Chapitre 1 Introduction

1.1 Symfony est un Framework PHP (Cadre de travail).

Il va donc vous offrir un cadre de travail. Contrairement au cours de PHP ou l'on devait créer tous nous même, il vous fournira un ensemble de classe que vous allez pouvoir utiliser pour créer une application WEB plus facilement. Il va vous donner une structure pour organiser votre projet.

C'est un Framework qui utilise la structure MVC (Model View Controller) qui est une manière de séparer son code plus proprement. Le **Model** c'est un peu comme l'administrateur, tout ce qui est source de données (la communauté de Symfony utilise doctrine). La **View**/vue (gérer généralement en **twig** qui est un moteur de Template) c'est un peu comme le marketing, tout ce qui est affichage. Le **Controller** c'est un peu comme le CEO de l'entreprise, c'est lui qui donne les directives, afin de dire au model récupère moi tel information en base de données et ensuite dire à la vue d'afficher l'information. C'est le chef d'orchestre. Le MVC se retrouve dans de nombreux Frameworks.

A l'heure actuelle où je crée ce cours, nous venons de passer à la version 7 de Symfony. J'ai fait un cours avec la version 5 et 6, sachez qu'il n'y a pas forcement de gros changement majeur. Il y a eu quelques petits changements entre la version 5 et 6 de Symfony parce que PHP passait de la version 7 à 8. Ce qu'on verra dans ce cours sera transférable sur la version 8, 9 etc. de Symfony.

Symfony 7 utilise PHP à partir de la version 8.2 donc pour pouvoir travailler dessus il faut vérifier que vous avez une version PHP >= 8.2.

1.2 Programme

Nous débuterons en explorant le Framework Symfony. Nous installerons Symfony, examinerons sa structure et créerons nos premières pages pour comprendre son fonctionnement de base. Ensuite, nous aborderons des concepts plus avancés à travers des exercices pratiques. Au cours de cette formation, nous élaborerons un petit site pour gérer des recettes de cuisine. Par la suite, nous plongerons dans des sujets plus spécifiques, tels que le fonctionnement interne, les systèmes de permission, etc. Enfin, nous approfondirons nos connaissances avec des travaux pratiques sur des cas concrets.

À ce stade, vous devriez être capable d'utiliser Symfony de manière autonome et d'explorer de nouveaux concepts par vous-même.

Pour pouvoir bien suivre ce cours, il est essentiel de maîtriser PHP. Si les termes comme "classe", "propriété publique ou privée", ou "instance" ne vous sont pas familiers, il serait difficile de comprendre les concepts avancés de Symfony. Il est aussi essentiel de maîtriser SQL, que les notions de tables, relations, clé primaire etc. soient acquise. De plus, vous devriez être à l'aise avec l'utilisation du terminal pour exécuter des commandes (CMD), car nous en aurons besoin pour interagir avec le Framework, par exemple pour générer du code. Enfin nous verrons un peu Composer, car cela vous aidera à installer le Framework et à ajouter de nouvelles dépendances.

Chapitre 2 Installation et découverte

2.1 Installation

Dans ce chapitre nous allons installer et découvrir la structure du Framework Symfony. Pour avoir plus d'information sur Symfony, il existe le site officiel qui est la documentation principale du Framework : www.symfony.com . Comme pour mon cours de PHP (php.net) et de MySQL(sql.sh), je vous recommande de mettre ce site en favoris, comme ça si vous avez besoin d'informations sur le Framework, vous y aurez accès directement.

Allons sur le site dans documentation Symfony Doc, ensuite dans getting started Setup et Installation. Vérifier tout d'abord que vous avez bien une version 8.2 de PHP ou supérieur dans le terminal en tapant « **php -v** ». Ceux qui ont une ancienne version, je vous invite d'abord à regarder dans Wamp quelle version de PHP sont disponible :

Si dans Wamp vous n'avez que des versions inferieurs à PHP 8.2, je vous invite à mettre à jour votre WampServer, pour pouvoir avoir les dernières versions de PHP en allant sur : https://wampserver.aviatechno.net/?lang=fr lors de l'installation de la mise à jour de Wamp n'oubliez pas de prendre les Microsoft Visual C++ à partir de 2010 dans les 2 versions x86 et x64. Une fois que votre WampServer est à jour, vous pourrez directement choisir votre version de php que vous voulez installer sur ce site. ATTENTION, une fois fini il faut modifier vos variables d'environnements système, pour choisir cette nouvelle version de PHP.

On va aussi installer **composer**. Il va nous permettre de gérer toutes les dépendances de notre projet, donc tous les packages dont le projet va dépendre. Je l'expliquerai par après.

Allez sur https://getcomposer.org/download/ télécharger et installer « **composer-setup.exe** » :

Il n'y a rien à cocher lors de l'installation.

Une fois fini lancer votre CMD et taper la commande :

- **composer -v**

Cette commande, vous donnera la version de « **composer** » sur votre machine.

Composer est installé maintenant il suffit d'installer Symfony. Allez sur https://symfony.com/download pour voir comment installer sur LINUX, MAC et Windows. Je vais montrer comment le faire sur Windows. Les personnes sur Linux et Mac, il suffit de suivre le tuto sur le site.

Une fois que c'est fait nous allons installer « **Scoop** » qui est un programme d'installation en ligne de commande pour Windows.

Tapez dans la barre de recherche Windows « **PowerShell** » et exécuter. Dans le terminal PowerShell taper cette commande qui va installer scoop :

- **Set-ExecutionPolicy** -ExecutionPolicy RemoteSigned -Scope CurrentUser
- **Invoke-RestMethod** -Uri https://get.scoop.sh | **Invoke-Expression**

Une fois installé, **quittez PowerShell** et lancer votre **CMD** (ligne de commande Windows).

Taper « **scoop** » pour voir s'il est bien installé. Si c'est le cas maintenant on va installer **symfony-cli** grâce à cette commande :

- **scoop install symfony-cli**

A partir d'ici les personnes sous Linux et Mac qui ont fini les installations vous pouvez nous rejoindre et vérifier que vous avez Symfony CLI installé sur vos machines en tapant dans la console (cmd) :

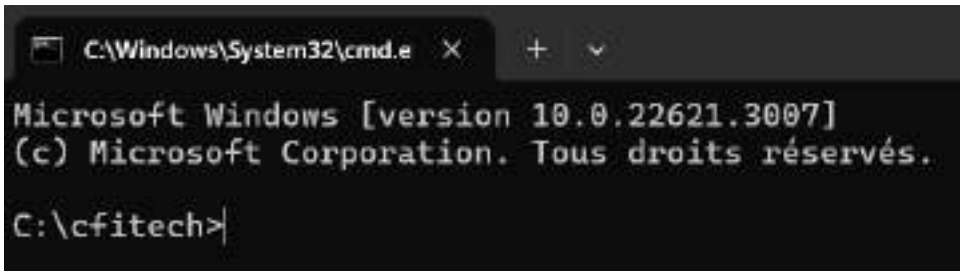
- **symfony -v** (ici il va vérifier la version)

J'ai décidé de passer par symfony cli en ligne de commande. Il faut savoir qu'il est possible de lancer symfony en passant par composer.

2.2 Création d'un projet

Maintenant que nous avons l'environnement installé de symfony, nous allons créer un nouveau projet. Je vous invite à aller dans votre PC et créer un projet à la **racine C:\.** Vous allez créer un nouveau dossier « **cfitech** » et entrer dedans.

Vous lancerez le cmd dans ce dossier.



Et là nous allons créer notre premier projet Symfony en tapant la commande :

- symfony new TutoSymfony

Ici on va donc créer un nouveau projet qui s'appelle « **TutoSymfony** ». A noter que je n'ai pas précisé la version de symfony comme c'est fait sur la doc officiel.

Cette commande va automatiquement créer le dossier correspondant et installer les différentes dépendances qui sont nécessaires. Il faudra ensuite rentrer dans ce dossier avec la commande « **cd TutoSymfony** ». Une fois dedans vous pouvez tapez « **code .** » pour ouvrir Visual Studio Code.

Avec la commande qu'on a tapée, on a installé le squelette c'est-à-dire une version de symfony qui est ultra légère et qui ne contient aucun des modules qui va être nécessaire à la création d'une application WEB. On n'a pas par exemple de système pour générer des vues HTML, on n'a pas la partie connexion à la base de données etc. Avec cette commande on devra rajouter par après les dépendances dont on a besoin pour créer une applications WEB.
On va donc ajouter une commande en plus qui va justement installer tous ceux dont on aura besoin pour une application WEB et c'est via composer qu'on peut le faire.

Composer va nous permettre de gérer toutes les dépendances de notre projet, donc tous les packages dont le projet va dépendre. Quand elles sont installées, elles vont se retrouver au niveau du dossier **vendor**. Donc à chaque fois qu'on fait « **composer require nomDuComposant** » on pourra le retrouver dans vendor.

A noter qu'on peut écrire :

- **composer req** comme raccourci pour composer **require**
- **composer rem** comme raccourci pour composer **remove**

On va donc installer webapp dans notre projet symfony en tapant :

- **composer require webapp**

Ça va installer tout un tas de dépendances supplémentaire. Il va nous demander si on veut inclure une configuration docker, et on dira oui. Normalement nous verrons docker beaucoup plus tard dans le cadre de votre formation.

A noter que si vous voulez on peut directement créer un projet symfony complet pour une application web en mettant directement dans la commande le webapp, donc dans C:\cfitech :

- **symfony new TutoSymfony --webapp**

C’est cette commande que je vous recommande de faire à chaque création de nouveau projet symfony. Après le faire de l’autre manière c’est totalement pareil.

Pour vérifier la version php de Symfony :

- **symfony php -v**

Pour vérifier la liste de version php prise en charge par Symfony :

- **symfony local:php:list**

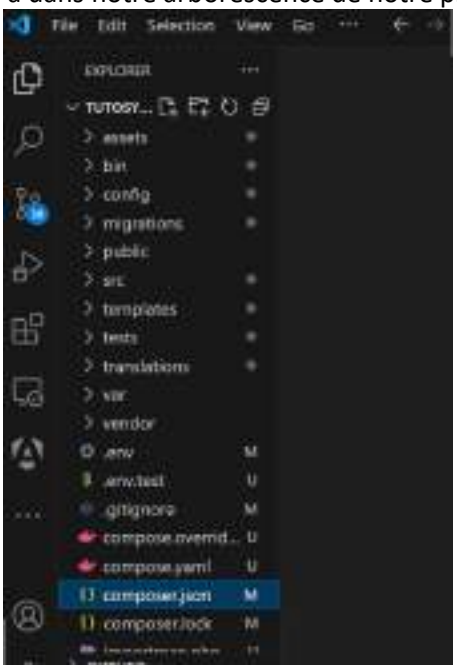
Pour changer la version de php dans le projet Symfony :

- **echo 8.*.* > .php-version**

Vous avez remarqué qu’il y a énormément de fichier dans votre projet, on va un peu les parcourir.

2.3 Description des dossiers et fichiers

On va un peu voir tous ce qu’on a dans notre arborescence de notre projet Symfony.



Le répertoire **assets/** contient des fichiers javascript et CSS.

Le répertoire **bin/** contient le principal point d’entrée de la ligne de commande : console. Vous l’utiliserez tout le temps. Il contient un deuxième fichier qui est phpunit, qui est une sorte de pont avec l’outil de test.

Le répertoire **config/** est constitué d’un ensemble de fichiers de configuration sensibles, initialisés avec des valeurs par défaut. Un fichier par paquet. Vous les modifierez de temps en temps. Ça utilise la syntaxe **yaml** qui est basé sur l’utilisation de l’indentation comme en python.

Le répertoire **migrations/** qui va contenir les migrations, qui sera très utilisé pour les informations concernant nos migrations en base de données.

Le répertoire **public/** est le répertoire racine du site web, et le script index.php est le point d'entrée principal de toutes les ressources HTTP dynamiques. Quand on va héberger notre application Symfony, c'est ce dossier la qui servira de racine pour notre serveur Web.

Le répertoire **src/** héberge tout le code (nos classes) que vous allez écrire ; **c'est ici que vous passerez la plupart de votre temps.** Par défaut, toutes les classes de ce répertoire utilisent le *namespace* App. C'est votre répertoire de travail, votre code, votre logique de domaine. Symfony n'a pas grand-chose à y faire. Dans le fichier composer.json à la racine du projet, si vous l'ouvrez, vous pourrez voir dans autoload qu'on a le namespace APP qui est branché sur le dossier src.

Le répertoire **templates/** va contenir nos vues HTML qui utilisent une extension twig.

Le répertoire **tests/** qui permettra d'écrire nos tests.

Le répertoire **translations/** va contenir les traductions parce que Symfony va nous permettre de créer un site multi langues.

Le répertoire **var/** contient les fichiers temporaires, les logs et les fichiers générés par l'application lors de son exécution. Vous pouvez le laisser tranquille. C'est le seul répertoire qui doit être en écriture en production.

Le répertoire **vendor/** contient tous les paquets installés par Composer, y compris Symfony lui-même. C'est notre arme secrète pour un maximum de productivité. Ne réinventons pas la roue. Vous profiterez des bibliothèques existantes pour vous faciliter le travail. Le répertoire est géré par Composer. N'y touchez jamais.

Le fichier **.env** va surtout permettre de gérer nos variables d'environnements. Il va nous permettre de configurer un peu le Framework. C'est ici qu'on configurera la connexion à notre base de données.

Le fichier **.gitignore** qui est en rapport avec git, pour indiquer les fichier qu' il ne faudra pas tracker.

Le fichier **composer.json** c'est le fichier de configuration de composer.

Le fichier **composer.lock** va nous permettre de locker spécifiquement les versions qui ont été installé.

Le fichier **symfony.lock** qui est en rapport avec Symfony Flex, la manière de gérer les applications Symfony.

Voilà une petite explication grosso modo des dossiers et fichiers qui composent notre projet. Je vous ai dit qu'il y a plus de 9000 fichiers et plus de 1000 dossiers lorsque le projet est créé avec webapp.

2.4 Lancer notre serveur

Si vous êtes toujours dans le dossier de votre projet dans la console. On va lancer le serveur :

- symfony server:start

Votre serveur est lancé et vous pouvez aller sur votre navigateur et taper 127.0.0.1 :8000 et normalement vous tomberez sur une page comme ça :

C'est une page qui est automatiquement générée par Symfony et qui nous présente les premières étapes à faire. En bas de la page nous avons un outil hyper important qui est une debug bar. Elle est activée par défaut quand on est en mode DEV. On l'utilisera pour débuguer justement. Il permet de voir aussi sous quelle version de Symfony tourne votre application (en bas à droite).

Il faut savoir qu'on peut lancer le serveur en back grounds en ajoutant un « **-d** », on peut le faire via la commande :

- **symfony server:start -d** (ou le raccourci **symfony serve -d**)

Pour lancer automatiquement la page locale après avoir lancer le serveur on fait :

- **symfony open:local**

Pour arrêter le serveur on peut faire la commande :

- **symfony server:stop** (**Si ça ne fonctionne pas faites Ctrl+C dans votre terminal)**

Pour pouvoir voir tous les serveurs lancer :

- **symfony server:list**

Pour lancer un serveur Symfony en **HTTPS** (sécurisé) :

- **symfony server:ca:install** (accepter le certificat)
- **symfony serve -d**

A noter que vous pouvez lancer plusieurs projets en même temps, il allouera juste les serveurs en incrémentant de 1 à chaque fois ex : 8000, puis 8001, puis 8002 etc.

Chapitre 3 : Création de nos premières pages

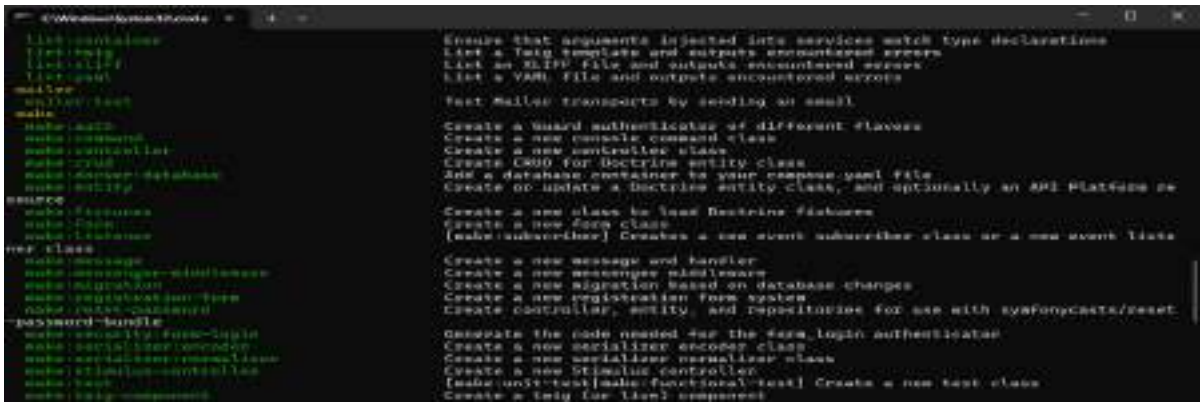
On va créer notre première page avec Symfony. Par défaut on arrivait sur la page d'accueil « **Welcome to Symfony 7** ». Pour changer ça on va devoir créer dans un premier temps un **Controller**. C'est une classe qui va contenir différentes méthodes, chaque méthode correspondra à une url particulière. On pourrait créer nous-mêmes ces classes mais Symfony permet de les créer automatiquement via la console (CMD).

3.1 La page index et les routes

Pour créer un **Controller**, il faut regarder les commandes

- **symfony console**

Cette commande vous liste toutes les commandes à votre disposition dans votre projet.



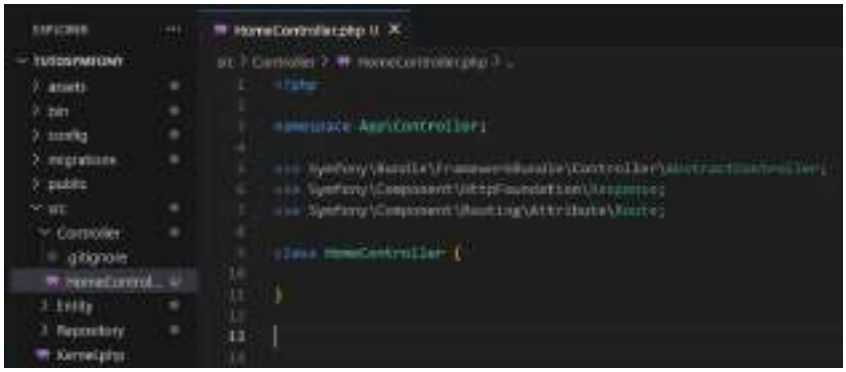
On verra qu'il y a un descriptif de chacune des commandes disponibles, on fera donc :

- **symfony console make:controller nomController**(pas obliger de noter Controller, il le rajoutera)

On va l'appeler **HomeController** ce qui donne donc **symfony console make:controller Home**.

Automatiquement il crée 2 fichiers, le **Controller HomeController.php** qui est une classe PHP dans le dossier **/src** et la vue (**View**) **index.html.twig** dans le dossier **/templates**.

Dans VScode allez là où se trouve votre Controller et ouvrez-le. Dans un premier temps on va supprimer ce qu'il y a dedans et vous aurez quelques choses comme ça :



On va repartir d'une classe simple.

Un Controller, je rappelle que c'est tout simplement une classe qui va contenir des méthodes et chaque méthode va correspondre à une page.

On va créer notre première méthode qui sera l'index sans paramètres donc notre page d'accueil et non plus la page d'accueil Symfony.

Les méthodes dans les Controllers doivent toujours renvoyer une réponse (qui est un Objet qui provient de Symfony : **Response**). Attention celle qui vient de HttpFoundation.

Donc je vais retourner une Response, qui lorsqu'on le construit, reçoit en premier paramètre le contenu, on va mettre « **Hello World !!! ».**

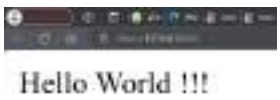
- **return new Response("Hello World !!!") ;**

Maintenant il va falloir expliquer au Framework que l'URL racine va correspondre à cette méthode-là. Il y a deux manières de faire. Je vais vous montrer les deux bien que j'en utiliserai qu'une seule. Les deux sont importantes à savoir.

La première manière c'est d'aller dans le dossier config à la racine de votre projet, dans le fichier « **routes.yaml** ». Là on va donner le nom de notre route donc « **home** », lui donner son chemin avec « **path** » qui sera la racine donc « **/** ». On écrit ensuite « **controller** » à qui on donnera le nom de notre controller donc en précisant son namespace et la méthode qu'on veut exécuter qui est donc index « **App\Controller\HomeController::index** » :

En gros j'explique à mon Framework que lorsqu'on a l'URL racine « **/** » il faut qu'il aille chercher cette action « **index** ». Attention dans YAML l'indentation est importante.

Une fois que c'est fait vous pourrez retourner dans votre navigateur et faire un refresh. Vous verrez votre Hello World !!! :



Voici la première manière de faire, qui était l'ancienne manière de faire. Beaucoup d'autres Frameworks utilisent ce système de mettre dans un fichier à part toutes les routes.

Vous verrez en Django et en React que ça se fait aussi. Le plus important c'est que vous compreniez comment ça fonctionne.

Maintenant je vais vous montrer la deuxième manière de faire qu'on fera tout le temps dans mon cours de Symfony.

Mettez en commentaire ce qu'on vient de faire dans « **routes.yaml** ». Les commentaires dans les fichiers YAML se font avec « # », vous pouvez faire dans Visual studio « Ctrl+/ ».

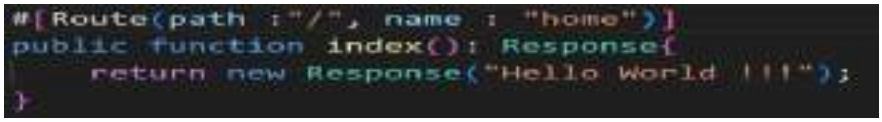
La partie en haut, ça permet de dire à Symfony, tu peux scanner les Controllers et si tu vois certains attributs, tu peux enregistrer les routes automatiquement.
On peut retourner dans notre Controller. C'est là qu'on va ajouter des attributs routes, directement au-dessus de nos méthodes qu'on a créées. Cet attribut vient du Namespace :

```
-   use Symfony\Component\Routing\Attribute\Route;
```

Qui était déjà présent dans notre HomeController. Le « **use** » c'est comme un require ou un include, ça permet d'importer. Voici la syntaxe des routes :

- #[Route("leCheminDeLUrl", "nomDeLaRoute")]

Le premier paramètre est donc le chemin de l'URL donc « **/** » et le deuxième paramètre c'est le nom de notre route « **home** ». On va nommer les paramètres, ça permettra de mieux les différencier.



Si vous retournez actualiser votre navigateur vous verrez toujours Hello World !!! Voilà la manière qu'on va utiliser à chaque fois. On a directement la route qui est lié à notre action. C'est plus pratique mais c'est bien de connaitre aussi que l'autre manière existe.

Remettez aussi le « **extends AbstractController** » à côté de la classe. C'est de l'héritage, donc notre controller **HomeController** hérite de la classe **AbstractController**. Ça va donc nous fournir des méthodes génériques. Par exemple grâce à cette classe parent, on pourra faire « $this->redirect » qui est fournis donc dans **AbstractController**.

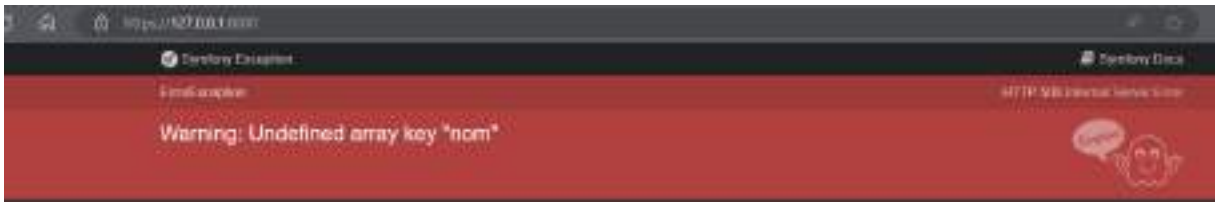
On a vu l'objet **Response** mais on va aussi parler d'un autre objet très important (**Request**) qui va nous permettre de gérer ce qui rentre dans le Framework. Rappelez-vous en PHP on pouvait mettre des informations dans l'URL et les récupérer. Ici on va essayer de faire ça.

Je vais mettre « **?nom=Dunia** »

En PHP dans le code on aurait utilisé une variable globale donc j'aurai mis « $_GET['nom'] ».

```php
#[Route(path :"/", name : "home")]
    public function index(): Response{
        return new Response("Hello ".$_GET['nom']);
    }
```

Et si vous testez vous allez voir que ça va bien afficher Hello Dunia. Ça fonctionne parce que je rappelle que Symfony c'est du PHP. Mais ce n'est pas la manière optimale à faire dans Symfony. Si maintenant vous enlevez de l'URL « **?nom=Dunia** », ça va nous faire une erreur.

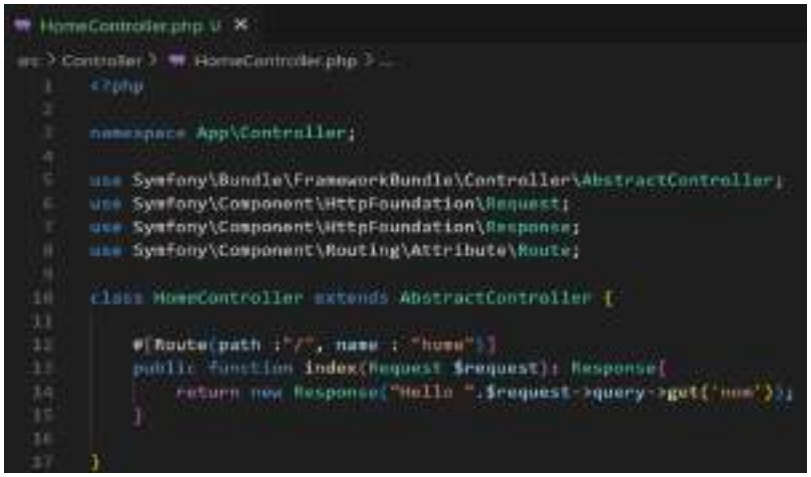


Nous on va plutôt mettre dans notre méthode « **index** » un paramètre de type **Request** (**Attention** il faut importer celui de **HttpFoundation** comme Response.)

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

Normalement Visual Studio vous l'importe automatiquement quand vous le choisissez.

Ensuite si on veut récupérer des paramètres dans l'URL, on a dans cet objet « **$request** » des méthodes utiles. Je dis je veux récupérer dans query le « **nom** » ce qui donne :



Si vous testez vous verrez que ça bien permis de récupérer la valeur de « **nom** » dans votre URL.
Si on enlève la suite dans l'URL, on aura plus l'erreur qu'on avait eu en utilisant basiquement la variable globale, il dira juste hello. On peut rajouter une valeur par défaut dans le **get()**.

```
-   $request->query->get('nom', 'World !!!')
```

Si vous n'avez rien dans l'URL il mettra Hello World !!! sinon si on met ?nom=Deo il mettra Hello Deo.

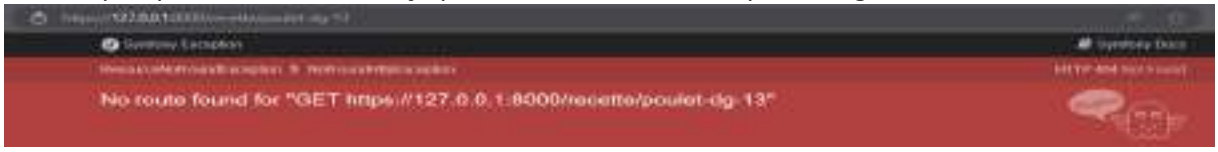
Il faut savoir qu'on peut comme en PHP utiliser un **var_dump()** pour savoir ce qu'il y a dans une variable. A savoir qu'on peut utiliser ces variantes qui afficherons d'une meilleure manière. Mettez en commentaire à chaque fois quand vous utilisez une manière différente :

```php
#[Route(path :"/", name : "home")]
public function index(Request $request): Response{
    //le var_dump classique en PHP
    var_dump($request);
    //le var_dump amélioré avec lequelle on peut rajouter si on veut un die derriere
    pour arreter la suite du code
    dump($request);
    die;
    //ici c'est la meme chose que de faire un Dump puis un Die => dd()
    dd($request);

    return new Response("Hello ".$request->query->get('nom', 'World !!!'));
}
```

On voit bien dans notre navigateur qu'il a créé un objet avec des attributs à NULL. Avec le dump en enlevant « **die** » vous remarquerez qu'il continuera l'affichage de la page.
A noter que quand maintenant on fera un **dump($request)** sans le « **die** », les données se retrouveront dans la barre de debug de Symfony qui a disparu pour le moment. A vous de choisir ce que vous préférez.

3.2 Une autre page avec route particulière

Je vais maintenant essayer de créer une route particulière. Imaginons que dans mon URL je veuille quelques choses comme ça pour la route « /recette/poulet-dg-13 » :



Bien entendu Symfony me dit qu'il n'a pas trouvé cette route donc erreur. Mais on va essayer de faire en sorte que ça passe. Pour faire ça on va devoir tout d'abord créer un nouveau controller.

Exos

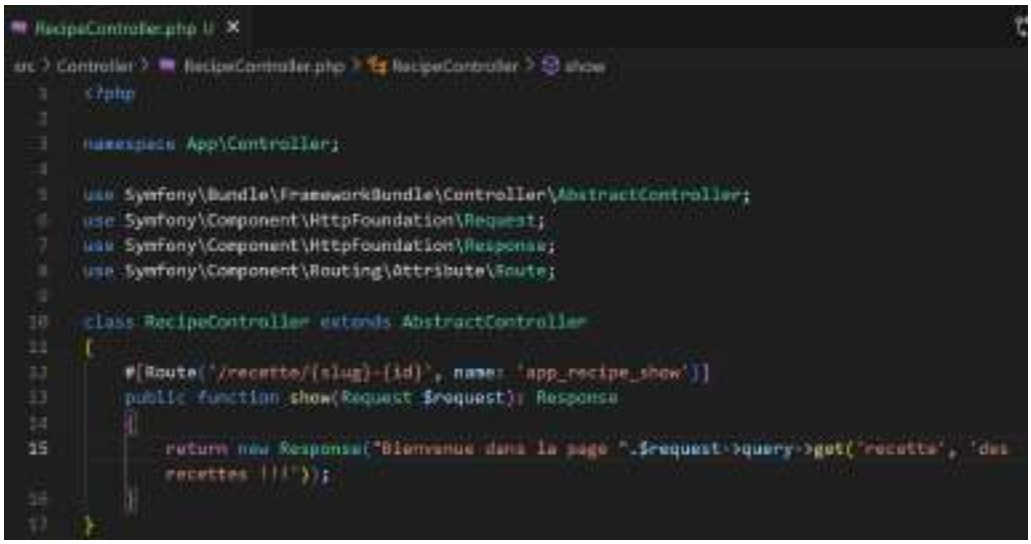
1) **Essayez de créer un Controller RecipeController qui permettra de gérer des recettes de cuisines.**
2) **Elle aura une méthode « show », une route avec comme chemin « /recette » et comme nom de route « app_recipe_show ».**
3) **Je veux qu'elle m'affiche dans votre navigateur « Bienvenue sur la page des recettes !!! »**

   Bienvenue dans la page des recettes !!!

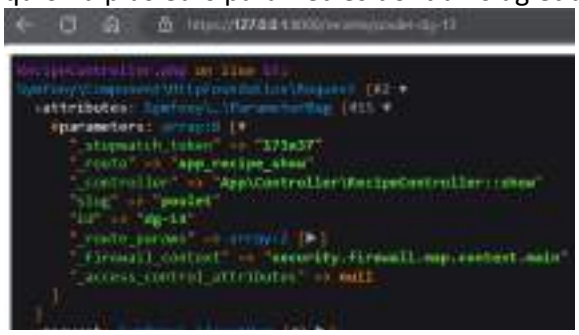
4) **(Difficile) Essayez de faire en sorte que si je mets « ?recette=Pundu » il me met :**

   Bienvenue dans la page Pundu

On va un peu modifier notre méthode « **show** » dans le « **RecipeController »** pour qu'elle fonctionne avec « **/recette/poulet-dg-13** ». On va y ajouter des paramètres entre accolade dans le chemin, un Slug qui représentera le nom de la recette et un id qui sera le numéro de la recette.



Si j'actualise ma page en mettant bien l'URL avec « **/recette/poulet-dg-13** », ça va fonctionner.
Si on fait un **dump** et **die** de **$request**, on peut voir dans « **attributes** » puis « **parameters** » qu'on a plusieurs paramètres dont un slug et un id. Il y a un petit problème non ?



On peut voir que « **slug »** contient juste « **poulet** » et que « **id »** contient par contre « **dg-13** ». Nous on voulait que « **id** » contienne juste **13** bien entendu.
Déjà pour que ce soit plus lisible on va faire un DD sur ces 2 attributs qui nous intéressent. Faites ça :

```
dd($request->attributes->get('slug') , $request->attributes->get('id'));
```

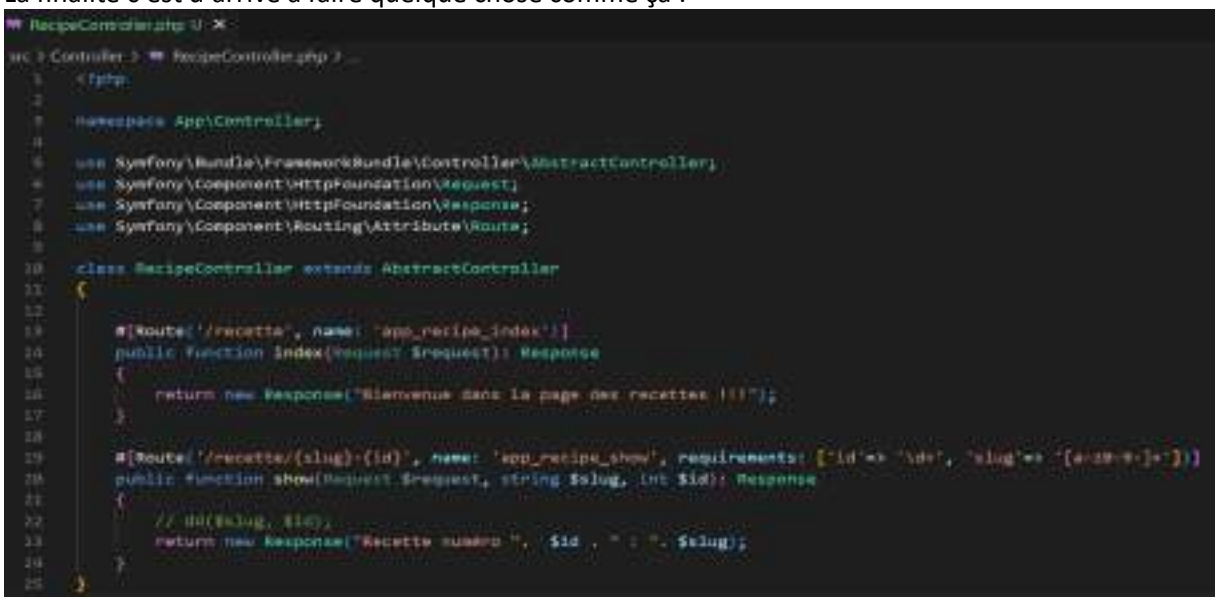
Là on verra mieux notre problème. Par défaut quand on met slug en paramètre, il ne sait pas trop ce qu'il doit prendre, on va devoir rajouter des précisions grâce aux « **requirements** ». On va lui dire que pour « **id** » je ne veux que des nombres et que pour « **slug** » je peux avoir des lettres en minuscule, des nombres et des tirets. C'est avec des expressions régulières qu'on fera ça (RegEx) :

```
#[Route('/recette/{slug}-{id}', name: 'app_recipe_show', requirements: ['id'=> '\d+', 'slug'=> '[a-z0-9-]+']])
```

Si on veut que notre « id » soit en entier, on peut remplacer dans le dd au niveau de l'id, **get** par **getInt**.
Il faut savoir qu'on aurait pu simplifier encore plus l'écriture dans le dd. En mettant en paramètres dans la méthode **show**, **une chaine de caractère $slug et un entier $id**. Attention, il faut que ce soit exactement les mêmes noms que dans votre chemin. Et ensuite les utiliser comme variable.

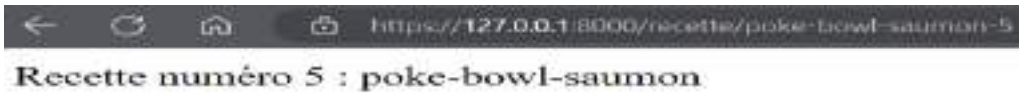
```
#[Route('/recette/{slug}-{id}', name: 'app_recipe_show', requirements: ['id'=> '\d+', 'slug'=> '[a-z0-9-]+']])
public function show(Request $request, string $slug, int $id): Response
{
    // dd($request->attributes->get('slug') , $request->attributes->getInt('id'));
    dd($slug, $id);
    return new Response('Bienvenue dans la page '.$request->query->get('recette', 'des recettes !!!'));
}
```

Voilà on a fait un petit tour sur l'utilisation de base d'un Controller. Ce qu'il faut retenir c'est qu'avec Symfony lorsqu'on souhaite créer et gérer une nouvelle URL donc une nouvelle page, on va créer une nouvelle action/méthode dans un controller. On peut soit créer un nouveau controller, soit tous simplement rajouter une méthode/action à un controller existant.
La finalité c'est d'arrivé à faire quelque chose comme ça :



Ce qui donnera 2 pages, une pour l'index et une qui montre la recette qu'on veut.
L'index :

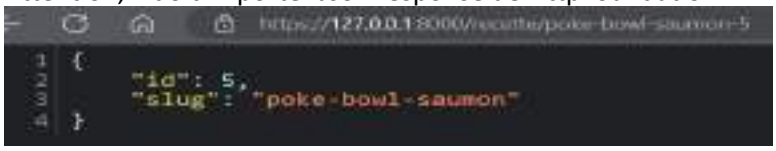
Et le show :



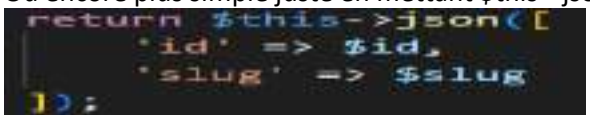
Si vous voulez faire en sorte que nos données sortent sous forme de Json dans le show, on peut utiliser l'objet **JsonResponse** comme type de retour. Il suffit de changer le return par :

```
return new JsonResponse([
    'id' => $id,
    'slug' => $slug
]);
```

Attention, il doit importer JsonResponse de HttpFoundation.



Ou encore plus simple juste en mettant $this->json :



Le format Json est très important car la plupart des Api l'utilise.
A noter que si vous voulez savoir toutes les routes de votre projet, il suffit de taper dans la console :

- **symfony console debug:router**

On y verra les routes liées au Framework, à la sécurité, au profiler et nos routes qu'on a créées dans HomeController et RecipeController. Pour l'instant tout est en ANY mais on y reviendra par après.

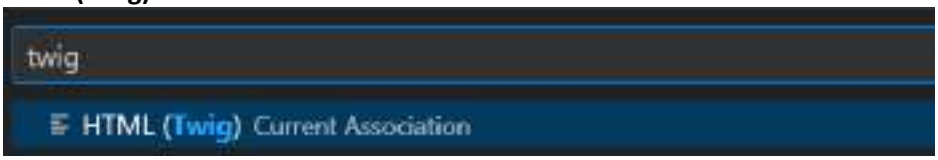
Chapitre 4 : Le moteur de Template Twig

Nous allons voir dans ce chapitre comment réaliser des pages web HTML avec le moteur de Template Twig. On a vu jusqu'à présent comment renvoyer du texte mais nous on veut maintenant renvoyer de jolies pages html. Le Template c'est la partie View dans le MVC.
Il faut savoir que Twig est un moteur de Template qui n'est pas spécifique à Symfony, c'est un langage qui peut être utiliser ailleurs. Ici on utilisera plus de PHP dans nos pages visuelles comme on le faisait dans le cours de PHP. Toute la logique se fera en Twig. A noter qu'en python avec Django vous verrez Ninja qui est son moteur de Template. Les deux se ressemblent. Pour twig, je vous recommande, de mettre en favoris le site de la documentation officiel qui est [Documentation - Twig - The flexible, fast, and secure PHP template engine (symfony.com)] (Faites CTRL + clic sur ce lien).

4.1 Notre première page Twig

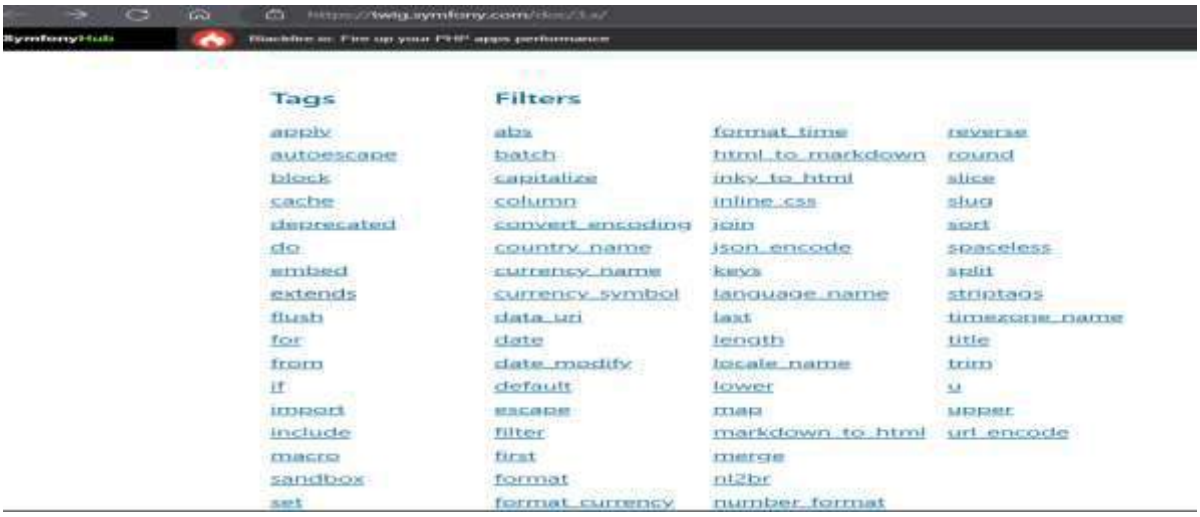
Par défaut quand on crée un controller automatiquement on a des fichiers Twig qui ont été créé. Vous pouvez aller dans la racine de votre projet, dans votre dossier « **templates »** pour les voir. Prenez le « **index.html.twig** » dans le dossier « **recipe** » :

Voilà à quoi ressemble un fichier twig. On va tous d'abord un peu améliorer le visuel en téléchargeant sur Visual Studio l'extension twig. Puis vous retournez sur votre fichier « **index.html.twig** » et en bas à droite vous cliquer sur HTML, une barre de recherche va s'ouvrir et vous cliquez sur « **Configure File Association for '.twig'** » ensuite tapez twig puis cliquez sur **HTML(Twig)** :



Maintenant tous vos fichier **html.twig** seront directement compris par VS code comme HTML(Twig). Voilà le visu que vous devriez avoir, avec ce qu'il y a dans les % en couleur etc. :

Si je prends ce fichier, on a à la première ligne quelque chose entre accolade et pourcentage. Ça doit vous rappelez l'héritage donc un extends suivi d'une autre page. En effet toutes nos pages twig hériterons de base de la page twig « base.html.twig ». Si vous allez sur le site officiel de twig dans la doc et que vous allez en bas, vous trouverez le extends :



Tags

apply
autoescape
block
cache
deprecated
do
embed
extends
flush
for
from
if
import
include
macro
sandbox
set

Filters

abs
batch
capitalize
column
convert_encoding
country_name
currency_name
currency_symbol
data_uri
date
date_modify
default
escape
filter
first
format
format_currency
format_time
html_to_markdown
inky_to_html
inline_css
join
json_encode
keys
language_name
last
length
locale_name
lower
map
markdown_to_html
merge
nl2br
number_format
reverse
round
slice
slug
sort
spaceless
split
striptags
timezone_name
title
trim
u
upper
url_encode

La doc est en anglais donc il suffira de traduire pour ceux qui ne comprennent pas l'anglais.

