medium rounded-lg text-sm px-5 py-2.5 me-2 mb-2 dark:bg-green-600 dark:hover:bg-green-700 dark:focus:ring-green-800">Créer le produit</button>
  <input type="hidden" id="produits_token" name="produits[_token]" value="5911.47-bhq3_Uw99j4rn2wn2SV7NL1TKV3XpOTpU8dw_d8w. j4zd5JiX8H8v2czSq1Pu0m715xwIb0yRCXU6t5cGNeGR5_7-_Zc1Yy_u3w">
</form>
```

Si vous avez fait attention, Symfony a créé un input caché nommé « **produits[\_token]** », cet input, permet de sécuriser les envoies des formulaires.

Maintenant, si on clique sur notre bouton, rien ne se passe, aucun produit n'est créé, car il nous manque encore une étape, celle de la validation du formulaire.

Pour ça, allez voir votre contrôleur, puis ajoutons une condition :

```
#[Route('/produit/p/new', name: 'app_produit_new')]
public function newProduit(Request $request): Response
{
    $product = new Produits();
    $form = $this->createForm(ProduitsType::class, $product);
    $form->handleRequest($request);

    if($form->isSubmitted() && $form->isValid()){
        dd($form);
    }
}
```

Détaillons ça :

**if** = C'est notre condition « SI »

**\$form** → **isSubmitted()** = Le formulaire est envoyé

**&&** = Et

**\$form** → **isValid()** = Le formulaire est valide

Alors il nous retourne le **dd** de notre formulaire :

## Créer un produit

Créer le produit

```
MainController.php on line 62:
Symfony\Component\Form\Form {#815 ▾
  -config: Symfony\Component\Form\FormBuilder {#816 ▶}
  -parent: null
  -children: Symfony\Component\Form\Form {#817 ▶}
  -errors: []
  -submitted: true
  -clickedButton: null
  -modelData: App\Entity\Produits {#704 ▶}
  -normData: App\Entity\Produits {#704 ▶}
  -viewData: App\Entity\Produits {#704 ▶}
  -extraData: []
  -transformationFailure: null
  -defaultDataSet: true
  -lockSetData: false
  -name: "produits"
  -inheritData: false
  -propertyPath: null
}
```

```
-viewData: App\Entity\Produits {#632 ▾
  -id: null
  -nom: "test"
  -description: "test"
  -price: 44
  -stock: null
}
```

Dans le viewData, on voit bien ce qu'on a entré dans les champs de notre formulaire.

Souvenez-vous, nous avons créé la variable **\$product** que nous avons implémenté à notre formulaire, donc, si on fait un **dd** de **\$product**, Symfony aura pré-créé le produit :

```
MainController.php on line 62:
App\Entity\Produits {#632 ▾
  -id: null
  -nom: "test"
  -description: "test"
  -price: 44
  -stock: null
}
```

On peut voir ici, qu'il a mis dans le nom : test ce qu'on a entré dans nos champs.

Pour l'envoyer dans notre base de données, on utilisera ces valeurs !

On peut aussi voir, que l'id est le stock sont « null », c'est tout à fait normal, car, nous n'avons pas défini le stock et l'id s'incrémentera seul à l'insertion dans notre base de données.

Occupons-nous du stock. Dans notre condition, ça nous dit :

Si notre formulaire est envoyé et qu'il est valide, alors il nous exécute ce qu'y se trouve dans la condition, c'est à partir de là que nous allons définir la valeur par défaut de notre stock.

```
$product->setStock(true);
```

```
if($form->isSubmitted() && $form->isValid()){
    $product->setStock(true);
    dd($product);
}
```

Donc si on renvoie notre formulaire, nous aurons ceci :

```
MainController.php on line 63: Comme on le voit, notre stock est devenu valide.
App\Entity\Produits {#632 ▼
  -id: null
  -nom: "test"
  -description: "test"
  -price: 44
  -stock: true
}
```

Et pour terminer, pour valider l'envoi et ainsi incrémenter notre base de données avec notre nouveau produit, il faudra faire appel de nouveau à l'**EntityManagerInterface** :

```
#[Route('/produit/p/new', name: 'app_produit_new')]
public function newProduit(Request $request, EntityManagerInterface $entity): Response
{
```

Dès que vous avez intégré l'interface manager, on devra l'appeler tout en utilisant la fonction **persiste()** et la fonction **flush()** :

```
if($form->isSubmitted() && $form->isValid()){
    $product->setStock(true);

    $entity->persist($product);
    $entity->flush();

    return $this->redirectToRoute('app_main');
}
```

**\$entity → persist(\$product)** = Notre produit est bien créé.

**\$entity → flush()** = On pousse notre produit dans la base de données.

Puis nous on créer une redirection sur notre page index.html.twig, là où nos produits sont affichés.

Faisons un nouveau test de création de produit :

### Créer un produit

Souris Razor X

La meilleure souris gaming

30










Créer le produit

Retour au produit

### Souris Razor X

La meilleure souris gaming  
30€  
Quantité : 1

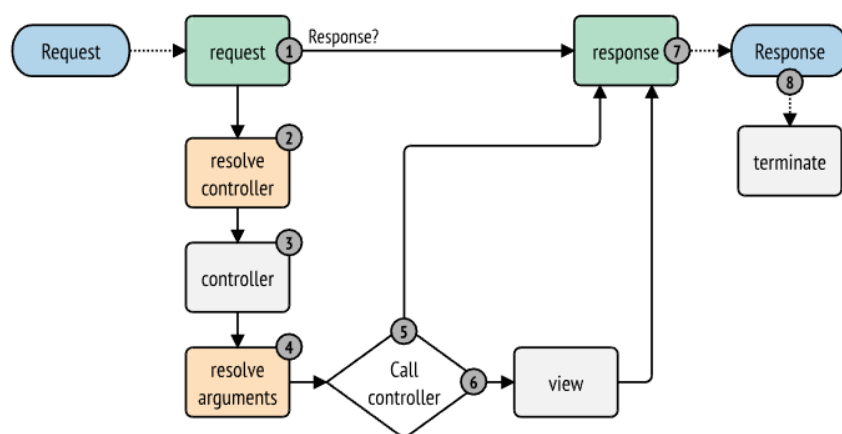
On a bien notre produit de créé !

<input type="checkbox"/>	 Éditer	 Copier	 Supprimer	1	Pommes	On aime les pommes ici !	3	2
<input type="checkbox"/>	 Éditer	 Copier	 Supprimer	2	ordinateur	ordinateur gaming prêt à l'emploi pour vos meille...	1000	1
<input type="checkbox"/>	 Éditer	 Copier	 Supprimer	3	Souris Razor X	La meilleure souris gaming	30	1

## Les request et response

Cette partie sera brève, nous en reparlerons plus en profondeur dans un chapitre dédié si ce PDF n'est pas trop long. Mais je tenais à en parler quand même, car nous avons fait appel au request ainsi qu'au response à plusieurs reprises sans vraiment savoir à quoi cela correspondait.

Regardons d'abord ce schéma. On peut y voir un cheminement peu compréhensible au premier abord, mais, qui en fait contient toute la logique de Symfony dans ses exécutions de requêtes.



Symfony a en son programme, c'est ce qu'on appelle, le HTTP KERNEL : <https://symfony.com/>

Le HTTP KERNEL fonctionne en 8 étapes comme vu dans le schéma précédent :

1. L'utilisateur demande une ressource dans un navigateur ;
2. Le navigateur envoie une requête au serveur ;
3. Symfony donne à l'application un objet Request ;
4. L'application génère un objet Response en utilisant les données de l'objet Request ;
5. Le serveur renvoie la réponse au navigateur ;
6. Le navigateur affiche la ressource à l'utilisateur.
7. Le navigateur a traité la réponse
8. La réponse se termine

Chacune des actions est appelée événement, donc dans notre exemple précédent lors de l'ajout en base de données, on a fait appel à la classe Request pour créer un événement à notre navigateur.

Renders a view.

```
<?php
protected function render(string $view, array $parameters = [], ?Response $response = null): Response { }
```

En exemple, notre « render » dans le contrôleur, lui nous renvoie une réponse (la création des vues).

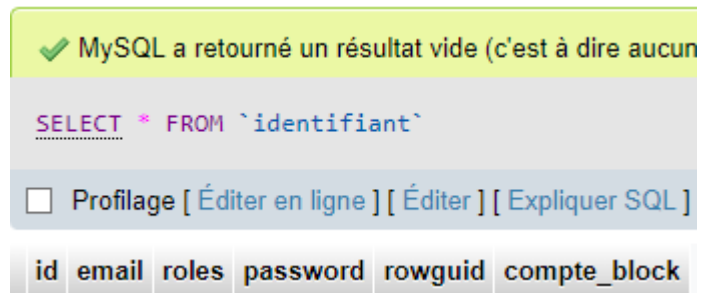
Si on fait un **dd** de **\$request** vous verrez que Symfony nous retourne tout un tas d'informations traité par Symfony pour son bon déroulement.



# Gestion de l'authentification

Souvenez-vous au tout début de la formation, nous avons créé une table identifiant qui comporte :

- email
- roles
- password
- rowguid
- compte\_block



Pour cela, nous avons utilisé la commande :

```
symfony console make:user
```

Revenons sur notre entité « identifiant », on avait par défaut laissé « email » en unique, cela correspond donc, qu'un utilisateur ne peut se créer de compte s'il utilise un email déjà enregistré dans la base de données.

```
#[ORM\Column(length: 180, unique: true)]  
private ?string $email = null;
```

← On le voit avec l'option « unique : true »

Comme vous le savez maintenant, si nous voulions ajouter une colonne à notre table « identifiant », nous utiliserons la commande :

```
symfony console make:entity
```

Puis entrer « identifiant ». Faisons-le pour ajouter la colonne « pseudo ».

```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)  
$ symfony console make:entity  
  
Class name of the entity to create or update (e.g. GentleKangaroo):  
> identifiant  
identifiant  
  
Your entity already exists! So let's add some new fields!  
  
New property name (press <return> to stop adding fields):  
> pseudo
```

Rendez-vous dans notre fichier identifiant.php qui se trouve dans « src/entity/identifiant.php »

Ajoutons à notre colonne pseudo, l'option unique :

```
#[ORM\Column(length: 255)]  
private ?string $pseudo = null;
```

```
#[ORM\Column(length: 255, unique: true)]  
private ?string $pseudo = null;
```

Créons la migration et poussons là dans notre base de données.