Allons voir ce qu'il y a maintenant dans le fichier « **base.html.twig** ». Je rappelle que ce fichier est créé directement durant la création de votre projet Symfony. Si vous y jetez un coup d'œil, on voit que ça une structure classique de fichier HTML.

```
templates > base.html.twig
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Welcome!{% endblock %}</title>
        <link rel="icon" href="data:image/svg+xml,<svg xmlns=%22http://www.w3.org/2000/svg%22 viewBox=%220 0 128 128
        {% block stylesheets %}
        {% endblock %}

        {% block javascripts %}
            {% block importmap %}{{ importmap('app') }}{% endblock %}
        {% endblock %}
    </head>
    <body>
        {% block body %}{% endblock %}
    </body>
</html>
```

Première chose qu'on peut constater, c'est qu'on a à différend endroit des parties « **block** » et chacune de ces parties est nommée. On a un « **block title »** qui changera en fonction du bloc title qu'on mettra dans nos pages (Par défaut c'est Welcome !), un « **block stylesheets** » pour la partie CSS, un « **block javascript** » pour la partie JS et un « **block body** » qui aura ce qu'on aura dans nos pages donc son contenu. Chacun de ces blocs se termine par « **endblock ».**

Retournons dans notre index. On va modifier « **index.html.twig** » dans « **templates/recipe** ». Donc on supprime tous ce qu'il y a dans le bloc body et on change de title dans le bloc title :

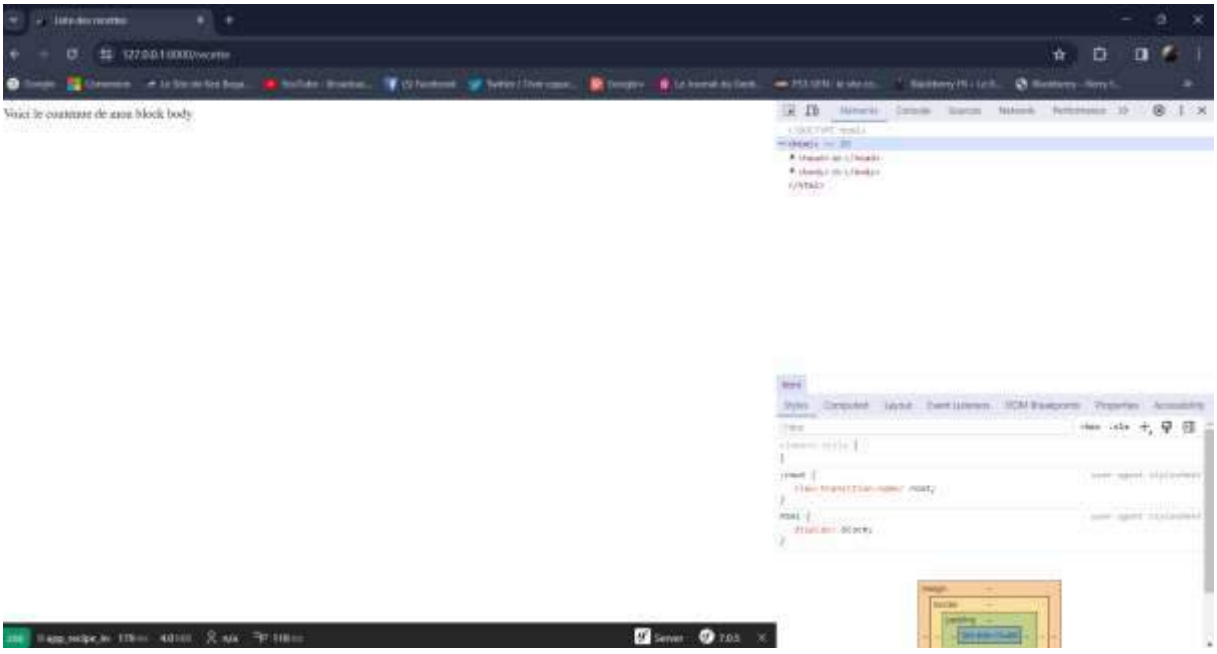
```
templates > recipe > index.html.twig
{% extends 'base.html.twig' %}

{% block title %}Liste des recettes{% endblock %}

{% block body %}

{% endblock %}
```

Tous ce que vous allez mettre dans le bloc body sera directement afficher sur votre page. Mettez du texte ou un lorem ipsum dans le bloc. Ce que j'aurai mis dans mon bloc body de « **index.html.twig** », va se situer dans mon block body de « **base.hmtl.twig** ». Maintenant il nous reste plus qu'à faire en sorte que dans notre controller, quand je suis sur la méthode index, il me renvoie vers le Template « **index.html.twig** ».

Retournons dans notre RecipeController dans la méthode/action index. Rappelez-vous, quand on allait sur notre navigateur ça nous affichait juste « **Bienvenue dans la page des recettes** ». Maintenant on va faire en sorte qu'il retourne un « **render** » qui si vous regarder le type de retour c'est bien un Objet de type Response. C'est une méthode de la classe **AbstractController** dont hérite tous nos controllers.

```php
#[Route('/recette', name: 'app_recipe_index')]
public function index(Request $request): Response
{
    return $this->render('recipe/index.html.twig');
}
```

Si vous actualisez votre navigateur dans l'URL /recette, vous verrez votre texte que vous avez introduit, vous aurez le titre de votre onglet et en plus de ça, ce sera le grand **retour de la barre de debug de Symfony**. Vous pouvez même inspecter la page et vous y trouverez ce qu'on retrouve d'habitude dans une page HTML :

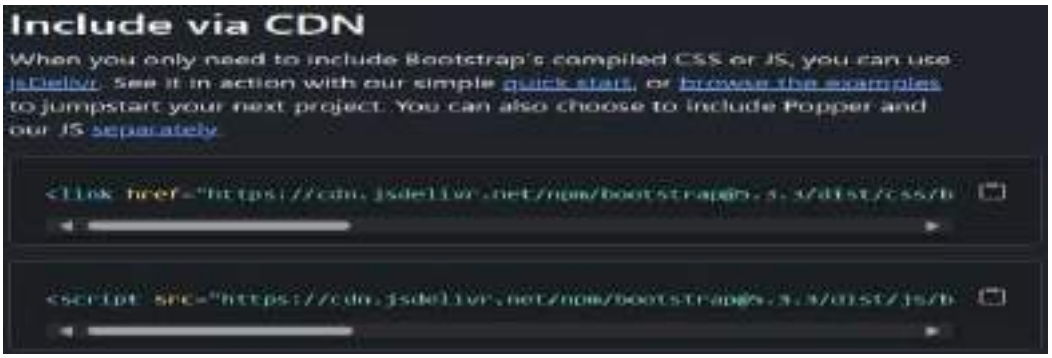


Pour changer votre couleur de background suffis d'aller changer dans assets/styles/app.css.

A noter que pour le bloc title, s'il n'y a que du texte vous pouvez l'écrire comme ça :

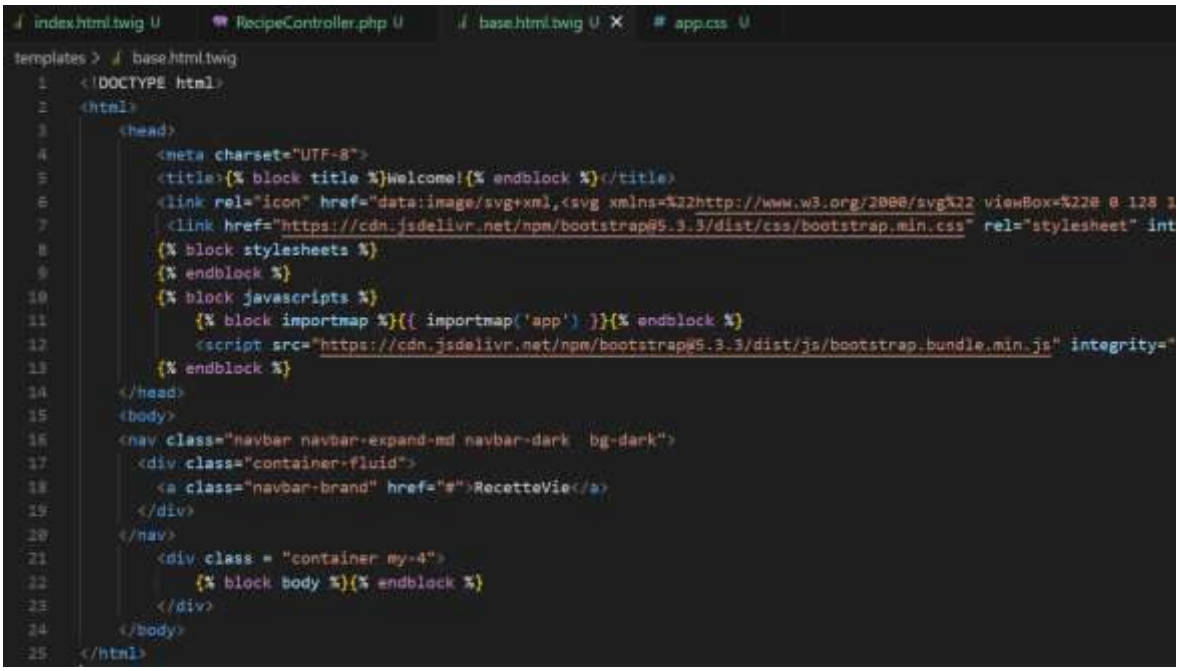
```twig
{# {% block title %}Liste des recettes{% endblock %} #}
{% block title "Liste des recettes" %}
```

On va rajouter un peu de CSS pour embellir notre page. Comme toujours avec moi ce sera via Bootstrap donc vous pouvez déjà aller sur le site www.getbootstrap.com. Récupérez le CSS et JS.



Dans « **base.html.twig** », je mets le **CSS au-dessus** du **bock style Sheets**. Je le mets au-dessus parce qu'il va être utilisé par l'ensemble des pages. Cela permettra de définir par après si on le souhaite notre propre CSS pour certaine page. Et le **JavaScript** dans le **bloc Javascript**.

On va aussi entourer le bloc body par une div pour que ce soit un peu centré et aussi lui donner un peu de marge. Et j'ai été récupéré une navbar simple sur Bootstrap :

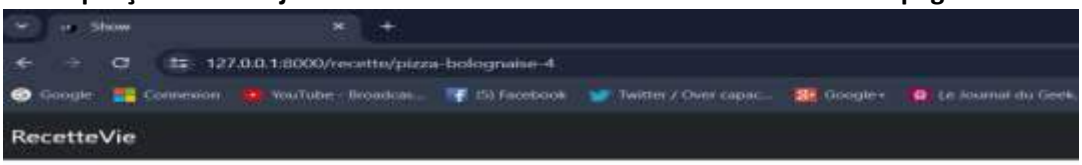


Ce qui me donne ce résultat, avec une petite navbar :

Maintenant notre page index ressemble un peu à quelques choses par contre notre page show, elle affiche toujours notre Json ou juste une phrase.

Exos

Essayez de faire la même chose pour la page show sans récupérer les données. Faites en sorte que ça m'affiche juste en title « Show » et un h1 « bienvenu dans la page SHOW ».



Pour pouvoir faire l'exercice ci-dessus, vous avez dû réfléchir à toute les étapes et liaisons qu'on a dû faire pour la page d'index.

Normalement il suffisait juste de changer le return de la méthode show et de créer un nouveau fichier de View.

4.2 Récupérer les données d'un Controller vers un Template

Maintenant on va un peu modifier notre page « **show.html.twig** » pour pouvoir justement récupérer le slug et l'id. Il faut savoir qu'avec le « **$this->render()** » on met en premier paramètre le chemin vers le Template. Mais on peut aussi rajouter d'autres paramètres dont un tableau associatif qui représentera justement notre slug et notre id.

De ce fait on envoi dans notre Template les données qui seront dans ce tableau associatif :

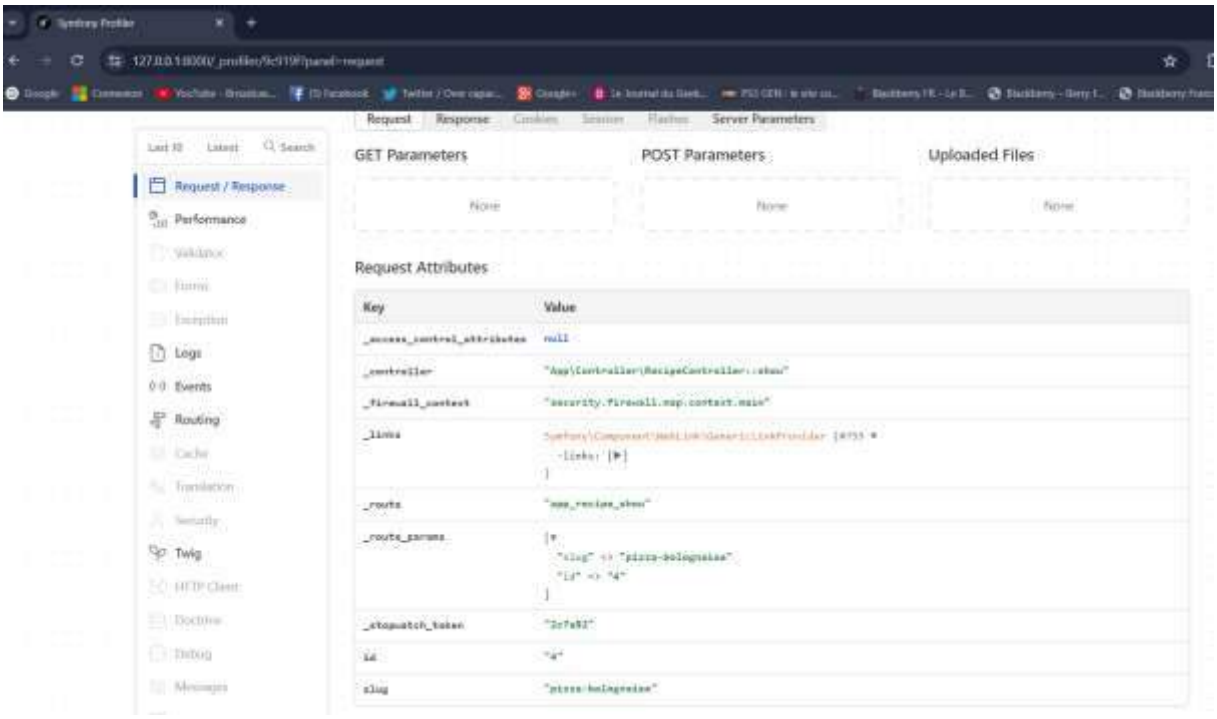
```php
#[Route('/recette/{slug}-{id}', name: 'app_recipe_show', requirements: ['id'=> '\d+', 'slug'=> '[a-z0-9-]+'])]
public function show(Request $request, string $slug, int $id): Response
{
    return $this->render('recipe/show.html.twig', [
        'slug' => $slug,
        'id' => $id
    ]);
}
```

En faisant ça, maintenant dans mon fichier « **show.html.twig** », j'aurai accès à ces variables.

Si vous actualisez vous aurez toujours le même résultat.

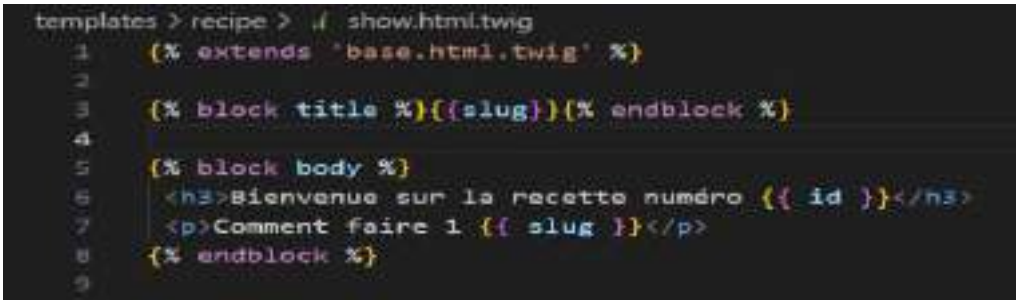
Bienvenu sur ma page SHOW

Si vous allez dans la barre de debug de Symfony en bas à gauche, et que vous cliquez sur le 200. Vous allez tomber sur cette page et on voit bien si vous descendez un peu qu'on a bien récupérer le slug et l'id que j'ai mis.



Il ne reste plus qu'à maintenant vous montrez comment on récupère en twig les données. Retournons dans notre Template « **show.html.twig** ».

En Twig {{…}} signifie qu'on veut **afficher** quelques choses. Donc si je veux afficher une variable je peux tout simplement mettre le nom de la variable dans ces doubles accolades.



Et voici ce que ça donnera :

Bienvenue sur la recette numéro 2

Comment faire 1 poulet-moambe

Au niveau du Twig, il y a une protection qui fait en sorte que quand vous mettez dans une variable du HTML au niveau du controller et que vous l'envoyez dans la vue, il n'interprètera pas ce code.

Bienvenue sur la recette numéro 2

Comment faire 1 poulet-moambe <h2>Bonjour</h2>

C’est une sécurité en plus par rapport à ce qu’on aurait pu écrire avec du PHP.

Imaginons maintenant que dans les paramètres j’ai une variable qui est un tableau. Ce qui donnerait quelques choses comme ça :

```php
return $this->render('recipe/show.html.twig', [
    'slug' => $slug,
    'id' => $id,
    'user' => [
        "firstname" => "Julien",
        "lastname" => "Dunia"
    ]
]);
```

En Twig on mettra d’abord user ensuite au lieu de faire comme en PHP « **->** » on utilisera plutôt un «**.** » pour pouvoir accéder aux éléments de « **user** » qui est le nom de ma variable qui est de type tableau. On fait donc « **user.firstname** » pour récupérer le prénom. Ce principe marche pour les tableaux mais aussi pour les objets qu’on verra plus tard. Dans le Template je ferai comme ça :

```twig
{% block body %}
 <h3>Bienvenue sur la recette numéro {{ id }}</h3>
 <h4>Comment faire 1 {{ slug }} </h4>
 <p>Recette créé par {{ user.firstname }} {{ user.lastname }}</p>
 {# <p>Recette créé par {{ user.firstname ~" "~ user.lastname }}</p> #}
{% endblock %}
```

J’ai mis deux versions. Celle en commentaire c’est la version avec la **concaténation** en **Twig** qui est le tilde « **~** ». Commentaire en Twig c’est avec **{#...#}**. Le résultat final donnera ceci :

4.3 Les filtres Twig

Vous venez de voir comment on affiche en Twig « **{{…}}** », on a vu les Blocs en Twig{% %}, on a vu comment on concatène en Twig « **~** », on a vu comment on récupère une donnée d’un tableau en Twig « . », les commentaires **{#...#}…**

Nous allons maintenant voire comment on peut appliquer des filtres de Twig. C’est très simple, la liste des filtres est disponible sur la documentation officielle de Twig. Il suffit de mettre un Pipe « **|** » suivi du nom du filtre.

Imaginons, je prends le filtre « **title** », ce filtre si je lis la documentation, ça me permet de mettre les premières lettres d'une chaîne de caractères de plusieurs mots en majuscule. On va voir ce que ça donne :

On voit bien qu'il l'a appliqué le filtre. Il y a aussi moyen de mettre tout en majuscule, ou de changer la manière dont une date est affichée, ou encore de masqué du texte avec une longueur définit etc… Il y en a plusieurs qu'on va utiliser tout au long du cours.

Il y a aussi une partie fonction sur le site de Twig, où l'on pourra utiliser par exemple min, max, range etc. Bien entendu il y a la possibilité de faire des boucles et des conditions. Nous verrons tout ça au fur et à mesure.

4.4 Passer d'une page à une autre

Vous avez vu en HTML qu'on pouvait passer d'une page à une autre avec le href. Ici aussi on l'utilisera donc vous connaissez. Ce sont juste les chemins qui seront un peu différends.

Si dans mon onglet donc dans « **base.html.twig** », je veux mettre la route qui me permet d'aller dans la page d'accueil. On utilise en fait une fonction « **url** » qui prend en paramètre le nom de la route. Rappelez-vous que « **home** » est le nom de la route de mon site racine « **/** », on le voit dans « **HomeController** » :

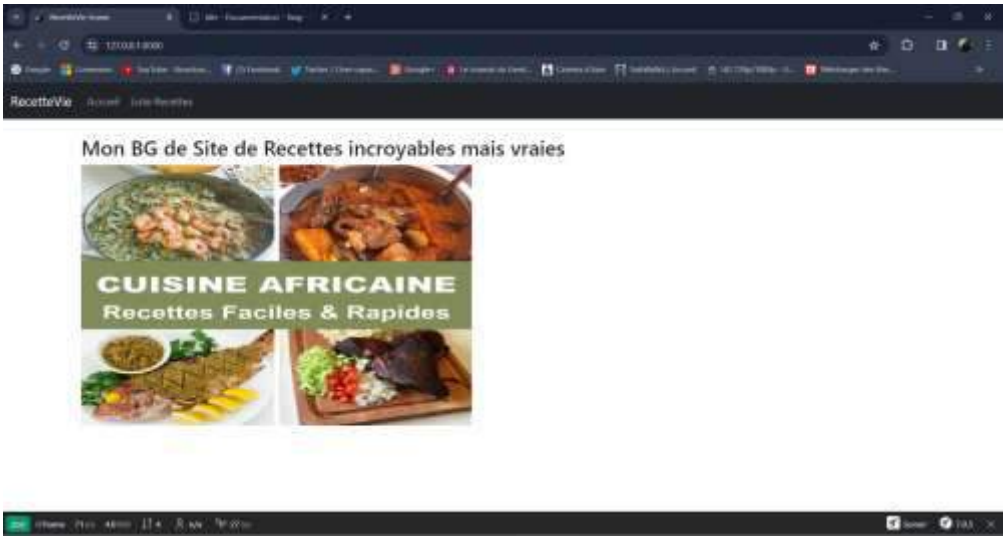
```
<a class="navbar-brand" href={{ url('home') }}>RecetteVie</a>
```

La maintenant si je retourne sur mon site, désormais quand je clique sur « **RecetteVie** », il m'amènera vers la page d'accueil.

Exos

1) **Rajouter un onglet « Accueil » qui me ramènera vers la page d'accueil (home).**
2) **Rajouter un onglet « Liste Recettes » qui montrera la page des recettes.**

3) **Essayer de faire en sorte que quand vous êtes dans la page d'accueil, on voit aussi la barre de navigation et donnez un petit titre ainsi qu'une image.**



Il faut savoir qu'on peut utiliser la fonction « **url** » comme on l'a fait jusqu'ici mais il y a aussi son équivalent en mettant comme fonction « **path** ». Les deux se valent et feront la même chose, la seule différence c'est que « **path** » ne montrera pas le chemin complet dans l'url, on le verra plus tard :

```
<li class="nav-item ">
  <a class="nav-link" href={{ path('home') }}>Accueil</a>
</li>
<li class="nav-item ">
  <a class="nav-link" href={{ url('app_recipe_index') }}>Liste Recettes</a>
</li>
```

En Twig on peut comme en PHP inclure un autre fichier dans un fichier. En PHP rappelez-vous, on utilisait « **include** » et « **require** ». Avec Twig on utilisera le « **include** » de Twig. Dans la documentation officielle, il se trouve dans la partie Tags.

On va l'implémenter dans notre code. On va tout d'abord créer un dossier dans le dossier « **templates** » qui s'appelle « **layouts** » ensuite dans ce dossier crée un autre dossier qui s'appelle « **partials** ». Et dans ce dernier dossier, on va créer un fichier « **_nav.html.twig** ». J'insiste sur le Under score « _ ». C'est pour bien dire que ce fichier est une partie de code d'où le nom partials. Donc dans ce fichier qui se trouve « **/templates/layouts/partials/_nav.html.twig** » vous y mettrez toute la balise **<nav>** qui se trouvait dans « **base.html.twig** »:

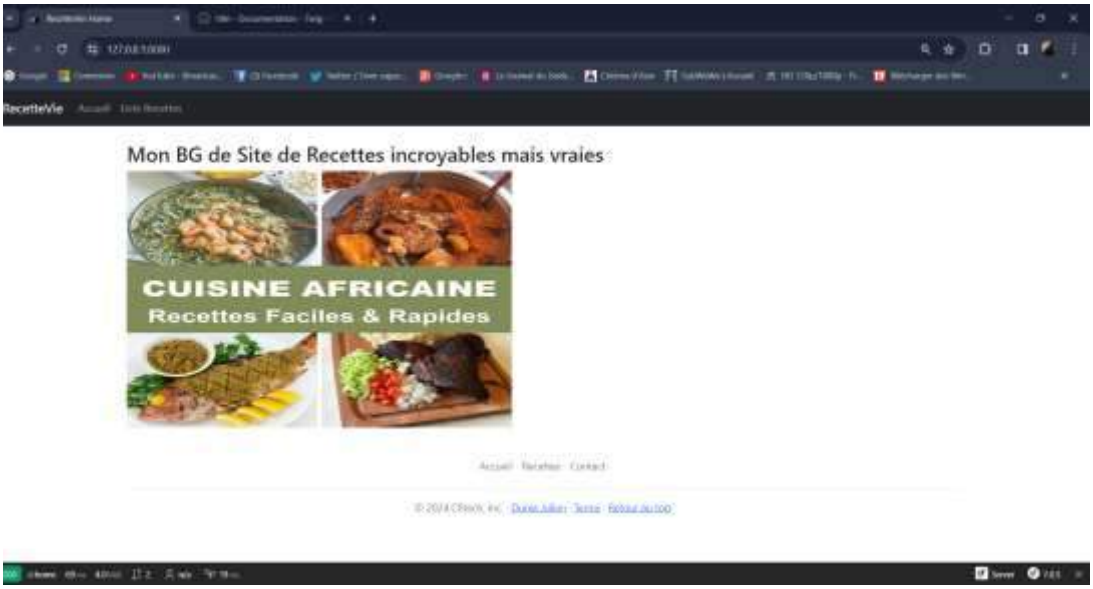
Maintenant il ne reste plus qu'à utiliser ce fichier. Donc dans « **base.html.twig** » vous mettrez :

```
    </head>
    <body>
        {{ include('/layouts/partials/_nav.html.twig')}}
        <div class = "container my-4">
            {% block body %}{% endblock %}
        </div>
    </body>
```

Si vous réactualisez votre site, il fonctionnera sans problème toujours avec la barre de navigation fonctionnelle.

Exos

4) **Essayer de me faire pareil avec un footer que vous irez prendre sur Bootstrap, celui de votre choix. Pareil que pour la barre de navigation, je veux que vous utilisiez avec le include en le mettant bien dans le dossier partials.**



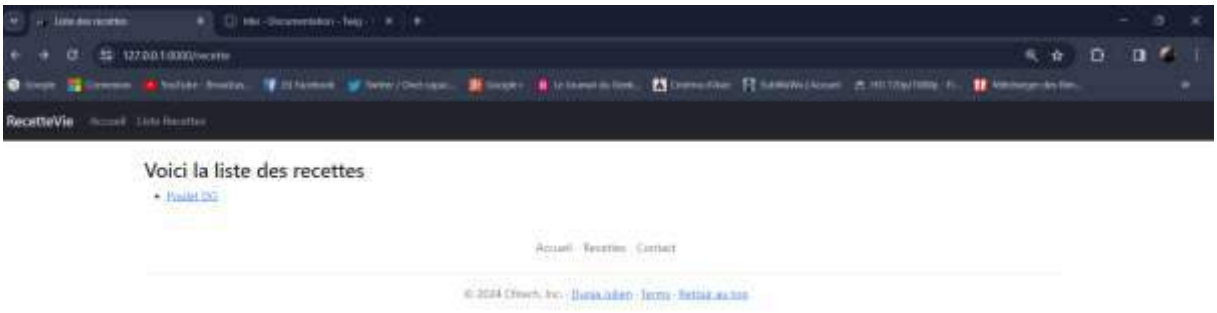
Il nous reste une seule chose à faire c'est d'afficher notre page « **/recette/uneRectte-unId** ». Mais on a un petit problème, c'est qu'il faut mettre un « **slug** » et un « **Id** » justement.

Allons dans le Template « **recipe/index.html.twig** » et ajoutons un lien. On va réutiliser la fonction « **url** » ou « **path** », les deux font pareil, pour accéder à la page recette Show. Ici on rajoutera en plus dans les paramètres, un tableau associatif contenant un id et un slug de la manière de JavaScript :

templates > recipe > index.html.twig

```
1  {% extends 'base.html.twig' %}
2  {# {% block title %}Liste des recettes{% endblock %} #}
3  {% block title "Liste des recettes" %}
4  {% block body %}
5   <h3>Voici la liste des recettes</h3>
6   <ul>
7      <li>
8          <a href={{ url('app_recipe_show', {id:4, slug:'poulet-dg'})}}>Poulet DG</a>
9      </li>
10   </ul>
11 {% endblock %}
```

Ici donc on a spécifié en second paramètre ce qu'on aura comme slug et id de l'url.



On voit maintenant qu'il me créé bien la page concernant la recette poulet DG.

Exos

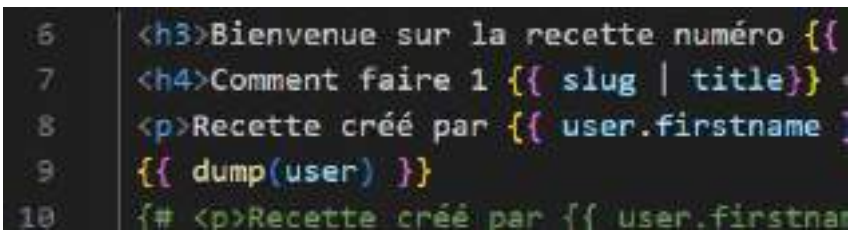
5) **Créez 2 autres liens de recette de votre choix. Que ça nous les affiche bien.**

Avec ce système de route, on peut décider si on veut de changer l'url, ça ne perturbera pas votre site web, parce que tant qu'on a le même nom de route, il acceptera.

Dorénavant nous utiliserons toujours « **path** », car il ne met pas le chemin absolu quand on fait inspecter. Il ne reprend donc pas tout le domaine :

Vous pouvez changer tous vos url en path. La fonction « **url** » sera plus utile quand il y'aura le système de mail.

Il faut aussi savoir qu'en Twig on peut utiliser le « **dump** » :



Ici on voit bien qu'il a donné ce qu'il y a dans la variable user.

On va terminer avec les onglets actives quand on est sur la page. Il faut savoir qu'on pourrait utiliser le même système qu'on a utilisé dans le cours de PHP pour rendre actif ou non un onglet. Donc en créant une variable « $**nav** » pour chacune des pages et ensuite testé si c'est bien la valeur qu'on souhaite. Mais ici avec Twig c'est beaucoup plus simple. On va utiliser « **app** », qui est une variable globale qui nous permet d'accéder à différente chose. Vous pouvez voir la liste en faisant un dump de app. Nous ici on va l'utiliser pour récupérer la route courante. Si je fais un **{{ dump(app.current_route) }}**, il me donne bien la route de la page dans laquelle je suis :

Pour rendre actif nos onglets, on va aller dans notre fichier « **_nav.html.twig** » et rajouter des ternaires :

```
<li class="nav-item ">
    <a class="nav-link {{ app.current_route == 'home' ? 'active' : '' }}" href={{ path('home') }}>Accueil</a>
</li>
<li class="nav-item ">
    <a class="nav-link {{ app.current_route == 'app_recipe_index' ? 'active' : '' }}" href={{ path('app_recipe_index') }}>Liste Recettes</a>
</li>
```

Maintenant si vous cliquez sur « **Accueil** » ou « **Liste Recettes** » l'onglet sera en surbrillance.

J'aimerai faire en sorte que si je clique sur une de mes recettes, l'onglet « **Liste Recettes** » reste en surbrillance. En gros si je clique sur ma recette « **poulet DG** », je veux que « **Liste Recettes** » reste actif. Pour faire cela on remplacera « **==** » par « **starts with** » qui en anglais veut dire commence par/avec :

```
<li class="nav-item ">
    <a class="nav-link {{ app.current_route starts with 'app_recipe_' ? 'active' : '' }}" href={{ path('app_recipe_index') }}>Liste Recettes</a>
</li>
```

Ce qui donnera bien en surbrillance « **Liste Recettes** » :

Chapitre 5 : Connexion à la base de données avec Doctrine ORM

Dans ce chapitre, on va voir comment on communique avec une base de données. Vous verrez que c'est assez simple et rapide grâce à **Doctrine ORM.** Notre objectif ici c'est de faire en sorte que les recettes de cuisine soient stockées dans une base de données et que l'on puisse les récupérer pour les afficher sous forme de liste etc.

5.1 Doctrine ORM

Un mapping objet-relationnel (en anglais **Object-Relational Mapping ou ORM**) est un type de programme informatique qui se place en interface entre un programme applicatif et une base de données relationnelle pour simuler une base de données orientée objet. Ce programme définit des correspondances entre les schémas de la base de données et les classes du programme applicatif. On pourrait le désigner par-là « comme une couche d'abstraction entre le monde objet et monde relationnel ». ***Wikipedia définition.***

Symfony utilise un **ORM** (**Objet-Relational Mapping),** il va nous permettre d'interagir avec la Base de Données avec des Objets. On aura donc des objets qui vont représenter des données qu'on a dans nos tables. Vous avez dû faire un projet en PHP qui était un site web avec PHP/HTML-CSS/MySQL ou je vous demandais de créer une base de données et des tables ainsi que dans PHP les classes représentant ces tables. Vous avez vu que c'était long et fastidieux de gérer cela avec des objets. Ici ce ne sera pas le cas, grâce à Doctrine ORM, c'est Symfony qui va se charger de tout à notre place.

Exemple de si on crée une classe/entité Recette

```
class Recette{
        private int $id;
        private string $nomRecette;
        private string $description;
        Getter/Setter …
}
```

L'ORM va faire le mapping vers le monde relationnel, il va créer une table « recettes » ou on aura ces champs, on n'aura pas besoin de gérer manuellement. On travaillera dans le monde PHP et l'ORM se charge de gérer tout ce qui est SQL.

Quand on va créer une nouvelle recette dans un controller :

```
$recette = new Recette;
$recette->setNomRecette('Couscous Agneau');
$recette->setContent('Recette de couscous agneau pour 4 personnes en 90 min');
$entityManager-> …
```

Une fois faite, on utilisera un entitymanager afin de sauvegarder notre recette. **On aura donc pas besoin d'écrire de SQL**, c'est l'ORM qui va tout gérer. Même si on change de SGBD, doctrine va se charger de générer le code SQL approprié, il va même générer un identifiant automatiquement en auto-incrémente.

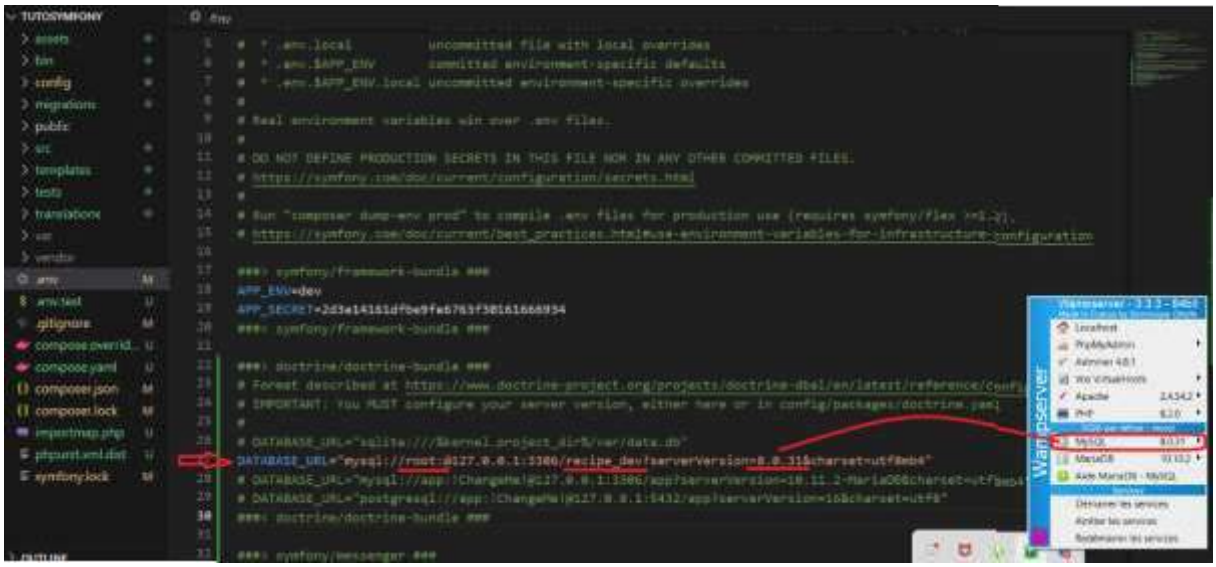
ORM fait donc le mapping du monde Objet au monde Relationnel.

Il faut savoir que Doctrine n'est pas spécifique à Symfony. Il est juste proposé par défaut. Il fonctionne avec plusieurs systèmes de gestion de base de données (MySQL, PostgreSQL, SQLite, Maria DB, …). Nous ici bien entendu on utilisera MySQL vu que c'est ce que vous avez vu avec moi.

Vous comprenez que l'orienté objet sera plus que présent dans Symfony et pourquoi il faut être à l'aise avec cela.

5.2 Création de notre base de données

Pour créer une base de données il faut d'abord modifier la variable d'environnement de votre projet. Il se trouve à la racine, dans le fichier « **.env** ». Ici on va utiliser une base de données MySQL et non PostgreSQL, donc on décommente la ligne MySQL et on commente la ligne PostgreSQL (#) :



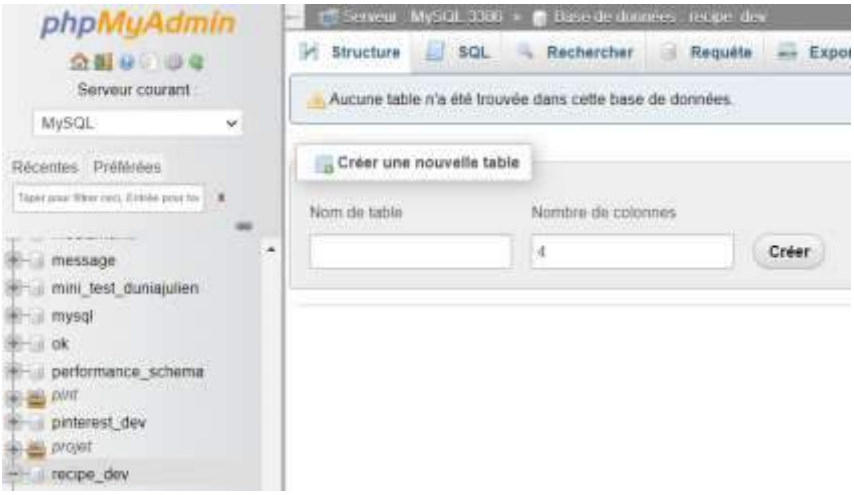
On remplace le **premier « app » par « root »** et on supprime « **!ChangeMe!** » qui représente le mdp. Ceux qui sont sur mac ou linux, vous devez mettre le mot de passe avec lequel vous vous connectez sur PHPMyAdmin(**root**). Le « **deuxième app »** on met le nom de la Base de données qui sera « **recipe_dev** ». Pour la version du server, il est préférable d'avoir la même que celle que vous voyez sur Wamp. Si ce n'est pas le cas, ce n'est pas grave, mais ça peut potentiellement créer des warnings dans votre base de données.

Il faudra bien entendu ouvrir Wamp d'abord avant de lancer la commande de création puis vérifier dans phpMyAdmin :

- **symfony console doctrine:database:create** (ca va créer la base de données)

Pour supprimer la Base de données :

- **symfony console doctrine:database:drop --force**



On peut aller voir dans PhpMyAdmin, il a bien créé notre base de données. Maintenant on va passer à la création de nos entités/tables.

5.3 Création de notre première entité

Nous allons maintenant créer notre première entité (Entity en anglais). Une entité est un objet en PHP qui va permettre de représenter chaque enregistrement/élément dans notre table en MySQL.

Pour créer une entité il faudra tout simplement aller dans le terminal et taper la commande :

- **symfony console make:entity**

Avec cette manière, il demande ensuite quel nom voulez-vous donner à votre Entité/Classe.

Il faut savoir qu'on peut aussi taper directement avec le nom de l'entité qui suit :

- **symfony console make:entity NomDeVotreEntité**

Attention une entité est représentée sous forme de classe PHP. Cette commande va nous permettre de créer une Classe avec les attributs qu'on voudra ainsi que les getters et setters de ces attributs.

Nous allons créer une classe/Entity « **Recipe** » (Recette en anglais). Faites donc l'une des deux manières de créer une entité. Il faut posera ensuite une question :

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

On n'utilisera pas Symfony UX Turbo, du coup on peut taper « **no** » MAIS ici ce qu'on voit en jaune sous crochet à la fin d'une question c'est la valeur par défaut si on ne tape rien. Donc je vais juste appuyer sur « **enter** ».

Vous pourrez déjà voir qu'il a créé 2 fichiers très important dans le dossier /src :

- « **Recipe** » qui se trouve dans le dossier **/Entity**. C'est là qu'on aura toutes nos entités/classes qui seront par la suite directement lié à nos tables en Base de Données.
- « **RecipeRepository** » qui se trouve dans le dossier **/Repository.** C'est là qu'on aura nos repositories lié à chacune de nos entités. On verra un peu plus tard que c'est là qu'on aura nos requêtes vers la base de données.

```
created: src/Entity/Recipe.php
created: src/Repository/RecipeRepository.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):
> |
```

Symfony vous dit en blanc que :

- L'entité a été généré et qu'il faut maintenant ajouter de nouveaux champs. Vous pourrez toujours ajouter d'autres champs plus tard manuellement ou en réexécutant cette commande de création de l'entité.

En vert il vous demande d'introduire un nouveau champ ou d'appuyer sur « **enter** » pour arrêter l'ajout de champs.

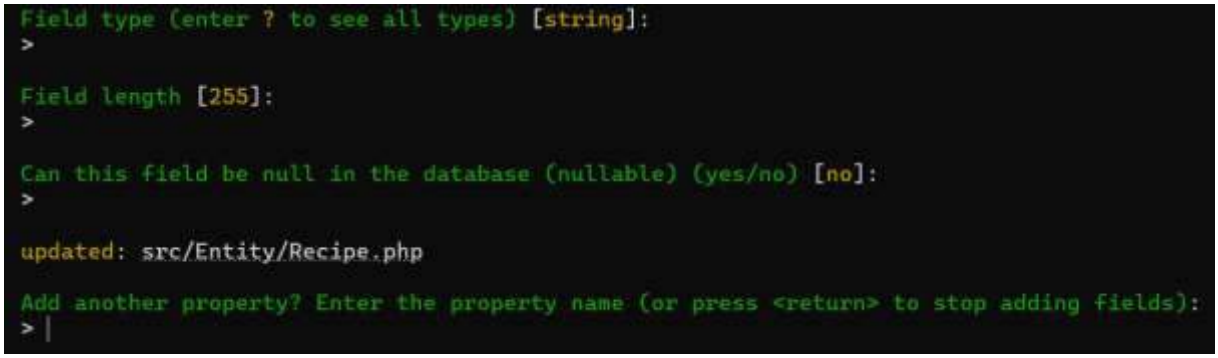
Nous bien entendu on va rajouter un premier attribut/champ qui sera « **title** » pour le titre.

Il demande ensuite de quel type, on veut que soit notre attribut.

Si vous voulez voir tous les types possibles pour une variable en symfony il suffit de taper « **?** ».

Par défaut on peut voir qu'en jaune entre crochet c'est « **[string]** ». Bien entendu on appuiera « **enter** » parce qu'un titre est une chaîne de caractère.

Il vous demandera ensuite de spécifier la taille, un peu comme on le faisait en base de données. Et par défaut c'est 255, on gardera cette taille. Il vous demande ensuite si on veut que le champ puisse avoir comme valeur null. On mettra « no » donc par défaut « enter ».



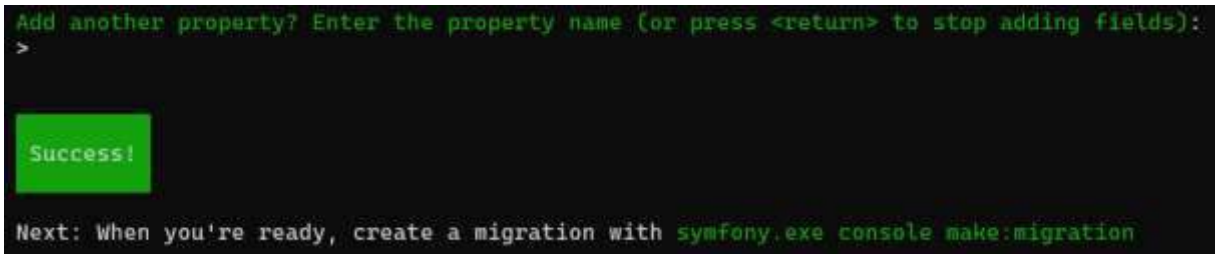
Symfony nous indique ensuite qu'il a bien mis un jour la classe « **Recipe** ». Et vous demande si vous voulez rajouter encore une fois un nouvel attribut. Si vous ne voulez pas ajouter de champ appuyer sur « **enter** » et il quittera.

Si vous avez quitté par erreur et voulez rajouter un champ à votre entité déjà existante, il suffit comme je l'ai dit plus haut, de refaire la commande de création d'une entité en donnant bien exactement le même nom que votre classe existante.

Exos

1) **Rajouter un attribut « slug » qui va permettre de générer une jolie URL**
   - **slug, string, 255, no null**
2) **Rajouter un attribut « content » qui va permettre de voir le contenu de ma recette, en gros une description :**
   - **content, text, no null**
3) **Rajouter ensuite une date de création qu'on va appeler « createdAt » :**
   - **createdAt, datetime_immutable, no null**
4) **Et pour finir, ajouter « updatedAt » qui sera la date de modification :**
   - **updatedAt, datetime_immutable, no null**

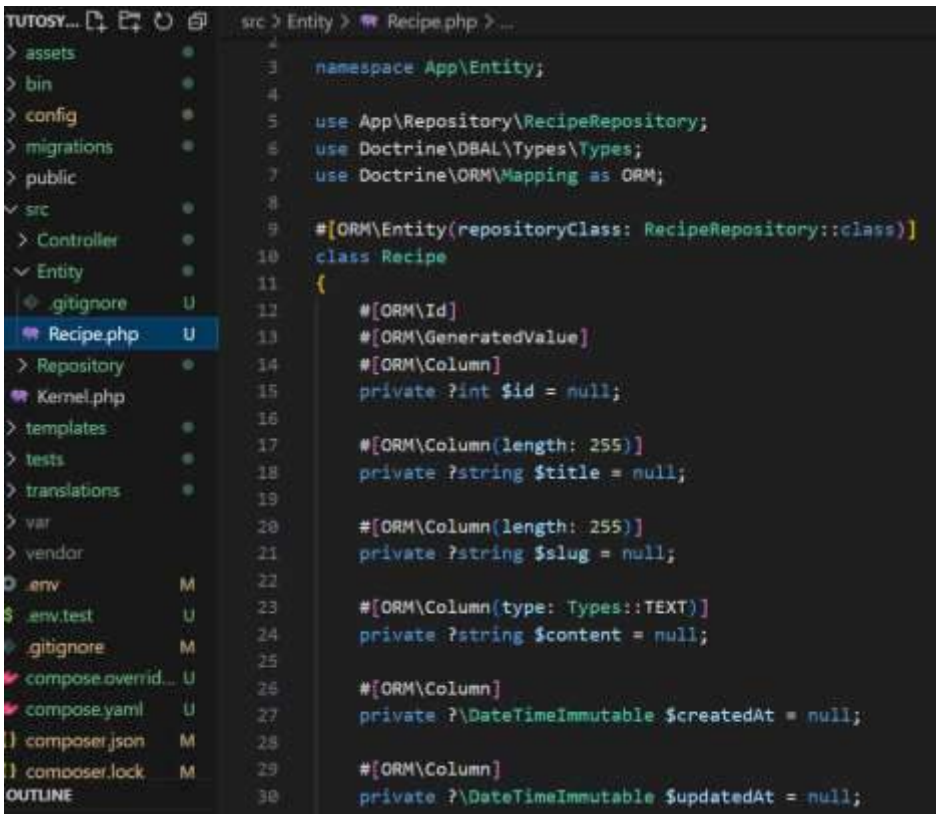
Ce qui est bien avec symfony, c'est que dès que vous terminez l'ajout des attributs, il vous donne c'est quoi la suite des étapes.



On voit qu'il me dit « **Success** » ensuite, quand tu seras prêt, créé une migration avec la commande « **symfony console make:migration** ».

Une fois finit, vérifier bien que vous avez mis les bonnes données dans votre entité et nous allons commencer la migration vers la base de données.

Il suffit d'aller dans votre racine du projet dans « **/src/entity** ». Vous y trouverez votre classe « **Recipe.php** » avec tous les attributs que vous avez inscrits via symfony :



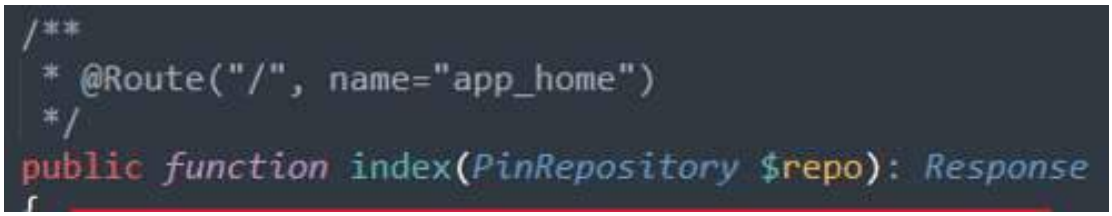
Vous voyez que c'est une classe classique de PHP, il nous a généré ça automatiquement. Si vous regardez attentivement, vous verrez qu'on a bien les getters et setters. Il n'y a pas de constructeur !!! Chose intéressante, il n'y a pas le « **setId()** », je vous avais dit qu'on ne pourra jamais changer l'identifiant d'un champ.

Il y a par contre des choses que vous ne connaissez pas. Accrochez-vous, il faut savoir qu'on peut utiliser des métadonnées.

5.3.1 Les métadonnées avec attributs vs annotations

Une métadonnée est une donnée servant à définir ou décrire une autre donnée, quel qu'en soit le support. Ces métadonnées sont sous formes d'annotations ou d'attributs dans Symfony.

Il faut savoir qu'avant PHP 8, on utilisait des annotations qui ressemblaient à ça pour les métadonnées dans Symfony :



Par exemple, si je devais donner une route dans le même projet qu'on est en train de faire en Symfony 5, qui n'utilise pas PHP 8, j'aurai fait comme ça, en utilisant les annotations grâce au **« @ »** :

```php
//Métadonnées avec les annotations avant PHP 8
/**
 * @Route('/recette', name: 'app_recipe_index')
 */
public function index2(Request $request): Response
{
    return $this->render('recipe/index.html.twig');
}
```

Ça ressemble à des commentaires mais ça n'en est pas. Il est possible que sur la multitude de fichier que vous avez dans votre projet, vous pouvez avoir des annotations. Parce que les fichiers ont été écrit avant que php 8 ne sorte bien entendu.

Depuis PHP 8, les « **annotations** » ont été "remplacé" par ce qu'on appelle des « **attributs** ». Ça reste des métadonnées mais sous une autre structure en utilisant « **#[]** » :

```php
//Metadonnées avec les attributs, depuis PHP 8
#[Route('/recette', name: 'app_recipe_index')]
public function index(Request $request): Response
{
    return $this->render('recipe/index.html.twig');
}
```

On utilise celle-ci pour notre projet actuel qui je le rappel est sous symfony 7 donc php 8.2 minimum. On appelle ces métadonnées des « **attributs** » mais pour éviter trop de confusion je parlerai d'annotations par moment sinon vous risquez de confondre avec les attributs de classe qui sont privés.

Dans notre classe « **Recipe.php** » on a donc aussi des métadonnées (attributs/annotations) qui ont été créés. Ces attributs/annotations permettent à l'ORM de comprendre comment les données seront sauvegardées dans notre base de données. On a par exemple au-dessus de la classe « **ORM\Entity(…)** » qui signifie que cette classe sera représenté dans notre base de données avec comme nom de table « **recipe** ». Par défaut il prend le nom de la classe qu'on a mais rappelé vous qu'en base de données les tables sont au pluriel parce qu'il y a plusieurs enregistrements dans une table. Nous ici on va donc **rajouter** un **attribut/annotation** pour lui donner le nom de la table qu'il aura en base de données :

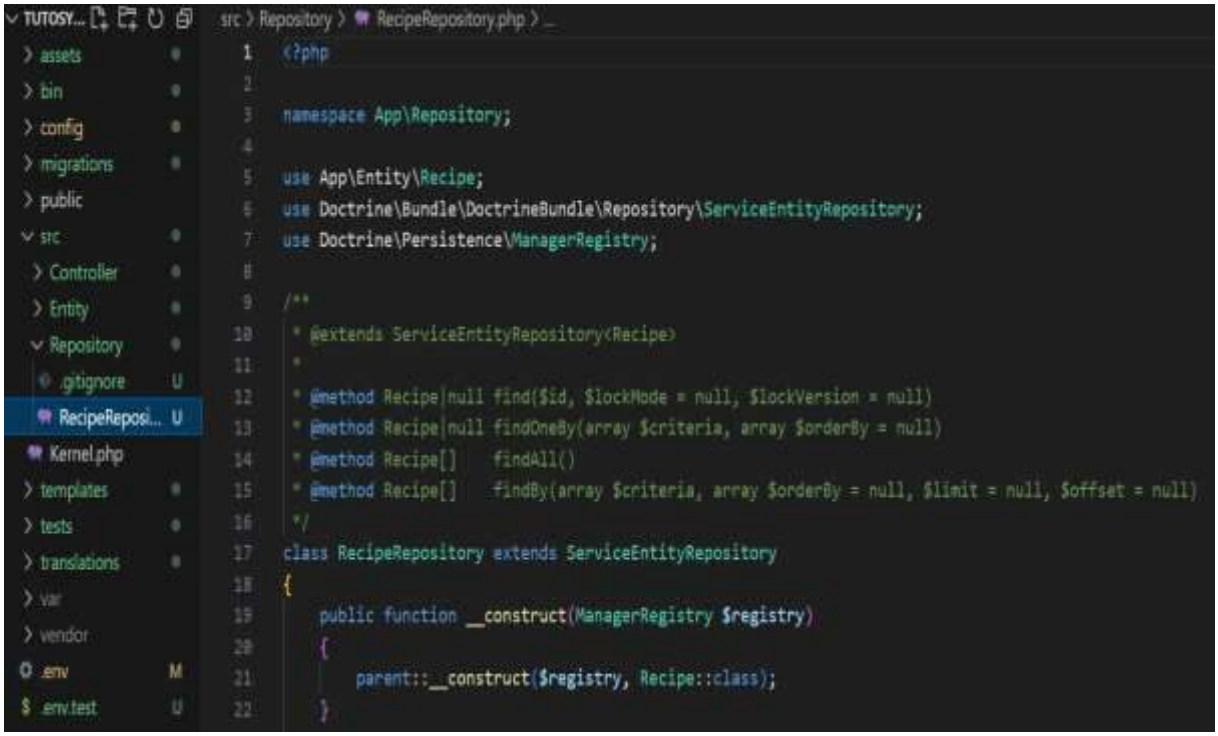
```php
#[ORM\Entity(repositoryClass: RecipeRepository::class)]
#[ORM\Table(name: "recipes")]
class Recipe
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

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

On peut aussi voir pour l'attribut de classe « **$id** » qu'il dit que ce sera un identifiant dans la base de données, que ce sera une valeur qui sera généré donc en gros auto-incrémenté et que ce sera une colonne (donc un champ). Pareil si je prends ensuite « **$title** » qui sera une colonne donc un champ en chaîne de caractère avec comme taille 255. On aurait pu rajouter par exemple que le champ doit être unique, ou des options supplémentaires etc., on en verra par après. Si vous regardez l'attribut « **$content** » vous voyez qu'il est précisé le type « **text** » et non un « **varchar** » en base de données malgré qu'on voie que c'est déclaré en « **string** » dans notre classe. Ce sont donc bien des métadonnées, donc des données en plus qui concerneront la base de données.

Pour conclure, quand on crée une entité via symfony, ce n'est rien de plus qu'une classe classique PHP avec des métadonnées. Et c'est grâce à doctrine qu'il va savoir comment il doit sauvegarder les informations en base de données.

5.3.2 Le repository

On a aussi un autre fichier qui a été créé après notre entité. Il se trouve dans « **/src/Repository** » et s'appelle « **RecipeRepository.php** ».



Donc ça, ça va être une classe qui va permettre de récupérer des enregistrements. C'est dans cette classe qu'on va créer des méthodes qui nous permettrons d'extraire des données depuis la base de données et elle va contenir par défaut quelques méthodes qui vont permettre de récupérer un ou plusieurs enregistrements. Donc à chaque fois on va avoir un repository qui va être associé à une entité. C'est donc ce repository qui sera chargé de récupérer les recettes. On pourra rajouter des méthodes spécifiques plus tard, pour l'instant on va se contenter de ce qu'il nous propose qui est déjà plus que suffisant.

5.3.3 Configuration de createdAt et updatedAt

Avant de faire la migration vers la base de données, on va faire en sorte que quand on créera une recette, la date actuelle sera envoyée automatiquement en base de données et que quand on modifiera une recette, il enverra aussi la date actuelle de modification.

Dans la classe « **Recipe** » qui se situe dans le dossier **src/Entity** on va ajouter **HasLifecycleCallbacks** :

```php
#[ORM\Entity(repositoryClass: RecipeRepository::class)]
#[ORM\Table(name: "recipes")]
#[ORM\HasLifecycleCallbacks]
class Recipe
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
```

« **HasLifecycleCallbacks** » viens de ORM et ça va permettre de pouvoir utiliser **prePersit** et **preUpdate**, on va écouter les différents évènements du cycle de vie.

On va ensuite crée une méthode qu'on rajoutera dans notre classe Recipe. Elle pourrait avoir n'importe quel nom, mais généralement les **updatedAt** et **createdAt** sont des timestamps (des évènements liés au temps). On l'appellera « **updateTimestamps** » :

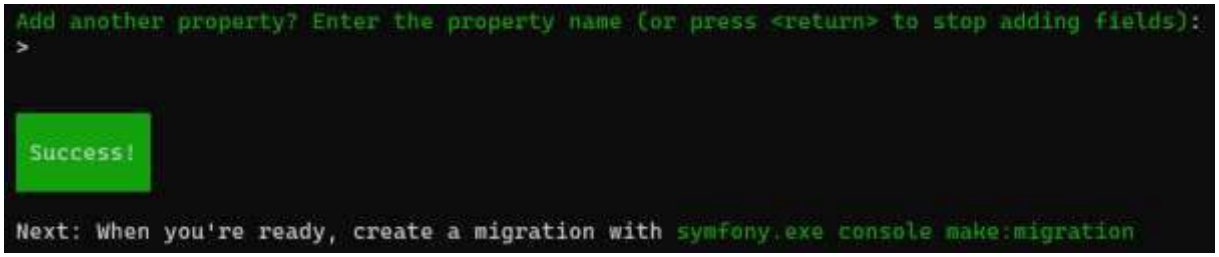
```php
36      #[ORM\PrePersist]
37      #[ORM\PreUpdate]
38      public function updateTimestamps()
39      {
40          if ($this->getCreatedAt() === null) {
41              $this->setCreatedAt(new \DateTimeImmutable);
42          }
43          $this->setUpdatedAt(new \DateTimeImmutable);
44      }
```

A chaque fois avant de persister les données en base de données ou à chaque fois avant qu'on fasse une mise à jour sur une donnée de nos recettes, il passera par cette méthode. C'est grâce au **PrePersist** et **PreUpdate**. La méthode vérifie si le createdAt est vide, si c'est le cas, il lui donne la date et heure actuelle. Et quoi qu'il arrive dès qu'on rentre dans cette méthode donc qu'on crée une nouvelle recette ou qu'on modifie une recette, il mettra la date et heure actuelle dans updatedAt.

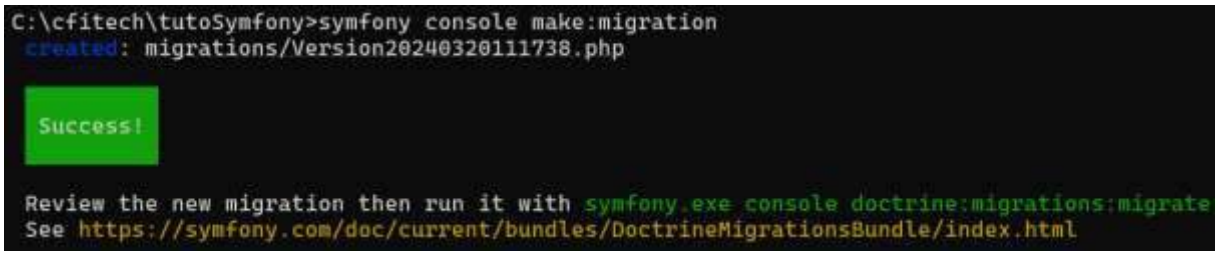
Une fois que vous aurez rajouter cela, on pourra cette fois ci faire la migration.

5.4 Création de notre table en base de données

On va maintenant mettre notre entité en base de données. Rappelez-vous, je vous ai dit que Symfony vous donne les étapes à suivre après avoir créé votre entité.



On a été modifié le nom de la table dans l'entité avec l'attribut « **ORM\table** ». Maintenant on va préparer la migration en tapant la commande indiquée « **make:migration** » :

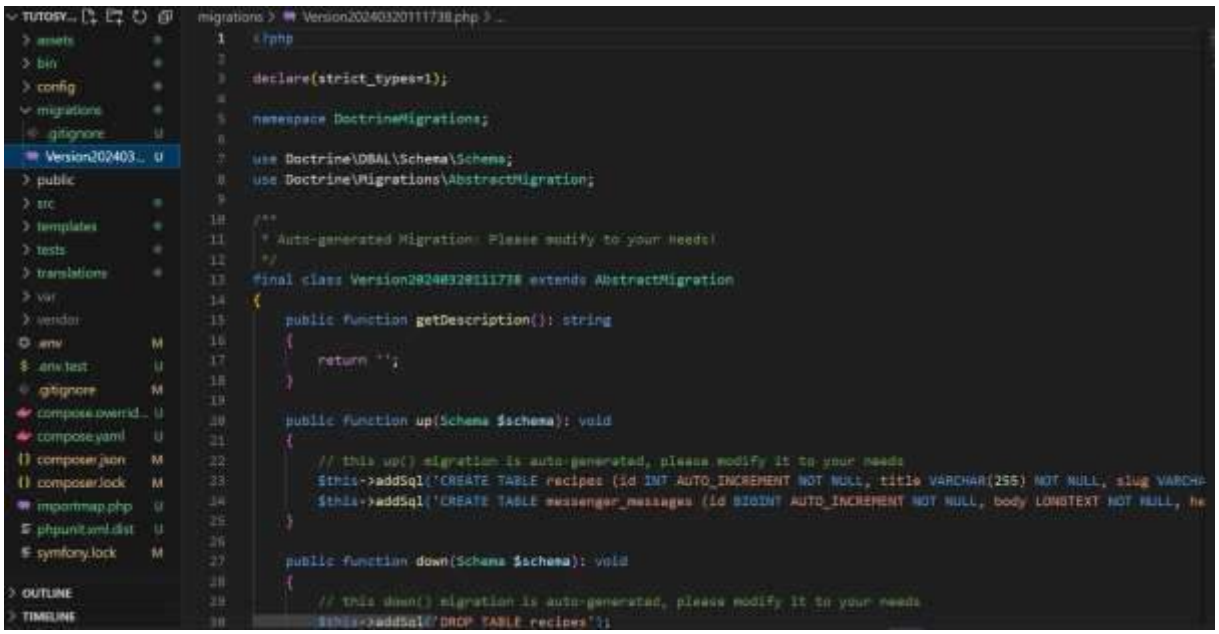


Il nous dit que ça a bien marché. Il nous a créé un fichier dans le dossier « **migrations** » à la racine du projet. Derrière ce nom de fichier barbare « **Version20…** » se cache une structure bien précise. Si je prends le nom du fichier de mon image et que je le décortique ça donne :

- **Version 2024 03 20 11 17 38**

Traduction, certains l'auront déjà compris, c'est le moment où on a fait la migration donc le 20 Mars 2024 à 11h17 et 38 sec. L'heure est en GMT 0, donc si vous rajoutez une 1 ou 2 heures, vous aurez l'heure exacte.

Symfony vous donne une nouvelle fois ce qu'il faut faire après cette commande. Mais avant de faire ça, allons dans notre fichier de migrations.



Et ce fichier php est une classe, « **final** » ça veut dire qu'elle ne peut pas être parent.

On a une méthode « **getDescription()** » qui retourne une chaîne de caractère. Cette méthode va servir à décrire ce qu'on a fait ou ce qu'on a rajouté qu'on va migrer (exemple : migration de ma classe Voiture en base de données, ou encore, rajout d'un champ dans la table Voiture qui est déjà en base de données ou suppression d'un champ etc.).

Prenez la ligne 17, on va mettre un commentaire :

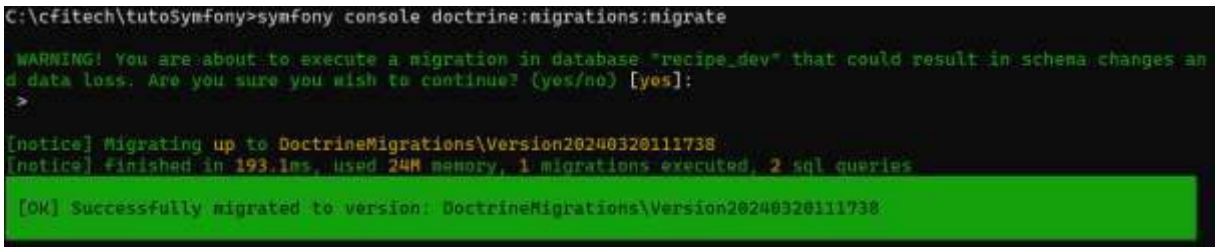
- **Migration de notre entité Recipe en base de données**

Dans la méthode « **up()** » on peut voir que c'est là qu'il a généré la requête SQL qui va justement créer notre table « **recipes** » en base de données. Il crée aussi une autre table qu'on verra beaucoup plus tard. On ne touchera rien dans cette méthode.

La méthode « **down()** » c'est dans le cas où on veut retourner en arrière donc il va supprimer notre table. En fait c'est retourner à l'état avant qu'on ait fait la migration. On ne touchera pas cette méthode.

Il nous reste maintenant plus qu'à continuer ce que nous propose Symfony donc exécuter et faire cette migration en base de données.

Pour ce faire on fait la commande proposée par symfony « **doctrine:migrations:migrate** », il posera une question (Vous êtes sur le point d'exécuter une migration dans la base de données « **recipe_dev** » qui pourrait entraîner des changements de schéma/table et perte de données. Etes-vous sur de vouloir continuer ?) :



On voit que c'est réussi. Nous pouvons allez voir dans notre base de données. Vous pouvez aller dans PhpMyAdmin et vous verrez bien votre table « **recipes** » qui a été rajouté ainsi que ses champs.

On peut voir aussi qu'on a une table « **doctrine_migration_version** », elle va retenir l'ensemble des migrations qui ont été exécutées. Notre premier enregistrement dans cette table correspond au fichier qu'on a dans notre projet dans le dossier « **migrations** ». Ce qui veut dire que si je refais une migration sans avoir modifié quoi que ce soit dans mon projet, il ne fera rien, parce qu'il verra qu'on a déjà fait cette migration.

Si on se dit qu'on a oublié un champ et qu'on veut qu'il soit en base de données. Il faudra bien entendu passer d'abord par la modification de l'entité. Je répète ne faites rien manuellement, on le fera via ligne de commande.

Exos

5) **On va créer un champ qui servira à déterminer combien de minutes il faut pour préparer une recette. Ajoutez un attribut « duration » dans votre entité « Recipe », il sera de type entier et attention il sera nullable.**
6) **Faites ensuite la migration en suivant et reprenant bien les étapes !!!**

Maintenant on va essayer de récupérer des recettes en base de données.

5.5 Récupération et affichage des recettes en base de données

Dans un premier temps nous allons tricher un peu et créer nos recettes directement dans PhpMyAdmin pour ensuite pouvoir tester de les récupérer et afficher sur notre site.

On va créer une première recette en base de données : « **Poulet Mayo** » qui est une recette congolaise, à tester les amis 😉. Je rappel on ne met pas d'id, c'est en auto incrémente. Pour le **title** c'est « **Poulet Mayo** », le **slug** c'est « **poulet-mayo** », le **content** je vais vous envoyer, les dates vous ne touchez pas et pour **duration** vous mettrez « **60** ».

id	title	slug	content	created_at (DC2Type:datetime_immutable)	updated_at (DC2Type:datetime_immutable)	duration
1	Poulet Mayo	poulet-mayo	Du poulet et de la mayonnaise, la base pour moi. L...	2024-03-29 09:50:39	2024-03-29 09:50:39	60

Exos

7) **Créez 2 autres recettes différentes, récupérer les sur internet mais gardez la même structure.**

Maintenant revenons sur notre site. J'aimerai bien récupérer la liste des recettes qui sont donc en base de données.

Maintenant retournons dans notre code, dans notre controller « **RecipeController** ».

On va avoir besoin de notre « **Repository** », c'est là qu'on aura toutes les requêtes concernant une Entité (Chapitre 5.3.2). Quand on crée une entité, il nous crée son repository automatiquement. On y reviendra un moment donné plus en détails.

On va rajouter à notre méthode « **index** » en paramètre « **RecipeRepository $repository** ». Vérifiez bien qu'il l'a bien importé dans votre controller. En faisant ça symfony va automatiquement initialiser ce repository et faire en sorte que ça fonctionne.

Maintenant si on veut récupérer les recettes, on peut utiliser une des méthodes fournies de base.

```
/**
 * @extends ServiceEntityRepository<Recipe>
 *
 * @method Recipe|null find($id, $lockMode = null, $lockVersion = null)
 * @method Recipe|null findOneBy(array $criteria, array $orderBy = null)
 * @method Recipe[]    findAll()
 * @method Recipe[]    findBy(array $criteria, array $orderBy = null, $limit = null, $offset = null)
 */
class RecipeRepository extends ServiceEntityRepository
{
```

Nous on va utiliser dans la méthode « **index** » le « **findAll()** » qui récupèrera tous les enregistrements et qui les renverra sous forme de tableau de Recettes. On va mettre ce tableau dans une variable et on l'enverra dans notre Template. Vous pouvez tenter de faire un dd pour voir ce qu'il y a dedans.

```
#[Route('/recette', name: 'app_recipe_index')]
public function index(Request $request, RecipeRepository $repository): Respo
{
    $recipes = $repository->findAll();
    dd($recipes);
    return $this->render('recipe/index.html.twig');
}
```

Vous retrouverez vos recettes qui sont en base de données.

Exos

8) **Essayez d'envoyer les recettes (le titre, le slug et l'id) sur notre page web. Réfléchissez à un moyen de le faire en faisant l'essentiel dans votre méthode index. Adaptez votre Template pour récupérer les données et les afficher comme on l'avait fait.**

Je vais vous montrer la bonne pratique pour récupérer et afficher les données dans nos Templates.

Pour envoyer à nos vues, on pourra retourner dans notre « **RecipController** » et modifier la méthode index.

N'oubliez pas de supprimer ou mettre en commentaire ce que vous avez fait pour l'exercice ci-dessus qui je rappel n'est pas la bonne pratique.

On va envoyer nos recettes à nos vues sous forme de tableau de Recette récupérer en BD.

```php
#[Route('/recette', name: 'app_recipe_index')]
public function index(Request $request, RecipeRepository $repository): Response
{
    $recipes = $repository->findAll();
    return $this->render('recipe/index.html.twig', [
        'recipes' => $recipes
    ] );
}
```

Il faudra ensuite aller dans les vues, dans « **/template/recipe/index.html.twig** ».

Et là on va voir quelque chose de nouveau en twig. C'est l'utilisation des boucles. Vous pouvez retrouver la boucle « **for** » qu'on va utiliser dans la documentation officielle de Twig. Ce « for » ressemble plus à un « **foreach** » en php. N'oubliez pas de supprimer les li en trop, on va en garder qu'un seul qui sera entouré par notre boucle. Ne pas oublier non plus le « **endfor** » qui ferme.

```twig
<h3>Voici la liste des Recettes</h3>
<ul>
   {% for recipe in recipes %}
      <li>
         <a href={{ path('app_recipe_show', {id : recipe.getId(), slug : recipe.getSlug()})}}>{{ recipe.getTitle() }}</a>
      </li>
   {% endfor %}
</ul>
```

Si vous retourner sur votre site, vous verrez qu'il continuera a fonctionné en récupérant bien les données en base de données. Et ici bien entendu c'est dynamique, si je rajoute une recette, elle apparaitra la bien sûr, pareil si je supprime etc.

On peut un peu améliorer notre code et faire plus tot ceci. J'ai rajouté une else avec un for. Il est pratique en twig de montrer un message spécial quand il n'y a pas encore de données en base de données. Il suffit de mettre un « else » qui je rappel signifie sinon et donner ce qu'on fait en cas de liste vide. Ici j'ai rajouté un message.

```twig
<h3>Voici la liste des Recettes</h3>
<ul>
   {% for recipe in recipes %}
      <li>
         <a href={{ path('app_recipe_show', {id : recipe.id, slug : recipe.slug})}}>{{ recipe.title }}</a>
      </li>
   {% else %}
      <p>Pas encore de recette</p>
   {% endfor %}
</ul>
```

On peut aussi remarquer qu'ici dans cette version ci-dessus, j'ai récupéré les propriétés qui sont privé normalement. Mais dans le cadre de Twig, on peut directement mettre les propriétés privées. Quand je mets directement, il va d'abord regarder s'il y a un getter, vu qu'il en trouve un pour chacune de mes propriétés, c'est comme s'il faisait les getters.

Cette syntaxe est facile à lire au niveau des templates.

5.5.1 Affichage d'une recette dans show (cRud)

Maintenant on va toucher un peu à la page de « **Show** ». On va rajouter un « **RecipeRepository** » en paramètre de cette méthode. Et rappelez-vous, on peut récupérer en base de données une recette complète en donnant juste son id. On va utiliser les méthodes fournies. On pourrait utiliser la méthode « **findOneBy** », je l'ai mise en commentaire, mais en gros elle va chercher la recette en fonction du slug reçu en paramètre. Nous on va plutôt utiliser la méthode **« find($id)** » qui trouve la recherche en fonction de l'ID qui je le rappel est unique.

```php
#[Route('/recette/{slug}-{id}', name: 'app_recipe_show', requirements: ['id'=> '\d+', 'slug'=> '[a-z0-9-]+'])]
public function show(Request $request, string $slug, int $id, RecipeRepository $repository): Response
{
    //ca nous permet de recuperer une recette à partir du slug donné en paramètre
    // $recipe1 = $repository->findOneBy(['slug' => $slug]);
    // dd($recipe1);
    //ca nous recupere une recette correspondant à l'id
    $recipe = $repository->find($id);
    return $this->render('recipe/show.html.twig', [
        'slug' => $slug,
        'id' => $id,
        'user' => [
            "firstname" => "Julien",
            "lastname" => "Dunia"
        ]
    ]);
}
```

Tentez de faire des « **dd** » pour bien vérifier ce qu'il y a dedans. Pour l'instant je ne l'envoi pas dans mon Template. Il faut d'abord qu'on voit ce qu'il se passe dans le Profiler Symfony (debug barre).

1) **Cet onglet nous permet d'expliquer comment cette requête a été généré, quel controller a été appelé. On voit aussi le Status (200 c'est que c'est Ok).**
2) **C'est la performance**
3) **C'est la mémoire qui est utilisé**
4) **C'est l'historique des changements de page ou d'action passé (une espèce de log)**
5) **C'est pour savoir si on est connecté ou non. On verra ça quand on introduira les utilisateurs**
6) **C'est les templates qui sont utilisé**
7) **C'est la partie base de données qui ne s'affiche que lorsqu'on a une requête en BD**
8) **Amusez-vous à naviguer**

On va cliquer sur la partie base de données. Vous y verrez plusieurs informations dont la requête SQL qui est fait pour obtenir justement notre recette.

Il y a un tas d'information dans le profiler de Symfony, je vous invite à explorer de vous-mêmes.

Exos

9) **Essayez d'adapter votre code pour que notre page « show » s'affiche bien avec les données qu'on à récupérer grâce au find(id).**
10) **Essayez quand vous êtes dans une recette de changer l'url manuellement en donnant un autre nom de recette. Que remarquez-vous ?**

Il y a donc un petit problème avec le slug au niveau de l'url. Comment pouvons-nous y remédier ?

C'est très simple, il suffira de vérifier qu'on a bien le slug qui est égale, si ce n'est pas le cas on fera une redirection vers une route directement.

```php
#[Route('/recette/{slug}-{id}', name: 'app_recipe_show', requirements: ['id'=> '\d+', 'slug'=> '[a-z0-9-]+'])]
public function show(Request $request, string $slug, int $id, RecipeRepository $repository): Response
{
    $recipe = $repository->find($id);
    if ($recipe->getSlug() !== $slug){
        return $this->redirectToRoute('app_recipe_show', ['slug' => $recipe->getSlug(), 'id' => $recipe->getId()]);
    }
    return $this->render('recipe/show.html.twig', [
        'recipe' => $recipe
    ]);
}
```

Ça nous permettra de nous assurer qu'on a toujours une seule et même url pour chacune des recettes.

Exos

11) **Changez un peu votre Template show pour qu'il affiche de cette manière ci (pour la description utiliser le filtre twig « | nl2br » :**

Toutes les recettes s'affichent normalement. On récupère bien les données de notre base de données et on les affiche de la manière qu'on souhaite.

5.5.2 Création d'une requête personnalisée

Maintenant on va essayer de faire une requête personnalisée qui permettra d'afficher par exemple toutes les recettes qui prennent moins d'une heure ou moins de 30 minutes etc.

Pour faire cela on va allez au niveau du repository et créer notre propre requête SQL qui va récupérer les données qu'on souhaite. En SQL on ferait la requête :

- **« SELECT * FROM recette WHERE duration < valeur ORDER BY duration ».**

Nous ici on va faire cette requête en utilisant un « **QueryBuilder** » dans le cadre de Doctrine. C'est quelque chose qui permet de construire une requête SQL. L'avantage c'est que c'est lui qui va traduire ça et générer la requête en fonction du système de gestion de Base de Données que vous aurez choisi (MySQL, PostgreSQL, SQLite etc.).

Ouvrez votre fichier « **RecipeRepository** » et créons cette méthode :

```php
/**
 * @return Recipe[]
 */
public function findRecipeDurationLowerThan(int $duration) : array{
    return $this->createQueryBuilder('r')
    ->where('r.duration <= :duration')
    ->orderBy('r.duration', 'ASC')
    ->setParameter('duration', $duration)
    ->getQuery()
    ->getResult();
}
```

Cette méthode recevra à chaque fois une durée en paramètre, qui sera en minutes. Elle retourne un tableau de Recette. Ce n'est pas obligatoire mais on rajoute un peu de PHP doc (annotation) pour lui expliquer que la fonction retourne certes un tableau mais bien précis qui sera de type Recette.

Dans le « **return** » on construit la requête en donnant comme alias « **r** » pour la table « **recipes** ». Nous avons par après la condition avec le « **where** » puis le « **order by** » que vous connaissez. Le « **setParameter** » sert à savoir comment il va renseigner les paramètres, donc ici on dit qu'on a une champ duration qui va obtenir la valeur qu'on aura mis quand on appellera cette méthode. C'est un peu comme une clé valeur. Le « **getQuery** » sert à générer la requête, qui va nous donner un objet Query qui est un objet de Doctrine. Et finalement on fait un « **getResult** » qui va nous permettre d'obtenir le résultat de cette requête sous forme de tableau.

Maintenant qu'on a créé cette méthode, on peut la tester dans notre RecipeController.

```php
//Metadonnées avec les attributs, depuis PHP 8
#[Route('/recette', name: 'app_recipe_index')]
public function index(Request $request, RecipeRepository $repository): Response
{
    //ca nous crée une liste de recette qu'il recupère en bd
    // $recipes = $repository->findAll();
    //ca nous crée une liste de recettes qu'il recupère en bd dont la durée est de moins de 60 minutes
    $recipes = $repository->findRecipeDurationLowerThan(60);
    return $this->render('recipe/index.html.twig', [
        'recipes' => $recipes
    ]);
}
```

Retournez sur votre site et rafraichissez, vous verrez bien qu'il aura pris toutes les recettes qui prennent moins d'une heure comprise. Si vous allez dans votre Profiler Symfony (barre d'outils symfony), vous verrez dans la partie doctrine, qu'il a bien fait notre requête qu'on a créé dans le Repository.

Voila donc comment on peut construire des requêtes particulières pour vos futurs site.

Si vous voulez limitez vos résultats, imaginons vous avez 1000 recettes, il suffira de mettre une « **LIMIT** » comme en SQL sauf que ça se fait dans notre repository. Si vous voulez que 10 résultats affichez :

```php
    ->orderBy('r.duration', 'ASC')
    ->setMaxResults(10)
    ->setParameter('duration', $duration)
    ->getQuery()
```

5.6 Persister nos données avec l'EntityManager et Repository (CrUD)

Je reviens un peu sur ce qu'est un **Repository** pour que vous compreniez mieux **l'EntityManager**.

Un Repository Doctrine est un objet dont la responsabilité est de récupérer une collection d'objets. Les repositories ont accès à deux objets principalement

- **EntityManager** : Permet de manipuler nos entités ;
- **QueryBuilder** : Un constructeur de requêtes qui permet de créer des requêtes personnalisées, que nous avons vu précédemment.

Chaque entité dispose d'un repository par défaut accessible dans tous les contrôleurs de nos applications. D'ailleurs vous avez pu voir qu'on a un dossier repository avec RecipeRepository.

L'EntityManager est l'objet permettant d'effectuer les opérations liées à l'altération des données, à savoir les **requêtes** de type **INSERT**, **UPDATE** et **DELETE**. Au niveau de votre application, même si vous manipulez des entités, qui correspondent aux données présentes en base, il ne suffit pas de modifier la valeur d'une propriété pour que Doctrine fasse la synchronisation en base de données. Vous devez gérer ces opérations avec **l'EntityManager**.

Afin de persister nos recettes dans la Base de Données, on va utiliser **l'ENTITYMANAGER**, qui va nous permettre de gérer nos entités.

5.6.1 Création d'une recette

Pour l'instant nous avons triché et créé des recettes en passant directement par la base de données. Bien entendu on ne fera plus jamais de cette manière à partir de maintenant.

Pour pouvoir créer une recette dans le code on va retourner dans RecipeController. On va avoir besoin d'une classe qui s'appelle « **EntityManagerInterface** » qui provient de Doctrine ORM. On va le rajouter en paramètre et on l'appellera « **$em** » pour EntityManager. Cette classe est responsable de mémoriser toutes les entités que vous avez dans votre application. Lorsqu'un repository récupère les entités, automatiquement elles sont sauvegardées dans l'EntityManager. Donc actuellement, elle a mes 3 recettes qui sont en Base de données. Nous on va rajouter une 4ème recette. Nous allons dans un premier temps passer par la page d'index, mais vous comprendrez que plus tard on aura une page de création de recette via un formulaire. Dans un premier temps rajoutez l'EntityManagerInterface en paramètre :

```php
#[Route('/recette', name: 'app_recipe_index')]
public function index(Request $request, RecipeRepository $repository, EntityManagerInterface $em): Response
{
```

```php
$recipe = new Recipe();
$recipe->setTitle('Omelette');
$recipe->setSlug('omelette');
$recipe->setContent('Prenez des oeufs, cassez les et ensuite battez les en rajoutant du sel.');
$recipe->setDuration(6);
$recipe->setCreatedAt(new DateTimeImmutable());
$recipe->setUpdatedAt(new DateTimeImmutable());
```

Pour la création de notre objet, étant donné que Symfony ne crée pas de constructeur, vous l'aurez fait comme ça. De l'orientée objet pur et dur. Donc on crée notre objet vide ensuite on le modifie grâce aux setters. Il faut savoir qu'on peut simplifier un peu l'écriture.

Certains l'ont remarqué que dans notre entité/classe Recipe, les setters retourne quelques choses. C'est ce qu'on appelle des « **fluent setter** », qui est un Type de setter qui renvoie sa classe et permet donc d'être appelé dans une chaîne de responsabilité. En gros dès qu'on modifie le titre par exemple via le « setTitle() », bah ça me retourne l'objet courant directement après modification, donc je peux directement enchaîner les différentes méthodes les unes sur les autres.

Et il faut savoir aussi que quand on crée un objet on n'est pas obligé de mettre de parenthèse.