```
neoxq@DESKTOP-G5027E0 MINGW64 /d:/wamp/www/symfony/Formation/MonProjet (master)
$ symfony console d:m:m

[WARNING] You are about to execute a migration in database "monprojet" that could result in schema changes and data loss. Are you sure you wish to continue? (yes/no) [yes]:
>

[WARNING] You have 2 previously executed migrations in the database that are not registered migrations.

>> 2023-12-11 22:31:51 (DoctrineMigrations\Version20231211223023)
>> 2023-12-16 19:41:00 (DoctrineMigrations\Version20231216194014)
```

id	email	roles	password	rowguid	compte_block	pseudo
----	-------	-------	----------	---------	--------------	--------

NB : après avoir créé le fichier de migration, pour la lancer, on peut raccourcir la commande avec :

**symfony console d:m:m**

Pour la création des formulaires d'inscription et de connexion, il existe une façon de le faire rapidement, mais nous allons le voir sans cette facilité pour bien comprendre les formulaires et comment les requêtes sont exécutés.

## Le sérialisation

Restons dans notre fichier entité, nous allons devoir y ajouter un paramètre supplémentaire le sérialisation :

```
#[ORM\Entity(repositoryClass: IdentifiantRepository::class)]
class Identifiant implements UserInterface, PasswordAuthenticatedUserInterface, \Serializable
{
```

Le sérialisation sert à convertir un objet dans un format spécifique (XML, JSON, etc ..). Nous au contraire, cherchons à garder :

- L'email
- Le mot de passe
- Le pseudo
- Le rowguid
- Le compte\_block

Désérialiser, pour que nos contenus soient des contenus brut. Il faudra également ajouter à la fin du fichier entité :

```
public function serialize()
{
    return serialize([
        $this->id,
        $this->email,
        $this->pseudo,
        $this->password,
        $this->compte_block,
        $this->rowguid
    ]);
}
```

```
public function serialize()
{
    return serialize([
        $this->id,
        $this->email,
        $this->pseudo,
        $this->password,
        $this->compte_block,
        $this->rowguid
    ]);
}
```

```
public function unserialize($serialized): void
{
    list(
        $this->id,
        $this->email,
        $this->pseudo,
        $this->password,
        $this->compte_block,
        $this->rowguid
    ) = unserialize($serialized);
}
```

```
public function unserialize($serialized): void
{
    list(
        $this->id,
        $this->email,
        $this->pseudo,
        $this->password,
        $this->compte_block,
        $this->rowguid
    ) = unserialize($serialized);
}
```

Pour en comprendre sur le sérialisation, je vous invite à lire la documentation Symfony :  
<https://symfony.com/doc/current/components/serializer.html>

L'utilisation de le sérialisation servira à éviter divers problèmes que l'on pourrait rencontrer dans le futur (*avec notre projet d'application*).

## Le formulaire

Maintenant, il faut créer un nouveau contrôleur qui lui sera dédié à notre connexion :

```
symfony console make:controller SecurityController
```

Il faudra modifier le chemin ainsi que le nom de l'app dans notre contrôleur en login :

```
class SecurityController extends AbstractController
{
    #[Route('/login', name: 'app_login')]
    public function index(): Response
    {
        return $this->render('security/login.html.twig', [
            'controller_name' => 'SecurityController',
        ]);
    }
}
```

Pensez à changer le nom du fichier index.html.twig dans le dossier security →

```
▼ security
  ✓ login.html.twig
```

On fait un léger retour en arrière sur la formation, souvenez-vous, quand nous avons créé la table identifiant, on a eu le message suivant :

```
created: src/Entity/Identifiant.php
created: src/Repository/IdentifiantRepository.php
updated: src/Entity/Identifiant.php
updated: config/packages/security.yaml
```

Allons voir le fichier « security.yaml ». La partie qui nous intéresse est la suivante :

```
main:
    lazy: true
    provider: app_user_provider
```

Nous allons ajouter des éléments dans cette partie (*normalement, ça se fait tout seul si nous utilisons la commande de création d'authentification et d'inscription.*)

Retournons dans notre « security.yaml », on va y ajouter notre formulaire de connexion, ainsi que son chemin et nous lierons le « provider » à notre form\_login :

```
form_login:
  login_path: app_login
  check_path: app_login
  provider: app_user_provider
```

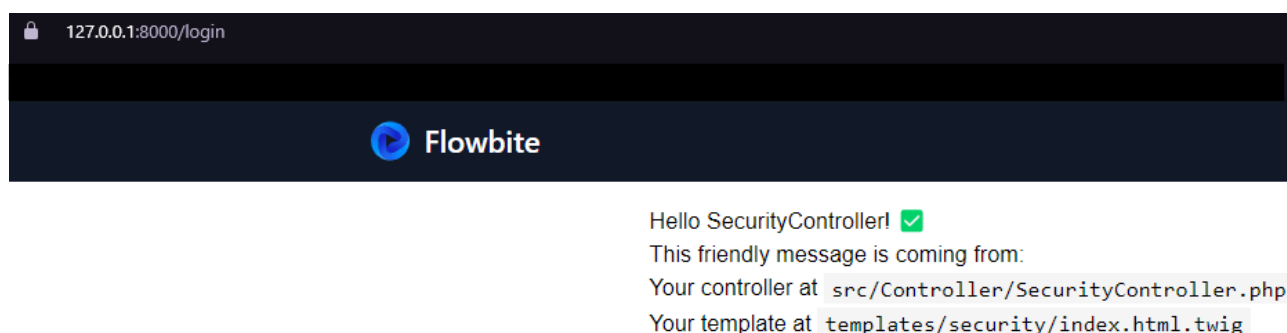
Il faut absolument que le nom soit celui affiché dans la Route

Dans notre cas, c'est app\_login, donc il faudra adapter le nom en fonction de votre nom choisi.

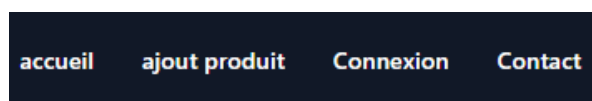
*NB : Si vous voulez qu'à la connexion soit redirigé, il faudra ajouter dans le form\_login :*

**`default_target_path: app_main`**

Et si on se rend sur notre page login voici ce que nous avons :



Ajoutons dans notre navbar, un lien vers notre page de connexion pour nous faciliter les différents voyages dans notre page web (*maintenant, vous savez faire*).



Retournons encore une fois dans notre contrôleur, pour lui ajouter des paramètres :

```
class SecurityController extends AbstractController
{
    #[Route('/login', name: 'app_login')]
    public function index(Request $request, AuthenticationUtils $authenticationUtils): Response
    {
        return $this->render('security/login.html.twig', [
            'controller_name' => 'SecurityController',
        ]);
    }
}
```

On remet encore une fois un **Request**, car on fait appel à un événement pour interagir avec notre base de données et on ajoute **AuthenticationUtils** qui sera là pour gérer les connexions (*gestion des erreurs, etc.*).

Ajoutons encore des choses dans notre contrôleur :

```
$error = $authenticationUtils->getLastAuthenticationError();
$lastPseudo = $authenticationUtils->getLastUsername();
```

```
$error = $authenticationUtils->getLastAuthenticationError();
$lastPseudo = $authenticationUtils->getLastUsername();
```

Détaillons ça rapidement :

**getLastAuthenticationError** = Retourne les erreurs liées à la connexion.

**getLastUsername** = Permet de ne pas vider le champ « email » si l'utilisateur se trompe.

Et comme d'habitude, on les ajoute à notre **render()** pour pouvoir les appeler sur notre page login.

```
return $this->render('security/login.html.twig', [
    'controller_name' => 'SecurityController',
    'error'=>$error,
    'lastpseudo'=>$lastPseudo
]);
```

Maintenant, créons notre formulaire complet à la main, rendez-vous dans notre page login.html.twig, supprimez tout sauf l'**extends** et la balise **TWIG** body.

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

{% block body %}

{% endblock %}
```

Dans le balisage « body » ajoutons notre formulaire :

```
<form action="{{path('app_login')}}" method="post" class="mt-5 flex flex-col">
    <input type="text" name="_username" value="{{lastpseudo}}" placeholder="Votre email" class="mb-5"/>
    <input type="password" name="_password" placeholder="Votre mot de passe" class="mb-5"/>
    <button type="submit" class="">Connexion</button>
</form>
```

Ajoutons du style :

```
<form action="{{path('app_login')}}" method="post" class="mt-5 flex flex-col">
    <input type="text" name="_username" value="{{lastpseudo}}" placeholder="Votre email" class="mb-5"/>
    <input type="password" name="_password" placeholder="Votre mot de passe" class="mb-5"/>
    <button type="submit" class="focus:outline-none text-white bg-green-700 hover:bg-green-800 focus:ring-4 py-2.5 me-2 mb-2 dark:bg-green-600 dark:hover:bg-green-700 dark:focus:ring-green-800">Connexion</button>
</form>
```

Dans notre formulaire, on a ceci :

```
_username
et
password
```

Il faut bien le spécifier, car c'est avec ça que notre formulaire renverra les bonnes données.

Résultat de notre page login :

Connexion

Vous pouvez essayer de vous connecter, vous n'y arriveriez pas, car aucun utilisateur n'est enregistré dans la base de données.

Affichons également les erreurs :

```
{% if error %}  
    {{ error.messageKey }}  
{% endif %}
```

Vous connaissez les conditions, donc, s'il y a une erreur, il m'affiche l'erreur.

Si vous tentez de vous connecter, vous aurez ceci : `Invalid credentials`.

Le `.messageKey`, permet de retourner seulement le message d'erreur, si on l'enlève et qu'on a une erreur, on a le détail complet de notre erreur et on ne veut pas qu'un utilisateur lambda ait accès à ce genre d'information.

---

```
Symfony\Component\Security\Core\Exception\BadCredentialsException: Bad  
credentials. in D:\wamp\www\symfony\Formation\MonProjet\vendor\symfony\security-  
http\Authentication\AuthenticatorManager.php:256 Stack trace: #0  
D:\wamp\www\symfony\Formation\MonProjet\vendor\symfony\http-  
foundation\Session\Storage\NativeSessionStorage.php(156): session_start() #1  
D:\wamp\www\symfony\Formation\MonProjet\vendor\symfony\http-  
foundation\Session\Storage\NativeSessionStorage.php(280):  
Symfony\Component\HttpFoundation\Session\Storage\NativeSessionStorage->start()
```

# Les fixtures

Les fixtures permettent de créer de fausses données, tout en gardant la structure de notre classe. C'est très important à savoir et à maîtriser.