```php
//Version avec l'utilisation de fluent setter
$recipe = new Recipe;
$recipe->setTitle('Omelette au Fromage')
    ->setSlug('omelette-au-fromage')
    ->setContent('Prenez des oeufs, cassez les et ensuite battez les en rajoutant du sel.')
    ->setDuration(6)
    ->setCreatedAt(new DateTimeImmutable())
    ->setUpdatedAt(new DateTimeImmutable());
```

Une fois que votre objet est créé, c'est là qu'au niveau du **EntityManager**, on va lui dire, je veux que tu **persistes** l'objet (**persist**) équivalent du add sur GitHub. En gros de persister les données, de sauvegarder la présence de cet objet. Mais une fois que c'est persisté, donc que l'objet est présent, il faudra faire une dernière étape qui est le **flush**, qui est un peu comme le push après avoir fait toutes les étapes sur GitHub. Le **flush** va envoyer, les données persistées, directement en base de données.

- **$em->persist($recipe);**

Il va permettre d'informer notre EntityManager qu'on veut persister notre objet.

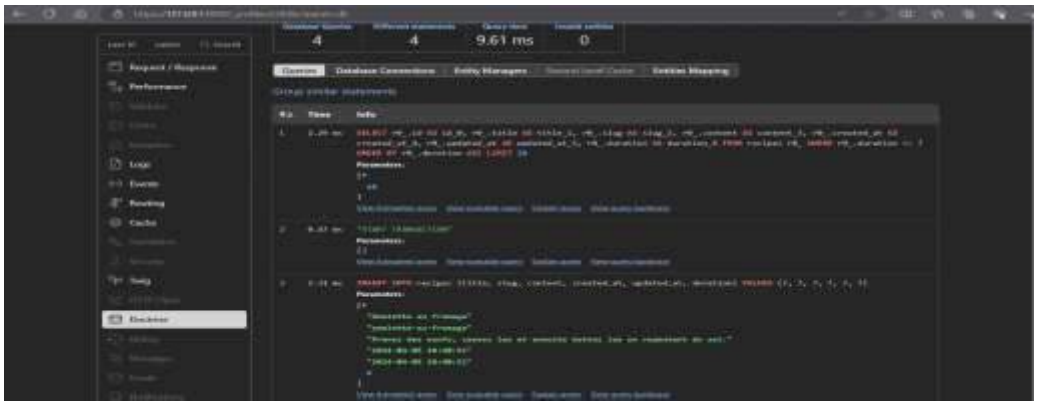
- **$em->flush();**

Le flush c'est comme si on disait go pour persister les données en Base de Données. Il exécute la requête.

A partir d'ici **ATTENTION**, à chaque fois que vous allez recharger votre page de la liste des recettes, il créera une recette en base de données. Donc prenez bien vos précautions et faites qu'**UN SEUL REFRESH**. Ensuite vous irez mettre en commentaire le persist et flush.

Une fois fait, si vous cliquez sur le profiler symfony, dans la partie doctrine, vous pourrez voir qu'il a fait 4 requêtes.

En cliquant dessus vous verrez bien qu'il a fait l'**INSERT INTO** :



Dernière étape vérifiez bien qu'en base de données dans phpMyAdmin, il a bien rajouté une 4ème recette.

id	title	slug	content	created_at	updated_at	duration
1	Poulet Mayo	poulet-mayo	Du poulet et de la mayonnaise, la base pour moi. L...	2024-03-29 09:50:39	2024-03-29 09:50:39	60
2	Pondu (Saka-saka)	pondu-saka-saka	Pour ceux et celles qui ne savent pas ce qu'est le ...	2024-03-29 10:41:35	2024-03-29 10:41:35	130
3	Poulet DG	poulet-dg	Aujourd'hui, je vous présente une recette tout dro...	2024-03-29 10:41:35	2024-03-29 10:41:35	55
4	Omelette au Fromage	omelette-au-fromage	Prenez des oeufs, cassez les et ensuite battez les ...	2024-04-05 10:40:51	2024-04-05 10:40:51	6

Voilà tous ce qui se passera lors de nos créations de recette.

5.6.2 Modification d'une recette

On va maintenant essayer de modifier une recette, voir le fonctionnement. N'oubliez pas que tous ce qu'on fait ici c'est de voir comment ça marche. On fera le CRUD via des formulaires bien entendu plus tard.

Retournons dans notre « **RecipeController** », on va mettre en commentaire la création. Et on va tous simplement modifier le titre de la deuxième recette.

```php
#[Route('/recette', name: 'app_recipe_index')]
public function index(Request $request, RecipeRepository $repository, EntityManagerInterface $em): Response
{
    //RECUPERATION DES RECETTES EN BD
    //ca nous crée une liste de recette qu'il recupère en bd
    $recipes = $repository->findAll();

    //ca nous crée une liste de recettes qu'il recupère en bd dont la durée est de moins de 60 minutes compris
    // $recipes = $repository->findRecipeDurationLowerThan(60);


    //MODIFICATION
    //Je modifie le nom de la 2eme recette
    $recipes[1]->setTitle('Saka Madesu');
    $em->flush();

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

N'oubliez pas le « **flush** » car sans lui ça ne modifiera rien en base de données.

Normalement si vous retournez sur votre site, vous verrez pour la 2ème recette que le nom aura changé, il sera différent du slug. **Si vous avez une erreur**, il se peut qu'il faille supprimer un composant en tapant : composer remove symfony/ux-turbo. Et là ça marchera.

On peut voir qu'il y a eu 4 requêtes dans la barre de Symfony. Si vous cliquez dessus vous pourrez voir qu'il a bien fait la modification :

Si vous refaite refresh sur votre site, il ne remodifiera plus bien entendu, parce qu'il verra que ça a déjà le titre « **Saka Madesu** ». Donc il ne fera qu'une seule requête.

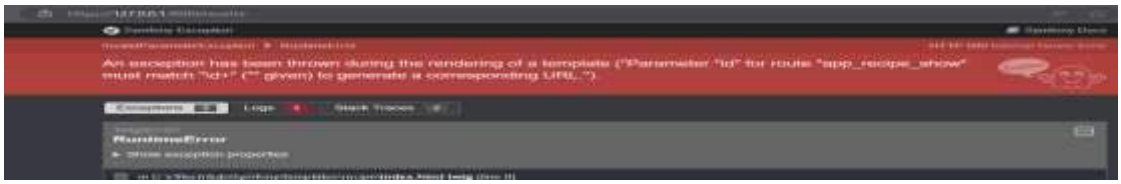
Je vais remettre mon ancien titre pour qu'il soit en raccord avec mon slug et content en base de données. Je refais une modification du coup avec l'ancien titre.

5.6.3 Suppression d'une recette

Maintenant il nous reste à voir comment on peut supprimer une recette. On va toujours le faire dans le « **RecipController** » dans la méthode index. Donc mettez en commentaire la modification. Si je veux supprimer ma dernière recette qui est dans ma base de données la 5ème.

```
//SUPPRESSION
//Vais supprimer la 5ème recette.
$em->remove($recipes[4]);
$em->flush();
```

Si vous faites un refresh dans votre page, il donnera une erreur mais c'est normal.



Suffis de mettre en commentaire la suppression dans votre code et vous aurez votre liste sans votre dernière recette qui aura été supprimé.

Nous avons donc vu grosso modo comment fonctionne le CRUD en Symfony. On y reviendra avec les formulaires bien entendu car l'utilisateur lui ne verra pas une seule ligne de code.

Il faut savoir que pour un peu décharger nos méthodes en paramètres, on pourrait enlever le fameux « **RecipeRepository** » qu'on reçoit. Et n'utiliser que le « **EntityManagerInterface** ».

```
#[Route('/recette', name: 'app_recipe_index')]
public function index(Request $request, EntityManagerInterface $em): Response
{
    //RECUPERATION DES RECETTES EN BD
    //ca nous crée une liste de recette qu'il recupere en bd
    // $recipes = $repository->findAll();
    //c'est la meme chose
    $recipes = $em->getRepository(Recipe::class)->findAll();
```

Exos

12) **Rajoutez 2 recettes via le code.**
13) **Modifiez le titre, le slug et le contenu de l'une des deux recettes que vous avez créées.**
14) **Supprimez la recette que vous n'avez pas modifier.**
15) **(Dans le projet Magasin Voiture ou sportvie) Créez deux voitures, dans le code toujours, mais je veux que la création se fait dans une nouvelle page, donc un onglet création.**
16) **(Dans le projet Magasin Voiture ou sportvie) Modifiez un élément différent de vos deux enregistrements en cliquant sur un onglet donc une page modification.**
17) **(Dans le projet Magasin Voiture ou sportvie) Supprimez un des deux enregistrements que vous aurez créé en cliquant sur un onglet supprimer.**

Nous avons fini ce chapitre, le prochain ce sera la découverte et l'utilisation des formulaires. Je vous recommande de bien refaire les étapes pour bien les comprendre.

Chapitre 6 : Les formulaires

Dans ce chapitre nous allons voir comment utiliser et créer des formulaires avec Symfony. Vous savez tous faire des formulaires. En php on aurait dû faire le formulaire en HTML puis se connecter à la base de données, récupérer les données en base de données, etc. c'est un peu long. Ici ils sont directement gérés par Symfony, beaucoup de chose seront automatisés.

6.1 Création d'un formulaire

Pour créer un formulaire rien de plus simple, il suffit de taper la commande :

- symfony console make:form

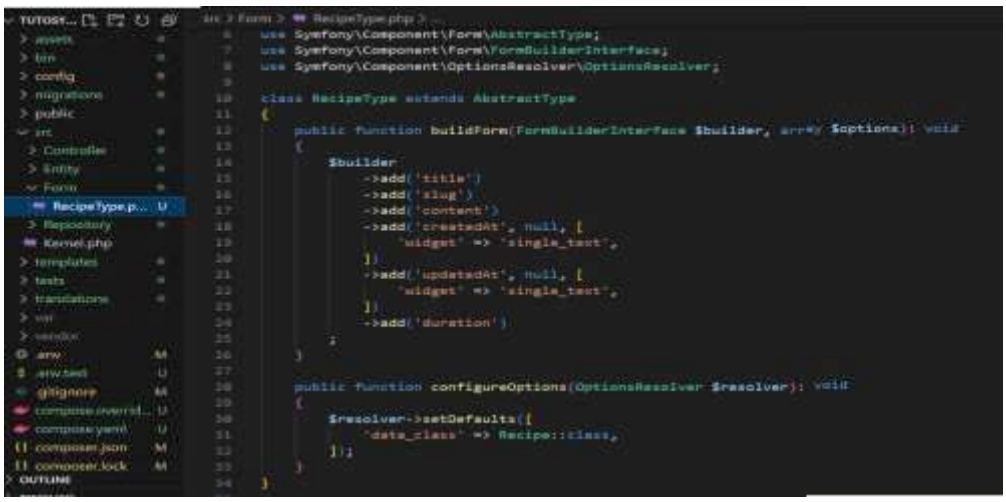
Et comme pour l'entité ou le controller il vous demandera de taper le nom du formulaire par la suite.

Et bien entendu on pourra aussi taper directement à coter de la commande le nom du formulaire. Donc je ferai plutôt :

- **symfony console make:form Recipe**

Il vous demandera ensuite à quelle entité voulez-vous qu'on lie ce formulaire. Bien entendu ce sera avec notre entité « **Recipe** ». **ATTENTION** notez bien avec la première lettre en majuscule **Recipe** comme on a créé notre classe. Il créera le fichier et rajoutera au nom du formulaire « **Type** » ce qui donnera **« RecipeType.php ».**

Ce fichier ce se retrouvera dans « **src/Form** » qui est le dossier ou on trouvera tous nos formulaires.



Et vous voyez qu'automatiquement dans la méthode « **buildForm** », qui veut littéralement dire construire le formulaire, il a rempli les champs en fonction des champs qui sont dans mon entité.

En bas il y a la méthode « **configureOptions** » qu'on ne touchera pas forcément. Mais elle sert à définir une **dataClass** qui est une option qui permet d'expliquer qu'à l'intérieur de ce formulaire on va traiter un objet de type **Recipe**.

On va maintenant créer nos pages pour gérer les formulaires.

6.2 Le formulaire de modification

On va commencer par la page du formulaire pour modifier les recettes. Retournons dans notre « **RecipeController** », rappelez-vous, **une page = une méthode**. On va donc devoir créer une nouvelle méthode « **edit** ».

```php
#[Route(path : '/recette/{id}/edit', name : 'app_recipe_edit')]
public function edit(Recipe $recipe) {
    dd($recipe);
}
```

Il faut savoir que Symfony va comprendre que si dans l'url/chemin de ma route je mets **{id},** ça correspondra à l'id de la recette que je donne en paramètres. Donc si dans mon navigateur je veux par exemple ma 4ème recette, je peux mettre la route « **/recette/4/edit** ».

J'obtiens bien la 4ème recette. Il a bien compris que le paramètre **{id}** c'est pour retrouver la recette. Il fait comme si on avait fait un « **find(id)** » avec le repository. Ça nous permet de gagner une ligne.

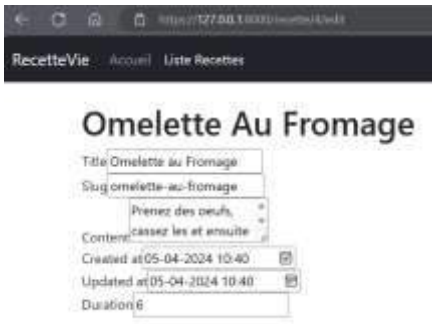
On va continuer à compléter la méthode pour qu'on renvoi cette recette vers une page « **templates/recipe/edit.html.twig** » qu'il faudra aller créer.

```php
#[Route(path: '/recette/{id}/edit', name: 'app_recipe_edit')]
public function edit(Recipe $recipe): Response
{
    // dd($recipe);
    //cette methode prend en premier paramètre le formulaire que l'on souhaite utiliser
    //en second paramètre elle prend les données
    $form = $this->createForm(RecipeType::class, $recipe);
    return $this->render('recipe/edit.html.twig', [
        'recipe' => $recipe,
        'monForm' => $form
    ]);
}
```

Et dans le fichier « **edit.html.twig** », c'est là que se trouvera notre visu du formulaire, basez-vous sur « **show.html.twig** ». On va afficher ensuite le formulaire de cette manière.

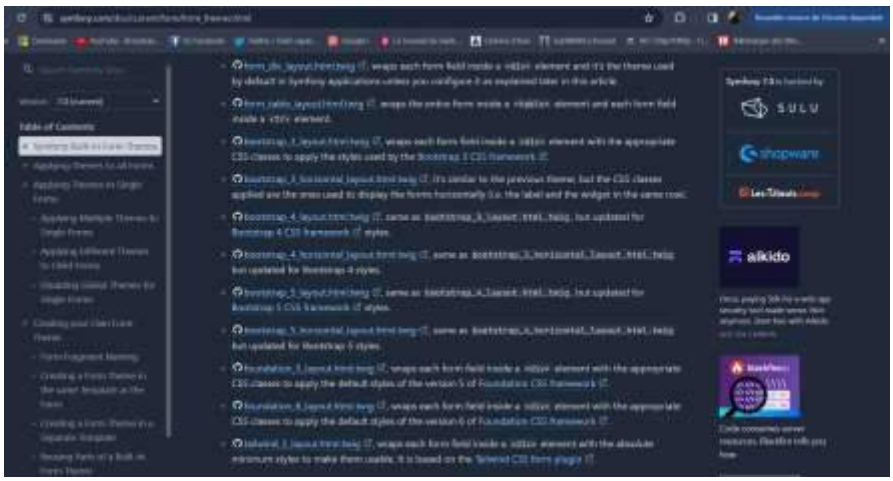
```twig
templates > recipe > edit.html.twig
1   {% extends 'base.html.twig' %}
2
3   {% block title %}#{{ recipe.id ~" "~ recipe.slug }}{% endblock %}
4
5   {% block body %}
6   <h1>{{ recipe.title | title }}</h1>
7
8   {#Attention, la form est la fonction fourni par symfony, et monForm c'est le nom qui vient de la méthode edit dans RecipController #}
9   {{ form(monForm) }}
10
11  {% endblock %}
```

Si vous retournez sur votre site et taper dans l'url « **/recette/2/edit** » par exemple, vous allez voir que ça va fonctionner et vous donnez ceci :



On voit que c'est magique et directement on a bien un formulaire déjà créer avec les données directement récupérer en base de données. Il nous a donc bien généré dynamiquement un formulaire complet. Si vous inspectez, vous pourrez voir qu'on a bien nos différents champs mais aussi un champ caché qui permet de valider que le formulaire provient bien de cette page-là, donc c'est une sécurité mise automatiquement.

C'est un formulaire brut, celui qu'est proposé sans faire de modification. Nous on va un peu le modifier pour le rendre plus correct. On va utiliser un thème. Ces thèmes vous pouvez les retrouvez en tapant « **form themes symfony**» sur google, vous tomberez sur le site de Symfony. Là il y a une liste de thème prédéfini et déjà installé dans tous vos projets SF. Je vais vous montrez comment les utiliser.



On va choisir ceux de Bootstrap 5 et on voit qu'il y en a deux. Je vais vous montrer le résultat de chacun, à vous de choisir lequel vous préférez. Pour notre projet « **TutoSymfony** » ici je vous impose de choisir entre ces deux-là. Mais rien ne vous empêche de tester les autres thèmes dans vos autres projets.

Copiez «bootstrap_5_layout.html.twig**»** ou «bootstrap_5_horizontal_layout.html.twig» et ensuite allez dans **config/packages/twig.yaml** :

Et si j'utilise l'autre :

Personnellement je garderais l'horizontal mais à vous de choisir.

On peut allez maintenant un peu modifier l'agencement du formulaire tout d'abord dans « **edit.html.twig** » par rapport au positionnement. Si j'enlève le mode horizontal de mes formulaires, on peut mettre en display flex plusieurs champs si on veut.

Le « **form_start** » c'est bien entendu pour dire ou le formulaire débute et le « **form_end** » où il va se terminer. A savoir que quand on met le « **form_end** », il continuera l'affichage du formulaire. C'est bon à savoir pour ceux qui veulent gérer l'affichage à leur manière.

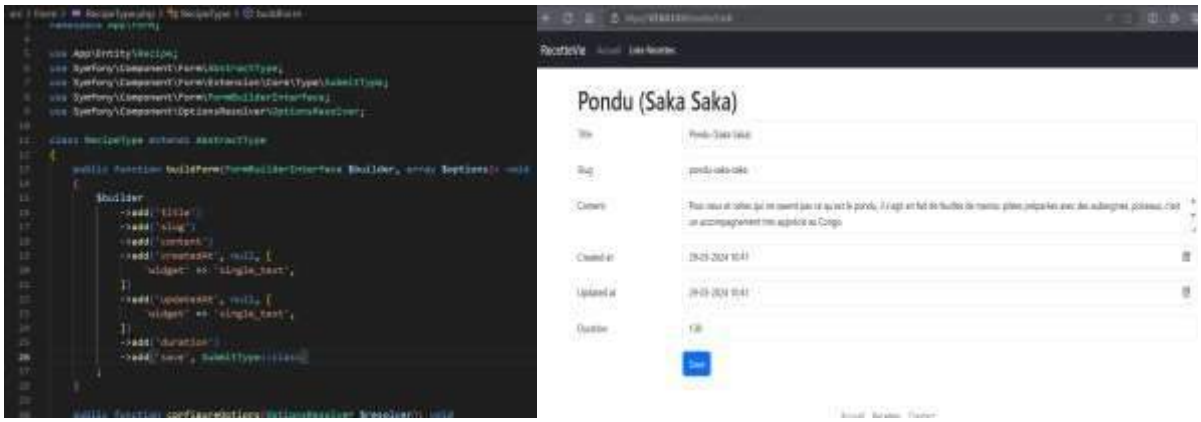
Dans la documentation officielle vous avez de nombreuses explications et possibilités :

https://www.symfony.com/doc/current/form/form_themes.html

Je rappelle qu'on est dans le « **edit** » donc maintenant je vais essayer d'avoir un bouton qui permettra de valider le changement de ma recette.

Il y a deux solutions pour les boutons. On peut le rajouter dans le fichier « **RecipeType** » ou dans le Template directement.

On va voir comment faire dans le **RecipeType**.



On voit bien qu'il a rajouté le bouton « **save** ». A savoir que si je veux mettre le bouton en français, je peux rajouter un label qui portera son nom.

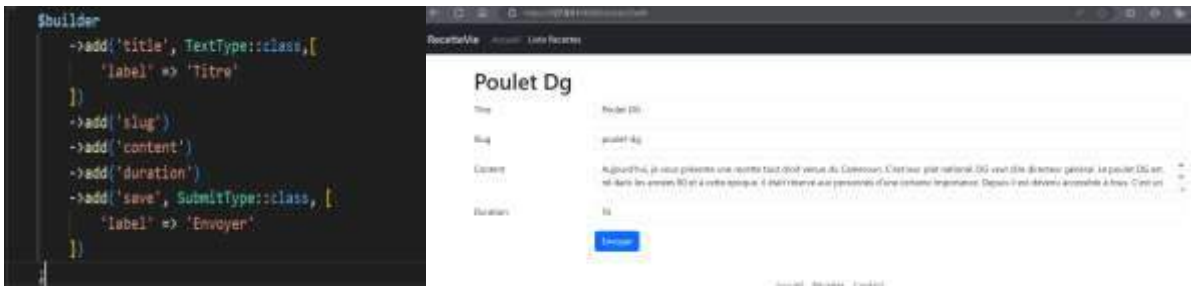
On peut changer le nom aussi de mes autres champs de la même manière avec le label.

Vérifier bien vos imports « **use Symfony\Component\Form\Extension\Core\Type\TextType** ». Par défaut les labels sont mis dans vos formulaires avec le nom du champ. On verra plus tard qu'on pourra directement faire la traduction de tout nos champs dans une autre langue. A savoir que toutes les options comme « **TextType** », « **ButtonType** » etc. sont disponible sur le site https://www.symfony.com/doc/current/reference/forms/types.html

Exos

1) **(Dans votre Projet Magasin Voiture ou SportVie) Ajoutez un formulaire pour l'entité Voiture. Et affichez le dans une page « update ». La route qui devra fonctionner dans votre url sera par exemple pour pouvoir éditer la deuxième voiture : /car/update/2**
2) **(Dans votre Projet Magasin Voiture ou SportVie) Rajoutez un bouton « update » en dessous du formulaire.**
3) **(Dans votre Projet Magasin Voiture ou SportVie) Je veux que les champs montrés dans votre formulaire soient en anglais et non en français.**

Revenons à notre projet principal. On va un peu modifier le formulaire. Normalement vous êtes tous d'accord avec moi pour dire que dans un formulaire on n'écrit pas la date de création manuellement, ni même la date de modification. Ça se fait généralement automatiquement. Donc on va les enlever de notre formulaire.



Il va falloir maintenant gérer le traitement, donc quand on clique sur « **Envoyer** », il modifie bien notre recette. Si vous cliquez sur « **Envoyer** », ça appellera la même page et on pourra voir en bas qu'il a fait un POST.

Pour pouvoir le rendre fonctionnel, on va retourner dans « **RecipeController** » dans notre « **edit** ».

On va pouvoir demander à notre formulaire de pouvoir gérer la requête. Pour faire cela, on a une méthode « **handleRequest** » à qui on passe une **Request** (requête). On va mettre en paramètre le Request (de HttpFoundation) dans notre méthode edit. On passera donc cette requête à notre formulaire.

Ça va regarder si la requête est en « **Post** » et si le formulaire a été soumis. Si c'est le cas, il va modifier l'entité « $recipe » pour la remplir avec les données provenant du formulaire.

Retourner sur votre page de modification de la recette. Changer le titre puis appuyer sur le bouton pour soumettre. Il ne se passera toujours rien. Maintenant mettez un die & dump de notre variable « $recipe » juste après le « handelRequest ». Et re-appuyer sur le bouton et vous verrez qu'il aura changé le titre :

Le formulaire va regarder pour chaque champ si on a une valeur, et pour chaque changement de valeur, il va utiliser les setters sur notre recette. Voilà ce qui se passe un peu avec le **handleRequest**.

Maintenant je peux lui demander est ce que le formulaire a été soumis grâce à une méthode « **isSubmitted** » et aussi si le formulaire est valide grâce à la méthode « **isValid** ».

Si c'est bon il faut maintenant envoyer nos données modifiées en base de données et on se souvient que c'était avec **l'EntityManager** avec le « **flush** » qu'on faisait cela. Ensuite je fais une redirection vers la page index par exemple.

```php
#[Route(path : '/recette/{id}/edit', name : 'app_recipe_edit')]
public function edit(Recipe $recipe, Request $request, EntityManagerInterface $em) : Response {
    // dd($recipe);
    //cette methode prend en premier paramètre le formulaire que l'on souhaite utiliser
    //en second parametre elle prend les données
    $form = $this->createForm(RecipeType::class, $recipe);
    $form->handleRequest($request);
    // dd($recipe);
    if ($form->isSubmitted() && $form->isValid()){
        $em->flush();
        return $this->redirectToRoute('app_recipe_index');
    }
    return $this->render('recipe/edit.html.twig', [
        'recipe' => $recipe,
        'monForm' => $form
    ]);
}
```

Après avoir fait le submit, il nous redirigera vers la liste des recettes et on pourra voir qu'il a bien fait le changement. Vous pouvez aller même voir en base de données ce sera fait. Je vais remettre l'ancien titre pour la cohérence des données.

Exos

4) **Essayez de faire en sorte que quand vous modifiez une recette, ça vous redirige vers la page show de cette recette avec la modification faites.**
5) **(Dans votre projet Magasin Voiture ou SportVie) Faites en sorte que la modification fonctionne et qu'il redirige bien vers la page show.**

6.2.1 Ajout de message Flash

Maintenant on aimerait informer l'utilisateur que la modification a bien été effectué. Pour cela on va utiliser un système de messages flash. Le principe c'est de sauvegarder les messages en sessions. On a une méthode qui est directement disponible qui est « **addFlash** ». Voici comment elle s'utilise :

```
if ($form->isSubmitted() && $form->isValid()){
    $em->flush();
    $this->addFlash('success', 'La recette a bien été modifiée');
    return $this->redirectToRoute('app_recipe_index');
}
```

Si vous testez de faire une modification, pour l'instant rien ne se passe. Pour que ça marche Il faudra aller dans nos templates. On va aller dans notre « **base.html.twig** », c'est là qu'on va utiliser les messages. On va faire un dump Twig de « **app.flashes** » :

```
<body>
    {{ include('/layouts/partials/_nav.html.twig')}}
    <div class = "container my-4">
        {{ dump(app.flashes) }}
        {% block body %}{% endblock %}
    </div>
    {{ include("/layouts/partials/_footer.html.twig") }}
</body>
```

On voit bien qu'il a sauvé le message flash qu'on avait écrit. On aura d'autres messages de différents types, par exemple error, info etc.

On va maintenant créer une petite logique pour introduire nos messages flash. On va créer un fichier « **_flash.html.twig** » dans le dossier « **/layouts/partials/** » :

templates > layouts > partials > _flash.html.twig

```
1  {% for type, messages in app.flashes %}
2  <div class="alert alert-{{ type }}">
3      {{ messages | join('. ') }}
4  </div>
5  {% endfor %}
```

Ici on utilise un foreach avec clé valeur pour lire le tableau de messages flash. Le filtre « **join** » est dans la documentation officiel de twig. Si on a plusieurs messages, il les séparera par un « . ».

```
<div class = "container my-4">
    {{ include('/layouts/partials/_flash.html.twig')}}
    {# {{ dump(app.flashes) }} #}
    {% block body %}{% endblock %}
```

Ici on verra bien qu'il nous a bien montré le message flash avec ce qu'on a mis. Il est en vert parce qu'on a mis comme type « **success** ».

Si vous réactualisez la page, le message disparait.

6.3 Le formulaire de création

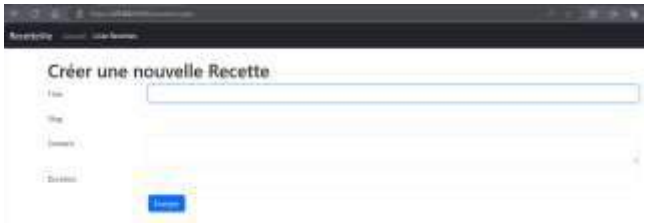
On va enfin pouvoir créer une recette directement à partir d'un formulaire. On fera comme pour le edit. Donc on va créer notre méthode « **create** » dans « **RecipeController** » :

```php
#[Route(path : '/recette/create', name : 'app_recipe_create')]
public function create(Request $request, EntityManagerInterface $em) : Response {
    $recipe = new Recipe;
    $form = $this->createForm(RecipeType::class, $recipe);
    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()){
        $em->persist($recipe);
        $em->flush();
        $this->addFlash('success', 'La recette '. $recipe->getTitle() .' a bien été créée');
        return $this->redirectToRoute('app_recipe_index');
    }
    return $this->render('recipe/create.html.twig', [
        'form' => $form
    ]);
}
```

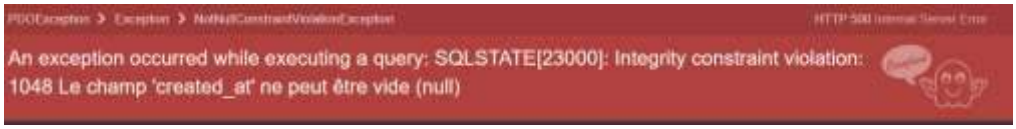
Elle ressemble beaucoup à notre méthode « **edit** ». La différence ici, c'est qu'on ne reçoit pas en paramètre une recette vu qu'on doit la créer. Et dans la partie condition, on fait un « **persist** » avant de l'envoyer en base de données. On n'a pas besoin d'afficher la recette dans le formulaire donc on envoie au Template que le formulaire.

Maintenant on a plus qu'à créer le Template « **create.html.twig** » qui affichera notre formulaire.

```twig
templates > recipe > create.html.twig
{% extends 'base.html.twig' %}

{% block title %}New Recette{% endblock %}

{% block body %}
    <h1>Créer une nouvelle Recette</h1>

    {{ form(form) }}

{% endblock %}
```



On a bien notre formulaire, par contre si vous le remplissez correctement, il vous génèrera une erreur liée aux dates de création et modification. Elles ne peuvent pas être null en BD.



En effet rappelez-vous, on avait enlevé les champs des dates dans notre formulaire « **RecipeType** ». Si les personnes doivent à chaque fois modifier eux même la date, ils ne s'en sortiraient plus.
On a plusieurs manières de gérer ce problème. La manière la plus simple c'est de mettre, au moment de la création et de la modification dans notre controller, des setters qui modifieront nos attributs concernés.

La maintenant ça fonctionnera bien.

6.4 Suppression d'une Recette

On va retourner une nouvelle fois dans notre « **RecipeController** » et ajouter une nouvelle méthode. On va l'appeler « **delete** » :

```php
#[Route(path : '/recette/{id}/delete', name : 'app_recipe_delete')]
public function delete(Recipe $recipe, EntityManagerInterface $em) : Response {
    $titre = $recipe->getTitle();
    $em->remove($recipe);
    $em->flush();
    $this->addFlash('info', 'La recette '. $titre . ' a bien été supprimée');
    return $this->redirectToRoute('app_recipe_index');
}
```

Si vous taper l'url avec votre dernière recette « **/recette/9/delete** » j'avais l'id 9 comme dernière recette. Il renverra vers notre liste et affichera le message comme quoi il a bien supprimé ma recette.

La recette okay a bien été supprimé

6.5 Gestion du Slug automatiquement

On va tout d'abord cacher le champ slug dans notre « **RecipeType** », pour qu'il ne soit plus afficher dans nos formulaires.

On peut faire ça grâce à un système d'événements au niveau des formulaires. C'est disponible dans la documentation officiel (**Form Events**). Pour faire simple, lorsqu'on soumet un formulaire, il y a différents événements qui sont envoyés (**Pre_submit, Submit et Post_submit**). La partie **Submit** c'est la partie qui va rentrer les informations dans notre modèle. On va devoir donc agir dans la partie **PRE_SUBMIT**.

Dans le fichier « **RecipeType** », on pourra rajouter un **addEventListener** et on lui passera l'événement **Pre_Submit** avec la fonction qu'on souhaite exécutée avant la soumission du formulaire. On va créer une méthode qui gèrera tout ça. **N'oubliez pas les imports !!!**

src > Form > RecipeType.php > RecipeType

```php
use Symfony\Component\Form\Event\PreSubmitEvent;
use Symfony\Component\Form\Extension\Core\Type\HiddenType;
use Symfony\Component\Form\Extension\Core\Type\TextareaType;

class RecipeType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('title', TextType::class,[
                'label' => 'Titre'
            ])
            ->add('slug', HiddenType::class)
            ->add('content',TextareaType::class)
            ->add('duration')
            ->add('save', SubmitType::class, [
                'label' => 'Envoyer'
            ])
            ->addEventListener(FormEvents::PRE_SUBMIT, $this->autoSlug(...))
        ;
    }

    public function autoSlug(PreSubmitEvent $event): void {
        $data = $event->getData();
        $slugger = new AsciiSlugger();
        if (empty($data['slug']) || $data['slug'] != strtolower($slugger->slug($data['title']))){
            $data['slug'] = strtolower($slugger->slug($data['title']));
            $event->setData($data);
        }
    }
```

6.6 Quelques améliorations

On va changer un peu le visuel de notre liste de recettes, **copiez coller la partie du body** dans votre Template « **/recipe/index.html.twig** », ici j'ai mis sous forme de carte avec une image qui pour l'instant est un lien par défaut :

```
{% block body %}
    <h1 align="center">Bienvenu dans la liste des Recettes</h1>

    <div class = "container text-center">
        <div class = "row">
            {% for recipe in recipes %}
                <div class="card max-w-sm col-4 bg-white rounded-lg border
border-gray-200 shadow-md dark:bg-gray-800 dark:border-gray-700 my-8 mr-4"
data-type="post">
                    <h2><a href={{ path('app_recipe_show', {id : recipe.id,
slug : recipe.slug})}}>{{ recipe.title }} </a></h2>
                    <img src =
"https://pbs.twimg.com/media/EdDUBw0WkAAQLJ1?format=jpg&name=large"
width="150" height="200">
                    <p>Submitted {{ recipe.createdAt |date }}</p>
                </div>
            {% else %}
                <p>Pas encore de recette</p>
            {% endfor %}
        </div>
    </div>
{% endblock %}
```

Ici on utilise un filtre « **date** » pour la date de création mais on le changera un moment donner pour qu'on voit correctement et avec la bonne date. Vous pouvez changer le lien de l'image si vous le souhaitez. On verra plus tard comment uploadé nos images de recette.

On va aussi modifier le visuel de nos formulaires edit et create. Voici le create **copier-coller le** :

```
{% extends 'base.html.twig' %}
{% block title 'Add Recipe'%}
{% block body %}
    <div class = "row">
        <div class= "col-md-6 mx-auto">
            <h1 align="center">Créer une nouvelle Recette</h1>
            {{form(monForm)}}
        </div>
    </div>
{% endblock %}
```

Attention depuis mon controller j'envoie « **monForm** », si vous avez un autre nom, adaptez le code.

Exos

6) **Faites la même structure d'affichage pour « edit.html.twig ».**
7) **(Bonus) Essayez de trouver comment changer le bouton « Envoyer » en « Editer » dans mon formulaire « edit » sans modifier le bouton du create.**

On va aussi rendre la nav bar un peu plus responsive. Copier mon code et mettez-le dans « **_nav.html.twig** » (J'ai modifié le code pour les onglets en surbrillance) :

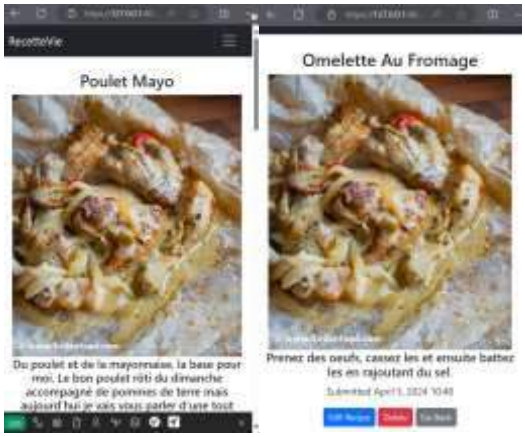
```
<nav class="navbar navbar-expand-sm navbar-dark bg-dark">
    <div class="container-fluid">
        <a class="navbar-brand" href={{ path('home') }}>RecetteVie</a>
        <button class="navbar-toggler" type="button" data-bs-toggle="collapse"
data-bs-target="#navbarsExample03" aria-controls="navbarsExample03" aria-
expanded="false" aria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
      </button>
      <div class="collapse navbar-collapse" id="navbarsExample03">
        <ul class="navbar-nav me-auto mb-2 mb-sm-0">
          <li class="nav-item ">
            <a class="nav-link {{ app.current_route  == 'home' ? 'active' :
'' }}" href={{ path('home') }}>Accueil</a>
          </li>
          <li class="nav-item ">
            <a class="nav-link {{ app.current_route == 'app_recipe_show' or
app.current_route == 'app_recipe_index' ? 'active' : '' }}" href={{
path('app_recipe_index') }}>Liste Recettes</a>
          </li>
          <li class="nav-item ">
            <a class="nav-link {{ app.current_route == 'app_recipe_create' ?
'active' : '' }}" href={{ path('app_recipe_create') }}>Ajouter une Recette</a>
          </li>
        </ul>
        </div>
    </div>
</nav>
```

Ce qui donnera ceci par exemple si on minimise la page.

On va aussi modifier notre fichier « **show.html.twig** » pour que ce soit plus agréable à voir.

Exos

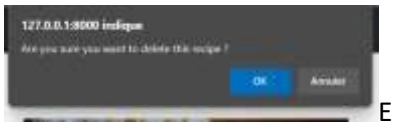
8) **Dans le show, essayez de faire un visu de ce style, avec notre recette centrée. Rajoutez une image url et des boutons qui renvoi bien vers la page Edit quand on appui sur le bouton « Edit Recipe » et qui renvoi sur la page de la liste des recettes si on appuie sur « Go Back ». Pareil pour le bouton delete qui devrait supprimer votre recette.**



Rappelez-vous, tous ce qui est en base de données quand on le supprime, c'est irréversible. Sans message de confirmation c'est un peu dangereux de le faire. On va implémenter maintenant la confirmation si on veut supprimer ou non une recette. Je vous envoi le code de mes 3 boutons :

```
<a class="btn btn-primary btn-sm" href={{ path('app_recipe_edit', {id:
recipe.id}) }}>Edit Recipe</a>
        <a class="btn btn-danger btn-sm" href="#"
onclick="event.preventDefault(); confirm('Are you sure you want to delete this
recipe ?') && document.getElementById('js-recipe-delete-
form').submit();">Delete</a>
        <a class="btn btn-secondary btn-sm" href={{ path('app_recipe_index')
}}>Go Back</a>
        <form id="js-recipe-delete-form" action={{ path('app_recipe_delete',
{id: recipe.id}) }} method="post" style="display:none;">
        </form>
```

On a un du JavaScript, avec un **onclick** que vous connaissez (quand on clique on met ce qu'il va se produire), on voit que c'est un **event.preventDefault()** ; il va faire en sorte que quand on clique dessus il n'ira pas voir le lien href=#. Le **confirm**, c'est pour proposer une fenêtre de confirmation. Ici si on confirme il va faire la suite donc aller prendre l'élément en donnant son ID (« **js-recipe-delete-form** »), donc ici faire un appel du formulaire en bas qu'on a empêché d'être montré via le « **display : none** » qu'on a rajoutée. Il va donc exécuter le formulaire qui va supprimer la Recette. Si on ne confirme pas la suppression, il ne fera rien.



En dessous, version plus simple juste avec le onclick :

```
<a class= "btn btn-danger btn-sm" href={{ path('app_recipe_delete', {id:
recipe.id}) }} onclick="return confirm('Are you sure you want to delete this
recipe?')">Delete</a>
```

6.7 Revenons un peu sur TWIG

On va un peu utiliser le langage twig. Au chapitre 4 j'avais dit qu'on y reviendrait.

Petit rappel : En **twig** quand on met **{% …%}** et du code entre, c'est pour mettre par exemple une opération, ici je vais vous montrer comment on fait une assignation. Le **set** sert à définir une variable dans twig. Je veux le nombre de Recette que j'ai en base de données via la page index.

Exos

9) **Essayez de le faire pour que ça m'affiche quelque chose comme ça, n'oubliez pas que vous avez la documentation pour trouver un moyen de le faire :**

J'en ai 8 en BD.

On met dans la variable **nbRecipe** la taille du tableau des recettes. Pour ce faire on déclare d'abord cette variable via le « **set** » ensuite on lui assigne la taille du tableau qu'on a reçu de notre controller. On a plus qu'à afficher le résultat :

```
{% block body %}
    {% set nbRecipe = recipes | length %}
    <h1 align="center">Bienvenu dans la liste des Recettes ({{ nbRecipe }})</h1>
    <div class = "container text-center">
```

Quand on a **{{…}}** c'est un appel à une variable, une fonction php, ou un Template Twig, ici c'est pour afficher le nbRecipe qui recevra le nombre de recette que j'ai en Base de Données.

On a pu voir aussi qu'on peut utiliser **{% %}** pour faire une boucle avec condition. On peut bien entendu aussi faire une simple condition, retrouver des exemples du « **If** » dans la doc officielle.

```
{% if users|length > 10 %}
    ...
{% endif %}
```

Maintenant je vais vous montrer d'autre filtre intéressant.

Pour tronquer un titre trop long, on peut le faire en allant dans la zone où l'on affiche le titre, donc dans **index.html.twig** en ajoutant « **u.truncate »** qui est une fonction qui recevra en paramètre la taille de caractère avant de tronquer, en second paramètre elle reçoit une option (genre '...' qui se mettra à la fin du troncages) et pour finir on peut ensuite mettre un booléen pour compter ou pas les '...' :

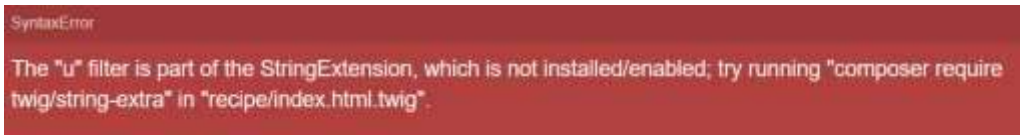
```
<h2><a href={{ path('app_recipe_show', {id : recipe.id, slug : recipe.slug})}}>{{ recipe.title |u.truncate(20,'...',true)}} </a></h2>
```

Ici en gros je veux faire en sorte que nos titres de recette dans notre page index, soient affiché que de 20 caractères. Si y a plus il rajoute « … ». Le true veut dire qu'il rajoutera … après le dernier mot.

Exos

10) **(Résoudre le problème) Essayez d'exécuter et de faire en sorte que ça fonctionne.**

Voici donc l'erreur qu'on a eu :



Si vous comprenez l'anglais et savez utilisez symfony, c'est facile. Il nous dit que :

- Le filtre « **u** » fait partie de l'extension « **StringExtension** » mais qu'il n'est pas installé/activé. Essayez de lancer « **composer require twig/string-extra** »

Il arrive par moment que Symfony vous aide, il suffit des fois de juste lire et comprendre l'erreur. Je vais donc faire ce qu'il me dit.

On voit bien ici qu'il a tronqué les titres qui dépassent 20 caractères et a rajouté « **…** » qui sont compris dans le compte. Si on remplaçait le « **true** » par le « **false** » ça donnerait ceci :

Il mettra les « … » après le dernier mot après 20 caractères.

Il y a aussi un autre filtre qu'on va utiliser qui est sympa. C'est le filtre « **ago** » pour les dates. Si vous voulez montrer depuis combien de jour la recette a été posté, il suffit de le rajouter :

Pour que ce filtre fonctionne il faut installer le composant :

- **composer require knplabs/knp-time-bundle**

On voit que c'est en anglais mais si je voulais je pouvais le mettre en français en rajoutant juste :

Maintenant dans notre index on verra depuis combien de temps ça été mis en ligne et quand on ira dans show, là on verra la date de création sous forme de date.

Exos

11) **Essayez de m'afficher la date de création sous le format Jours Mois Années dans show.**
12) **Affichez aussi la date de modification au format Jours Mois Années Heures Minutes.**

On verra que ça changera bien mes dates dans le format que je souhaite.

6.8 Date et fuseau horaire

On avait remarqué dans notre projet, que quand on créait une nouvelle recette, il nous donnait pour createdAt, une heure de création des heures de décalage en moins. On peut le voir dans l'exo 12.

En fait Symfony de base quand on ne le configure pas, va utiliser l'heure UTC (l'heure universelle donc celle à GMT+0). Pour changer de fuseau horaire et donc choisir celui de Paris/Bruxelles Europe en GMT +1/2, il suffit de créer le fichier "**php.ini**" dans la racine de votre projet.

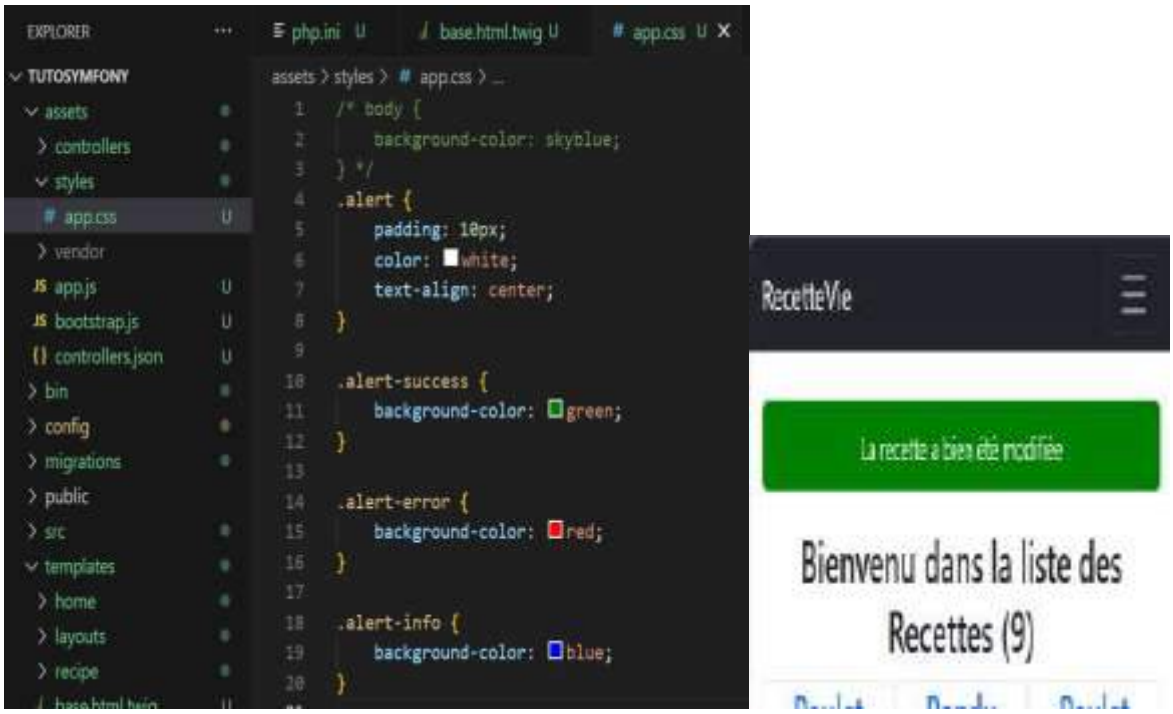
Et de définir le **date.timezone** en **Europe/paris** :

Redémarrez votre serveur !!! Pour qu'il prenne en compte la modification.

Essayer maintenant de créer une Recette et d'aller vérifier son heure de création en cliquant sur la recette créée. Vous pourrez voir que la recette a bien été créée à l'heure exacte. Ce sera pareil quand vous modifierez une recette.

6.9 Ajout de son propre CSS

Je vais tout simplement modifier le style de mes messages flash, en rajoutant du CSS dans le fichier « **assets/styles/app.css** » :



C'est là que vous pourrez gérer votre CSS du projet de manière générale.

Il y a plusieurs moyens de le faire. On pourrait par exemple créer un fichier styles.css dans « **public/css/styles.css** » et y mettre notre CSS. Ensuite allez dans « **base.html.twig »** et l'appeler en le mettant dans le bloc style Sheets de twig :

```
{% block stylesheets %}
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/di
<link rel="stylesheet" href={{ asset("/css/styles.css") }}>
{% endblock %}
```

A vous de choisir, vous êtes plus doué que moi pour le front-end.

6.10 BONUS CRUD

Nous avons donc réalisé le **CRUD** (**C**reate, **R**ead, **U**pdate, **D**elete) pour une entité/classe :

- Page de création d'une Recette (**Create**)
- Pages Index et Show qui récupèrent les données en Base de Données (**Read**)
- Page éditer une Recette (**Update**)
- Dans la page Show on a un bouton pour supprimer une Recette (**Delete**)

Avant de continuer, il faut savoir qu'on a une commande qui permet de créer automatiquement pour nous le CRUD pour une entité, en donnant même les pages avec formulaire etc. Donc ce n'est pas juste du côté du Controller, mais même au niveau des vues (templates).

ATTENTION CHANGEZ DE PROJET !!!

On va le tester dans le SIDE projet « **magasin_voitures ou sportvie**». Rajoutez une nouvelle entité/classe « **Moto** » **qui aura les 5 mêmes attributs que** « **Voiture** ».

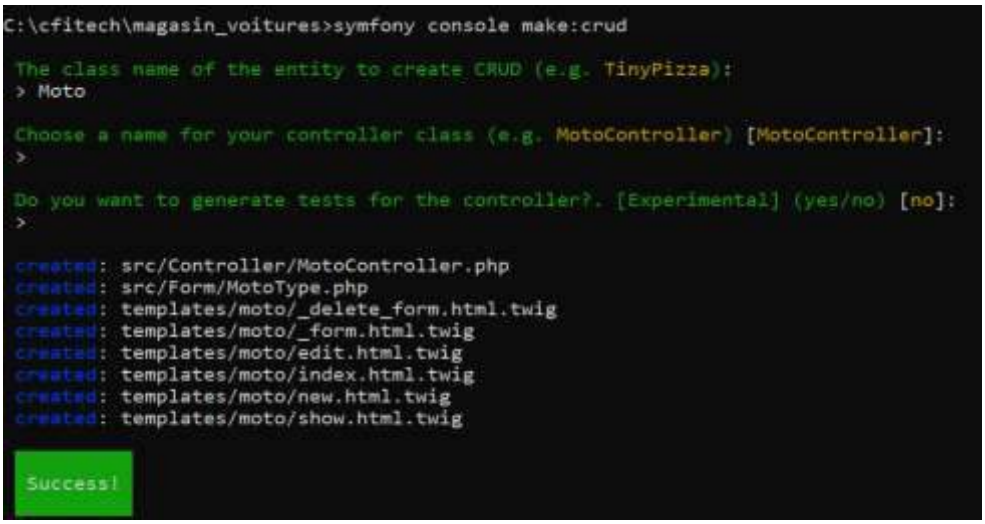
- **symfony console make:entity Moto**

Une fois votre entité créée, faites une migration vers la base de données :

- **symfony console make:migration**
- **symfony console doctrine:migrations:migrate**

Pour finir il faudra juste lancer la commande :

- **symfony console make:crud**



Comme par magie il nous a créé la plupart des fichiers qu'on avait rajouter manuellement. Pareil il a créé le Controller et le Formulaire. Il faudra pas oublier de faire la migration. Il suffit maintenant d'aller sur la page d'index de notre MotoController « **/moto** ».

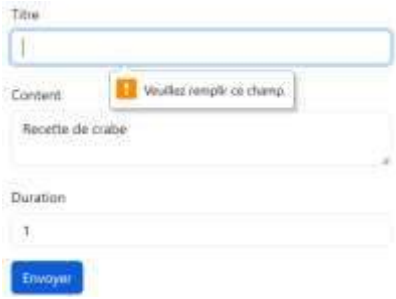
Voilà en moins de 5 minutes comment faire ce qu'on a fait jusqu'à maintenant coté fonctionnement.

Chapitre 7 : La validation des données

Dans ce chapitre nous allons maintenant parler de la validation des données. On a utilisé des formulaires mais le problème c'est qu'un utilisateur mal intentionné peut nous poster des recettes avec un titre beaucoup trop court, une description vide, un titre identique, ou encore des gros mots etc. Pour remédier à tous ces problèmes on va avoir besoin de valider les choses.

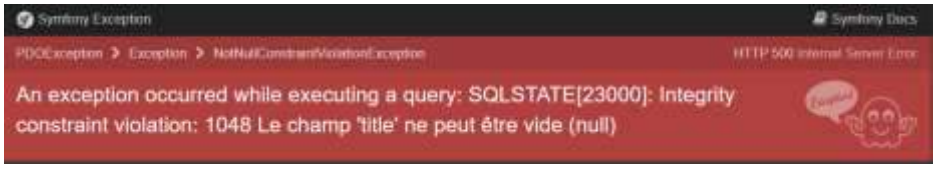
7.1 Introduction

Vous avez tous fait du front-end et savez que si on fait inspecter une page, on peut potentiellement modifier un comportement qui de base a été défini par le développeur du site web. Je prends le simple exemple de notre projet recette dans la page d'ajout d'une recette.



Quand vous laissez votre titre vide, il y a une protection parce que le formulaire est en require. Mais suffis que j'aille dans inspecter, et je peux modifier cela.

Si je suis mal intentionné et que j'enlève la partie **required**, ça va tenter un forcing et le mettre en base de données. Ça nous sortira une erreur très explicite.



Heureusement en base de données, il y a une autre protection mais vous voyez que j'ai pu créer du coup un objet de type recette sans Titre.

Deuxième exemple. Allons dans mon **RecipeController**, dans la condition de la méthode create, et **enlevons**-le **$form->isValid**. Et mettons ceci comme contrainte dans **RecipeType** :

Si je vais dans ma page de création de recette et que je fais inspecter.

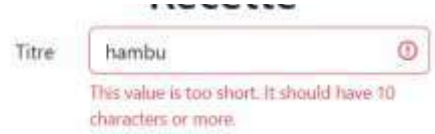
Je peux lui dire par exemple que je veux minimum 5 lettres au lieu de 10. Je rajoute minlength=5 :

```
<input type="text" id="recipe_title" name="recipe[title]"
required="required" class="form-control" minlength="5">
```

J'aurai pu même ne rien mettre dans le HTML au niveau de l'inspection que se serait passé. Parce que la contrainte que je donne n'est pas appliquée vu qu'on ne vérifie plus si le formulaire est Valide. D'où l'importance d'avoir mis dans notre méthode le fameux $form->isValid().

Je ne suis pas un expert dans ce qui est de l'inspection, mais il faut savoir qu'il est fortement recommandé de mettre des validations cotés serveur et non coté client. Les protections coté client peuvent être détourné en passant par le HTML.

Si je remets le isValid dans ma méthode create, il protégera.



Par défaut il donne un message en anglais. Il faut savoir qu'on peut modifier directement le message en paramètre dans le RecipeType. Il y a aussi moyen de modifier dans la partie traduction qu'on verra plus tard.

```
'constraints' => new Length(min: 10, minMessage: "Valeur trop petite, 10 minimum")
```

Au final ça fonctionnera assez bien, mais on ne le fera pas de cette manière. La plupart des développeurs Symfony ne le font pas via le formulaire mais directement à la source, c'est-à-dire au niveau de la classe qui est utilisé. Dans notre cas on le fera dans notre entité Recipe. Enlevons notre contrainte dans RecipeType.

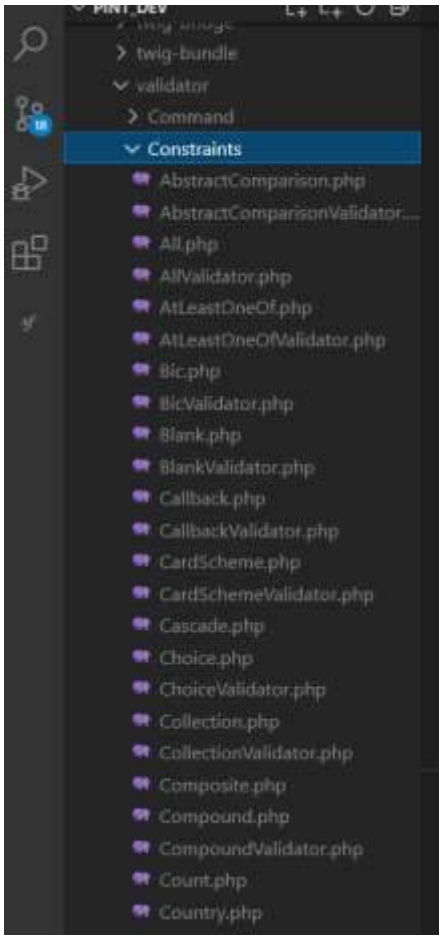
```
public function buildForm(FormBuilderInterface $builder, array $options): void
{
    $builder
        ->add('title', TextType::class,[
            'label' => 'Titre'
        ])
        ->add('slug', HiddenType::class)
        ->add('content',TextareaType::class)
```

7.2 Le composant Validator

Le composant **validator**, est un composant officiel de Symfony. Normalement quand on crée un projet en webapp, ce composant est déjà disponible.

Le composant validator permet de gérer la validation des données des formulaires. Par exemple une Recette ne pourra pas avoir moins de 10 caractères, ou qu'un champ ne peut pas être vide au niveau du formulaire etc.

Ce composant se trouve dans « **vendor/symfony/validator/ »**, on pourra voir dans le dossier « **Constraints** », qu'il y a plein de contraintes (classe). On pourra valider par exemple des pays, le mail, la date etc.



Ici dans notre formulaire nous allons utiliser donc validator pour, dans un premier temps, ne pas permettre d'enregistrer une recette quand le champ est vide. De toute façon cela produisait une erreur au niveau de la base de données quand on essayait de valider un champ vide. Dû au fait que l'attribut titre ne peut pas être null (no null).

7.3 Ajout des contraintes dans notre entité

Dans la classe Recipe, il faudra tout d'abord importer via un « **use** » toutes les contraintes de **Validator**, ensuite rajouter dans les paramètres de **title** et de **content** la contrainte (constraints) **NotBlank** qui va ici faire en sorte que les champs ne peuvent pas être vide.

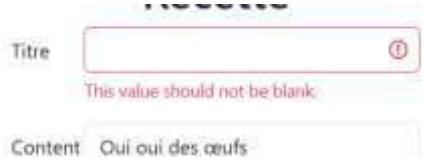
```php
use Symfony\Component\Validator\Constraints as Assert;
```

Ici le mot **Assert** est juste un **alias**, on aurait pu mettre ce qu'on veut après le « **as** », c'est celui qu'on utilisera dans les annotations. On parle **d'Assertion** :

```php
#[ORM\Column(length: 255)]
#[Assert\NotBlank()]
private ?string $title = null;
```

On appel la classe NotBlank dont le chemin complet est :
Symfony\Component\Validator\Constraints\NotBlank.

Vu qu'on utilise notre formulaire et qu'il a été associé à notre entité Recipe, il va lire cette validation. Ici c'est une validation coté serveur donc plus sécurisé.

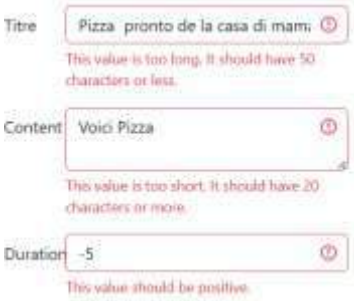


Pour voir le message, il faut enlever required en inspectant.

Toutes ces contraintes sont retrouvables dans la documentation officielle de symfony.

Exos

1) **Essayez de faire de même pour le « content » donc qu'il reçoit NotBlank.**
2) **Essayez de trouver un moyen de faire en sorte que « title » soit de minimum 10 caractères.**
3) **Essayez de faire en sorte que « content » soit de minimum 20 caractères.**
4) **Essayez de trouver un moyen de faire en sorte que « duration » ne peut pas être négative.**
5) **Essayez de faire en sorte qu'une recette ne dure pas plus de 24h.**
6) **Rajoutez pour « title » maximum 50 caractères.**



On sait bien entendu combiner plusieurs Assertions/Contraintes l'une à la suite des autres. Je peux bien entendu avoir un **NotBlank** suivit d'un **Lenght** :

```php
#[ORM\Column(length: 255)]
#[Assert\NotBlank()]
#[Assert\Length(min: 10)]
#[Assert\Length(max: 50)]
private ?string $title = null;
```

```php
#[ORM\Column(nullable: true)]
#[Assert\Positive()]
private ?int $duration = null;
```

Je rappelle que ce sont des protections cotés serveurs, même si c'est via les formulaires qu'on le voit. Et vous avez un tas de contraintes que vous pouvez rajouter.

Exos

7) **Tentez de mettre tous les messages en français**

8) **Essayez de trouver un moyen de faire en sorte qu'on ne puisse pas donner comme titre « Merde ».**

Dans cet exemple on a un mot < 10 caractères

9) **Implémentez des validations dans vos projets. (Magasin voiture et personnel)**

7.4 <u>Contraintes personnalisée</u>

Si je reviens sur la question 8, on a un petit problème que certain ont certainement remarqué. En effet, si je mets comme titre « Recette de Merde », le validator va laisser passer parce qu'il se dit le titre n'est pas égale à « Merde ».

Moi je veux que le ou les termes que je trouve inappropriés soient interdits.

Pour se faire, je peux décider de créer une contrainte qui empêcherai une liste de mot que je définirai.

Allons dans notre cmd et tapons cette commande « **symfony console make:validator** » ensuite donnons le nom « **InappropriateWords** » :

Il nous créé 2 Classes dans « **src/Validator** » et nous dis qu'on peut y rajouter notre logique.

La classe **InappropriateWords**, elle va représenter la contrainte, c'est comme la classe **NoBlank** ou encore **Length** qu'on a utilisé dans nos contraintes. On va la modifier :

```php
#[\Attribute(\Attribute::TARGET_PROPERTY | \Attribute::TARGET_METHOD | \Attribute::IS_REPEATABLE)]
class InappropriateWords extends Constraint
{
    //On crée un constructeur qui reçoit la liste des mots et le message approprié quand ca ne va pas
    //Je lui rajoute aussi les paramètres que reçoit le constructeur parent
    //a l'interieur on appel le constructeur parent
    public function __construct(public array $listWords = ['merde','wesh','imbécile'], public string $message = 'This contains an
    inappropriate word "{{ inappropriateWord }}".', mixed $options = null, ?array $groups = null, mixed $payload = null)
    {
        parent::__construct($options, $groups, $payload);
    }
}
```

Le message je le remet en anglais, on verra comment traduire plus tard.

Ensuite on a la classe **InappropriateWordsValidator** qui servira à recevoir, via une méthode, une valeur et en second paramètre une contrainte. On va aller rajouter notre logique et reprendre l'erreur qu'on enverra qui est déjà là :

```php
class InappropriateWordsValidator extends ConstraintValidator
{
    public function validate($value, Constraint $constraint)
    {
        /* @var InappropriateWords $constraint */

        if (null === $value || '' === $value) {
            return;
        }

        //on met d'abord le mot introduit dans le formulaire en minuscule
        //on fait ensuite une boucle pour varifié que le mot intruduit n'est pas dans notre liste
        //si il est dans notre liste, il envoi une violation avec notre message définit dans InappropriateWords.php
        $value = strtolower($value);
        foreach($constraint->listWords as $inappropriateWord){
            if(str_contains($value, $inappropriateWord)){
                $this->context->buildViolation($constraint->message)
                ->setParameter('{{ inappropriateWord }}', $inappropriateWord)
                ->addViolation();
            }
        }
    }
}
```

On n'oublie pas d'aller rajouter dans la classe **Recipe,** cette nouvelle contrainte pour notre titre. J'ai enlevé l'ancienne contrainte NotEqualTo.

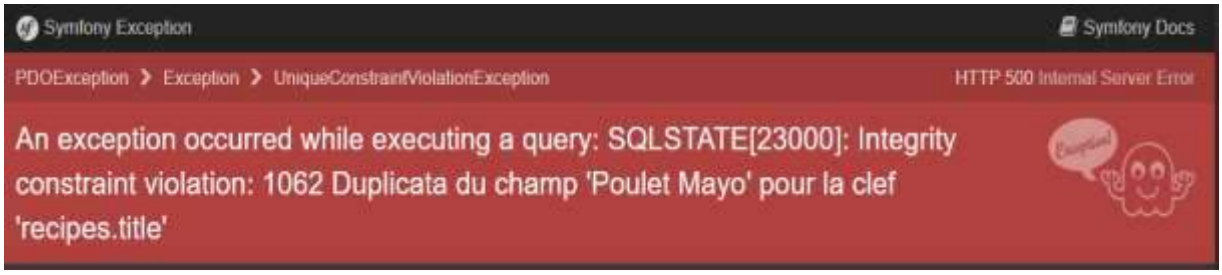
```php
#[ORM\Column(length: 255)]
#[Assert\NotBlank(message: "Votre titre ne peut pas être vide")]
#[Assert\Length(min: 10,max: 50, minMessage: "Vous devez avoir u
#[InappropriateWords()]
private ?string $title = null;
```

Si je retourne sur mon site et que je tente d'écrire des mots de ma liste ça me donnera ceci :

7.5 Rendre un champ unique

Je vais essayer de faire en sorte qu'on ne puisse pas avoir le même titre pour nos recettes. On avait déjà fait quelques choses comme ça dans le cours de MySQL, en rendant un champ unique et ça avait bien marché. Si j'essaye de faire pareil, donc que je vais modifier en base de données mon champ title et le met en unique.

Normalement quand je vais créer une recette, si un titre existe déjà il ne pourra pas le rajouter. Mais me sortira une erreur coté Base de Données :



On a bien la protection du fait qu'on ne puisse pas avoir le même titre, par contre j'aimerai mieux gérer ça. Je peux rajouter une autre protection coté serveur toujours, qui quand on introduira un titre dans notre formulaire, il ira d'abord vérifier lors de la création de ma recette en Base de Données si le titre existe déjà. S'il existe il devra me sortir un message.

Ce sera grâce à une contrainte spéciale qui s'appelle « **UniqueEntity** », disponible bien entendu comme toutes les contraintes, dans la documentation officielle de Symfony.

Cette contrainte se met directement au niveau de votre classe/entité contrairement à celles qu'on a vu précédemment. Je mets une contrainte sur le titre, mais j'aurai pu en rajouter autant que je veux.

N'oubliez pas l'import, qui normalement se rajoute automatiquement pour la plupart :

```
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
```

Elle vient de Doctrine. Donc c'est bien tout ce qui concerne la base de données.

Exos

10) **Implémentez tous ce qu'on a fait dans le projet magasin_voiture ainsi que dans vos projets.**

7.6 Ajout d'un champ Image

Nous allons rajouter un nouvel attribut/champ « **imageName** » dans notre entité **Recipe**.

Exos

11) **Vous connaissez désormais la démarche à suivre pour le faire. Je vous donne juste les paramètres :**
    - **imageName, string, 500, yes (null)**

    Une fois finis vous me montrerez chacun votre nouveau champ dans la base de données. Voici le résultat à avoir :

id	title	slug	content	created_at	updated_at	duration	image_name
1	Poulet Mayo	poulet-mayo	Du poulet et de la mayonnaise, la base pour moi. L...	2024-03-29 09:50:39	2024-03-29 09:50:39	60	NULL
2	Pondu (Saka Saka)	pondu-saka-saka	Pour ceux et celles qui ne savent pas ce qu'est le ...	2024-03-29 10:41:35	2024-03-29 10:41:35	130	NULL
3	Poulet DG	poulet-dg	Aujourd'hui, je vous présente une recette tout dro...	2024-03-29 10:41:35	2024-03-29 10:41:35	55	NULL

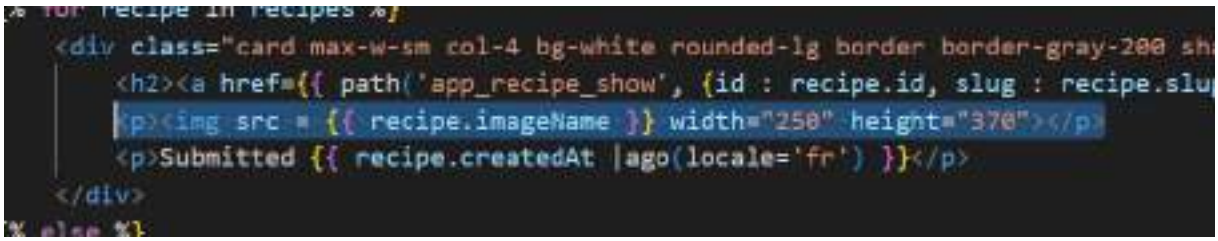
12) **Essayez de faire en sorte que votre formulaire de création l'affiche :**

Nous allons utiliser une url pour utiliser les images dans un premier temps. On implémentera plus tard un outil nous permettant de charger les images dans notre projet.

Pour qu'il s'affiche dans nos formulaires, il fallait aller dans **src/Form/RecipeType** rajouter ceci :

Grace à ça on rajoutera dans nos formulaires de **creation et de modification de recette** un champ avec l'url de l'image.

Il faut ensuite allez dans **index.html.twig** et modifier :



Et aussi allez dans **show.html.twig** et rajouter ceci :

```
<div align = "center" class= "col-md-6 mx-auto">
    <article>
        <h1>{{ recipe.title | title }}</h1>
        <img style="max-width: 400px;" src={{ recipe.imageName }}>
        <h5>{{ recipe.content | nl2br }}</h5>
        <p>Submitted {{ recipe.createdAt|date("d/m/Y") }}</p>
        <p>Updated {{ recipe.updatedAt|date("d/m/Y H:i") }}</p>
```

Ici j'ai mis d'une manière la balise <img>.

On va aussi rajouter une image par défaut quand on crée une Recipe. Il suffit d'aller dans votre entité **Recipe.php** et mettre un lien à coté de votre attribut **$imageName** :

```
private ?string $imageName =
"https://upload.wikimedia.org/wikipedia/commons/a/ac/No_image_available.svg";
```

Tentez de créer une recette, vous verrez que vous aurez le lien par défaut. Et l'image sera générée une fois la recette créée.

Exos

13) **Mettez des images différentes pour toutes vos recettes existantes.**

Voilà vous pouvez bien entendu modifier l'affichage comme vous le souhaitez pour la taille des images que ce soit dans la page d'accueil ou sur la page de détails par rapport à une Recette.

Je le redis encore pour ce qui est de l'esthétique, vous pouvez appliquer vos connaissances en HTML, CSS et JAVASCRIPT, pour un peu styliser comme vous le souhaitez votre site.

Attention à noter que vous devez du coup être en ligne pour pouvoir afficher les images qui sont je le rappelle dans ce cas des liens url (venant d'internet). On va voir un peu plus tard comment charger nos propres images dans notre projet.

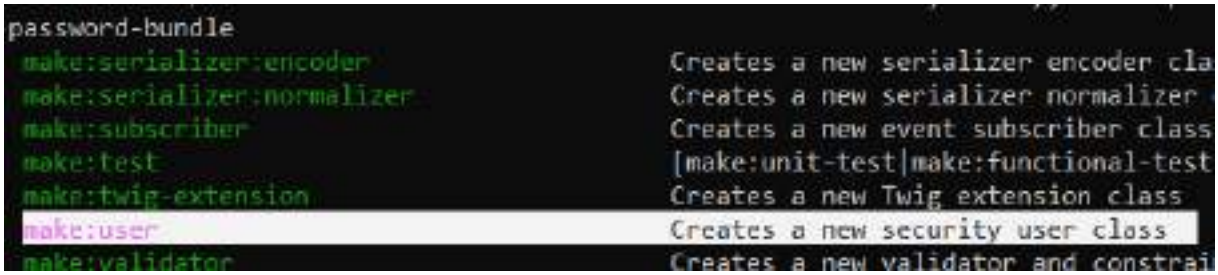
Chapitre 8 : Les utilisateurs

Dans ce chapitre nous allons ajouter la notion d'utilisateur. Jusqu'à présent, on pouvait faire le CRUD sur toutes les recettes. A partir d'ici on va essayer de faire en sorte qu'un utilisateur peut gérer ses recettes qu'il crée. Pour faire ça, on devra vous vous en doutez lier les utilisateurs aux recettes. L'utilisateur pourra aussi dans un premier temps s'inscrire via un formulaire de registration, ensuite il pourra se connecter quand on fera la partie authentification et gérer son compte.

8.1 <u>Création d'un Utilisateur</u>

Une manière vous vient à l'esprit, c'est de crée l'Entity/entité **User** en faisant « symfony console make:entity User » puis fournir les champs : firstname, lastname, password, email etc. Mais nous n'allons pas le faire comme ça. Symfony a une commande qui crée directement l'utilisateur, donc on n'aura déjà une base, c'est ensuite à nous de modifier à notre guise les attributs, méthodes, etc.

Si vous faites « **symfony console** » vous verrez la liste des commandes que Symfony peut faire pour ce projet.



Dans la liste on pourra remarquer qu'il y a « **make:user** » qui va nous permettre très facilement de créer notre entité User.

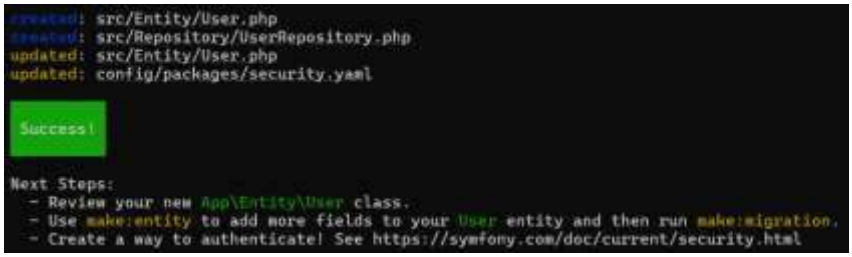
On rajoutera une partie authentification et sécurité par la suite.

Commençons, créons d'abord notre utilisateur :

- **symfony console make:user**

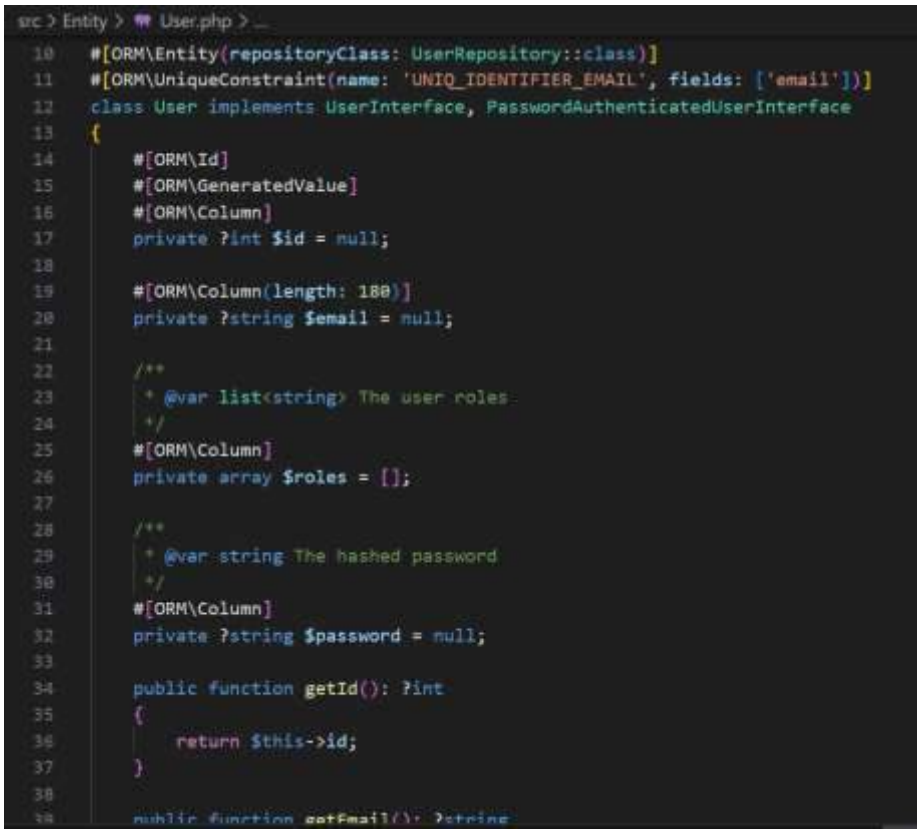
On va garder le nom **User** par défaut, on va stocker les données de l'utilisateur dans la base de données. On nous demande notre identifiant visuelle (un champ/attribut unique). Ici on va utiliser l'adresse mail donc ce qui est par défaut. On va nous demander pour le hachage et vérification du mot de passe, on va dire oui. C'est pour qu'on ne les stocks pas en clair dans la base données. Il est préférable de ne jamais stocker les mots de passe en clair.

Une fois finit, il nous a créé 2 fichiers et mis à jour d'autres :



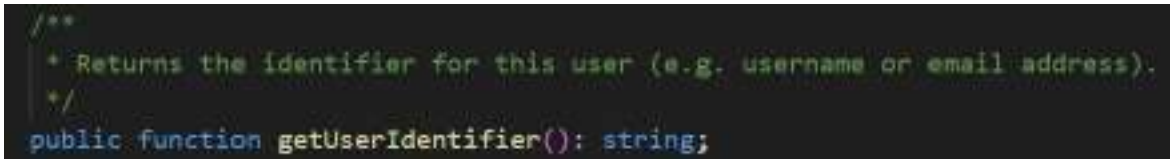
Il nous dit ensuite d'aller voir la classe et de rajouter des champs si on le veut.

On va d'abord aller voir notre classe **User :**

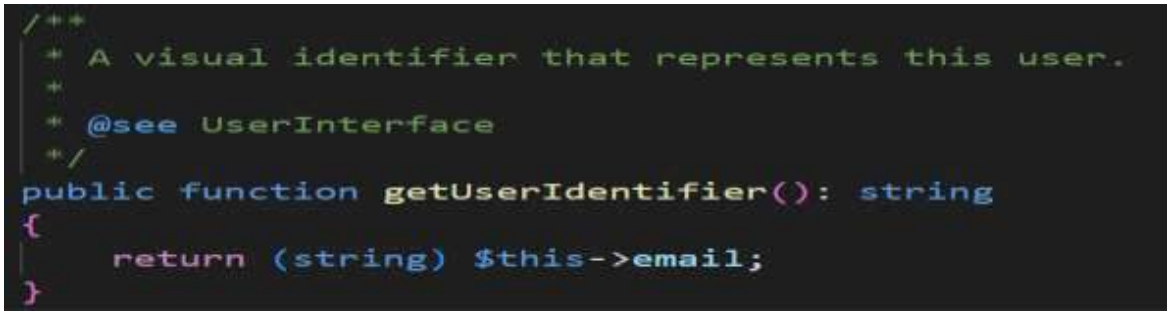


Première chose qu'on peut remarquer dans cette entité User, c'est qu'à la différence de l'entité Recipe qu'on avait créé avec un **make:entity**, cette classe/entité **User** implémente deux interfaces. Rappel : Au niveau de la **Programmation Orienté Object**, les **interfaces** c'est comme des contrats. Donc si on implémente une interface c'est comme si on signe un contrat, qui lie notre entité User à ces deux interfaces. Dans ces interfaces on a des entêtes de méthodes qui n'ont pas de corps. Le contrat ici c'est que pour être implémenté par ces deux interfaces, il faut que la classe User remplisse ces méthodes « **vide** ».

Dans l'interface « **UserInterface** » on a juste l'entête :



Dans la classe « **User** » on remplit la méthode :



En gros c'est comme si on promet à nos interfaces de donner à toutes leurs méthodes un corps. On n'est pas obligé d'avoir du contenu dans une méthode qui vient d'une interface, on peut juste mettre les accolades si on le souhaitait :

- Public function blablabla(){

  }

On va rajouter deux attributs à notre classe User.

Exos

1) **Rajoutez deux attributs à notre entité User :**
   - **firstname, string, 255, no null**
   - **lastname, string, 255, no null**

   Je veux avoir ce résultat en base de données :

	#	Nom	Type	Interclassement	Attributs	Null	Valeur par défaut
☐	1	id	int(11)			Non	*Aucun(e)*
☐	2	email	varchar(180)	utf8mb4_unicode_ci		Non	*Aucun(e)*
☐	3	roles	json			Non	*Aucun(e)*
☐	4	password	varchar(255)	utf8mb4_unicode_ci		Non	*Aucun(e)*
☐	5	firstname	varchar(255)	utf8mb4_unicode_ci		Non	*Aucun(e)*
☐	6	lastname	varchar(255)	utf8mb4_unicode_ci		Non	*Aucun(e)*

On va créer maintenant encore 2 attributs que vous connaissez, **createdAt** et **updatedAt**.

On comprend directement qu'on aura la même chose dans la classe Recipe et la classe User.

Une pratique courante, c'est que pour ne pas à chaque fois réécrire ce qu'on a déjà, on va créer un trait qui permettra juste d'être appelé avec **createdAt** et **updatedAt** ainsi que leurs getters, setters et fonction.

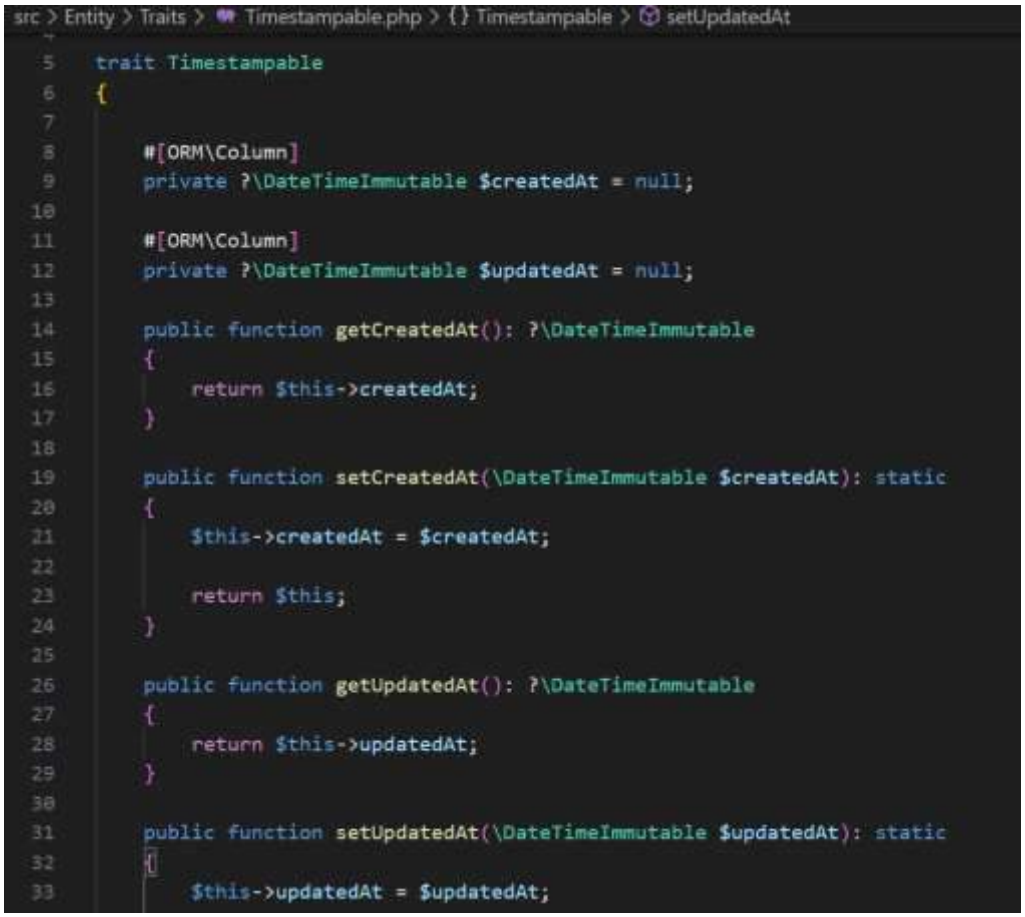
On va tout d'abord créer **un nouveau dossier** « **Traits** » dans (**Entity/Traits**). Dans ce nouveau dossier on va créer un nouveau fichier s'appelant « **Timestampable.php** ».

Voici l'entête, on précise qu'on est en PHP, on donne le **Namespace** (donc ou le fichier se trouve) et on donne comme nom de trait (**Timestampable**) :

```php
<?php
namespace App\Entity\Traits;
use Doctrine\ORM\Mapping as ORM;

trait Timestampable
{

}
```

Le principe des **traits** est de permettre de contourner les limites imposées par l'héritage simple de **PHP**. Le but est de permettre de créer de nouvelles méthodes et de nouvelles propriétés que l'on pourra ajouter à nos différentes classes de manière horizontale.