Nous allons installer le pack des fixtures pour que Symfony génère nos données.

Je vous invite tout de même à lire la documentation pour en apprendre plus :

<https://symfony.com/doc/current/the-fast-track/en/17-tests.html#defining-fixtures>

Voici la commande à exécuter dans notre terminal GIT bash :

```
symfony composer req orm-fixtures --dev
```

Laisser le temps au paquet de s'installer. Dès que c'est terminé, lancer la commande :

```
symfony console make:fix
```

Entrer le nom de votre fixtures, ici, nous allons mettre userFixtures.

```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)
$ symfony console make:fix

The class name of the fixtures to create (e.g. AppFixtures):
> userFixtures

created: src/DataFixtures/UserFixtures.php

Success!
```

Maintenant, comme affiché sur le résultat de l'image ci-dessus, dans le dossier « src/DataFixtures », nous avons bien notre fichier créé :

```
▼ DataFixtures
  AppFixtures.php
  UserFixtures.php

namespace App\DataFixtures;

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

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

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

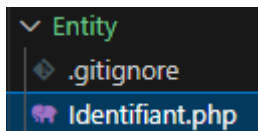
Tout en restant dans notre fichier fixtures, nous allons lui définir les paramètres nécessaires à la création de nos utilisateurs :

Les fixtures s'appuie sur l'**ObjectManager**, ce qui nous permet une meilleure gestion de nos objets.

Ajoutons cette ligne à notre fichier :

```
$user = new Identifiant();
```

**\$user** prend comme valeur la classe **Identifiant** que l'on peut retrouver dans nos entités :



Avez-vous remarqué, que dans notre fixture, il y a la fonction **flush()**. Tout comme l'ajout de produit, nous allons utiliser la fonction **persist()** pour lui envoyer la structure de notre objet **Identifiant**.

```
class UserFixtures extends Fixture
{
    public function load(ObjectManager $manager): void
    {
        $user = new Identifiant();
        $manager->persist($user);
        $manager->flush();
    }
}
```

Nous avons presque terminé la création de notre utilisateur fictif. Il manque encore deux – trois étapes. La première consiste à insérer des données par défaut. La seconde serait de créer un constructeur pour notre mot de passe et la dernière serait de hasher notre mot de passe.

Commençons par les données :

```
public function load(ObjectManager $manager): void
{
    $user = new Identifiant();

    $user->setEmail('test@test.fr');
    $user->setPseudo('test');
    $user->setRowguid('123456');
    $user->setCompteBlock('0');

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

```
$user->setEmail('test@test.fr');
$user->setPseudo('test');
$user->setRowguid('123456');
$user->setCompteBlock('0');
```



Passons au « **constructeur** », déjà à quoi sert un constructeur ?

*Les classes qui possèdent une méthode constructeur appellent cette méthode à chaque création d'une nouvelle instance de l'objet, ce qui est intéressant pour toutes les initialisations dont l'objet a besoin avant d'être utilisé.*

Voilà comment il est défini : `__construct(mixed . . . $values = ""): void`

```
class UserFixtures extends Fixture
{
    private $passhash;

    public function __construct(UserPasswordHasherInterface $passhash)
    {
        $this->passhash = $passhash;
    }

    public function load(ObjectManager $manager): void
    {
    }
```

Dans notre constructeur, on va lui définir un paramètre qui est [UserPasswordHasherInterface](#). Cette classe permet l'utilisation de hasher par défaut de Symfony, qui est [Bcrypt](#). Également utilisé dans notre security.yaml :

```
security:
    # https://symfony.com/doc/current/security.html#registering-the-user-hashing-passwords
    password_hashers:
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
```

On peut voir que nous avons défini une variable en private pour pouvoir l'appeler par la suite dans nos fonctions. Je vous invite à prendre connaissance de la POO (programmation orientée objet) :

<https://grafikart.fr/tutoriels/visibilite-public-private-556>

Cette classe requiert deux arguments, l'utilisateur concerné et le mot de passe à hasher :

```
$user->setPassword($this->passhash->hashPassword($user, "123"));
```

Désormais, toutes nos informations sont pré-crées, il ne reste plus qu'à lancer nos fixtures :

```
symfony console doctrine:fixtures:load
```

```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)
$ symfony console doctrine:fixtures:load

Careful, database "monprojet" will be purged. Do you want to continue? (yes/no) [no]:
> yes
```

Le message nous dit que notre base de données va être purgée pour incorporer notre fixture. Faites oui.

	id	email	roles	password	rowguid	compte_block	pseudo
<input type="checkbox"/> Éditer <input type="checkbox"/> Copier <input type="checkbox"/> Supprimer	1	test@test.fr		\$2y\$13\$XN6UbFtds1PNju7TbapF.DYKnC6sMhiSFSDfNOMfVM...	123456		0 test

Notre utilisateur est bien créé. Mais en contrepartie, nous avons perdu nos produits créés.

Pour le moment, ce n'est pas grave, car nous sommes encore en phase de développement, lorsque vous serez en phase de déploiement, l'utilisation des fixtures sera à proscrire, car les futurs utilisateurs seront eux de réelle personne.

Essayons de nous connecter avec nos informations choisies :

Connexion

Dans cet exemple, j'ai entré le bon email, mais un mauvais mot de passe. Comme ce ne sont pas les bons identifiant, une erreur nous sera retournée : **Invalid credentials.**

Par contre, si on entre « 123 » comme écrit dans notre fixture :

Logged in as test@test.fr

Authenticated Yes

Roles ROLE\_USER

Inherited Roles none

Token class UsernamePasswordToken

Firewall name main

test@test.fr 20 ms 2 in 0.0

Nous sommes bien connectés !

```
#[Route('/logout', name: 'app_logout')]
public function logout()
{}
```

Maintenant, que l'on est connecté, on veut pouvoir se déconnecter pour ça, rendez-vous dans notre contrôleur de connexion puis ajouter une Route qui n'aura aucun paramètre

Puis de nouveau se rendre dans notre security.yaml pour définir la déconnexion

```
main:
  lazy: true
  form_login:
    login_path: app_login
    check_path: app_login
    provider: app_user_provider

  logout:
    path: app_logout
    target: app_login
```

Donc si on se rend sur la page logout depuis notre navigateur, nous pourrons constater que nous ne sommes plus connecté :

127.0.0.1:8000/logout

Authenticated No

Firewall name main

n/a 33 ms

Pour que notre navbar s'adapte et change le lien connexion en déconnexion, il faudra ajouter une condition pour lui dire que si on est connecté, tu m'affiches « déconnexion », sinon tu m'affiches « connexion » :

```
{% if app.user %}
<li>
    <a href="{{path('app_logout')}}"
</li>
{% else %}
<li>
    <a href="{{path('app_login')}}"
</li>
{% endif %}
```

accueil ajout produit Connexion Contact

*Non connecté*

accueil ajout produit Déconnexion Contact

*Connecté*

Détaillons rapidement la condition :

**{{ }}** = Balisage TWIG

**{% if %}** = Condition SI

**app.user** = L'utilisateur est connecté

**{% else %}** = Sinon

**{% endif %}** = Fin de la condition

Voilà, notre formulaire de connexion est à présent terminé, vous avez déjà appris beaucoup de choses, nous feront également le formulaire d'inscription, mais avec la méthode dites plus « simple » dans un des prochains chapitres.

# Les flash

Les flash sont des alertes à durée limitée, ça nous permet de mettre plus en forme le design de notre site web et d'avoir un rendu plus agréable à nos alertes qui de base sont ignobles :

Invalid credentials.

Votre email

Votre mot de passe

Connexion

Invalid credentials en est la preuve vivante. On aimerait que cette alerte soit présente, comme ça l'utilisateur est notifié de l'erreur, mais ne s'occupe pas de cliquer quelque part pour fermer cette erreur.

Mettons en forme une alerte avec les composants flowbite. On ne va pas se compliquer la vie, prenons l'alerte la plus basique :

## Créer un produit

Success alert! Change a few things up and try submitting again.

Nom du produit

Description

Prix

Créer le produit

Retournons dans le contrôleur où nous avons la création de nos produits puis ajoutons notre flash :

```
$this->addFlash("success", "Ajout du produit réussit");
```

```
$entity->persist($product);  
$entity->flush();  
  
$this->addFlash("success", "Ajout du produit réussit");
```

*NB : changer la route pour rester sur notre page d'ajout de produit.*

Il va falloir créer une boucle sur nos messages, car **flash()** prend en compte toutes les alertes.

```
{% for Message in app.flashes('success') %}
  <div class="p-4 mb-4 text-sm text-green-800"
    |   {{ Message }}
  </div>
{% endfor %}
```

Détaillons la boucle :

**{{ }}** = Balisage TWIG

**for** = Boucle pour

**Message** = Variable qui prend les valeurs de nos flash

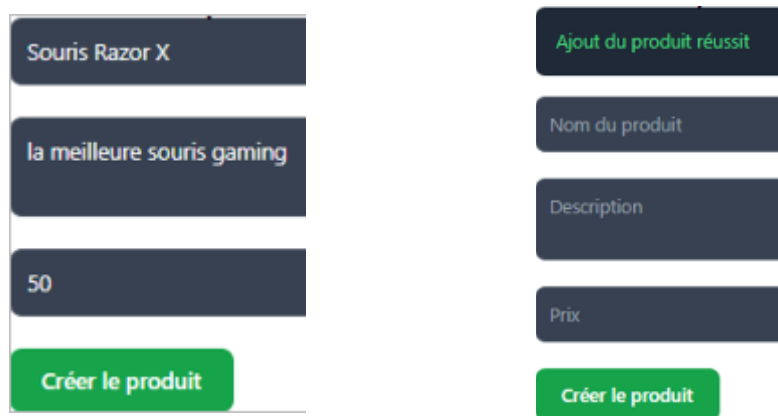
**in** = Dans

**app.flashes('success')** = Sélectionne le flash « success »

**{{ Message }}** = Affichage du message que l'on a créé

**endfor** = Fin de la boucle

Vérifions ça en créant un produit :



Petit bonus, si vous voulez que le nom du produit s'affiche, il faudra ajouter :

```
$this->addFlash("success", "Ajout du produit réussit ".$product->getNom());
```



Et si vous rafraîchissez la page, le message disparaît. D'où son nom **flash**.