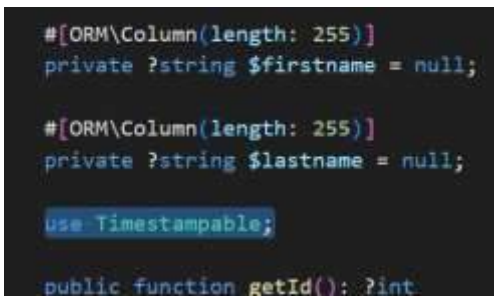
Dans ce « trait **Timestampable** » vous allez copier-coller tous ce qui est en rapport avec **createdAt** et **updatedAt** de la classe **Recipe**. Donc la déclaration des **2 attributs**, leurs **getters** et **setters** ainsi que la fonction.



ATTENTION, il ne faudra pas oublier de rajouter l'import du Timestampable dans la **classe User** :

```
use App\Entity\Traits\Timestampable;
```

Et ensuite il faudra utiliser ce fichier de cette manière ci :



C'est comme ça qu'on fait l'appel du fichier « **use Timestampable** ». Je l'ai mis en dessous de la déclaration de firstname et lastname, mais on aurait pu le mettre ailleurs.

Une fois finit faites une migration vers la base de données, vous connaissez les commandes maintenant.

#	Nom	Type	Interclassement	Attributs	Null	Valeur par défaut	Commentaires	Extra
1	id	int			Non	*Aucun(e)*		AUTO_
2	email	varchar(180)	utf8mb4_unicode_ci		Non	*Aucun(e)*		
3	roles	json			Non	*Aucun(e)*		
4	password	varchar(255)	utf8mb4_unicode_ci		Non	*Aucun(e)*		
5	firstname	varchar(255)	utf8mb4_unicode_ci		Non	*Aucun(e)*		
6	lastname	varchar(255)	utf8mb4_unicode_ci		Non	*Aucun(e)*		
7	created_at	datetime			Non	*Aucun(e)*	(DC2Type:datetime_immutable)	
8	updated_at	datetime			Non	*Aucun(e)*	(DC2Type:datetime_immutable)	

Exos

2) **Modifiez Recipe pour que lui aussi utilise le Timestampable.**
3) **Appliquez toutes les choses qu'on a vu dans vos « side Project » pour voir si vous saurez le refaire.**

N'oubliez pas de mettre sur GitHub vos avancées, c'est à vous d'avoir cet automatisme.

8.2 Hachage de mot de passe

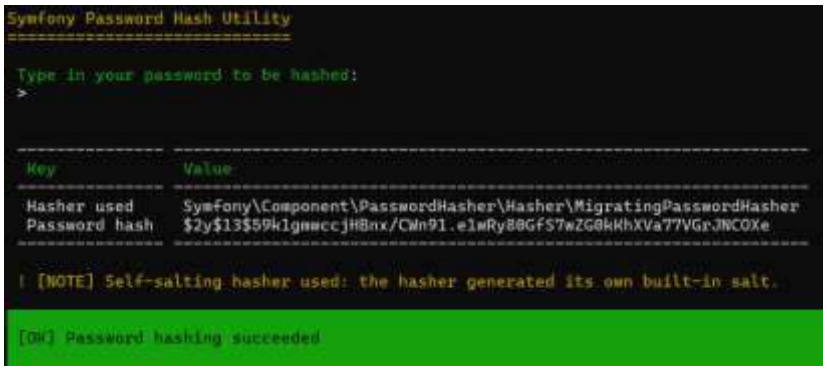
Symfony permet de cacher nos mots de passe en base de données grâce au hachage. D'ailleurs il existe une commande dans Symfony pour utiliser le système de hachage de mot de passe. Si vous faites « **symfony console** » vous pourrez la retrouver :

Ici il va nous permettre d'encoder un mot de passe, en gros d'haché un mot de passe d'un Utilisateur.

Lancer la commande :

- **symfony console security:hash-password**

Ensuite tapez comme mot de passe « **Bonjour** » :



Si vous tapez tous « **Bonjour** » comme mot de passe, vous n'aurez pas le même hachage.

C'est donc ce style de mot de passe haché qui va se retrouver dans ma base de données qui représente « **Bonjour** ». C'est le **Bcrypt**.

C'est via une fonction de hachage qui transforme notre mot de passe qu'il reçoit en paramètres. Les mots de passe sont différents grâce au système de **Salt cryptographique**. Je ne vais pas trop rentrer dans les détails mais en gros même si 10 utilisateurs ont comme mot de passe « **Bonjour** », le hachage sera à chaque fois différends parce il rajoute à chacun des mots de passe « **Bonjour** », un mot aléatoire avant et après que lui seul sait : exemple il va hacher pour un utilisateur en rajoutant (blaBonjourbla) et pour un autre (cocoBonjourcoco). Du coup quand il fait l'appel à la fonction de hachage, il recevra deux mots de passe différends. C'est ça le Salt cryptographique.

8.3 Création de la relation entre un Utilisateur et une Recette

Normalement vous avez vu les relations en base de données. Mais je vais quand même ici expliquer rapidement cette relation.

Pour faire simple un utilisateur quand il se connectera, pourra créer ses recettes. Donc ces recettes créées seront liées à cet utilisateur et pas à un autre. On verra que c'est lui l'auteur. Nous ici donc on veut faire en sorte qu'en base de données quand l'utilisateur crée une recette, elle sera reliée à lui. Vous l'aurez donc compris qu'un **utilisateur** pourra **avoir 0 à plusieurs recettes.** Et automatiquement aussi qu'une **recette** ne pourra **avoir qu'un seul auteur/utilisateur**.

Cette relation est appelée **ManyToOne**.

En faisant cette relation ça va créer une clé étrangère **user_id** dans la table Recette.

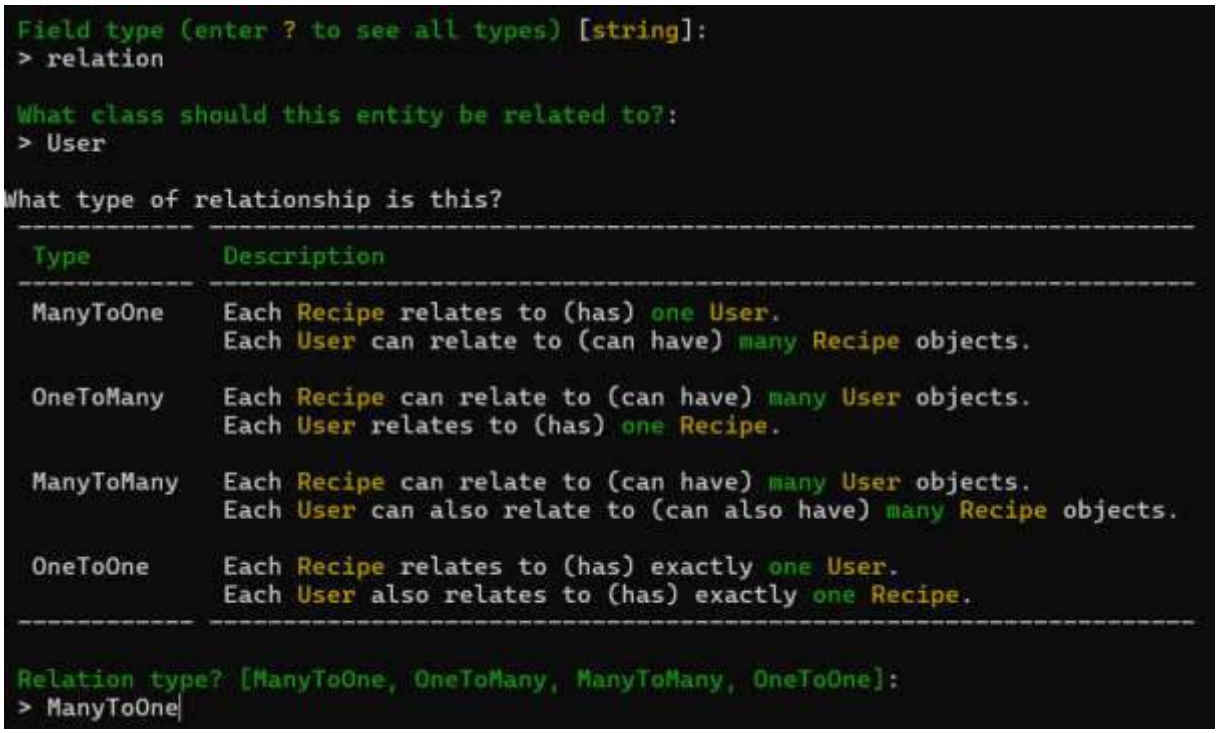
Essayons de rajouter cette **relation** en passant par l'ajout de ce nouvel attribut dans l'entité Recipe.

Faites donc :

- **symfony console make:entity Recipe**

Pour le nom de l'attribut on va mettre « **user** » ensuite pour son type c'est là qu'on va donner la relation. Il suffit de mettre « **?** » pour voir la liste des possibilités. On verra tout en haut qu'on a nos relations.

On sait déjà la relation que ce sera, mais on peut taper « **relation** », il nous posera des questions pour savoir de quelle relation on a besoin. Il nous demandera avec quelle classe voulez-vous lier la classe Recipe, on mettra bien entendu « **User** ».



On choisira donc celui qui match avec l'explication que j'ai mis sur la relation entre Recipe et User. C'est bien entendu « **ManyToOne** ».

Il nous pose ensuite encore une question, et nous demande est ce qu'on peut avoir une Recette qui n'a pas d'utilisateur associé. On va bien entendu mettre « **no** » parce qu'une Recette devra être créée par un utilisateur obligatoirement.

On nous demande ensuite, est ce qu'on veut qu'un User puisse modifier/accéder à des propriétés de Recette par exemple obtenir la liste de toutes les recettes que cet utilisateur aura créé, on dira « **yes** ».

Ensuite il demande, de donner le nom du champ qui va représenter la/les recette(s) d'un utilisateur. On va garder donc « **recipes** ».

Pour la dernière question, il demande si on supprime un utilisateur ça supprimera toutes ses recettes liées. Est-ce que ses recettes peuvent rester orphelins ? En gros avoir des recettes sans utilisateurs. On avait dit qu'une Recette désormais devra d'office avoir un utilisateur donc on va mettre oui « **yes** » quand on supprime un User ça supprimera toutes ses Recettes.

Une fois finit, il nous dira qu'il a bien modifié nos deux entités.

Dans **Recipe** il a rajouté tout simplement un nouveau champ « **user** » avec ses getters et setters :

```php
#[ORM\ManyToOne(inversedBy: 'recipes')]
#[ORM\JoinColumn(nullable: false)]
private ?User $user = null;
```

Ce champ représentera un objet de type User, donc l'utilisateur qui aura créé cette recette.

Dans la classe « **User** », il nous aura rajouter plusieurs choses.

```php
/**
 * @var Collection<int, Recipe>
 */
#[ORM\OneToMany(targetEntity: Recipe::class, mappedBy: 'user', orphanRemoval: true)]
private Collection $recipes;

public function __construct()
{
    $this->recipes = new ArrayCollection();
}
```

Les collections sont des tableaux améliorés. On utilise principalement pour des objets.

« Les collections offrent une manière plus puissante et flexible de travailler avec des ensembles de données en PHP par rapport aux tableaux simples. Elles fournissent des fonctionnalités supplémentaires, une meilleure organisation du code et une meilleure sécurité, ce qui en fait un choix préféré dans de nombreuses applications. »

Il nous a aussi rajouté le getter et non le setter, parce qu'une liste on la modifie juste en ajoutant et supprimant des éléments.

```php
/**
 * @return Collection<int, Recipe>
 */
public function getRecipes(): Collection
{
    return $this->recipes;
}

public function addRecipe(Recipe $recipe): static
{
    if (!$this->recipes->contains($recipe)) {
        $this->recipes->add($recipe);
        $recipe->setUser($this);
    }

    return $this;
}

public function removeRecipe(Recipe $recipe): static
{
    if ($this->recipes->removeElement($recipe)) {
        // set the owning side to null (unless already changed)
        if ($recipe->getUser() === $this) {
            $recipe->setUser(null);
        }
    }

    return $this;
}
```

Une fois que c'est fini, vous pouvez faire une migration. Vous verrez que vous aurez des erreurs lors de la migration.

Est-ce que quelqu'un sait m'expliquer pourquoi ?

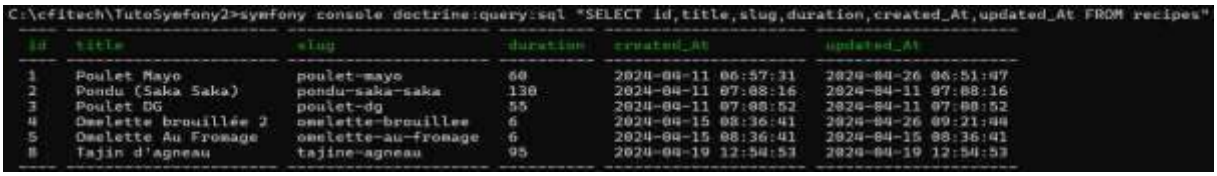
Vous verrez que votre site n'arrivera plus à créer de recette. Mais que dans votre base de données il aura bien rajouté **user_id** dans la table **recipes**.

On va régler ce problème un peu plus tard.

8.3.1 Commande SQL avec Symfony

On peut utiliser directement les commandes SQL dans notre terminal, pour par exemple afficher une recette ou toutes les recettes, créer, supprimer, modifier, etc. En gros utiliser du SQL pour faire des requêtes.

Pour pouvoir utiliser directement des commandes SQL par rapport à la base de données qu'on utilise, on peut taper la commande :



Vous pouvez supprimer une recette à partir d'ici par exemple.

Exos

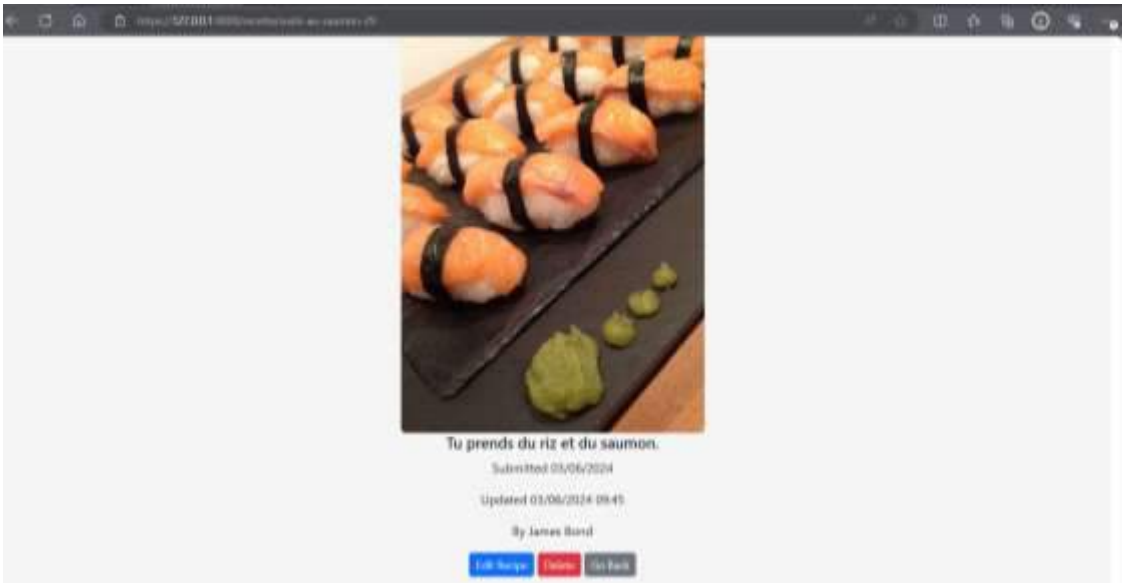
4) **Créez 2 utilisateurs en ligne de commande SQL sur symfony. Je vous impose un utilisateur : jamesbond@cfitech.be, [], jamesbond, James, Bond, datetime d'aujourd'hui, datetime d'aujourd'hui.**

5) **Vérifiez que vous avez bien user_id dans la table recipes en base de données en mode foreign key. Si pas refaite une migration.**

6) **Attribuez vos recettes aux utilisateurs**

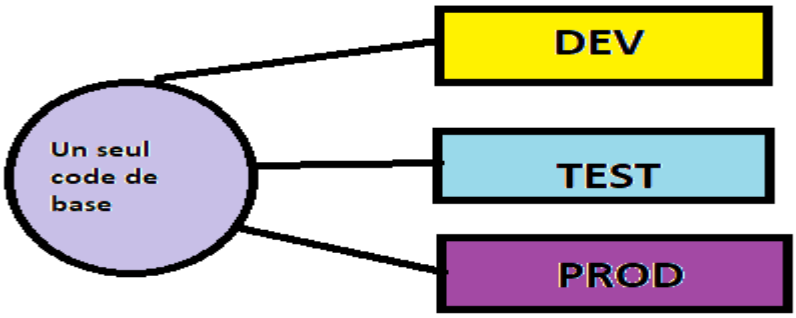
7) **Essayez d'afficher l'utilisateur qui a créé la recette :**

8) **Faites aussi pareil dans votre show, pour qu'on voit l'auteur de la recette :**



8.3.2 La notion d'environnement dans Symfony

Je vais vous présenter la notion d'environnement dans Symfony, vous avez déjà entendu parler du mode Dev (celui qu'on utilise car on développe l'application). Il y a aussi le mode Test et mode Prod.



La notion d'environnement c'est l'idée qu'un code de base peut être configurer différemment. Avec ce même code de base, en mode Dev j'ai des outils de débogage, en mode Test j'ai des outils pour

faire les Test (fonction test) et en mode Prod j'ai une application rapide en fonction des besoins et sécurisée.

C'est dans le fichier «.**env** » de votre projet que vous pouvez voir et modifier le mode. Si on met le mode prod, on va passer en mode production. Tout cela fonctionne grâce au système de configuration d'une application Symfony.

En passant en mode **prod** par exemple sur votre projet dans le fichier « **.env** », vous verrez que la Tool bar du Symfony profiler n'y sera plus.

Ensuite il faudra faire un cache clear dans la console :

- **symfony console cache:clear**

Ça permet de corriger certains bugs du au cache et aussi mettre à niveau votre projet, pour effacer les incohérences possibles. Dans notre cas, il permet d'actualiser le projet et les fichiers de configuration comme le « **.env** ».

Il se peut qu'après avoir fait la commande, il stop votre serveur donc il faudrait potentiellement le redémarrer.

Vous verrez que votre site en production ira beaucoup plus vite. Nous y reviendrons par après dessus.

Remettez votre site en mode dev dans le fichier d'environnement, n'oubliez pas de faire le cache clear.

Nous avons toujours notre problème lors de la création d'une recette au niveau du formulaire. Je rappelle que c'est parce qu'on a relié les deux entités « **Recipe** » et « **User** ».

Chapitre 9 : Authentification avec Symfony

9.1 Authentification et autorisation

Lorsqu'on parle de sécurité, en réalité on parle de 2 parties, la partie **authentification** et la partie **autorisation**.

Au niveau de la partie authentification, on vérifie tout simplement qui vous êtes.

Au niveau de la partie autorisation on vérifie est ce que vous avez les droits nécessaires afin d'effectuer différentes actions.

Au niveau de notre application, on pourra donner notre adresse email, de même que notre mot de passe. Et grâce à ces deux informations, on pourra vérifier qui nous sommes. Si ces identifiants sont corrects on pourra se connecter, dans le cas contraire on va nous dire que nos identifiants sont invalides.

Donc la partie authentification on vérifie seulement qui nous sommes.

Une fois authentifier on va pouvoir modifier que nos propres Recettes, et non celles des autres utilisateurs. C'est là que la partie autorisation joue.

Au niveau de Symfony toute la partie sécurité est générée par ce bundle « **SecurityBundle** ». Toute la documentation de ce bundle se trouve sur le site officiel de Symfony. Ce bundle tout ce qu'il fait, c'est intégrer le composant Security de Symfony, ce composant est composé de plusieurs autres sous composant :

```
> composer require symfony/security-core
> composer require symfony/security-http
> composer require symfony/security-csrf
> composer require symfony/security-guard
```

Exos

1) **La première chose qu'on va faire c'est de rajouter un onglet pour s'inscrire et un onglet pour se connecter dans notre barre de tache. (Register et Login en anglais)**

On va ensuite vouloir faire en sorte que quand un utilisateur est connecté il affiche dans la barre de navigation Ajouter une recette, Mon Compte (Account) et Se déconnecter (Logout). Et quand il n'est pas connecté, qu'il affiche S'enregistrer (Register/Sign Up) et Se loguer (Login).

Au niveau de Symfony, on peut accéder à l'utilisateur connecté en faisant juste « **{{ app.user }}** ». On va aller **modifier** le partials « **_nav.html.twig** » en mettant une **condition**. Si utilisateur existe donc différend de null, alors il affichera ce qu'il y aura dans la condition :

```
{% if app.user %}
<li class="nav-item ">
  <a class="nav-link {{ app.current_route == 'app_recipe_create' ? 'active' : '' }}" href={{ path
  ('app_recipe_create') }}>Add Recipe</a>
</li>
<li class="nav-item ">
  <a class="nav-link {{ app.current_route == 'app_account' ? 'active' : '' }}" href="#">Account</a>
</li>
<li class="nav-item ">
  <a class="nav-link" href="#">Logout</a>
</li>
{% else %}
<li class="nav-item ">
  <a class="nav-link {{ app.current_route == 'app_register' ? 'active' : '' }}" href="#">Sign Up</a>
</li>
<li class="nav-item ">
  <a class="nav-link {{ app.current_route == 'app_login' ? 'active' : '' }}" href={{ path
  ('app_login') }}>Login</a>
</li>
{% endif %}
```

Au niveau du lien pour login on va déjà mettre la route qu'on utilisera. Généralement les développeurs Symfony mettent comme route pour Login : « **{{ path('app_login') }}** ».

On va devoir créer cette route dans un Controller. Ici on ne créera pas un Controller comme on l'a fait pour **RecipeController** avec la commande « symfony console make:controller ».

On va faire via la commande « **make:security** » qui jusqu'à la version 6 de Symfony était make:auth :

- **symfony console make:security:form-login**

Il va nous poser des questions pour savoir de quel genre d'authentification on a besoin.

On nous demande quel nom de Controller pour notre classe. Ici on laissera par défaut « **SecurityController** ».

On nous demande est ce qu'on veut générer une URL pour la déconnexion, on va mettre **yes**.

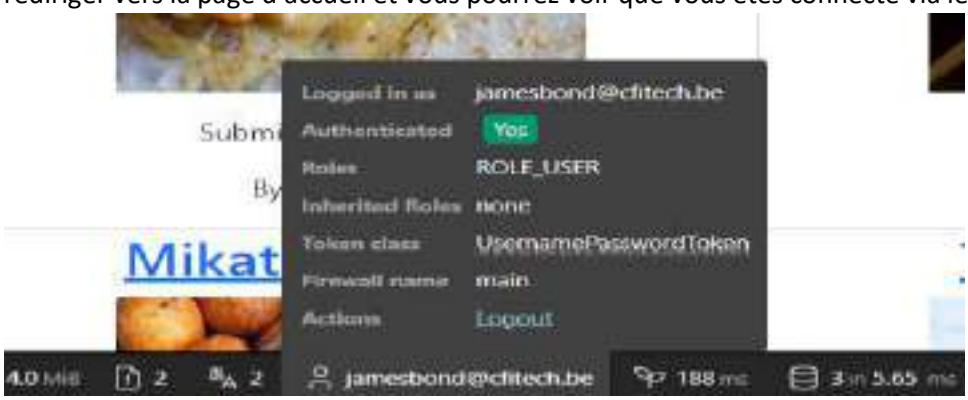
```
C:\cfitech\tutoSymfony>symfony console make:security:form-login

 Choose a name for the controller class (e.g. SecurityController) [SecurityController]:
 >

 Do you want to generate a '/logout' URL? (yes/no) [yes]:
 >

 Do you want to generate PHPUnit tests? [Experimental] (yes/no) [no]:
 >

 created: src/Controller/SecurityController.php
 created: templates/security/login.html.twig
 updated: config/packages/security.yaml

  Success!

 Next: Review and adapt the login template: security/login.html.twig to suit your needs.
```

On pourra voir qu'on a donc un nouveau Controller : **SecurityController**. Cette classe vient avec une méthode **login** et une méthode **logout**. Vous remarquez que la route pour la fonction login est bien **app_login** par défaut.

```php
src > Controller > SecurityController.php > ...
class SecurityController extends AbstractController
{
    /**
     * @Route("/login", name="app_login")
     */
    public function login(AuthenticationUtils $authenticationUtils): Response
    {
        // if ($this->getUser()) {
        //     return $this->redirectToRoute('target_path');
        // }

        // get the login error if there is one
        $error = $authenticationUtils->getLastAuthenticationError();
        // last username entered by the user
        $lastUsername = $authenticationUtils->getLastUsername();

        return $this->render('security/login.html.twig', ['last_username' => $lastUsername, 'error' => $error]);
    }

    /**
     * @Route("/logout", name="app_logout")
     */
    public function logout()
    {
        throw new \LogicException('this method can be blank - it will be intercepted by the logout key on your firewall.');
    }
}
```

Lancez votre site et normalement si vous cliquez sur le bouton login, vous tomberez sur cette page :

Connectez-vous ensuite avec jamesbond@cfitech.be et mot de passe **jamesbond.** Vous serrez rediriger vers la page d'accueil et vous pourrez voir que vous êtes connecté via le profiler Tool :



Vous remarquerez aussi que notre barre de navigation a changée :

Si on est connecté et qu'on tape par exemple dans le navigateur https://127.0.0.1:8000/login pour retourner sur la page login, il va nous montrer qu'on est connecté.

On va bien entendu faire en sorte que quand on tapera le lien du login ou qu'on fera un retour en arrière après s'être loggé, il nous retournera vers la page d'accueil. Parce qu'on ne doit pas avoir accès à la page de login alors qu'on est déjà connecté.

Il faudra aller dans **src/Controller/SecurityController** puis de **rajouter** le code qui vérifie si un utilisateur est connecté. Si c'est le cas, on donne un message flash d'erreur et on le redirige vers la page d'accueil « **home** » :

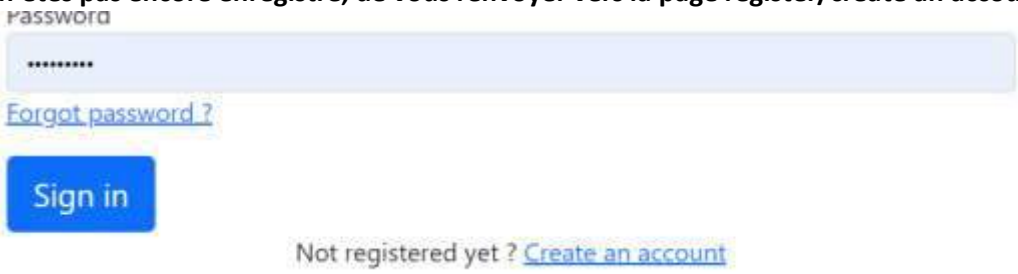
```
#[Route(path: '/login', name: 'app_login')]
public function login(AuthenticationUtils $authenticationUtils): Response
{
    if($this->getUser()){
        $this->addFlash('error', 'Already Connected !');
        return $this->redirectToRoute('home');
    }
    // get the login error if there is one
    $error = $authenticationUtils->getLastAuthenticationError();
```

Pour l'instant nous n'avons pas encore configuré le bouton pour se déconnecter (**logout**). Pour l'instant on pourra se déconnecter via le web profiler en cliquant sur Logout :

Exos

2) **Faites en sorte que quand on est connecté et qu'on clique sur Logout/se déconnecter de la barre de navigation, il se déconnecte.**
3) **Essayez de rajouter un petit lien en message de si on a oublié notre mot de passe :**

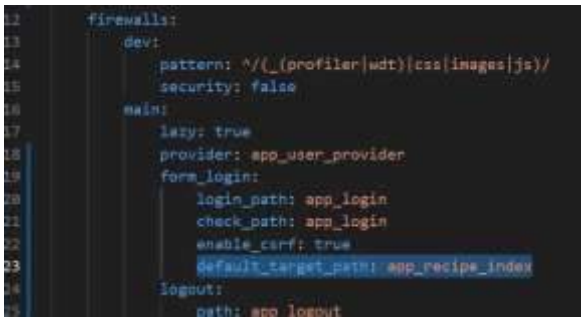
4) **Je vais aussi vous demander de rajouter une petite phrase en bas qui vous dis, si vous n'êtes pas encore enregistré, de vous renvoyer vers la page register/create an account :**



Essayer de centrer le message Not registered yet.

5) **Et pour finir essayer de centrer le formulaire login.**

On va faire en sorte qu'on soit redirigé vers la page de la liste des recettes quand on se connecte. Il suffit d'aller rajouter dans « **config/packages/security.yaml** » :



```
firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false
    main:
        lazy: true
        provider: app_user_provider
        form_login:
            login_path: app_login
            check_path: app_login
            enable_csrf: true
            default_target_path: app_recipe_index
        logout:
            path: app_logout
```

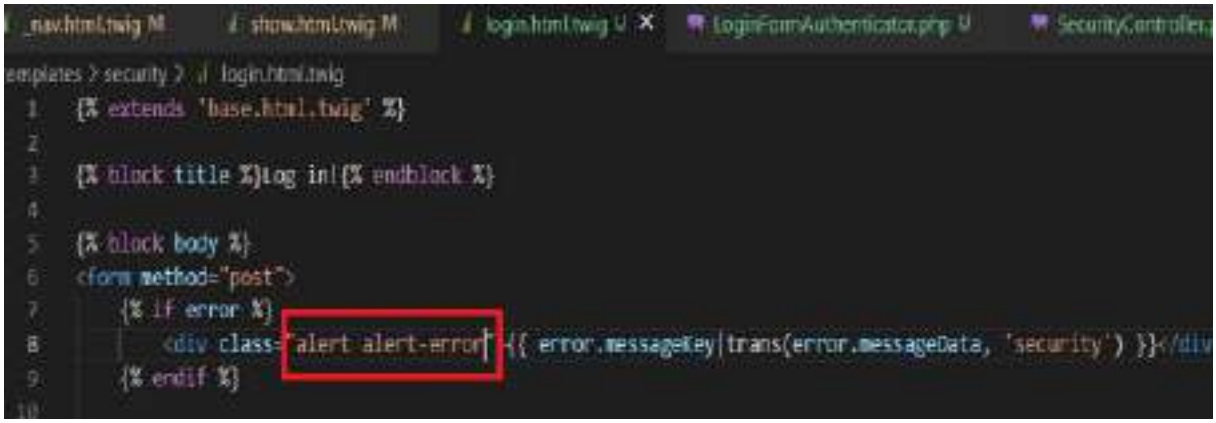
Vous pourrez faire un git commit.

Maintenant vous serez redirigé vers la liste des recettes quand vous vous connecterez mais plus tard on redirigera vers le profil de l'utilisateur quand on aura créé cette page.

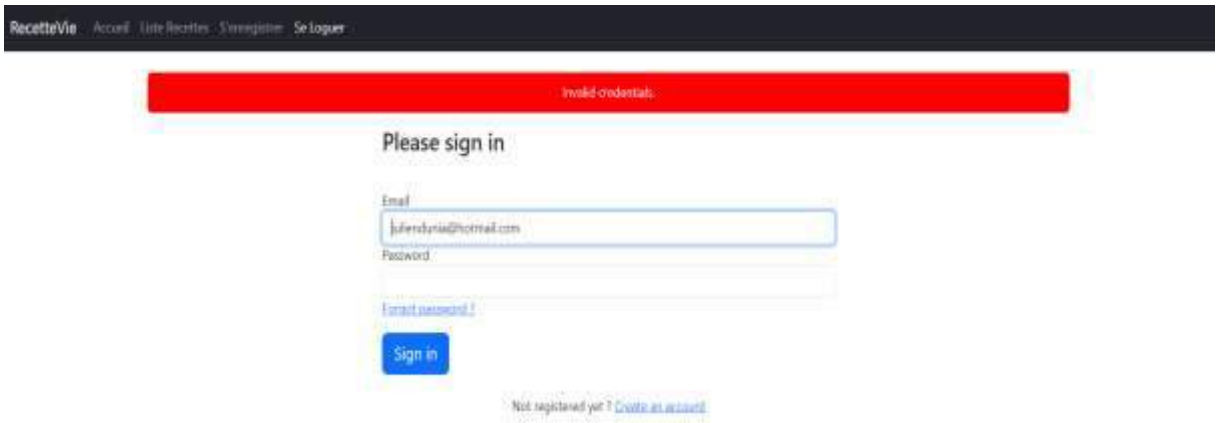
Par contre si vous introduisez un mauvais nom d'utilisateur ou mot de passe, il va vous sortir une erreur « **Invalid credentials** » :

On va un peu changer la couleur.

Dans le fichier « **templates/security/login.html.twig** » **remplacer** **alert-danger** par **alert-error**.



Vous obtiendrez normalement ce résultat.



Avant de continuer pour l'enregistrement on va un peu voire comment traduire « automatiquement » les choses.

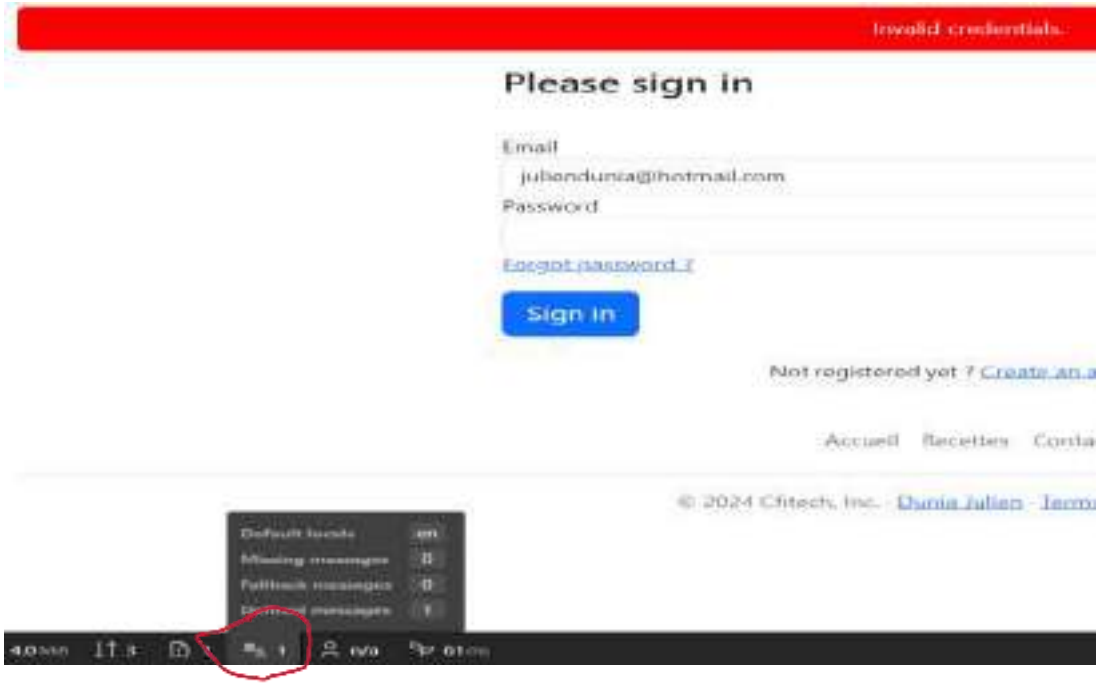
9.2 La traduction

Il existe plusieurs manières d'opérer pour la traduction de votre site, vous pouvez toutes les trouver dans la documentation officielle de Symfony dans Translations. Je vais vous donner un aperçu de comment ça peut fonctionner.

9.2.1 La traduction manuelle

Une des méthodes dont je vais vous parler, consiste à cibler manuellement toutes les phrases ou mots de votre site. ATTENTION ne pas confondre les mots de votre site avec ceux des données en base de données. Je m'explique, vos recettes sont en français dans votre base de données, ce ne sont donc pas des phrases/mots de votre site.

Retournons sur notre page de login, et on va changer ce message « **Invalid credentials** » qui venait quand on se trompait de mot de passe ou d'email.



On peut voir dans la debug bar de Symfony le « A » qui représente la partie traduction. Si vous cliquez dessus on peut retrouver ceci :

Ici ma langue locale par défaut c'est l'anglais d'où le « **en** ». Ce tableau récapitule les traductions manquantes selon lui dans votre page actuelle. On a le **domaine** qui est comme la catégorie là où se trouvera la traduction pour invalid credentials qui est « **security** ». Ensuite on a Message ID qui est le message d'origine et message Preview qui est censé avoir le message traduit en la langue qu'on souhaite.

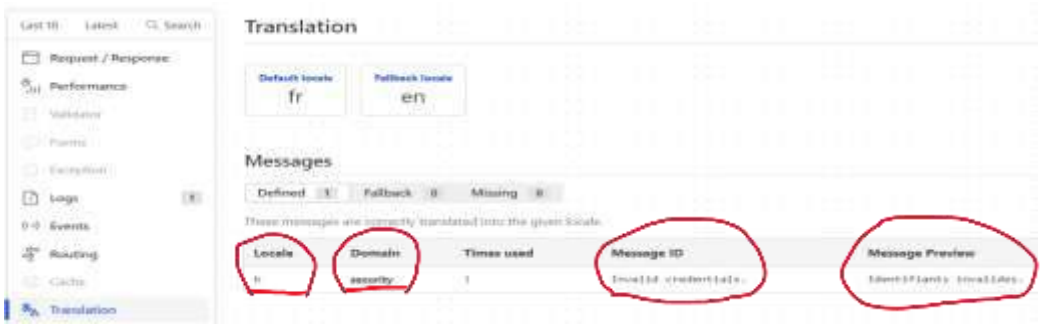
On va modifier ce message. Mais avant, on va d'abord rendre notre site avec la langue par défaut en français. Allez dans « **config/packages/translation.yaml** ». Ce fichier va nous permettre de piloter la traduction. On va mettre français pour la locale par défaut. La partie translator c'est pour dire ou ira nos fichiers de traductions et c'est aussi l'endroit providers ou vous pourrez mettre une traduction automatique mais on n'en est pas encore là.

```
config > packages > ! translation.yaml
framework:
    default_locale: fr
    translator:
        default_path: '%kernel.project_dir%/translations'
        fallbacks:
            - en
        providers:
```

Si déjà là vous allez sur votre site et que vous vous trompez lors de la connexion il vous enverra ce message normalement :

Identifiants invalides.

Si vous retournez dans translation, on pourra voir que maintenant la « **locale** » est en fr, le « **Domain** » c'est toujours « **security** », le « **Message ID** » représente le message d'origine et le « **Message Preview** » c'est pour l'instant la traduction qu'il nous donne automatiquement.



On peut bien entendu changer le message en décidant soi-même quelle traduction mettre. Pour ça il suffit d'aller dans le dossier translation à la racine de votre projet.

Il faut **créer** un fichier dans le dossier « **translations** ». Le fichier s'appellera **security.fr.yaml**, c'est un fichier de traduction. Ce n'est pas un hasard le nom qu'on donne. Le « **security** » vient du domaine et le « **fr** » vient du fait qu'on va faire un fichier pour les traductions en français.

Dans ce fichier on va mettre le « **Message ID** » qui représente je le rappelle le message de base et a côté on mettra le message que nous voulons qui soit affiché dans la langue fr. On va mettre un message en FR ('**Mot de passe ou email incorrect !**') :

```
translations > ! security.fr.yaml
1    Invalid credentials. : "Mot de passe ou email incorrect !"
```

ATTENTION, il faut que le Message ID soit exactement écrit comme il est montré. Donc ici j'ai mis « Invalid credentials. » avec le point à la fin. Si vous ne le mettez pas il ne reconnaitra pas.

Si ça ne marche pas il faut vider le cache en faisant :

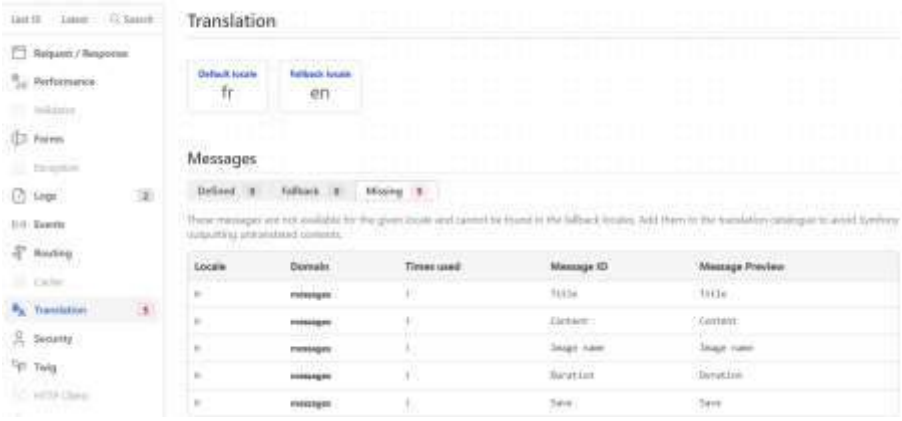
- **symfony console cache:clear**

Ce qui donne bien maintenant :

Faisons un autre exemple. Connectez-vous avec un utilisateur et allez dans la page de création d'une recette. Allons remodifier notre formulaire pour que tout soit dans un premier temps en anglais, enlever les labels qu'on avait mis pour que ce soit en français (Titre et Envoyer) :

```php
class RecipeType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('title', TextType::class)
            ->add('slug', HiddenType::class,[
                'required'=>false
            ])
            ->add('content',TextareaType::class)
            ->add('imageName')
            ->add('duration')
            ->add('save', SubmitType::class)
            ->addEventListener(FormEvents::PRE_SUBMIT, $this->autoSlug(...))
        ;
    }
```

Si vous allez dans translation sur votre site vous aurez ceci :



J'ai ceci par défaut quand je ne fais rien sur la page de création.

(Exos)

6) **Essayez de me traduire le formulaire pour qu'on voit ceci :**

Je rappelle qu'on ne peut toujours pas créer de recette.

On peut aussi faire plus propre de cette manière dans le RecipeType :

Et dans le fichier messages.fr.yaml :

Ça fonctionnera aussi. Attention par contre à bien mettre l'indentation et bien mettre les mêmes noms dans les 2 fichiers.

Voici une des manières de faire concernant les formulaires et messages.

Il faut savoir qu'on peut choisir tout ce qu'on veut traduire, par exemple si je veux traduire ma barre de navigation, il y a une manière de faire avec un tag en Twig qui s'appelle « **trans** ». Il suffit d'entourer la chaine de caractère qu'on veut traduire par :

- **{% trans %}** votre chaine de caractère à traduire **{% endtrans %}**

Il faut savoir qu'on peut faire aussi traduire de la même manière mais au lieu de mettre :

- {% trans %} votre chaine de caractère à traduire {% endtrans%}

On peut faire :

- **{{ 'Votre chaine de caractère à traduire' | trans }}**

C'est exactement la même chose.

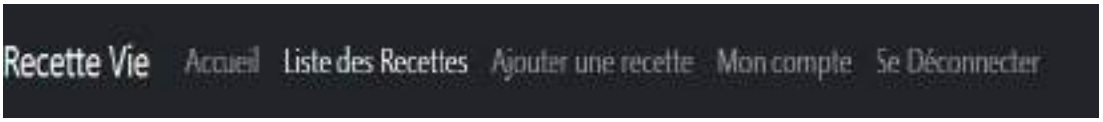
Rajoutez dans votre fichier **_nav.html.twig** à tous vos onglets ce tag la, voici un exemple :

```
<li class="nav-item ">
  <a class="nav-link {{ app.current_route  == 'home' ? 'active' : '' }}" href={{ path('home') }}>{{
  'navBar.home' |trans }}</a>
</li>
<li class="nav-item ">
  <a class="nav-link {{ app.current_route == 'app_recipe_show' or app.current_route ==
  'app_recipe_index' ? 'active' : '' }}" href={{ path('app_recipe_index') }}>{{ 'navBar.recipes' |
  trans }}</a>
</li>
{% if app.user %}
 <li class="nav-item ">
  <a class="nav-link {{ app.current_route == 'app_recipe_create' ? 'active' : '' }}" href={{ path
  ('app_recipe_create') }}>{{ 'navBar.addRecipe' |trans }}</a>
 </li>
 <li class="nav-item">
   <a class="nav-link {{ app.current_route == 'app_account' ? 'active' : '' }}" href='#'>{{ 'navBar
   account' |trans }}</a>
 </li>
```

La maintenant si vous allez sur n'importe quelle page vous pourrez voir dans le A de translation vos onglets qui attendent d'être traduit.

(Exos)

7) **Traduisez tous vos onglets en français de ce style.**
   Quand je suis connecté :



   Quand je suis déconnecté :

On reviendra sur la traduction automatique et la traduction via les url genre /fr ou /en un peu plus tard. Je vous conseille de continuer à travailler sur vos projets de côté pour bien maitriser. On apprend beaucoup de matière, le seul moyen d'y arriver seul à refaire les choses c'est de s'exercer.

Chapitre 10 : Inscription, système de mail et autorisations

10.1 Retour de l'ajout d'une recette

Avant de commencer à créer le formulaire d'enregistrement. Je vais d'abord faire en sorte qu'on puisse créer des recettes quand on se connecte avec un utilisateur.

Si vous essayez de créer un nouvelle Recette vous verrez une erreur pour l'instant.

Pour faire en sorte que la création de Recette fonctionne à nouveau. Il faut aller rajouter dans la **fonction create** de **RecipeController** ce bout de code :

```php
#[Route(path : '/recette/create', name : 'app_recipe_create')]
public function create(Request $request, EntityManagerInterface $em) : Response {
    $recipe = new Recipe;
    $form = $this->createForm(RecipeType::class, $recipe);
    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()){
        $recipe->setUser($this->getUser());
        $recipe->setCreatedAt(new DateTimeImmutable());
        $recipe->setUpdatedAt(new DateTimeImmutable());
        $em->persist($recipe);
        $em->flush();
        $this->addFlash('success', 'La recette '. $recipe->getTitle() .' a bien été créée');
        return $this->redirectToRoute('app_recipe_index');
    }
    return $this->render('recipe/create.html.twig', [
        'monForm' => $form
    ]);
}
```

On récupère l'utilisateur connecté grâce à **$this->getUser().** On récupère cette méthode parce qu'on hérite de **AbstractController**.

Là vous pourrez créer une nouvelle Recette sur votre site et on verra quel utilisateur l'aura fait.

10.2 Le Register

Passons maintenant à notre formulaire d'inscription.

Allons déjà **changer** nos **routes/liens** dans la **« _nav.html.twig »** les href de « **SignUp** » et « **Account** » path app_register pour le premier et app_account pour le second :

```
<li class="nav-item ">
  <a class="nav-link {{ app.current_route == 'app_account' ? 'active' :
  '' }}" href={{ path('app_account') }}>{% trans %}navBar.account{%
  endtrans %}</a>
</li>
<li class="nav-item ">
  <a class="nav-link" href={{ path('app_logout') }}>{% trans %}navBar.logout
  {% endtrans %}</a>
</li>
{% else %}
<li class="nav-item ">
  <a class="nav-link {{ app.current_route == 'app_register' ? 'active' :
  '' }}" href={{ path('app_register') }}>{% trans %}navBar.signUp{%
  endtrans %}</a>
</li>
```

Dans le « **security/login.html.twig »** on changera aussi le href de « Create an account ».

```
<div class="mb-3" align="center">
        Not registered yet ? <a href={{ path('app_register') }}> Create an account </a>
    </div>
```

Ici on aurait pu créer un controller RegistrationController avec symfony console make:controller puis créer une fonction register qui aura comme nom de route app_register etc.

Mais avec Symfony qui facilite certaines choses, tout comme on a fait avec **User** et **Auth**, on a une commande Symfony console qui permet directement de créer un formulaire d'enregistrement.

Avant de commencer on va d'abord installer un composant qui permet la vérification des emails :

- **composer require symfonycasts/verify-email-bundle**

Pour créer donc ce formulaire de registration avec Symfony, il suffit donc de faire :

- **symfony console make:registration-form**

On nous demande premièrement est ce qu'on veut rajouter une annotation **UniqueEntity**, en gros cette annotation va nous assurer qu'on ne puisse pas avoir deux utilisateurs avec la même adresse email (UserIdentifier). **YES**.

On nous demande ensuite est ce qu'on veut envoyer un mail de confirmation une fois que l'utilisateur s'est enregistré. **YES,** il nous dira d'installer un composant pour se faire.

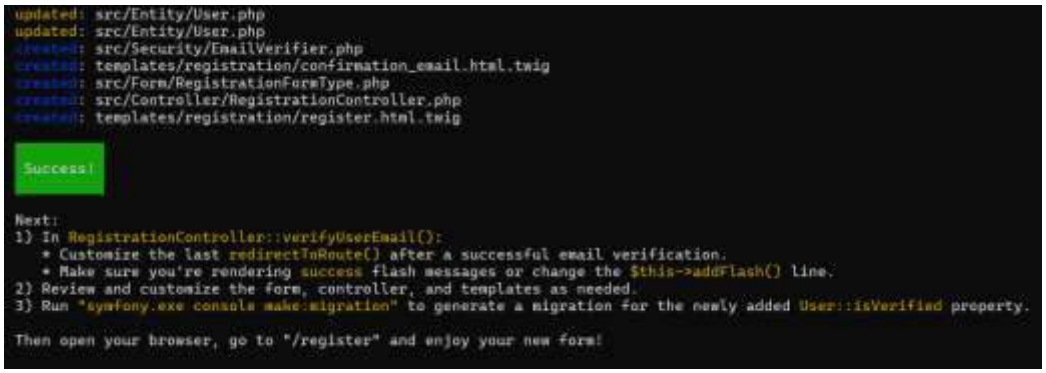
On nous demande si on veut inclure user_id dans le lien. **NO**

On nous demande le mail qui sera utilisé pour envoyer un mail de confirmation. Vous pouvez mettre **noreply@recettevie.be.** On pourra le changer si on veut.

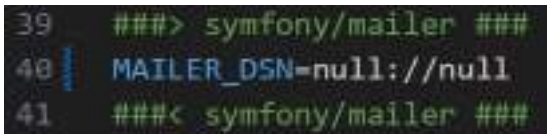
On nous demande quel nom sera associé à cette email. On pourra mettre **RecetteVie.** On pourra changer si on veut.

On nous demande pour finir si on veut connecter automatiquement l'utilisateur après inscription, on met **NO**.

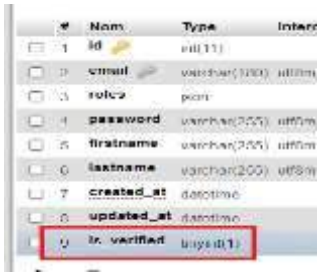
Il vous demandera aussi vers quel page, on sera redirectionné, vous prenez **app_login**.(20)



Une fois terminer on pourra voir qu'il a créé plusieurs fichiers. Il faudra ensuite aller dans le fichier « **.env** » pour **décommenter** la ligne concernant le mail :



On doit maintenant **faire** une **migration** pour prendre en compte les changements effectués surtout dans **User**, avec l'ajout d'un champ « **is_verified** ».



Si vous avez des problèmes de migration, vous pouvez faire une migration d'une version ciblée avec cette commande :

symfony console doctrine:migrations:diff

symfony console doctrine:migrations:execute DoctrineMigrations\leNomDeVotreFichierVersion

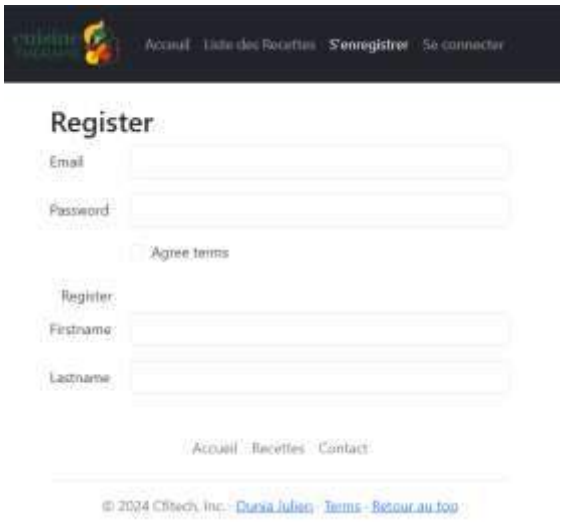
Vérifier si votre page **Register** fonctionne en cliquant sur Register, attention il faut être déconnecté :

Maintenant que notre base de données est à jour et que notre page Register fonctionne, on va essayer de **rajouter** à cette dernière les autres **champs** dont on a besoin pour s'enregistrer.

Pour ça, il faut aller dans le **fichier RegistrationFormType** dans le **dossier Form**. On **ajoute firstname** et **lastname** :

```
src > Form > RegistrationFormType.php > RegistrationFormType > buildForm
17      public function buildForm(FormBuilderInterface $builder, array $op
18      {
19          $builder
20              ->add('firstname')
21              ->add('lastname')
22              ->add('email')
23              ->add('agreeTerms', CheckboxType::class, [
24                  'mapped' -> false,
25                  'constraints' => [
26                      new IsTrue([
27                          'message' => 'You should agree to our terms.',
28                      ]),
29                  ],
```

On peut retourner sur le site pour voir ce que ça a fait et on verra que firstname et lastname se retrouve après register :



On va modifier l'agencement du formulaire et changer le bouton **Register** dans **register.html.twig** :

```
12      {{ form_start(registrationForm) }}
13          {{ form_row(registrationForm.firstname) }}
14          {{ form_row(registrationForm.lastname) }}
15          {{ form_row(registrationForm.email) }}
16
17          {{ form_row(registrationForm.plainPassword, {
18              label: 'Password'
19          }) }}
20          {{ form_row(registrationForm.agreeTerms) }}
21
22          <button type="submit" class="btn btn-primary">Register</button>
23      {{ form_end(registrationForm) }}
```

Avant de continuer, on va voir un peu une manière de traduire dynamiquement.

10.2.1 La traduction dynamique (part1)

Il y a une manière dynamique de générer les traductions. Symfony nous fournit une commande qui va scanner l'ensemble de votre code source et générer toutes les clés de traductions que vous avez :

- **symfony console translation:extract --dump-messages fr**

L'avantage c'est qu'il va aussi chercher pour les composants security ou autres, pour les erreurs de validations et vous les afficher. On peut voir d'ailleurs qu'il organise par domaine, et on peut constater que dans le domaine « **validators** » on a nos messages qu'on avait déjà mis en français.

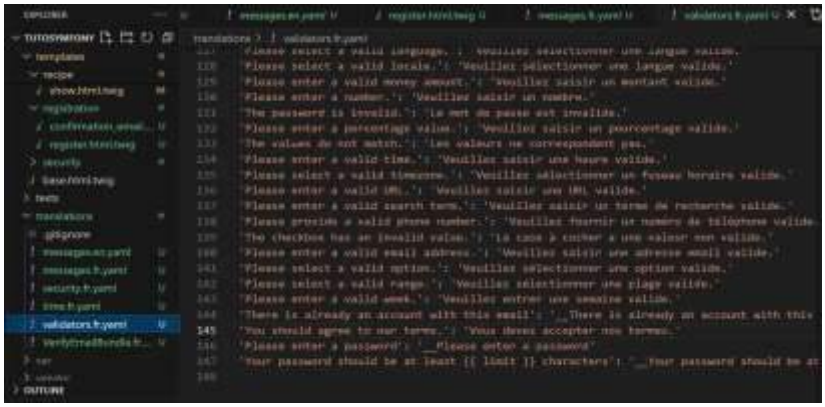
Avant de continuer allons enlever ces messages qu'on avait écrit manuellement en français, on va laisser les messages par défauts. J'ai aussi enlevé le « **(locale:fr)** » à côté du « **ago** » qu'on avait utilisé dans « **recipe/index.html.twig** » pour la date de création. Vu qu'on a mis notre locale en fr dans notre projet, il nous le traduit en français. On peut aussi enlever les message, minMessage et maxMessage qu'on avait mis pour les « **notBlank** », « **lenght** », « **positive** » etc. parce qu'il les traduira automatiquement pour nous.

Cette commande, que vous avez tapé, fait juste un debug, on a utilisé « **--dump** ». Et ça nous a listé les clés de traduction (les Message ID) qu'il a trouvé dans notre code.

On va maintenant lui demander d'extraire tout ça dans des fichiers au format yaml. Pour faire cela il suffit de faire une seconde commande :

- **symfony console translation:extract --force fr --format=yaml**

Et là il nous créera plusieurs fichiers dans le dossier translations, et traduira déjà une bonne partie. Il reconnaitra aussi tous les messages que vous voulez traduire dans les templates avec le tag **|trans**. Il faudra par contre traduire manuellement toutes les traductions ou vous voyez __.

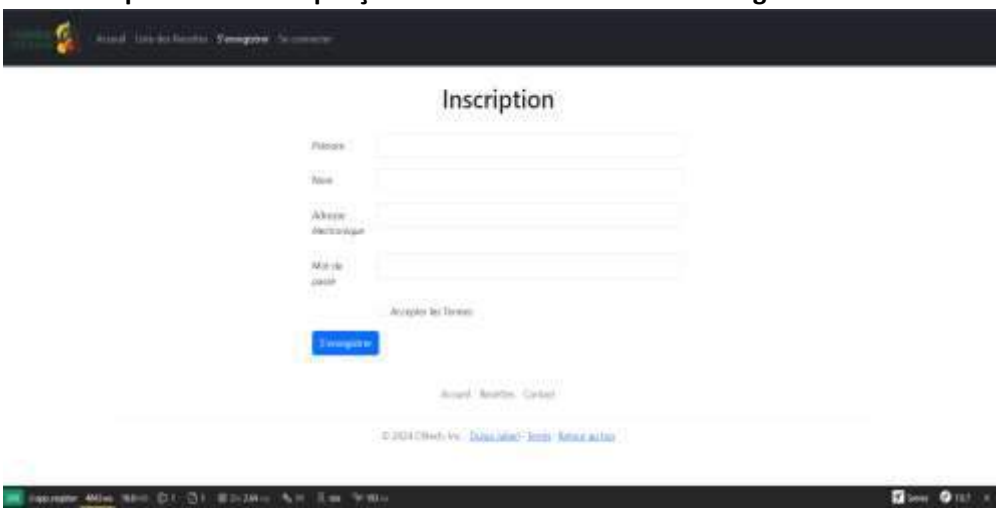


Même si vous rajoutez entre temps des nouveaux mots à traduire et que vous faites la commande, il ne supprimera pas les traductions que vous avez faits. On reviendra encore sur la traduction plus tard avec url /fr et /en.

Si on retourne sur notre page Register et qu'on essaye de s'enregistrer ça ne va pas marcher parce qu'il n'arrive pas à créer de date de création et modification pour User. On avait eu ce même problème quand on créait des recettes… Réglez ce problème.

Exos

1) **Essayez de faire en sorte que ça nous crée bien une date de création et de modification pour un User quand on s'enregistre. (Vérifiez en Base de Données s'il a créé l'utilisateur)**
2) **Essayez de centrer ce formulaire comme on l'a fait pour les autres (login, etc.).**
3) **Essayez de traduire ce formulaire en utilisant les bonnes méthodes de traduction. N'oubliez pas de vérifier que ça bien traduit surtout les messages de contraintes.**



4) **Ajoutez des contraintes pour tous les champs, ils ne peuvent pas être vide :**

5) **Ajoutez ces contraintes avec les traductions :**

10.3 Système de mail

Maintenant votre formulaire fonctionne, vous pouvez essayer d'enregistrer un utilisateur. Vous verrez que dans votre base de données il aura bien été enregistré mais le site va retourner une erreur parce qu'on n'a pas encore de page de profil (Account).

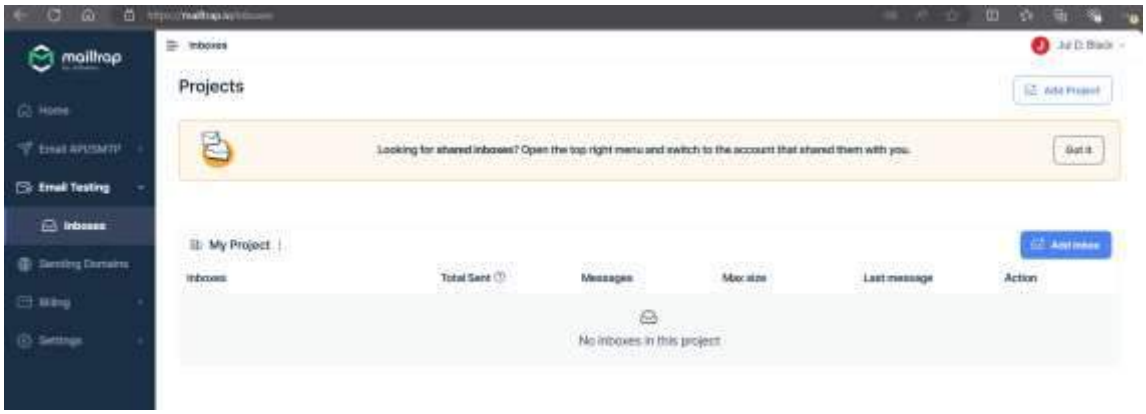
Exos

6) **Essayez de régler ce problème.**

On reviendra sur la page de profil plus tard.

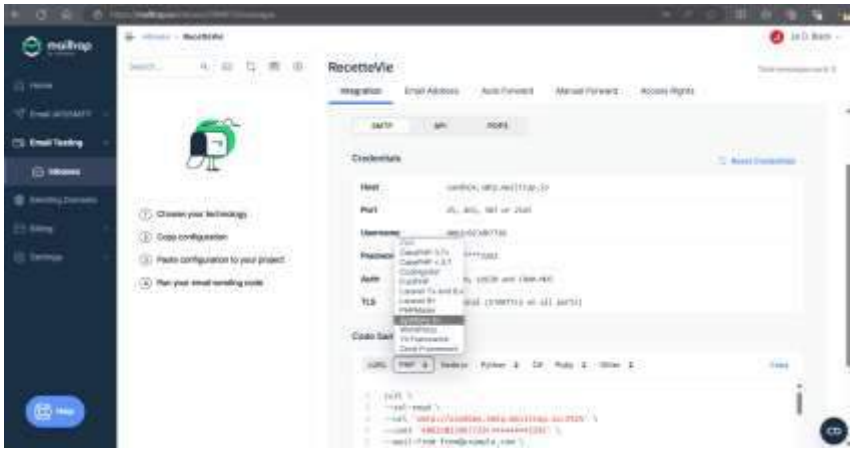
On n'a pas non plus géré la partie mail de confirmation. Pour y remédier, je vais vous montrer un service vraiment pas mal pour tester la vérification du mail.

Allez sur votre navigateur et tapez **mailtrap**. **Inscrivez-vous** sur ce site. Une fois inscrit et connecté, vous aurez une interface de boite mail (cliquez sur sandbox) :



Attention il faut supprimer la boite déjà existante par defaut. Vous pourrez ensuite cliquer sur « **Add Inbox »** et créez la boite **« RecetteVie »** puis sauvegarder.

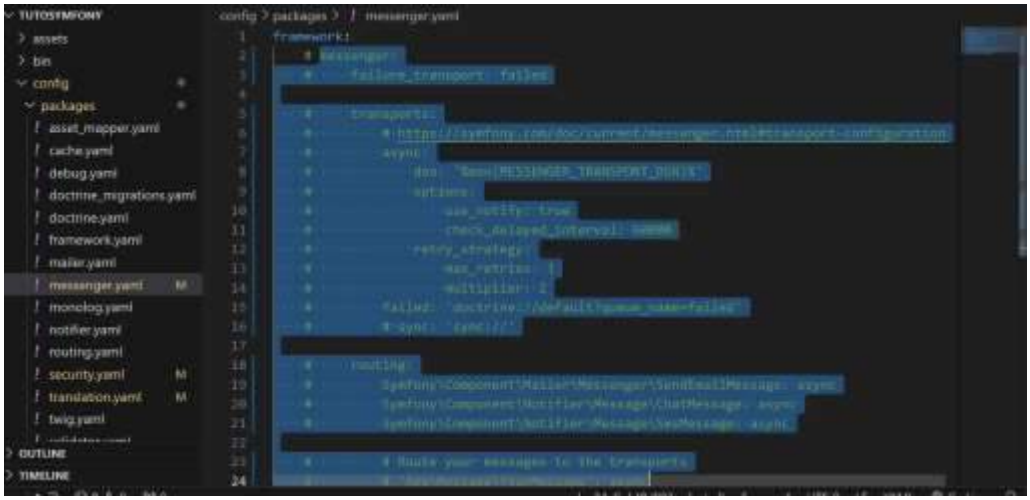
Si vous cliquez sur la boite, donc **RecetteVie,** que vous venez de créer, il vous donne des informations. Descendez un peu et un moment dans « **Code Sample** ». On va sélectionner notre langage de programmation PHP ensuite notre Framework Symfony 5+ :



Vous copiez la ligne de code qu'on vous donne en PHP Symfony 5+.

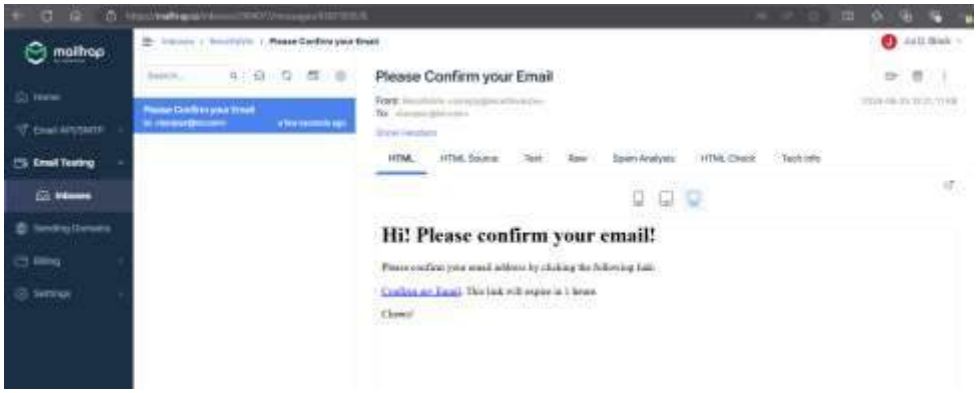
Vous allez la mettre dans le fichier « **.env »** à la place de celle qu'on avait décommenté :

Suite à un conflit avec « **messenger.yaml »** depuis Symfony 6, on va devoir aller mettre en commentaire les configurations provenant de ce fichier. Il faut aller dans **config/packages/messenger.yaml** et mettre tout en commentaire après Framework :

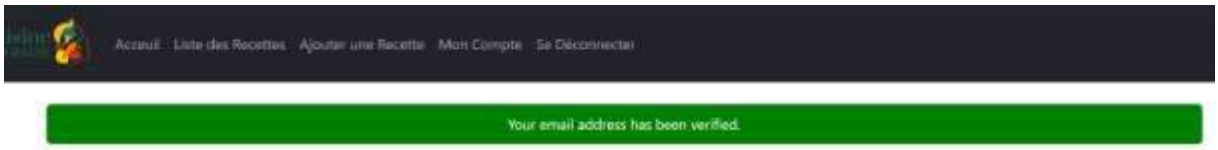


Pour finir il faudra ensuite aller dans le « **.env »** mettre en commentaire cette ligne :

La maintenant tenter de recréer un utilisateur. Vous n'aurez plus de problèmes. Si vous allez sur mailtrap, vous verrez que vous aurez reçu un mail de confirmation.



En cliquant sur **Confirm my Email**, vous verrez comme par magie, il vous redirigera vers votre page Login. Là vous vous connecterez avec cette adresse mail. Et vous verrez qu'il vous connectera bien avec un message comme quoi votre mail a bien été vérifié.



Vous pouvez aller voir en base de données, vous verrez bien is_verified a true dans le dernier utilisateur.

Vous pouvez modifier l'adresse mail, le nom et l'objet du mail dans **RegistrationController**

```
$entityManager = $this->getDoctrine()->getManager();
$entityManager->persist($user);
$entityManager->flush();

// generate a signed url and email it to the user
$this->emailVerifier->sendEmailConfirmation('app_verify_email', $u
    (new TemplatedEmail())
        ->from(new Address('noreply@pint.com', 'Pint'))
        ->to($user->getEmail())
        ->subject('Please Confirm your Email')
        ->htmlTemplate('registration/confirmation_email.html.twig'
);
```

Bien entendu vous pourrez aussi modifier le **message du mail**, ce sera dans le **templates confirmation.email.html.twig** qui se trouve dans **registration** :

```
templates > registration > ! confirmation_email.html.twig
<h1>Hi! Please confirm your email!</h1>

<p>
    Please confirm your email address by clicking the following link: <br><br>
    <a href="{{ signedUrl }}">Confirm my Email</a>.
    This link will expire in {{ expiresAtMessageKey|trans(expiresAtMessageData, 'VerifyEmailBundle') }}.
</p>

<p>
    Cheers!
</p>
```

Pour changer le temps qu'il faut avant que le message de confirmation expire il faudra aller créer dans **config/packages** un fichier **verify_email.yaml** et ensuite y mettre ce que vous voulez comme temps en seconde (86400 secondes c'est 24h) :

```
config > packages > ! verify_email.yaml
symfonycasts_verify_email:
    lifetime: 86400
```

Une fois finit stoppez votre serveur, faites un **cache:clear** et relancez le serveur. Avant de tester.

Quand vous avez confirmé votre mail, vous avez été redirigé vers la page **Register**. Vous pouvez changer ça. On veut rediriger vers la page d'accueil. Allez **modifier** dans **RegistrationController** :

```
// @TODO Change the redirect on success and handle or remove the flash message in your templates
$this->addFlash('success', 'Your email address has been verified.');

return $this->redirectToRoute('app_recipe_index');
```

10.4 Autorisations

On va mettre une **redirection** au cas où un utilisateur connecté tombe sur la page **Register**. Toujours dans **RegistrationController**, on va le rediriger vers sa page de profile :

```
src > Controller > RegistrationController.php > RegistrationController > register
20    class RegistrationController extends AbstractController
25
26        #[Route('/register', name: 'app_register')]
27        public function register(Request $request, UserPasswordHasherInterfa
28        {
29            if($this->getUser()){
30                $this->addFlash('error', 'Already Connected !');
31                return $this->redirectToRoute('app_account');
32            }
33            $user = new User();
34            $form = $this->createForm(RegistrationFormType::class, $user);
```

Maintenant faites en sorte qu'une fois enregistrer on est redirigé vers login et on nous dit bien qu'il faut aller checker sa boite mail pour cliquer sur le lien de confirmation.

On va un peu améliorer le code, on veut faire en sorte que si un utilisateur vient de s'enregistrer, tant qu'il n'a pas confirmé son mail, il ne pourra pas créer de recette.

Dans un premier temps allons dans **src/Controller/RecipeController.php** et rajoutons au tout début de la méthode **index()** :

```
        if($this->getUser()){
            /**
             * @var User
             */
            $user = $this->getUser();
            if(!$user->isVerified()){
                $this->addFlash("info","Your email address is not verified !");
            }
        }
```

En faisant ça, on aura un message flash quand un compte ne sera pas vérifié sur notre page de liste de recette.

Allons maintenant dans **templates/layouts/partials/_nav.html.twig** et allons rajouter une condition pour le chemin vers la page de création :

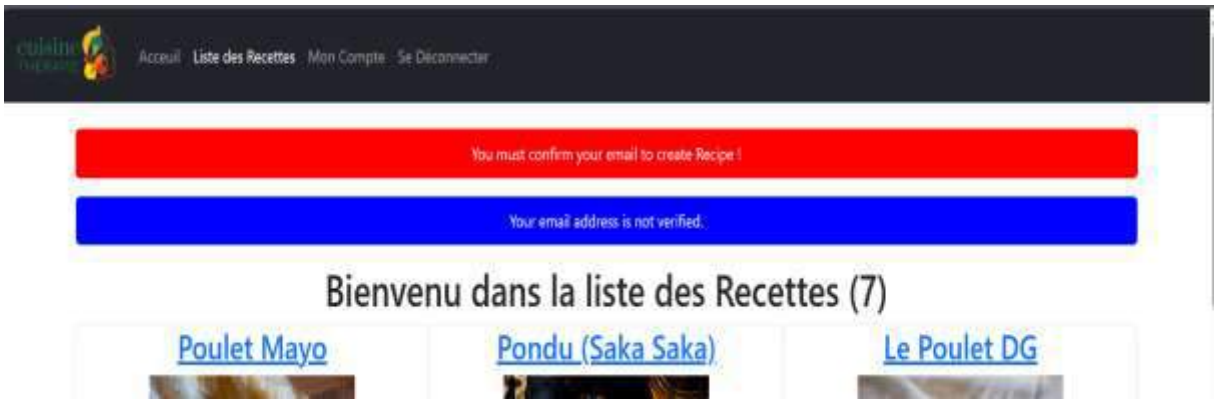
```
{% if app.user.isVerified %}
  <li class="nav-item ">
  <a class="nav-link {{ app.current_route == 'app_recipe_create' ? 'active' : '' }}" href={{ path
  ('app_recipe_create') }}>{{ 'navBar.addRecipe' |trans }}</a>
  </li>
{% endif %}
```

Ici on rajoute juste une condition qui va vérifier si l'utilisateur a bien été vérifié, si oui il montrera l'onglet concernant la création de recette sinon il ne montrera pas.

On va aussi quand même rajouter dans **RecipeController** dans la méthode **create**() au tout début ce bout de code :

```php
#[Route(path : '/recette/create', name : 'app_recipe_create')]
public function create(Request $request, EntityManagerInterface $em) : Response {
    if ($this->getUser()){
        if (!$this->getUser()->isVerified()) {
            $this->addFlash('error', 'You must confirm your email to create Recipe !');
            return $this->redirectToRoute('app_recipe_index');
        }
    }else{
        $this->addFlash('error', 'You must login to create Recipe !');
        return $this->redirectToRoute('app_login');
    }

    $recipe = new Recipe;
```

Il vérifie d'abord si l'utilisateur est connecté ou pas. S'il est connecté, Il va faire en sorte que si on essaye de passer par l'url en tapant la route de la création d'une Recette alors qu'on n'a pas confirmé son mail, il nous le dit avec un message flash et il nous redirige vers la page d'accueil.



Sinon il nous demande de nous connecter en nous redirigeant vers la page login.

On est clairement dans la partie autorisations ici.

Exos

Avant, allez en Base de données et rendez vos 2 premiers utilisateurs is_verified en mettant à 1.

7) **Toujours dans le RecipeController, vous pouvez aussi rajouter ce bout de code dans les méthodes edit et delete en modifiant le message.**

8) **Essayez de rajouter aussi des conditions pour qu'un utilisateur qui n'a pas créé une recette ne puisse ni la modifier ni la supprimer. En gros si l'utilisateur JamesBond a créé une recette, l'utilisateur JackBauer ne peut ni modifier ni supprimer la recette que JamesBond a créé.**

Voici ce que vous aurez dans la méthode edit et il faudra faire l' équivalent dans delete.

```php
#[Route(path : '/recette/{id}/edit', name : 'app_recipe_edit')]
public function edit(Recipe $recipe, Request $request, EntityManagerInterface $em) : Response {
    if ($this->getUser()){
        if (!$this->getUser()->isVerified()) {
            $this->addFlash('error', 'You must confirm your email to edit a Recipe !');
            return $this->redirectToRoute('app_recipe_index');
        }else if ($recipe->getUser()->getEmail() !== $this->getUser()->getEmail()){
            $this->addFlash('error', 'You must to be the user '. $recipe->getUser()->getEmail() . ' to edit
            this Recipe !');
            return $this->redirectToRoute('app_recipe_index');
        }
    }else{
        $this->addFlash('error', 'You must login to edit a Recipe !');
        return $this->redirectToRoute('app_login');
    }
```

Maintenant on recevra un message si j'essaye de supprimer ou modifier une Recette qui n'a pas été créée par un même utilisateur.

Exos

9) **Créez deux utilisateurs via le formulaire d'inscription, n'oubliez pas confirmer l'inscription en allant voir dans mailtrap. Une fois que votre inscription est totalement terminée, créer deux Recettes pour chacun de ces 2 utilisateurs. Testez ensuite si vous pouvez supprimer ou modifier en vous connectant sur l'utilisateur différend de son créateur.**
10) **Essayez de faire en sorte que quand vous allez sur le détail d'une Recette, si on n'est pas connecté, qu'on est connecté mais pas vérifié ou si on est connecté avec un utilisateur différent de celui qui a créé la recette, on ne voit ni le bouton edit ni le bouton delete.**

Astuce 1, allez voir ce qu'on a fait dans la navbar.
Astuce 2, en Twig quand on veut comparer deux choses si elles sont égales dans une condition on fait de cette manière-là :

En PHP : $voiture1 === $voiture2

En TWIG : voiture1 is same as (voiture2)

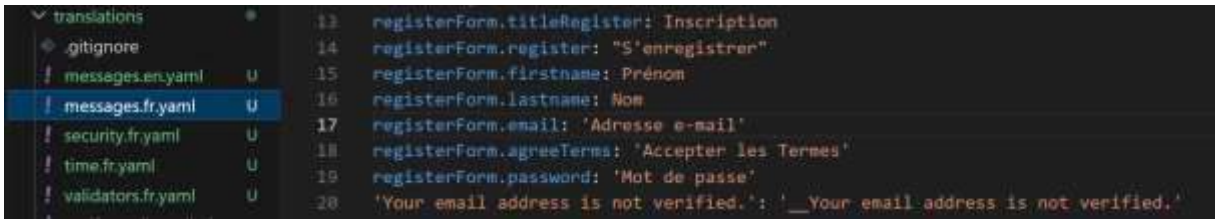
10.4.1 La traduction dynamique (part2)

Avant de continuer on va voir comment traduire aussi du coté de PHP. On a vu que par twig pour rendre une chaine de caractère traduisible il suffit de mettre le tag « **|trans** ». On a vu aussi que dans nos formulaires, en mettant le label, il arrivait à comprendre qu'on peut traduire cela. Mais comment faire pour traduire nos messages dans nos controller par exemple ? C'est ce qu'on va voir, grâce à une interface « **TranslatorInterface** ».

```php
use Symfony\Component\Routing\Attribute\Route;
use Symfony\Contracts\Translation\TranslatorInterface;

class RecipeController extends AbstractController
{

    //Metadonnées avec les attributs, depuis PHP 8
    #[Route('/recette', name: 'app_recipe_index')]
    public function index(Request $request, EntityManagerInterface $em, TranslatorInterface $translator):
    Response
    {
        if ($this->getUser()){
            if (!$this->getUser()->isVerified()){
                $this->addFlash('info', $translator->trans('Your email address is not verified.'));
            }
        }
```

On ajoute l'import, on le met en paramètre de la méthode et on l'utilise grâce à sa méthode trans().

La maintenant il suffit de faire la commande de symfony que je vous ai donné qui permet de trouver tous les messages qu'on peut traduire sur votre site. Et vous verrez qu'il l'a ajouté le message qu'on a mis entre parenthèse.



Il n'y a plus qu'à traduire en remplaçant le message commençant avec « __ ».

On voit que ça m'a bien traduit le message flash maintenant dans la langue locale de mon projet.

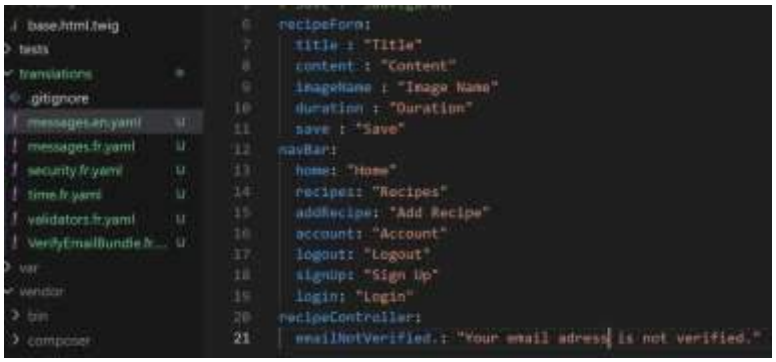
De base il est fortement recommandé d'utiliser les 'clés' comme on l'a fait pour registerForm, navBar et recipeForm. C'est pratique pour se retrouver. Il n'est pas recommandé de créer des traductions en prenant la langue par default (Anglais) comme clé, dans le sens ou si on veut changer le message d'origine c'est mieux de passer par un fichier « **messages.en.yaml** ».

Je vais un peu changer le message flash et utiliser un système de « clé » histoire de me retrouver d'où vient ce message.

Maintenant si je refais la commande de traduction, je peux aller voir dans messages.fr.yaml :

C'est plus pratique pour se retrouver aussi.

Si je veux mettre mon site en anglais « en », je peux faire le même procédé dans messages.en.yaml :



On reviendra une dernière fois sur les traductions plus tard.

10.5 Reset Password

On va faire en sorte que quand tu cliques sur le lien « **forgot password ?**», on atterrit sur une page qui nous demande notre email pour pouvoir reset le mot de passe.

Dans Symfony on a une commande qui permet justement de créer ce reset password.

Il faut d'abord installer un composant :

- **composer req symfonycasts/reset-password-bundle**

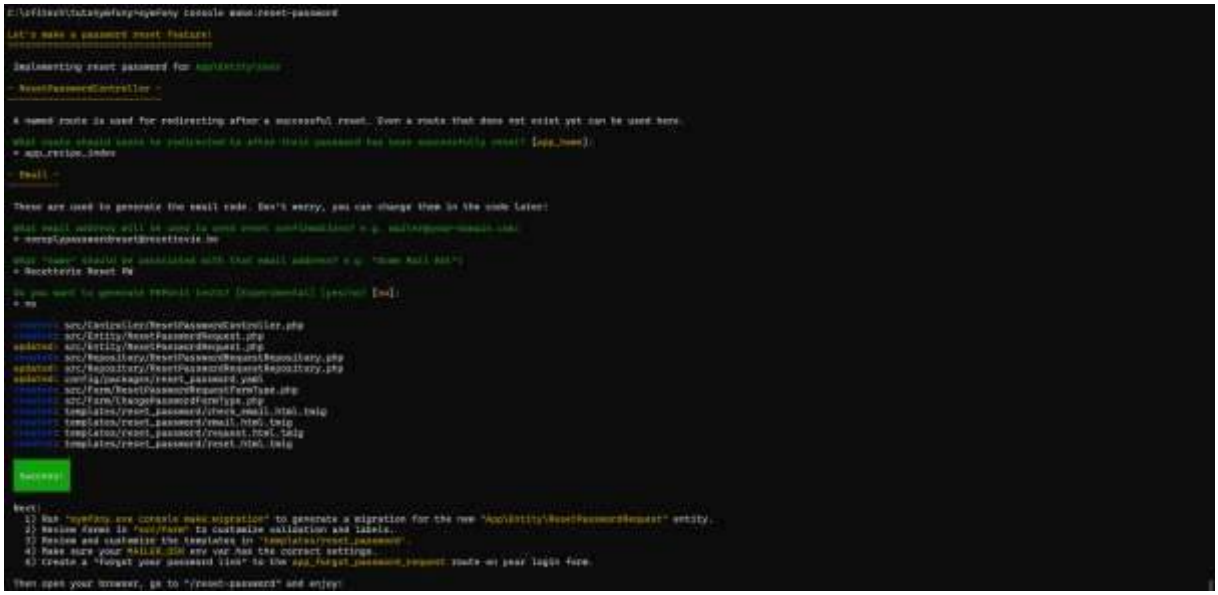
On pourra ensuite utiliser la commande Symfony :

- **symfony console make:reset-password**

On nous demande vers quelle route les utilisateurs devraient être redirigé. On va garder la page d'accueil (**app_recipe_index**).

Quel email serait utiliser pour les mails de réinitialisation. noreplypasswordreset@recettevie.be, on aurait pu mettre n'importe quoi.

Quelle sera son nom, on va mettre **RecetteVie Reset PW.** Et no pour la question sur les tests.



Vous voyez qu'il a fait des mises à jour sur plusieurs fichiers et qu'il a créé beaucoup de fichier. Après « Success ! » il vous donne ce qu'il faudra faire pour que ce soit fonctionnel.

Exos

11) Faites ce qui est demandé pour faire en sorte que tout fonctionne.

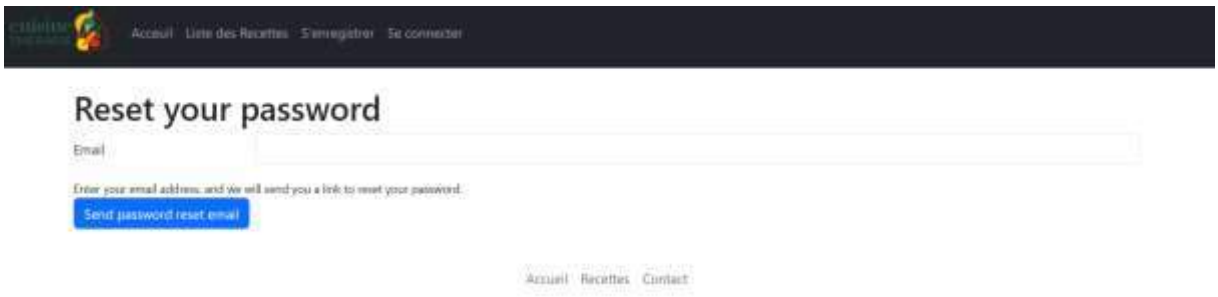
Voici ce qu'il fallait faire :

La migration de la nouvelle table.