# Formulaire d'inscription

Nous allons voir comment créer un formulaire d'inscription avec la méthode dite « facile ».

Pour cela, ajoutons notre formulaire avec la commande :

```
symfony console make:registration-form
```

```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/MonProjet (master)
$ symfony console make:registration-form
Creating a registration form for App\Entity\Identifiant

Do you want to add a #[UniqueEntity] validation attribute to your Identifiant class to make sure duplicate accounts aren't created? (yes/no) [yes]:
> 
```

Faire oui.

```
Do you want to send an email to verify the user's email address after registration? (yes/no) [yes]:
> no
```

Pour la vérification par « email », faire non, nous le verrons probablement sur la création du projet.

```
Do you want to automatically authenticate the user after registration? (yes/no) [yes]:
> no
```

Dites non à l'authentification automatique après l'inscription.

```
What route should the user be redirected to after registration?:
```

Symfony nous demande où nous devons rediriger nos utilisateurs après leur inscription, on choisira la page de connexion (affichée dans la liste) :

```
[14] app_main
[15] app_contact
[16] app_produit
[17] app_produit_new
[18] app_login
[19] app_logout
> 18
```

```
updated: src/Entity/Identifiant.php
created: src/Form/RegistrationFormType.php
created: src/Controller/RegistrationController.php
created: templates/registration/register.html.twig
```

Success!

Comme on le voit sur le screen ci-dessus, la commande nous à créé le contrôleur, la page template ainsi que la page type (*que nous avons créé pour la page connexion*).

Et en fait à partir de là, notre formulaire est presque terminé, il faut lui ajouter du style, et éventuellement enlever certains champs pour les des valeurs par défaut (comme la colonne `compte_block` qui sera à 0).

```
{% block body %}
    <section class="w-[30%] mx-auto mt-5">
        <h1 class="text-3xl mb-5">S'enregistrer</h1>

        {{ form_errors(registrationForm) }}

        {{ form_start(registrationForm) }}
        <div class="mb-5">
            {{ form_widget(registrationForm.email) }}
        </div>
        <div class="mb-5">
            {{ form_widget(registrationForm.plainPassword) }}
        </div>
        <div class="mb-5">
            {{ form_row(registrationForm.agreeTerms) }}
        </div>

        <button type="submit" class="btn">Register</button>
        {{ form_end(registrationForm) }}
    </section>
{% endblock %}
```

Puis retournons dans notre fichier type pour ajouter le champ pseudo ainsi qu'un placeholder :

```
$builder
->add('email', EmailType::class, [
    'attr'=>['placeholder'=>"Votre email"]
])
->add('agreeTerms', CheckboxType::class, [
    'mapped' => false,
    'constraints' => [
        new IsTrue([
            'message' => 'You should agree to our terms.',
        ]),
    ],
])
->add('pseudo', TextType::class, [
    'attr'=>['placeholder'=>"Pseudo"]
])
->add('plainPassword', PasswordType::class, [
    // instead of being set onto the object directly,
    // this is read and encoded in the controller
    'mapped' => false,
    'attr' => ['autocomplete' => 'new-password', "placeholder"=>"Votre mot de passe"],
    'constraints' => [
        new NotBlank([
            'message' => 'Please enter a password',
        ]),
        new Length([
            'min' => 6,
            'minMessage' => 'Your password should be at least {{ limit }} characters',
            // max length allowed by Symfony for security reasons
            'max' => 4096,
        ]),
    ],
]);
```

Minimum de caractères pour le mot de passe

Puis ajoutons à notre formulaire (*page register.html.twig*) :

```
<div class="mb-5">
|   {{ form_widget(registrationForm.pseudo) }}
</div>
```

Le formulaire a déjà une bonne tête, à vous après de le styliser à votre façon.

## S'enregistrer

Agree terms ☐

Register

Ajoutons seulement les valeurs par défaut de `compte_block` et `rowguid`. Pour ça, il faudra se rendre dans le contrôleur de notre formulaire d'inscription :

```
Controller
├── .gitignore
├── MainController.php
└── RegistrationController.php
```

Puis insérer ces deux petites lignes :

```
$user->setCompteBlock("0");
$user->setRowguid("123456");
```

```
$user->setCompteBlock("0");
$user->setRowguid("123456");
```

```
if ($form->isSubmitted() && $form->isValid()) {
    // encode the plain password
    $user->setPassword(
        $userPasswordHasher->hashPassword(
            $user,
            $form->get('plainPassword')->getData()
        )
    );

    $user->setCompteBlock("0");
    $user->setRowguid("123456");







    $entityManager->persist($user);
    $entityManager->flush();
    // do anything else you need here, like send an email

    return $this->redirectToRoute('app_login');
}
```

Et voilà, si on essaie de s'inscrire désormais, nous aurons bien un nouveau utilisateur, il faudra respecter les conditions du mot de passe (6 caractères, à vous de le changer si besoin dans *new Length*) :

Agree terms ☐

Register

<input type="checkbox"/>	 Éditer  Copier  Supprimer	1	test@test.fr		\$2y\$13\$XN6UbFtds1PNju7TbapF.DYKnC6sMhiSFSDfNOMfVM...	123456	0	test
<input type="checkbox"/>	 Éditer  Copier  Supprimer	2	aze@gmail.com		\$2y\$13\$MljrrwLXgnrw/svX5/ofC./7HewW6moVw52p9.8w/6e...	123456	0	aze

On peut constater que la création d'un formulaire est beaucoup plus simple. Surtout pour l'inscription / connexion. Mais il fallait voir comment créer un formulaire complet pour comprendre comment cela fonctionne ! C'est important de savoir quoi utiliser et comment l'utiliser.

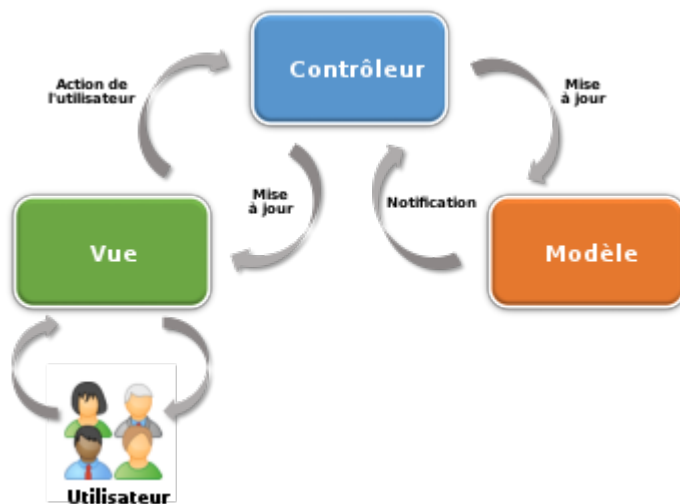
Je vous laisse libre d'essayer d'ajouter les erreurs et éventuellement un flash pour ceux-ci. Ça fera un bon exercice et vous pratiquerait sans devoir tout suivre à la lettre.



# Le modèle MVC

Avant de passer au projet, nous allons aborder différents points qui peuvent nous être utiles dans la réalisation d'un projet avec Symfony.

Le premier d'entre eux est le modèle MVC (*Modèle Vue Contrôleur*), depuis le début de cette formation, vous utilisez le modèle MVC sans même le savoir. On va détailler ça pour comprendre comment et à quoi ça sert.



Lien: <https://fr.wikipedia.org/wiki/Modèle-vue-contrôleur>

On peut voir via le schéma ci-dessus, que la vue communique avec le contrôleur, mais jamais avec le modèle et que le modèle lui, communique avec le contrôleur et jamais avec la vue. Alors que le contrôleur lui, communique avec les deux entités.

## La vue :

La vue est toute la partie cliente, soit celle que l'on voit dans notre navigateur ou dans notre cas, nos pages insérées dans « templates » (*index.html.twig, etc*).

## Le contrôleur :

Le contrôleur a le rôle le plus important dans le modèle MVC, c'est lui qui orchestre les requêtes côté client pour les envoyer côté serveur. Prenons par exemple notre ajout de produit (*cf. page 45*). Quand on est sur notre formulaire pour ajouter un produit, on lance une requête, que notre contrôleur va traduire et envoyer dans notre modèle pour son exécution.

## Le modèle :

Le modèle, lui, sert à instancier toutes les requêtes qui demanderont d'interagir avec la base de données. Par exemple dans notre cas, ce sont les différentes fonctions comme **persist()** ou encore **flush()**, et si nous avons créé une fonction, elle serait dans nos fichiers se trouvant dans « Repository ».

Cette façon de procéder existe depuis longtemps (1978), c'est une pratique connue pour ne jamais intégrer de fonction liée au back-end sur du front-end (*back-end* = *partie serveur*, *front-end* = *partie client*).

Il faut savoir, que cette méthode-là, ne nécessite pas d'utiliser Symfony, avec du PHP vanilla, c'est tout à fait possible. La seule contrainte sera qu'il faudra générer nos vues à la main et pas automatiquement comme le fait Symfony avec la fonction **render()**.

```
return $this->render('security/login.html.twig', [
    'controller_name' => 'SecurityController',
    'error'=>$error,
    'lastpseudo'=>$lastPseudo
]);
```

De plus, Symfony nous « simplifie » la vie en générant également les différentes Routes que l'on a besoin pour rediriger nos utilisateurs sur les différentes pages que l'on a créées. Comme par exemple une redirection sur notre page login, nous utilisons seulement un **path()**.

```
{{ path('app_login') }}
```

En PHP vanilla, il faudrait passer différentes étapes, pour que quand on clique sur notre connexion dans notre navbar, ce soit bien la page connexion qui nous est retournée :

```
public function routerRequete()
{
    if (isset($_GET['action'])) {
        switch ($_GET['action']) {
            case 'inscription':
                $this->ctrlUsers->inscription();
                break;
            case 'connexion':
                $this->ctrlUsers->connexion();
                break;
            case 'Admin':
                $this->ctrlUsers->PannelAdmin();
                break;
            default:
                throw new Exception("Action non valide");
        }
    } else {
        // Aucune action définie : affichage de tous les
        $this->ctrlProducts->getCategorie();
    }
}
```

Démo 1: *routeur.php en vanilla*

Il existe plein de façons de travailler, mais Symfony se base sur le modèle MVC, je vous invite à prendre plus ample connaissance de cette méthode pour comprendre toutes les subtilités du framework.

```
<?php
session_start();
require_once('controleur/Router.php');

$router = new Routeur();

$router->routerRequete();

?>
```

Démo 2: *index.php en vanilla*

# Service Container

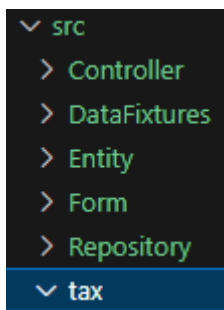
Nous allons aborder encore 2 parties avant de passer au projet, l'un d'eux est les services container, comme le modèle MVC, on l'a utilisé à plusieurs reprises sans même s'en rendre compte.

Un service container est par exemple `EntityManagerInterface`. Nous l'avons appelé à plusieurs reprises, en fait un service container est une fonction que l'on peut appeler n'importe où dans notre projet. Et ça depuis une classe créée spécialement pour ça.

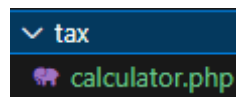
Prenons par exemple, pour nos produits, si on désire avoir un prix TTC (*toutes taxes comprises*), nous pourrions créer un service container et l'appeler depuis un constructeur ou même depuis notre contrôleur.

Tous les fichiers qui se trouvent dans « src » sont des services que l'on pourrait appeler.

Passons à notre exemple, dans le dossier « src » ajoutons un dossier « tax » :



Puis ajoutons un fichier PHP à ce dossier.



Dans le fichier calculator :

```
<?php
namespace App\Tax;

class calculator{

    public function caculTTC(float $prix)
    {
        return $prix * 1.2;
    }

}
```

```
<?php
namespace App\Tax;

class calculator{

    public function caculTTC(float $prix)
    {
        return $prix * 1.2;
    }

}
```

Maintenant, allons sur notre contrôleur de l'app\_main et ajoutons notre classe à notre constructeur :

```
protected $prix;
public function __construct(calculator $prix)
{
    $this->prix = $prix;
}

#[Route('/', name: 'app_main')]
```

```
#[Route('/', name: 'app_main')]
public function index(EntityManagerInterface $entity): Response
{
    $products = $entity->getRepository(Produits::class)->findAll();

    dd($this->prix->caculTTC(40));
}
```

Retournons sur notre index.html.twig :


```
MainController.php on line 29:
48.0
```

Nous avons bien le calcul du prix TTC.

Si, vous avez l'erreur suivante :

Cannot autowire service "App\Controller\MainController": argument "\$prix" of method "\_\_construct()" references class "App\Tax\calculator" but no such service exists. Did you mean "App\tax\calculator"?

Remplacer le nom du « USE » dans votre contrôleur, par celui recommandé :

<pre>namespace App\Controller;  use App\Entity\Produits; use App\Form\ProduitsType; use App\Tax\calculator;</pre>		<pre>namespace App\Controller;  use App\Entity\Produits; use App\Form\ProduitsType; use App\tax\calculator;</pre>
-------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------

Maintenant, regardons une autre façon d'appeler notre classe, de base, nous l'avions ajouté au constructeur, on peut faire appel à notre classe directement depuis les paramètres de notre fonction contrôleur :

```
class MainController extends AbstractController
{
    #[Route('/', name: 'app_main')]
    public function index(EntityManagerInterface $entity, calculator $prix): Response
    {
        $products = $entity->getRepository(Produits::class)->findAll();

        dd($prix->caculTTC(40));
    }
}
```

Vous aurez le même résultat, ça permet d'appeler la fonction directement là où on en a besoin. À vous d'adapter en fonction de vos besoins.

Il est important de comprendre que nous pouvons importer diverses fonctions via des librairies composer.

Premier site de paquet où l'on peut récupérer des fonctions déjà créées : <https://packagist.org>

Pour l'installation de paquet externe, je vous invite à chercher par vous-même, ce n'est pas le thème de la formation.

# Le projet concret : Le gestionnaire de mot de passe

Tout en reprenant la formation, nous allons :

1. Installer Symfony et installer les différents paquets nécessaires au projet
2. Création de la base de données (*avec un rapport OTM*)
3. Gestion de l'inscription / Connexion
4. Ajout de compte au gestionnaire

Je vous mets en garde d'avance, le style graphique sera strictement personnel et non poussé, vous adapterez la charte graphique à vos envies. Il sera également possible que je passe certaines étapes comme l'ajout de boutons, l'ajout de liens dans la navbar, etc.. Car ce sont des choses logiques. Sans elles, le site ne fonctionnerait pas correctement.

Ce projet est un exercice, qui vous permet de mettre en pratique ce que vous avez appris. Nous verrons des choses en plus tout au long de cet exercice. Donc, si vous veniez à ne pas comprendre une partie, pas de panique, elle sera sûrement expliquée, sinon vous aurez un lien redirigeant vers une initiation à ce qu'on fera au moment T.

Avant de commencer, je tiens à vous remercier d'avoir été jusqu'au bout de la formation Symfony. C'est un framework très performant et savoir l'utiliser pourrait vous être utile dans votre avenir en tant que développeur.

# Installer Symfony

Comme au tout début de la formation, ouvrez VSC, créer un dossier vide, ouvrez le, puis dans le terminal GIT BASH lancer la commande :

```
symfony new gestion_mdp --version="7.0.*" --webapp
```

*NB : Cette commande est fonctionnel pour une version supérieur ou égale à PHP 8.2.*

Entrons dans le dossier que nous à créé symfony.

```
cd gestion_mdp
```

Ajoutons le paquet logger :

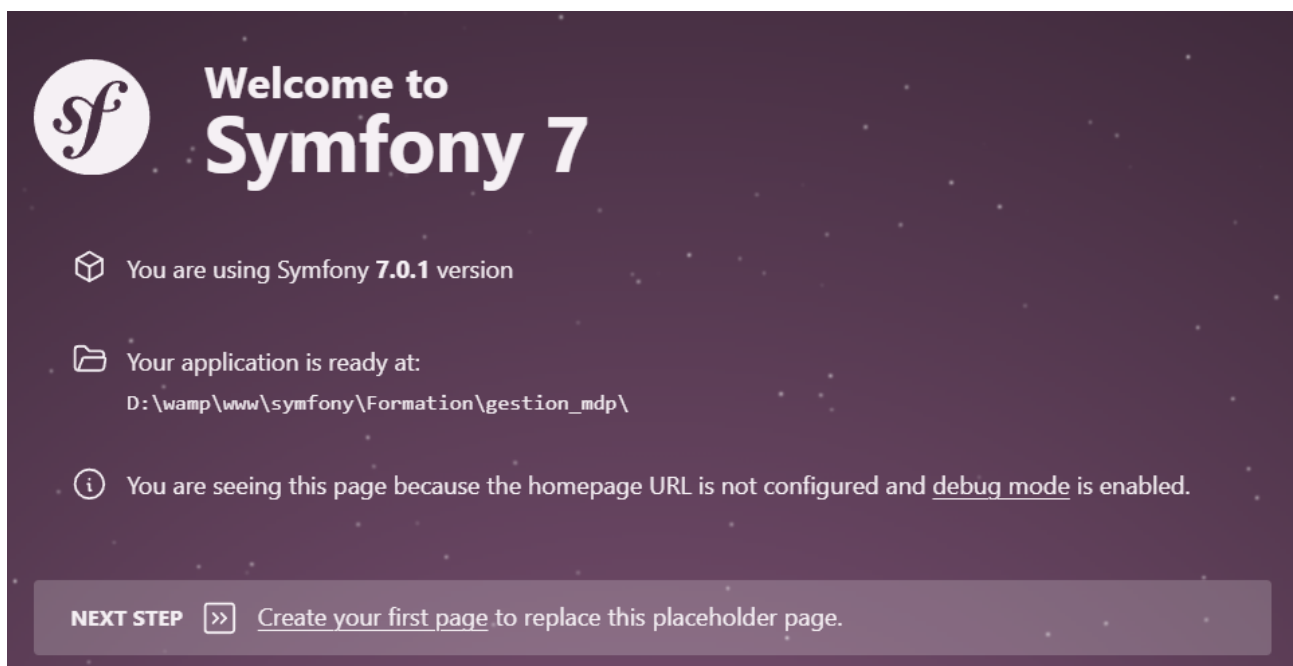
```
composer require logger
```

Installons le certificat SSL :

```
symfony server:ca:install
```

Testons si notre app est fonctionnelle :

```
symfony serve
```



Notre projet est fonctionnel, il manquera des paquets que nous installerons plus tard.

Ajoutons tailwind et flowbite dans notre base.html.twig via le CDN :

```
<script src="https://cdn.tailwindcss.com"></script>  
<link href="https://cdnjs.cloudflare.com/ajax/libs/flowbite/2.2.1/flowbite.min.css" rel="stylesheet"/>  
<script src="https://cdnjs.cloudflare.com/ajax/libs/flowbite/2.2.1/flowbite.min.js"></script>
```

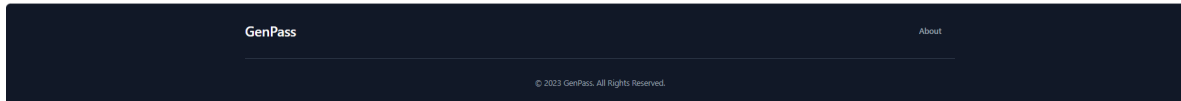
Ajoutons un footer et une navbar. À vous de choisir selon vos préférences.

Créons notre premier contrôleur pour y ajouter notre index.html.twig :

```
symfony console make:controller
```

Pensez à changer les chemins Route selon vos envies.

```
class MainController extends AbstractController
{
    #[Route('/', name: 'app_main')]
    public function index(): Response
    {
        return $this->render('main/index.html.twig', [
            'controller_name' => 'MainController',
        ]);
    }
}
```



La mise en page est terminée. Je vous laisse libre choix de gérer la mise en page. Le plus important sera l'utilisation des classes, etc.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>...
  </title>
  <script src="https://cdn.tailwindcss.com"></script>
  <link href="https://cdnjs.cloudflare.com/ajax/libs/flowbite/2.2.1/flowbite.min.css" rel="stylesheet"/>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/flowbite/2.2.1/flowbite.min.js"></script>

  <link rel="icon" href="data:image/svg+xml,<svg xmlns=%22http://www.w3.org/2000/svg%22 viewbox=%220 0 128
  {% block stylesheets %}...
  {% endblock %}

  {% block javascripts %}...
  {% endblock %}
</head>
<body>
  <header>...
</header>
  <main class="min-h-[75vh]">
    {% block main %}{% endblock %}
  </main>

  <footer class="bg-white rounded-t-lg shadow dark:bg-gray-900">...
</footer>
</body>
</html>
```

## Installer les différents paquets nécessaires au projet

Installons les paquets ORM et maker-bundle pour la gestion de notre base de données

```
composer require symfony/orm-pack
```

```
composer require --dev symfony/maker-bundle
```

## Création de la base de données

Dans notre projet, nous aurons besoin de deux tables :

1. Les utilisateurs avec leurs infos personnelles
2. Les comptes enregistrés

Avant de commencer la création de la base de données, parlons rapidement des différents modes de communication entre tables SQL.

**MTM** = many to many

**OTM** = one to many

**MTO** = many to one

**OTO** = one to one

Voyons ça de plus près. La méthode que nous allons utiliser pour définir notre structure SQL, s'appelle la méthode [MERISE](#). Cette méthode existe depuis 1970 et reste le b.a.-ba dans la création d'une base SQL.

Expliquons rapidement les différences :

### **Many To Many**

Imaginons, dans notre projet, nous voulions que nos utilisateurs aient accès à plusieurs haut-fait (*récompenses de fidélité par exemple*). Nous aurions une table « *haut-fait* » et notre table utilisateurs, chaque utilisateur pourrait avoir plusieurs haut-fait et plusieurs haut-fait peuvent être sur plusieurs utilisateurs (*Plusieurs pour Plusieurs*). Nous sommes bien dans la méthode MTM.

### **One To Many**

Ce cas est déjà plus présent dans notre projet actuel, nous voulons que chaque utilisateur ait plusieurs comptes enregistrés. Donc nous aurons une table « *utilisateurs* » et une table « *compte\_enregistre* ». Dans la table « *compte\_enregistre* » il pourrait y avoir plusieurs fois notre utilisateur, par contre, chaque compte enregistré ne sont associé qu'à un seul utilisateur (*Un pour Plusieurs*).



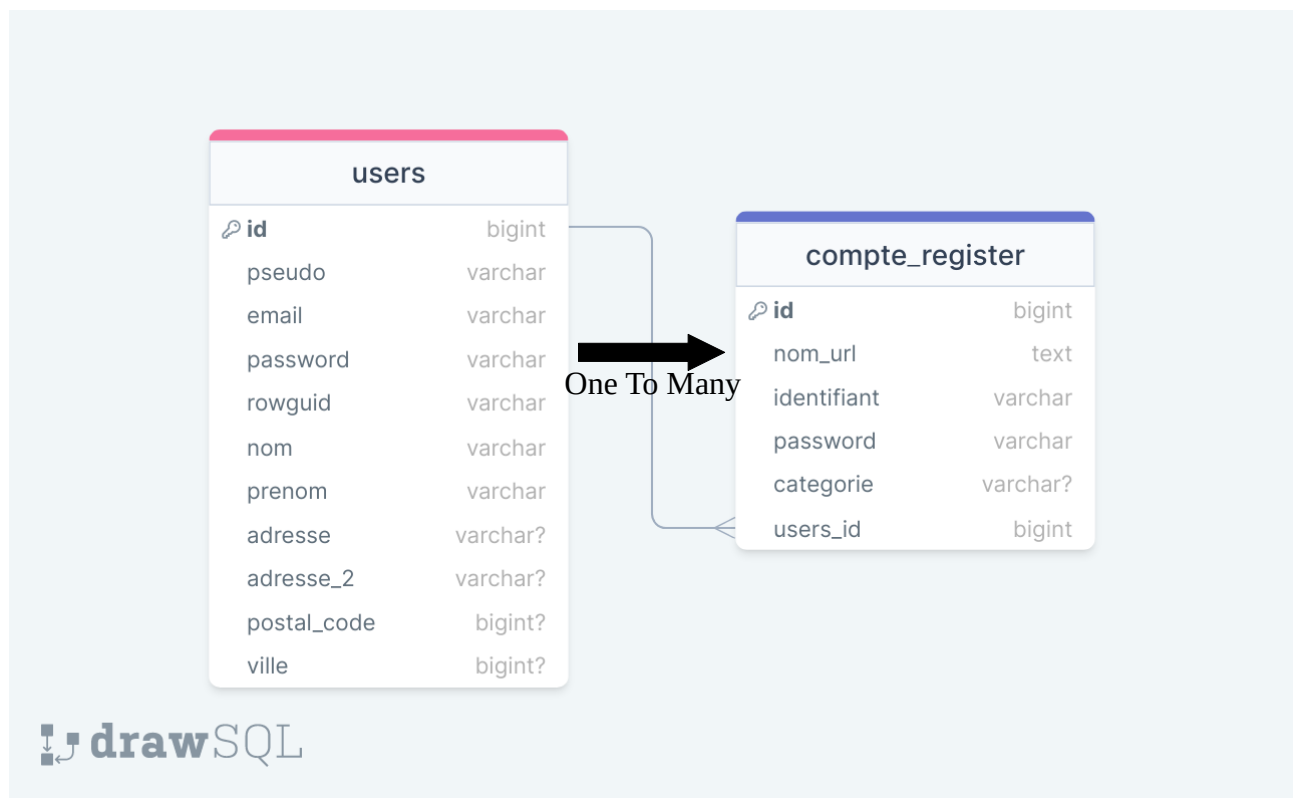
## Many To One

Dans le même cas que le OTM, nous pourrions avoir une table adresse et notre table utilisateurs. Dans la table adresse, il pourrait y avoir plusieurs adresse pour un seul utilisateur. (*Plusieurs pour Un*).

## One To One

C'est une méthode que nous utiliserons également, dans notre cas, ça sera avec les informations personnelles, chaque ligne d'informations sera liée à un seul utilisateur (*Un pour Un*).

Schéma de notre base SQL (on reste sur une structure facile à comprendre) :



Ici, nous aurons des clés étrangères, je vous invite à suivre le cours complet sur openclassroom :

<https://openclassrooms.com/fr/courses/mettez-en-relation-plusieurs-tables-avec-des-cles-etrangees>

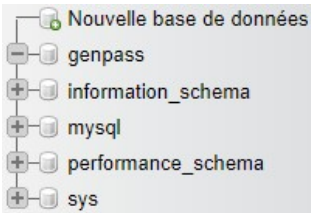
Depuis ce graphique, créons notre base, mais avant cela, modifions notre fichier **.env** pour lui donner les bonnes informations de connexion à notre SQL (*Dans la formation, je vous avais fait créer un fichier supplémentaire .env.local, dans ce projet pas besoin*).

```
# DATABASE_URL="sqlite:///kernel.project_dir/var/data.db"
# DATABASE_URL="mysql://app:!ChangeMe!@127.0.0.1:3306/app?serverVersion=8.0.32&charset=utf8mb4"
DATABASE_URL="mysql://root:root@127.0.0.1:3306/genpass?serverVersion=10.10.2-MariaDB&charset=utf8mb4"
```

Pour lancer la création de notre base SQL :

```
symfony console d:d:c
```

```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/gestion_mdp (master)
$ symfony console d:d:c
Created database `genpass` for connection named default
```



Créons ensuite nos tables, commençons par « *users* » :

```
symfony console make:user
```

```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/gestion_mdp (master)
$ symfony console make:user

The name of the security user class (e.g. User) [User]:
> users

Do you want to store user data in the database (via Doctrine)? (yes/no) [yes]:
>

Enter a property name that will be the unique "display" name for the user (e.g. email, username, uuid) [email]:
> email

Will this app need to hash/check user passwords? Choose No if passwords are not needed or will be checked/hashed
Does this app need to hash/check user passwords? (yes/no) [yes]:
>

created: src/Entity/Users.php
created: src/Repository/UsersRepository.php
updated: src/Entity/Users.php
updated: config/packages/security.yaml

Success!
```

Ajoutons ensuite les colonnes manquantes en suivant le schéma SQL.

Faisons juste une petite mise à jour sur notre entité « *users* », rendons la colonne pseudo, unique :

```
#[ORM\Column(length: 255, unique: true)]
private ?string $pseudo = null;
```

Pour la table « compte\_register » suivons également le schéma jusqu'à users\_id ou nous le lierons à users en ManyToOne :

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

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

What class should this entity be related to?:
> users
users

Is the ComptRegister.users_id property allowed to be null (nullable)? (yes/no) [yes]:
> no

Do you want to add a new property to users so that you can access/update ComptRegister objects from it - e.g. $users->getComptRegisters()? (yes/no) [yes]:
> yes

A new property will also be added to the users class so that you can access the related ComptRegister objects from it.

New field name inside users [comptRegisters]:
>

Do you want to activate orphanRemoval on your relationship?
A ComptRegister is "orphaned" when it is removed from its related users.
e.g. $users->removeComptRegister($comptRegister)

NOTE: If a ComptRegister may *change* from one users to another, answer "no".

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

updated: src/Entity/ComptRegister.php
updated: src/Entity/Users.php
```

Testons l'export de nos tables dans notre base SQL :

```
symfony console make:migration
```

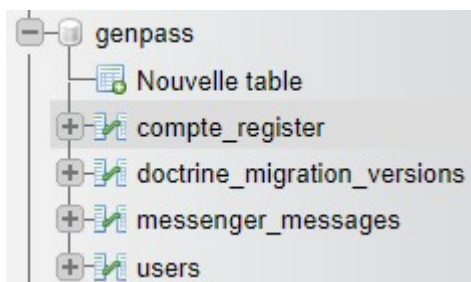
```
symfony console d:m:m
```

```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/gestion_mdp (master)
$ symfony console d:m:m

WARNING! You are about to execute a migration in database "genpass" that could result in schema changes and data loss. Are you sure you wish to continue? (yes/no) [yes]:
>

[notice] Migrating up to DoctrineMigrations\Version20231225142820
[notice] finished in 2075.9ms, used 24M memory, 1 migrations executed, 6 sql queries

[OK] Successfully migrated to version: DoctrineMigrations\Version20231225142820
```



Pour notre base de données, nous sommes bons.

Nous avons la structure, plus qu'à mettre en place l'inscription et la connexion

# Gestion de l'inscription / Connexion

Dans le projet concret, on utilisera la méthode rapide pour la création de nos formulaires de connexion et d'inscription.

## L'inscription

**symfony console make:registration-form**

Avec cette commande, nous aurons notre formulaire déjà pré-fait :

```
neox@DESKTOP-G5027E0 MINGW64 /d:/wamp/www/symfony/Formation/gestion_mdp (master)
$ symfony console make:registration-form
Creating a registration form for App\Entity\Users

Do you want to add a #[UniqueEntity] validation attribute to your Users class to make sure duplicate accounts aren't created? (yes/no) [yes]:
>

Do you want to send an email to verify the user's email address after registration? (yes/no) [yes]:
> no

Do you want to automatically authenticate the user after registration? (yes/no) [yes]:
> no

What route should the user be redirected to after registration?:
```

Pour la route de redirection, on met l'**app\_main** pour le moment, on pourra le changer par la suite.

Allez dans le contrôleur, puis changer les routes si vous le désirez.

**Inscription**

Accepter les terms ☐

Comme dans la formation, adapter le formulaire selon les différents champs que l'on a.

Ajouter du style et pensez à éventuellement changer le nombre de caractères par défaut du mot de passe.

Pour l'ajout du champ Répéter mot de passe, il faudra changer le type de « plainPassword » en RepeatedType et lui ajouter le type maître.

Pour les contraintes et l'ajout des placeholders je vous laisse chercher.

```
->add('plainPassword', RepeatedType::class, [
    // instead of being set on
    // this is read and encoded in the controller
    'type' => PasswordType::class,
```

id	email	roles (DC2Type:json)	password	pseudo	rowguid	nom	prenom	adresse	adre	pos	ville
1	aze@gmail.com	[]	\$2y\$13\$u1vWz8815AOV6K2zidc	test	387e48e3c428417	test	test	test	test	999	test

Notre formulaire fonctionne parfaitement.

## Les contraintes avec ASSERT

Je vous donne le lien pour en savoir plus, mais je vais essayer de l'expliquer au mieux :

<https://symfony.com/doc/current/validation.html#constraints>

Le validateur est conçu pour valider des objets par rapport à des contraintes (*c'est-à-dire des règles*). Afin de valider un objet, mappez simplement une ou plusieurs contraintes à sa classe, puis transmettez-la au service de validation. En arrière-plan, une contrainte est simplement un objet PHP qui fait une déclaration assertive.

Dans la vraie vie, une contrainte pourrait être : « *Le gâteau ne doit pas être brûlé* ».

Dans Symfony, les contraintes sont similaires : ce sont des assertions selon lesquelles une condition est vraie. Étant donné une valeur, une contrainte vous dira si cette valeur respecte les règles de la contrainte.

Pour utiliser les contraintes, il faut ajouter la ligne dans votre entité :

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

Nous allons ajouter une contrainte sur l'email, pour vérifier que c'est bien un email qui est entré dans le champ et pas simplement du texte.

```
#[Assert\Email(message: 'l'email {{ value }} n'est pas correct')]
```

```
#[Assert\Email(message: 'l'email {{ value }} n'est pas correct')]  
#[ORM\Column(length: 180, unique: true)]  
private ?string $email = null;
```

Dans le cas présent, si l'email n'est pas conforme, on pourra afficher l'erreur. Et nous pouvons faire ça avec n'importe quel champ (*sauf exception le mot de passe*).

Voyons pourquoi avec le mot de passe cela ne fonctionne pas du moins dans notre cas :

Dans notre entité « *users* », nous avons ceci pour le mot de passe :

```
#[ORM\Column]  
private ?string $password = null;
```

Alors que dans notre **formType** le mot de passe est définie par **plainPassword**:

```
->add('plainPassword', RepeatedType::class, [  
    // instead of being set onto the object directly,  
    // this is read and encoded in the controller  
    'type' => PasswordType::class,  
    'invalid_message' => 'The password fields must match.',  
    'first_options' => ['attr' => ['placeholder' => 'Mot de passe', "class"=>"mr-5"]],  
    'second_options' => ['attr' => ['placeholder' => 'Répéter mot de passe']],  
    'mapped' => false,  
    'required' => false,  
    'constraints' => [  
        new NotBlank([  
            'message' => 'Ce champs ne peut être vide.',  
        ]),  
        new Length(min: 5,  
            minMessage: "Le mot de passe doit contenir {{ limit }} caractères !",  
            max: 4096)  
    ],  
    'attr' => ['autocomplete' => 'new-password', "placeholder"=>"Mot de passe"]])
```

En effet, plainPassword prend la valeur de notre champ password, mais n'interagit pas avec notre variable \$password comme les autres champs.

```
$user->setPassword(  
    $userPasswordHasher->hashPassword(  
        $user,  
        $form->get('plainPassword')->getData()  
    )  
)
```

Pour notre email par exemple, on utilise bien la variable \$email :

```
#[ORM\Column(length: 180, unique: true)]  
#[Assert\Email(message: 'l'email {{ value }} n'est pas correct',)]  
private ?string $email = null;
```

```
->add('email', EmailType::class, [  
    "attr"=>["placeholder"=>"Email", "class"=>"w-full"],  
    'required'=>false  
)
```

Donc les « asserts » servent à donner des contraintes que notre formulaire doit respecter pour s'envoyer sans erreurs.

### Bonus affichage erreurs

Il se peut que vos erreurs ne s'affichent pas quand vous envoyez un formulaire. L'option --webapp installe par défaut des paquets qui entrent en conflit avec l'affichage de nos erreurs (*du moins, quand on utilise tailwind*).

Pour palier à ce problème, il suffit de se rendre dans notre base.html.twig et se supprimer les lignes suivantes :

Et il faudra bien spécifier chaque erreur à son champ :

```
<div class="mb-5">  
    {{ form_errors(registrationForm.email) }}  
    {{ form_widget(registrationForm.email) }}</div>
```

## Connexion

Comme pour l'inscription, il y a un moyen rapide de créer le formulaire de connexion :

```
symfony console make:auth
```

```
neox@DESKTOP-G5027E0 MINGW64 /d/wamp/www/symfony/Formation/gestion_mdp (master)
$ symfony console make:auth

What style of authentication do you want? [Empty authenticator]:
[0] Empty authenticator
[1] Login form authenticator
> 1
1

The class name of the authenticator to create (e.g. AppCustomAuthenticator):
> Login

Choose a name for the controller class (e.g. SecurityController) [SecurityController]:
> Login

Do you want to generate a '/logout' URL? (yes/no) [yes]:
>

Do you want to support remember me? (yes/no) [yes]:
>

How should remember me be activated? [Activate when the user checks a box]:
[0] Activate when the user checks a box
[1] Always activate remember me
> 1
```

Dans le LoginAuthenticator qui se trouve dans « src/Security », il faut activer une ligne pour que la redirection se fasse correctement à notre connexion :

```
public function onAuthenticationSuccess(Request $request, TokenInterface $token, string $firewallName): ?Response
{
    if ($targetPath = $this->getTargetPath($request->getSession(), $firewallName)) {
        return new RedirectResponse($targetPath);
    }

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

Commentez la ligne contenant `__FILE__` et dé-commenter la ligne juste au-dessus puis remplacer la route sur la page de votre choix.

### Connexion

Email

test@gmail.com

Password

.....

Connexion

Entrer les données de votre compte créé juste avant. Nous voilà enfin connectés :

Je vous laisse également gérer la mise en page de vos alertes, etc.

Logged in as

test@gmail.com

Authenticated

Yes

Roles

ROLE\_USER

Inherited Roles

none

Token class

PostAuthenticationToken

Firewall name

main

Actions

Logout

test@gmail.com

27 ms

1

## Ajout de compte au gestionnaire

Pour notre formulaire d'ajout de compte au gestionnaire, il va falloir créer un formulaire pour ça utiliser la commande :

```
symfony console make:form
```

Puis lier ce formulaire à l'entité « *CompteRegister* ». On peut ensuite voir que dans le type, on a bien nos différents champs :

```
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('nom_url')
        ->add('identifiant')
        ->add('password')
        ->add('categorie')
        ->add('users_id', EntityType::class, [
            'class' => users::class,
            'choice_label' => 'id',
        ])
}
```

Nous enlèverons users\_id et categorie. Ajoutons également un contrôleur pour notre formulaire gestionnaire :

```
symfony console make:controller
```

Retournons rapidement dans notre gestionType pour définir les types de nos champs et ajouter des placeholder :

```
$builder
    ->add('nom_url', TextType::class, [
        'attr' => ['placeholder' => "Nom ou url du site"]
    ])
    ->add('identifiant', TextType::class, [
        'attr' => ['placeholder' => "Identifiant de connexion"]
    ])
    ->add('password', PasswordType::class, [
        'attr' => ["placeholder" => "Mot de passe"]
    ])
```

Nous ajouterons des contraintes avec les asserts directement dans notre entité « *CompteRegister* ».

```
{% block main %}
<article class="flex flex-col items-center">
    {{ form_start(gestionForm) }}
    <div class="mb-5">
        {{ form_widget(gestionForm.nom_url) }}
    </div>
    <div class="mb-5">
        {{ form_widget(gestionForm.identifiant) }}
    </div>
    <div class="mb-5">
        {{ form_widget(gestionForm.password) }}
    </div>
    <button type="submit" class="w-full focus:outline-none">Ajouter
    {{ form_end(gestionForm) }}
</article>
```

Nous avons ajouté les champs nécessaires au formulaire, pour cela, inspirez-vous des contrôleurs déjà présent pour implémenter le vôtre correctement !



Maintenant, que le formulaire est « terminé », on va pouvoir s'attaquer au plus gros, l'insertion dans la base de données.

On sait qu'il faut :

1. Un nom ou une URL
2. Un identifiant
3. Un mot de passe.
4. L'id de l'utilisateur connecté.
5. Catégorie qu'on mettra à 0 par défaut.

On a déjà pré-fait trois de ses critères pour envoyer le formulaire. Pour le moment, préoccupons-nous de l'insertion de l'ID dans notre colonne « *users\_id* ».

Il faut savoir qu'il existe une fonction intégrée à Symfony qui nous permet de récupérer l'utilisateur connecté et ainsi afficher toutes ses informations :

```
$users = $this->getUser();  
dd($users);
```

Et comme c'est un utilisateur, on peut utiliser les fonctions « get » et « set » de notre classe Users :

```
$users = $this->getUser()->getId();  
dd($users);
```

```
GestionnaireController.php on line 23:  
4
```

```
GestionnaireController.php on line 23:  
App\Entity\Users {#943 ▾  
  -id: 4  
  -email: "test@gmail.com"  
  -roles: []  
  -password: "$2y$13$/vUDgP4k8Xi2v2lA/aDra.aqeI4FoCRrxaUvQn0Axn5c9k82ctxbs"  
  -pseudo: "teste"  
  -rowguid: "eebc7a90759a1afdc48e5578fa17486298e945464e2d613a1217266cdd2082"  
  -nom: "test"  
  -prenom: "test"  
  -adresse: null  
  -adresse_2: null  
  -postal_code: null  
  -ville: null  
  -compteRegisters: Doctrine_\PersistentCollection {#957 ►}  
  -isVerified: false  
}
```

Si votre [IDE](#) vous affiche du rouge sur la fonction getId(), c'est normal, le code fonctionnera tout de même !

Maintenant, qu'on sait comment récupérer l'ID, il va falloir le définir à notre colonne :

```
$user->setCategorie("0");  
$user->setUsersId($this->getUser(), $users);
```

Notre *users\_id* à besoin de deux paramètres pour s'exécuter

1. L'utilisateur référent
2. L'ID

```
<?php  
public function setUsersId(?users $users_id): static { }
```

avec le **\$this → getUser()** on lui définit l'utilisateur référent, puis avec **\$users** on lui définit l'ID, donc si on envoie un compte test, nous aurons ceci :

id	users_id_id	nom_url	identifiant	password	categorie
1	4	test	test	testt	0

On peut également constater que l'ID qui est associé à notre compte enregistré est bleuté et cliquable.

Si on survole l'ID voici qu'on a :

id	users_id_id	nom_url	identifiant	password	categorie
1	<a href="#">4</a>	test	test	testt	0
test@gmail.com					

Il a bien lié, notre compte utilisateur aux différents comptes que nous pourrions avoir (OTM).

Cliquons dessus pour voir, ça nous renvoie bien vers notre compte.

id	email
4	test@gmail.com

Pour plus de tests, créons-nous un nouveau compte et ajoutons un compte dans notre gestionnaire.

5 aze@gmail.com [] \$2y\$13\$JSXpTRnMG6.GW9ipJWmyweUzzjYA/3O811ed.e3vMcW... azerty

id	users_id_id	nom_url	identifiant	password	categorie
1	4	test	test	testt	0
3	5	azeok	aze	test	0

C'est parfait, on a bien un lien direct entre notre utilisateur connecté et les futurs comptes qu'il pourra enregistrer.

Affichons déjà un tableau avec nos comptes enregistré :

```
$events = $entityManager->getRepository(CompteRegister::class)->findBy(["users_id"=>$users]);

return $this->render('gestionnaire/gestion.html.twig', [
    'controller_name' => 'GestionnaireController',
    'gestionForm' => $form->createView(),
    'events'=>$events
]);
```

```
<tbody class="bg-white border-b dark:bg-gray-800 dark:border-gray-700">
    {% for compte in events %}
        <tr>
            <td class="px-6 py-4">{{ compte.NomUrl }}</td>
            <td class="px-6 py-4">{{ compte.identifiant }}</td>
            <td class="px-6 py-4">{{ compte.password }}</td>
        </tr>
    {% endfor %}
```

NOM / URL	IDENTIFIANT	MOT DE PASSE
azeok	aze	test
compte_aze	aze	testmdpaze

Comme on peut le voir, on arrive à afficher seulement les comptes que possède notre utilisateur grâce à la fonction **findBy()** qui va afficher seulement les résultats où l'`users_id` est égale à l'ID de notre utilisateur connecté.

*NB : Pour afficher les « nom\_url », il faut enlever le « \_ » et mettre des majuscules à chaque mot*

```
{{ compte.NumUrl }}
```

Ajoutons des boutons pour la suppression et la modification des comptes.

```
<td class="px-6 py-4">
  <button type="button" name="modifier" value="{{ compte.id }}"
  <button type="button" name="supprimer" value="{{ compte.id }}"
</td>
```

### La suppression :

```
if (isset($_POST['supprimer'])) {
    $del = $entityManager->getRepository(CompteRegister::class)->find($_POST['supprimer']);
    $entityManager->remove($del);
    $entityManager->flush();
}
```

Détaillons tout ça :

```
if (isset($_POST['supprimer'])) {
    $del = $entityManager->getRepository(CompteRegister::class)->find($_POST['supprimer']);
    $entityManager->remove($del);
    $entityManager->flush();
}
```

**if** = Condition Si

**isset(\$\_POST['supprimer'])** = Le bouton supprimé a été appuyé.

**\$del** = On cherche le compte qui contient l'ID du bouton.

**\$entityManager → remove(\$del)** = On supprime le compte que l'on a trouvé.

**\$entityManager → flush()** = On met à jour la table SQL

Donc maintenant, si on appuie sur notre bouton supprimer, le compte associé est bien enlevé !

Le petit souci est que, si on vient à modifier le numéro dans notre bouton via le code source HTML:

```
<button type="submit" name="modifier" value="9">
```

Ça supprimera la ligne associée au numéro.

On va pallier à ça avec une vérification d'`users_id` :

```
if (isset($_POST['supprimer'])) {  
    $del = $entityManager->getRepository(CompteRegister::class)->find($_POST['supprimer']);  
    if ($del->getUsersId()->getId() === $users) {  
        $entityManager->remove($del);  
        $entityManager->flush();  
    }  
}
```

Désormais, nous sommes protégés d'éventuel malin qui pourrait changer les numéros des boutons.

## La modification :

Passons au formulaire de modification, ajoutons d'abord un nouveau formulaire que l'on fera à la main dans notre vue (*là, où se trouve notre formulaire d'ajout de compte*) :

```
{% for compte in events %}  
    <tr>  
        <form method="post" class="flex flex-col">  
            <td class="px-6 py-4"><input type="text" class="text-black" name="NomUrl" value="{{ compte.NomUrl }}" /></td>  
            <td class="px-6 py-4"><input type="text" class="text-black" name="identifiant" value="{{ compte.identifiant }}" /></td>  
            <td class="px-6 py-4"><input type="text" class="text-black" name="password" value="{{ compte.password }}" /></td>  
            <td class="px-6 py-4 flex flex-col md:flex-row">  
                <button type="submit" name="modifier" value="{{ compte.id }}" class="focus:outline-none text-white bg-green-700 hov<br></td>  
                <button type="submit" name="supprimer" value="{{ compte.id }}" class="focus:outline-none text-white bg-red-700 hove<br></td>  
            </td>  
        </form>  
    </tr>  
{% endfor %}
```

Puis, nous allons ajouter presque le même script que pour la suppression à deux-trois détails près :

```
if (isset($_POST['modifier'])) {  
    $update = $entityManager->getRepository(CompteRegister::class)->find($_POST['modifier']);  
    if ($update->getUsersId()->getId() === $users) {  
        $update->setNomUrl(htmlentities(stripslashes($_POST['NomUrl'])));  
        $update->setIdentifiant(htmlentities(stripslashes($_POST['identifiant'])));  
        $update->setPassword(htmlentities(stripslashes($_POST['password'])));  
        $entityManager->flush();  
    }  
}
```

```
if (isset($_POST['modifier'])) {  
    $update = $entityManager->getRepository(CompteRegister::class)->find($_POST['modifier']);  
    if ($update->getUsersId()->getId() === $users) {  
        $update->setNomUrl(htmlentities(stripslashes($_POST['NomUrl'])));  
        $update->setIdentifiant(htmlentities(stripslashes($_POST['identifiant'])));  
        $update->setPassword(htmlentities(stripslashes($_POST['password'])));  
        $entityManager->flush();  
    }  
}
```

Au lieu du « remove », on « set » les nouvelles valeurs puis on pousse dans la base de données ce qui aura pour effet d'update.

Nous avons gardé la vérification users\_id pour éviter des modifications dans des comptes qui ne nous appartiennent pas !

NOM / URL	IDENTIFIANT	MOT DE PASSE	MODIFIER / SUPPRIMER	
<input type="text" value="azeaze"/>	<input type="text" value="az"/>	<input type="text" value="asdqsdqsd"/>	<button>Modifier</button>	<button>Supprimer</button>
<input type="text" value="qsdqsd"/>	<input type="text" value="merci :D"/>	<input type="text" value="azeazeaze"/>	<button>Modifier</button>	<button>Supprimer</button>

id	users_id_id	nom_url	identifiant	password	categorie
10	4	dfsdfsdfd	sdfsdf	sdfsdfsdf	0
11	5	azeaze	az	asdqsdqsd	0
14	5	qsdqsd	merci :D	azeazeaze	0

L'update fonctionne parfaitement. On va juste ajouter des contraintes au champ pour pas qu'on puisse avoir un mot de passe vide par exemple.

```
if (isset($_POST['modifier'])) {
    $update = $entityManager->getRepository(CompteRegister::class)->find($_POST['modifier']);
    if ($update->getUsersId()->getId() === $users && htmlentities(stripslashes($_POST['password'])) != "") {
        if(htmlentities(stripslashes($_POST['identifiant'])) != ""){
            if(htmlentities(stripslashes($_POST['NomUrl'])) != ""){
                $update->setNomUrl(htmlentities(stripslashes($_POST['NomUrl'])));
                $update->setIdentifiant(htmlentities(stripslashes($_POST['identifiant'])));
                $update->setPassword(htmlentities(stripslashes($_POST['password'])));
                $entityManager->flush();
            }else $this->addFlash("message", "Le champ Nom / Url ne peut être vide");
        }else $this->addFlash("message", "Le champ identifiant ne peut être vide");
    }else $this->addFlash("message", "Le champ mot de passe ne peut être vide");
}
```

On a mis en condition que si nos champs sont différents de vide alors c'est bon le script modifie notre ligne, sinon, on renvoie un message flash.

```
{% for message in app.flashes('message') %}
    {{message}}
{% endfor %}
```

Le champ mot de passe ne peut être vide

Et voilà, nous avons terminé notre projet. J'espère que vous avez appris des choses et que vous serez capable de réutiliser ce savoir pour votre avenir !

Si cette formation vous a plu, je vous invite à partager l'endroit où vous l'avez trouvé et à partager mon [GITHUB](#) ainsi que mon [LINKEDIN](#)

Lien du projet : [https://github.com/arkunis/Symfony\\_Formation](https://github.com/arkunis/Symfony_Formation)

## Merci encore !

Je vous ajoute un lien Discord pour que l'on puisse échanger si vous avez des questions :

<https://discord.gg/uTUC33zR5d>