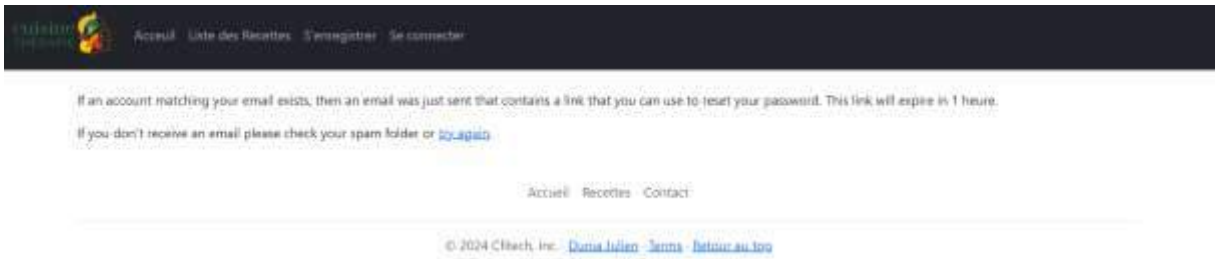
On peut ensuite aller mettre dans « **login.html.twig** » la route « **app_forgot_password_request** » pour le lien « **Forgot password ?»** qu'on peut bien entendu traduire si on le veut.

```
    <div class="mb-3">
        <a href={{ path('app_forgot_password_request') }}> {{'Forgot password ?'|trans}} </a>
    </div>

    <button class="btn btn-lg btn-primary" type="submit">
        {{'Sign in'|trans}}
    </button>
    </div>
</div>
</form>
<br>
<div class="mb-3" align="center">
    {{'Not registered yet ?'|trans}} <a href={{ path('app_register') }}> {{'Create an account'|trans}} </a>
</div>
```

Et là vous pourrez cliquer donc sur le lien qui vous redirigera sur cette page ci :

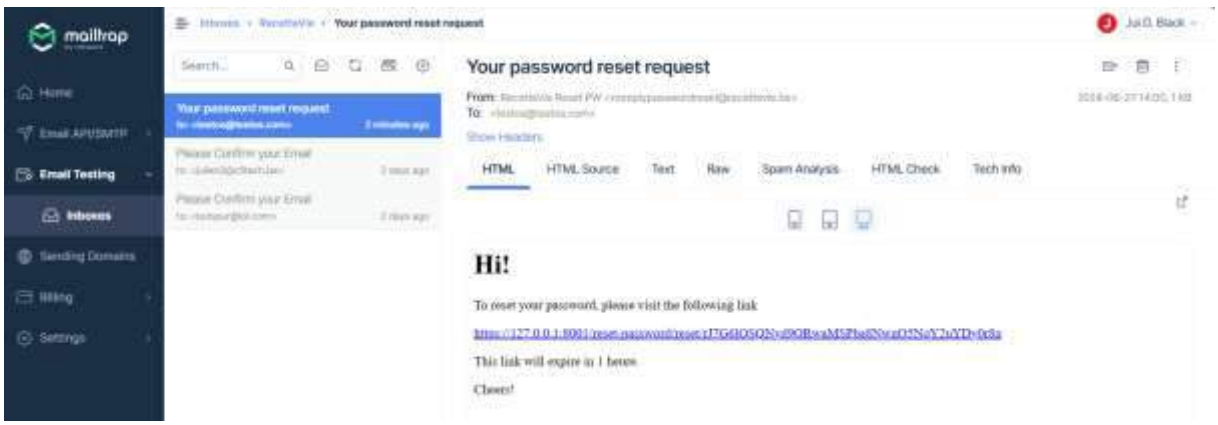


Mettez le mail d'un de vos utilisateurs puis validez, vous aurez ceci comme message :

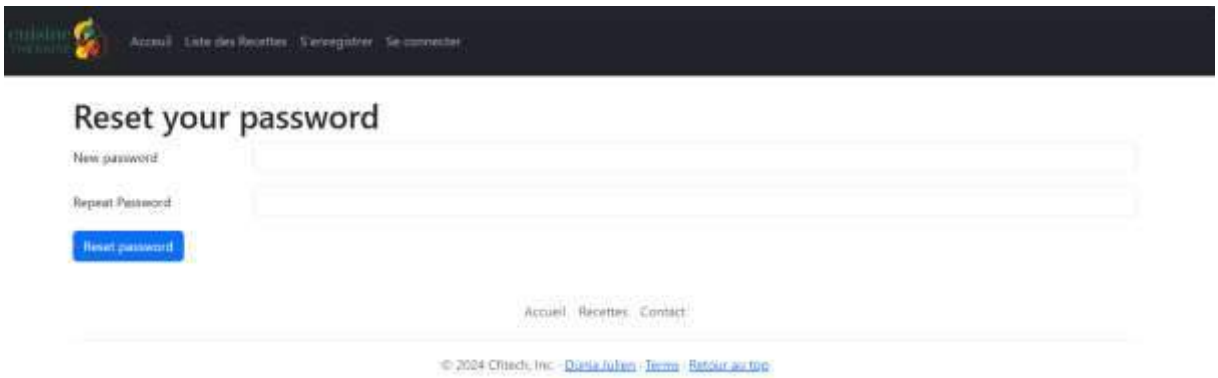


Le Try again vous renverra vers la page ou il faut introduire son email à reset.

Là maintenant il faut aller regarder au niveau de mailtrap. Vous verrez que vous avez un mail :



Dès que vous cliquez sur le lien de ce mail, cela va vous rediriger vers la page ou il faudra introduire votre nouveau mot de passe



Là vous pourrez mettre votre nouveau mot de passe 2 fois pour être sûr qu'ils sont les mêmes.

Attention par défaut Symfony vous redemande de créer un mot de passe de minimum 12 caractères et vous dis que le mot de passe ne doit pas être faible. Donc il voudra un mot de passe avec chiffre, caractères spéciaux, minuscule et majuscule. Pour éviter cela, allez dans « **form/ChangePasswordFormType** » et changer la limite à 6 et mettre en commentaire « **newPasswordStrength()** ». On va aussi enlever « **NotCompromisedPassword** » :

```
],
'first_options' => [
    'constraints' => [
        new NotBlank([
            'message' => 'Please enter a password',
        ]),
        new Length([
            'min' => 6,
            'minMessage' => 'Your password should be at least {{ limit }} char
            // max length allowed by Symfony for security reasons
            'max' => 4096,
        ]),
        // new PasswordStrength(),
        // new NotCompromisedPassword(),
    ],
    'label' => 'New password',
],
```

On peut bien entendu changer le mail, le nom, le message, l'objet du mail etc. C'est dans le Controller. On peut aussi bien entendu tout traduire.

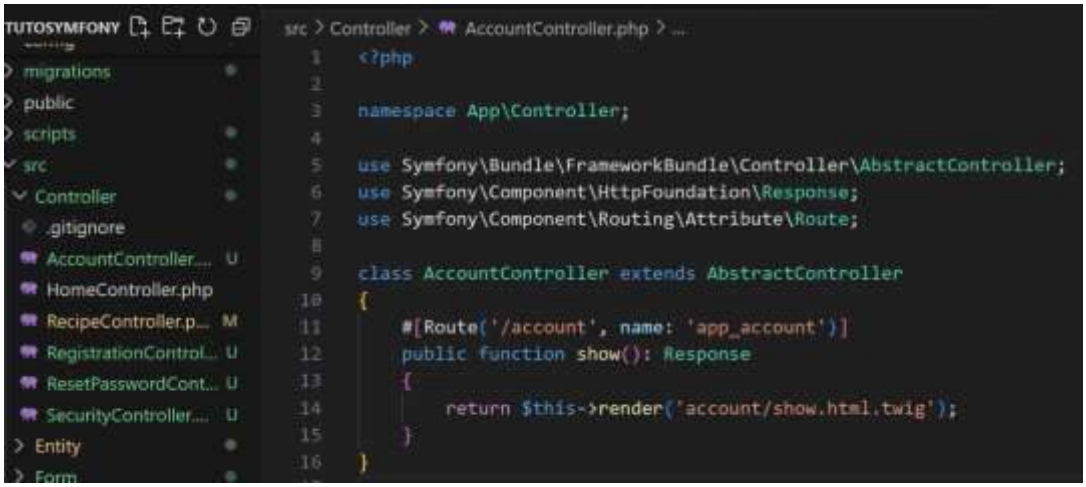
Exos

12) **Essayez de recentrer un peu tous ces formulaires et de les traduire.**

10.6 Page de profil

On va revenir sur notre page « **account** ». Normalement vous avez fait un bon « **symfony console make:controller account** » à l'ancienne. Ici nous n'allons pas utiliser la commande magique du CRUD, par ce qu'on a déjà le register qui s'occupe de la création d'un utilisateur.

Une fois créé, allez dans **AccountController** pour mettre comme nom de route **app_account** si ce n'est pas déjà fait**.** Et **changez** aussi le **nom** de la **méthode** ainsi que le **nom** du **template** :



Exos

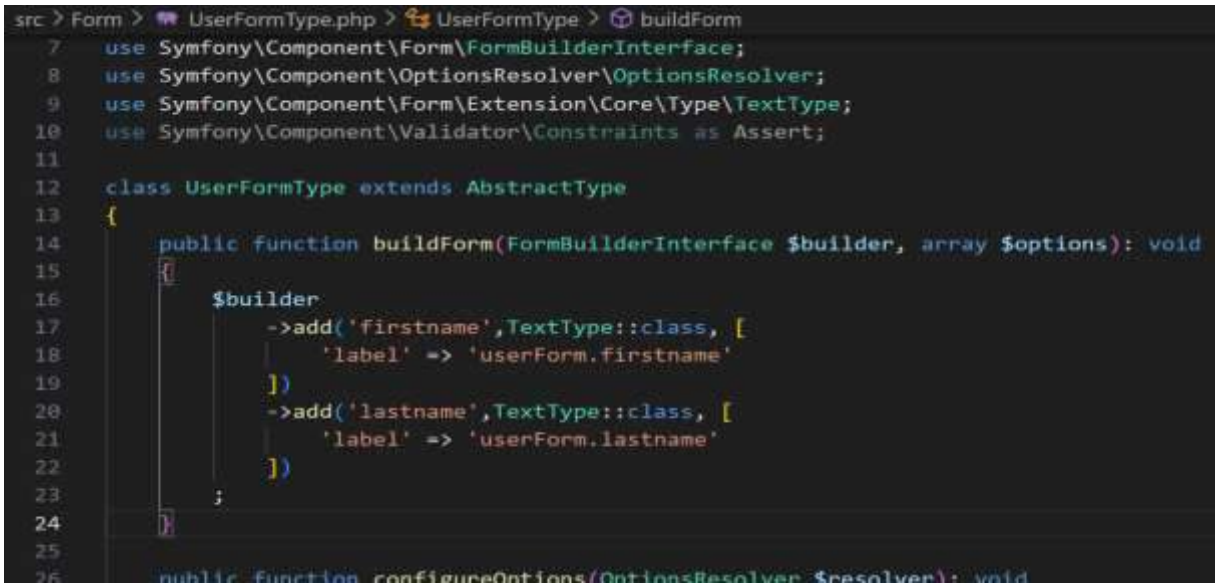
13) **Essayer de redirectionner l'onglet Account dans la barre de navigation.**

On modifiera ensuite le Template « **account/show.html.twig** » :

Ce qui donne ça dans un premier temps, on peaufinera après.

On va ensuite créer un **UserForm** pour pouvoir créer un formulaire qui édite un utilisateur :

On va dans **src/Form/UserFormType**, et on le modifie pour juste changer nom et prénom, plus tard on verra pour une image de profil. Donc la fonction buildForm donnera ça :



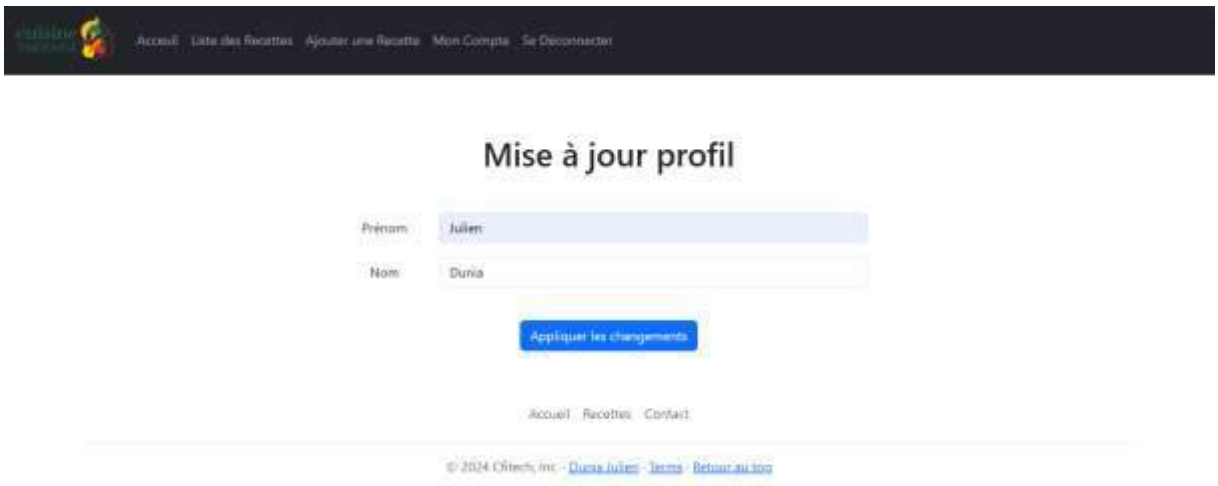
On pourra ensuite créer la méthode edit() dans **AccountController** **copier-coller** le code suivant :

```
#[Route('/account/edit', name: "app_account_edit")]
    public function edit(Request $request, EntityManagerInterface $em,
TranslatorInterface $translator): Response{
        /**
        * @var User
        */
        $user=$this->getUser();
        $form = $this->createForm(UserFormType::class, $user);
        $form->handleRequest($request);
        if($form->isSubmitted() && $form->isValid()){
            $user->setUpdatedAt(new DateTimeImmutable());
            $em->flush();
            $this->addFlash('success', $translator->trans('Account successfuly
updated !'));
            return $this->redirectToRoute('app_account');
        }
        return $this->render('account/edit.html.twig', [
                        'user' => $user,
                        'usermonForm' => $form->createView()
        ]);
    }
```

N'oubliez pas les imports manquants, basez-vous sur RecipeController.

On va créer la vue pour notre modification de profile (**templates/account/edit.html.twig**) :

Résultat final :



Quand je modifie ça marche bien :

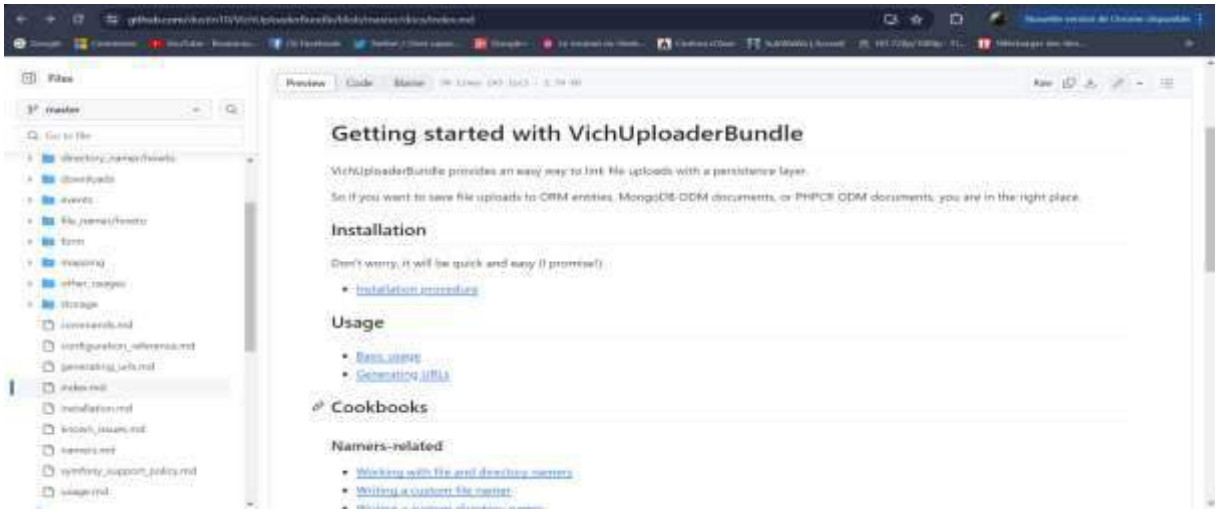
Exos

14) **Empêchez l'accès à /account et /account/edit si on n'est pas connecté.**

10.6 TP : Implémentation de VichUploader

On va ajouter **VichUploader** pour les images dans notre site. Donc maintenant on utilisera plus les url pour les images, mais directement des images qui proviennent de votre pc. On peut le retrouver sur l'excellent site https://packagist.org/ . Vous trouverez tous les composants PHP sur ce site.

Chaque année je demande à ce stade aux stagiaires de le faire tous seul en suivant les instructions officielles ici : https://github.com/dustin10/VichUploaderBundle/blob/master/docs/index.md



Je répète encore, vous êtes des développeurs web et vous devez être capable d'implémenter des composants dans vos projets par vous-même. On sait faire un tas de choses en Symfony ou même dans d'autres Frameworks, on ne peut pas tous vous apprendre en formation, mais on peut améliorer votre méthodologie par rapport à ça. En plus vous êtes une génération avec une plus grande assistance grâce aux IA.

Je vous donne juste quelques directives pour aider un peu ceux qui seront perdu. La plupart des documents que vous trouverez seront en anglais, vous pouvez toujours les traduire.

Exos

15) **D'abord installer VichUploader comme indiqué**
Modifier config/package/vich...yaml attention au nom et à l'indentation.
Ajouter ce qu'il faut rajouter dans l'entité (attributs, getters-setters etc.)
Faites une migration
Il faudra installer composer require liip/imagine-bundle pour la taille de l'image dans form.
Modifier le formulaire ou l'on veut rajouter l'image (quand on veut modifier le compte)

```
])
->add('imageFile', VichImageType::class, [
    'required' => false,
    'allow_delete' => true,
    'delete_label' => 'Delete profile image',
    'download_uri' => true,
    'image_uri' => true,
    'asset_helper' => true,
    'imagine_pattern' => 'avatar_thumbnail'
]);
```

Allez dans votre fichier config/packages/liip_imagine.yaml.

config > packages > ! liip_imagine.yaml

```yaml
# Documentation on how to configure the bundle can be found at:
liip_imagine:
    # valid drivers options include "gd" or "gmagick" or "imagick"
    driver: "gd"
    filter_sets:
        cache: ~
        avatar_thumbnail:
            quality: 75
            filters:
                thumbnail: { size: [120, 120], mode: inset }
```

Si vous avez problème de sérialisation il faut rajouter une fonction __serialisable() dans votre entity User :

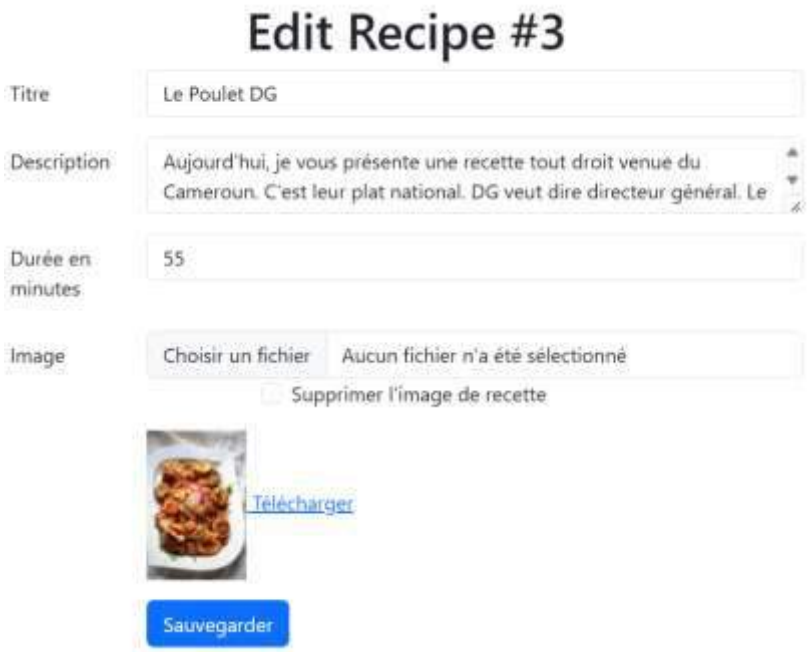
```php
/**
 * Return only the security relevant data
 *
 * @return array
 */
public function __serialize(): array
{
    return [
        'id' => $this->id,
        'email' => $this->email,
        'password' => $this->password,
    ];
}
```

Vous devrez avoir quelques choses comme ça si vous uploadé une image.

Pour voir l'image exemple dans show : <img style="max-width: 250px;" src={{ vich_uploader_asset(voiture, 'imageFile') }}>

16) **Rajouter image profile par défaut, vu que quand on crée un utilisateur, on ne lui demande pas de choisir une image. Voici mon image par défaut :**

17) **Essayez d'implémenter vichuploader pour les recettes. Ensuite remplacez vos anciennes images de recette avec lien d'internet par les images de VichUploader qu'on téléchargera dans notre projet. Mettez aussi une image par défaut aux recettes.**



10.6.1 Traduction dynamique (part final)

On va faire en sorte que notre site se sépare en 2 langues (vous pouvez faire autant de langues que vous voulez), moi je vais vous montrer pour l'anglais et le français. Vous devrez aller dans "**routes.yaml**" et rajouter la partie app. Attention c'est du yaml donc l'indentation est vitale.

```
config > ! routes.yaml
1   controllers:
2       resource:
3           path: ../src/Controller/
4           namespace: App\Controller
5       type: attribute
6   # home:
7   #     path: /
8   #     controller: App\Controller\HomeController::index
9   app:
10      resource: '../src/Controller/'
11      type: attribute
12      prefix: /{_locale}
13      requirements:
14          _locale: en|fr
```

Ce bout de code va nous permettre de mettre devant toutes nos routes la valeur de la variable {_locale} qui est dans mon cas la langue française ou anglaise. Donc à partir d'ici, pour que vos pages fonctionnent, commencer par mettre /en ou /fr :

Nous voyons bien que maintenant après avoir mis /en ou /fr ça fonctionne bien. Attention il faudra bien vérifier que vous traduisiez bien chacun des messages dans chacune des langues que vous avez. Ça ne fonctionne pas sur les données qui sont en base de données, pour cela je vous laisse vous-même effectuer vos recherches pour pouvoir le faire (il y a 2 possibilités).

En revanche on a un petit problème, c'est que maintenant quand vous redémarrez votre serveur et que vous ouvrez une nouvelle page a partir de la console. Vous allez tomber sur la page de base de symfony :

C'est normal, on lui a dit que toutes nos routes doivent être précédée de /en ou /fr. Mais ici on n'a pas l'une des deux options. Pour régler ce problème on va faire une redirection automatique de « / » vers « /en » ou « /fr ».

Pour que la redirection fonctionne quand vous lancer votre site. Rajouter ceci dans routes.yaml au-dessus de app :

```
root_redirect:
    path: /
    defaults:
        _controller:
Symfony\Bundle\FrameworkBundle\Controller\RedirectController::urlRedirectActio
n
        path: /en
        permanent: false
```

Et ajouter dans **app** le « **defaults** »... :

```
app:
    resource: '../src/Controller/'
    type: attribute
    prefix: /{_locale}
    requirements:
        _locale: en|fr
    defaults:
        _locale: en
```

Sauvegardez, stoppez votre serveur, faites un cache clear et redémarrez votre serveur avant de tester. Vous verrez que cette fois ci, il vous ferra une redirection vers /en dans mon cas ici.

On va pour finir, faire en sorte que nous ayons un bouton de traduction qui permet de switcher entre mes différentes langues.

Pour se faire, on va rajouter dans notre partials « **_nav.html.twig** » juste en dessous du « **
** » vous mettrez ceci :

```
<div class="dropdown">
                <button class="btn btn-secondary dropdown-toggle"
type="button" data-bs-toggle="dropdown"
                        aria-expanded="false">
                    {% if app.request.locale == 'en' %}
                         <span class="flag">GB</span> English
                    {% else %}
                         <span class="flag">FR</span> Français
                    {% endif %}
                </button>
                <ul class="dropdown-menu">
                    <li>
                        <a class="dropdown-item" href="{{
path(app.request.attributes.get('_route'),
                            app.request.attributes.get('_route_params')|merge(
{'_locale': 'en'})) }}">
                             <span class="flag">GB</span> English
                        </a>
                    </li>
                    <li>
                        <a class="dropdown-item" href="{{
path(app.request.attributes.get('_route'),
                            app.request.attributes.get('_route_params')|merge(
{'_locale': 'fr'})) }}">
                             <span class="flag">FR</span> Français
                        </a>
                    </li>
                </ul>
            </div>
```

Ce qui donnera un onglet :

Avec la possibilité de choisir soit Anglais soit Français en haut à droite.

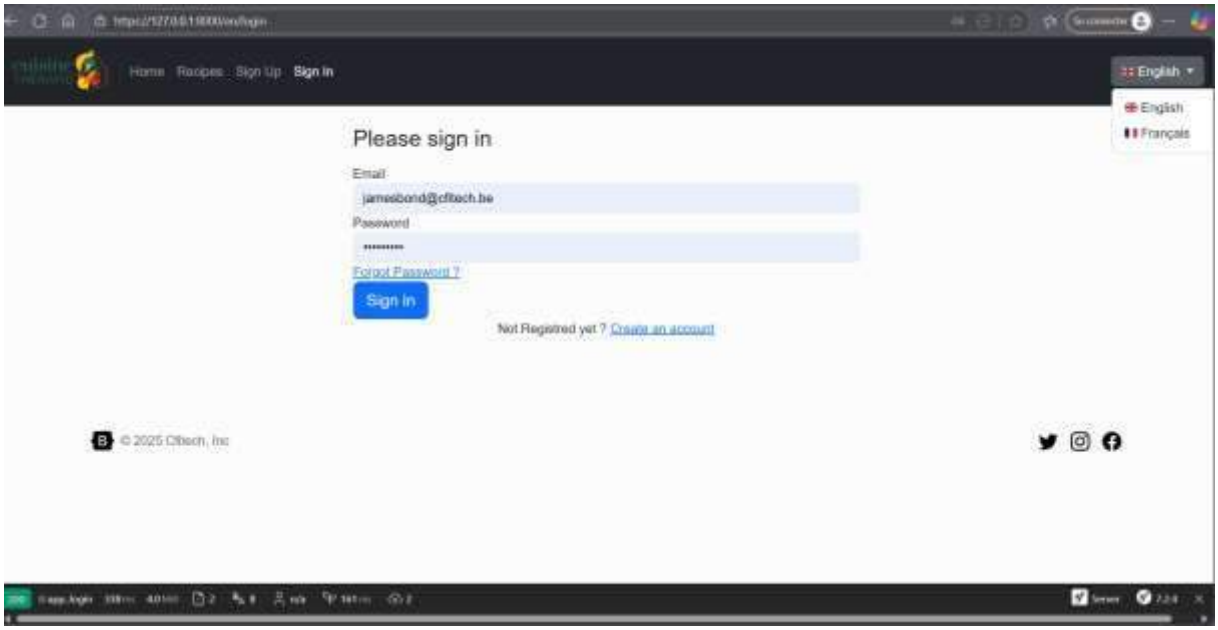
Chez la plupart des personnes sous **Windows** qui utilisent chrome ou edge, vous ne verrez pas les drapeaux. Pour les voir, il faudra rajouter un script en javascript dans « **base.html.twig** » dans le block javascript :

```
<script type="module" defer>
                import { polyfillCountryFlagEmojis } from
"https://cdn.skypack.dev/country-flag-emoji-polyfill";
                polyfillCountryFlagEmojis();
            </script>
```

Et ensuite allez dans « **assets/styles/app.css** » et rajouter ceci :

```
body {
  font-family: "Twemoji Country Flags", "Helvetica", "Comic Sans", serif;
}
```

Maintenant si vous retournez sur votre site vous verrez :



Ici se termine la première partie du cours de Symfony, ou on fait un site en front-end et back-end. Il y a plusieurs TPs que je vous donnerai à faire dont la pagination, la barre de recherche ou encore les commentaires.

Chapitre 11 : Les API avec API Platform

API Platform est un Framework web utilisé pour générer des API REST et GraphQL, se basant sur le patron de conception MVC.

Aller sur le site https://api-platform.com/ et cliquer sur get started

- **composer require api**

Vous pouvez déjà taper l'url de votre projet suivi de /api (votreUrl/api)

Il faudra aller mettre dans nos entités l'import

- use ApiPlatform\Metadata\ApiResource;

  #[ApiResource]

  class nomDeVotreClass {…}

Faites un **cache:clear** et redémarrez votre serveur.

Refaite **votreUrl/api** et vous verrez toutes les possibilités avec les requêtes HTTPS

On va utiliser Postman pour jouer avec ces requêtes (**CRUD**)

CREATE = **POST**

READ = **GET**

UPDATE = **PUT, PATCH**

DELETE = **DELETE**

Les méthodes **PUT** et **PATCH** ont des significations différentes :

PUT, remplace les données par celle qui sont envoyées dans la requête.

PATCH, permet la modification partielle d'une ressource en fusionnant les données envoyées avec les données déjà présentes ou grâce à l'utilisation d'opération de modification.

Téléchargez, installez et inscrivez-vous sur Postman.

Pour utiliser Postman il suffira de mettre les url que nous montre Api Platform.

Dans Postman (allez dans home, my Workspace)

a) Pour les **GET** il suffira de copier l'url et d'exécuter (Send). (Appuyez sur **disable SSL** si vous avez une erreur) (**Status 200**)
b) Pour le **POST** il suffira de copier aussi l'url, par contre il faudra faire plusieurs choses en plus (**Status 201**). D'abord allez dans Header et tapez comme clé : **Content-Type** et pour la valeur vous mettez : **application/ld+json**
Ensuite allez dans Body puis dans raw et c'est là que vous mettrez votre code json.

Exemple de code pour créer une Recette avec un utilisateur :

```
{
    "title" : "Recette Postman",
    "slug" : "recette-postman",
    "content" : "Voici la recette que j'ai créé avec postman",
    "duration" : 150,
    "imageName" : "default.svg",
    "user" : "/api/users/2"
}
```

Exemple de code pour créer un User, attention le mot de passe haché :

```
{
    "email" : "zidane@cfitech.be",
    "password" : "VOTRE MDP HACHÉ",
    "firstname" : "Zinedine",
    "lastname" : "Zidane",
    "isVerified" : true,
    "imageName" : "default.jpg",




}
```

c) Pour le **DELETE** il suffira de copier l'url de ce que vous voulez delete. (**Status 204**)

d) Pour le **PATCH** il suffira de copier l'url avec l'id de l'user que vous voulez modifier (**Status 200**). D'abord allez dans Header et tapez comme clé : **Content-Type** et pour la valeur vous mettez : **application/merge-patch+json**. Ensuite allez dans Body puis dans raw et c'est là que vous mettrez votre code json. Exemple de code pour changer le prénom d'un User :

```
{
    "firstname": "Christian",
}
```

e) Pour le **PUT** il suffira de copier l'url avec l'id de l'user que vous voulez modifier (**Status 200**) D'abord allez dans Header et tapez comme clé : **Content-Type** et pour la valeur vous mettez : **application/ld+json**. Ensuite allez dans **Body** puis dans **raw** et c'est là que vous mettrez votre code json.

Exemple pour changer le prénom d'un User, il faudra remettre toutes les informations des autres champs, c'est un peu comme quand vous modifiez un seul champ dans un formulaire, il renvoi les données :


```
{
    "email" : "zidane2@cfitech.be",
    "roles" : [
        "ROLE_USER"
    ],
    "password" :
"$2y$13$Sv9kwcEG6t814FGOZ9rdBOnM.Ewx6FWKPyo9342pRdecfORGUkyvW",
    "firstname" : "Zinedine",
    "lastname" : "Zidane",
    "imageName" : "default.jpg"
    "userIdentifier" : "zidane1@cfitech.be",
    "verified" : false
}
```


Voici une liste de status que vous rencontrerez le plus souvent.

Status 200 Ok

Status 201 Created

Status 204 Delete

Status 404 Not found

Status 415 Unsupported Media Type

Status 400 Bad Request

Status 401 Unauthorized

Status 422 Unprocessable Entity

Status 500 Internal Server

40) Partie Sécurité avec JWT

On va protéger notre API, il ne peut y avoir que les utilisateurs connectés qui peuvent utiliser.

On va aussi avoir besoin de OPENSSL qu'on peut télécharger ici https://slproweb.com/products/Win32OpenSSL.html version light c'est suffisant.

Une fois installer il faudra aller modifier les variables d'environnement pour le rajouter dans path le chemin du openssl (C:/program /Openssl/bin).

On peut aller ensuite voir la documentation officielle de JWT en tapant sur google lexik jwt, allez sur le GitHub, tout en bas il y a getting started.

Pour se faire on va installer JWT dans notre projet :

composer req lexik/jwt-authentication-bundle --ignore-platform-req=ext-sodium

On va ensuite devoir générer des clés SSL, elles vont permettre de délivrer des tokens JWT et donc s'assurer que ces tokens ont été délivré par votre serveur (que par les machines qui détiennent cette clé). Il faudra faire cette commande :

symfony console lexik:jwt:generate-keypair

(Si ça ne marche pas, créez d'abord un dossier jwt dans config et recommencer)

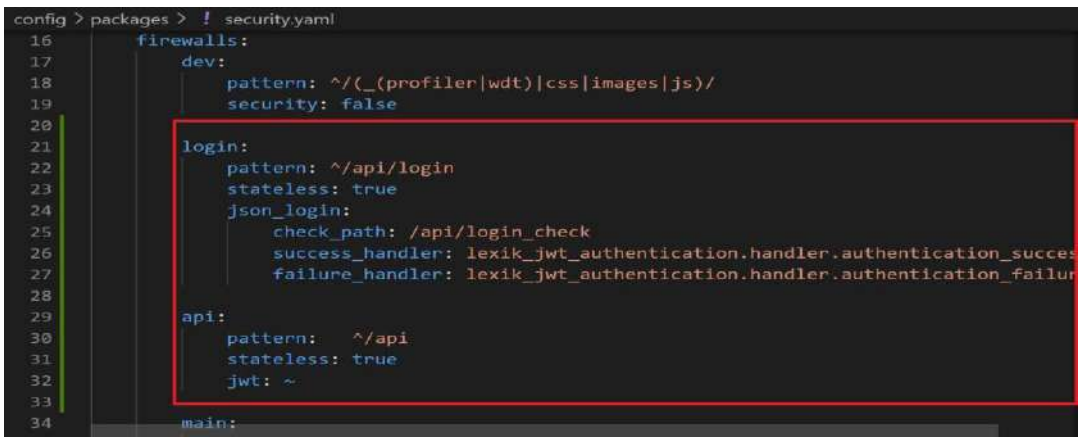
Si ça ne marche pas suivez ces étapes https://stackoverflow.com/questions/66252709/error-system-libraryfopenno-such-process

Vous remarquerez qu'il a créé deux fichiers dans le dossier **config/jwt**. Il a aussi ajouté des choses dans le fichier **.env**. Il a rajouté une clé **JWT_PASSPHRASE** qui a été utilisé pour créer les deux fichiers dans le dossier jwt. Ces deux fichiers (**public.pem** et **private.pem**) ne seront pas versionné (donc n'irons pas sur GitHub).

Attention lors de la production, il ne faudra pas utiliser la même clé **JWT_PASSPHRASE**, parce qu'elle va apparaitre dans votre dépôt git. Il faut que cette clé la donc dans .env n'apparaisse pas.

Maintenant qu'on a nos clés, il faudra paramétrer notre bundle.

Allons dans **config/packages/security.yaml** et modifier ce qu'il faut (suivez le tuto officiel **en LISANT BIEN**). Attention il faut que **login** soit avant **main** dans firewalls :



Et tout en bas rajouter dans access_control :

```
access_control:
    # - { path: ^/admin, roles: ROLE_ADMIN }
    # - { path: ^/profile, roles: ROLE_USER }
    - { path: ^/api/login, roles: PUBLIC_ACCESS }
    - { path: ^/api/users, roles: PUBLIC_ACCESS, methods: [POST] }
    - { path: ^/api/recipes, roles: PUBLIC_ACCESS, methods: [GET] }
    - { path: ^/api/, roles: IS_AUTHENTICATED_FULLY }
```

Ça nous permettra de créer des users (POST), d'accéder à la page Login et ça nous permettra aussi de voir tous les pins (GET) sans être connecté. La nôtre application sera protégée.

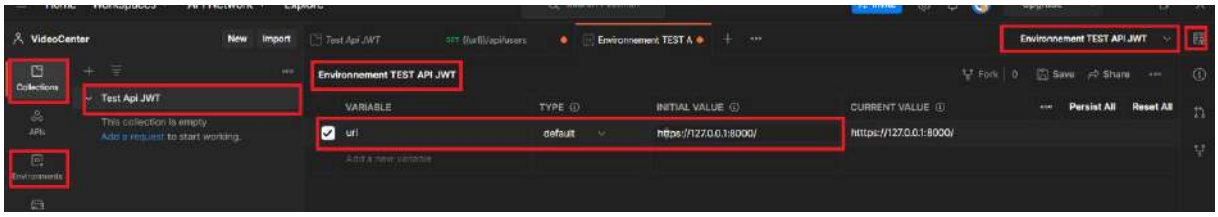
Il faudra aussi allez rajouter dans **config/routes.yaml :**

```
api_login_check:
    path: /api/login_check
```

Récupérer via le tuto, pour avoir la bonne indentation.

Quand c'est fini, faites un cache clear puis fermer et réouvrer votre serveur et essayer de faire /api.
Comme sur le site, il faudra donc se loguer pour pouvoir créer une recette etc.
On va tout d'abord créer des scénarios.
1er Scénario, Une personne qui s'inscrit puis une deuxième qui s'inscrit avec des champs différend mais un même email et mot de passe.
2ème Scénario, Une personne qui s'inscrit et se logue.
3ème scénario, Une personne qu'on a inscrit et logué, et créez une recette qu'on supprimera à la fin.

On va d'abord créer une nouvelle collection, qu'on va appeler « **Test_Api_Jwt** ».
On va aussi créer des environnements pour pouvoir utiliser des variables que l'on pourra réutiliser plusieurs fois si on le souhaite. Créez un **Environment** sur l'onglet à gauche, on va l'appeler « **Environnement_Test_Api_JWT** ».
On va rajouter dans cette environnement quelques choses qui nous sera utile, c'est l'URL.



Maintenant commençons ce **1er scénario** :
Cliquez sur votre Collections dans l'onglet à gauche sur « **Add a request** » et donnez-lui comme nom : **Create User 1**. Puis tentez de créer un utilisateur :

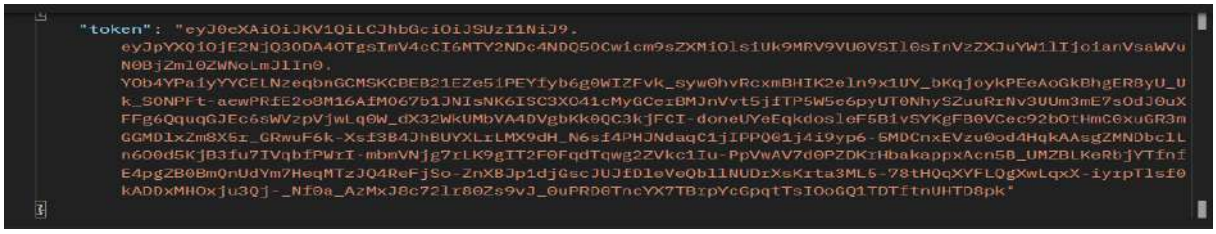
Email : vendredi@cfitech.be
Mot de passe : vendredi (haché)
Prénom : Vendredi
Nom : Vendredi
Est Vérifié : Vrai

Vérifiez bien que vous arrivez à vous connecter avec cet utilisateur.
Créez une nouvelle requête (clic droit) avec cette fois ci comme nom : **Create User 2**

Email : vendredi@cfitech.be
Mot de passe : vendredi (haché)
Prénom : Samedi
Nom : Samedi
Est Vérifié : Vrai

Tentez de créer. Vous verrez qu'il ne voudra pas parce que cet email a déjà été utilisé. Remplacez juste le mail par samedi@cfitech.be et changez le mot de passe aussi en samedi(haché).

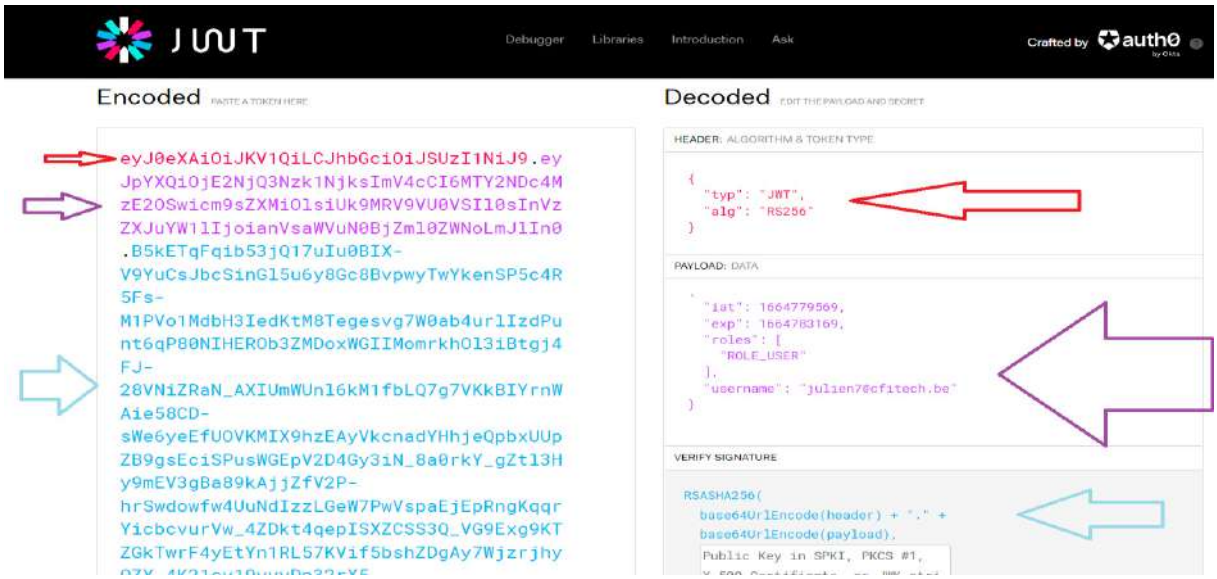
Avant de passer au scénario 2, testez de vous loguer avec un compte existant sur Postman. Ce sera avec login_check. Allez voir les informations sur votreServeur/api.
Si vous avez réussi, il vous donnera un Status 200 et le fameux token JWT. Ce token est le token d'authentification pour l'utilisateur avec lequel vous vous êtes connecté. Ces tokens sont **actifs pendant 1 heure**. Il faudra en recrée un autre après une heure.



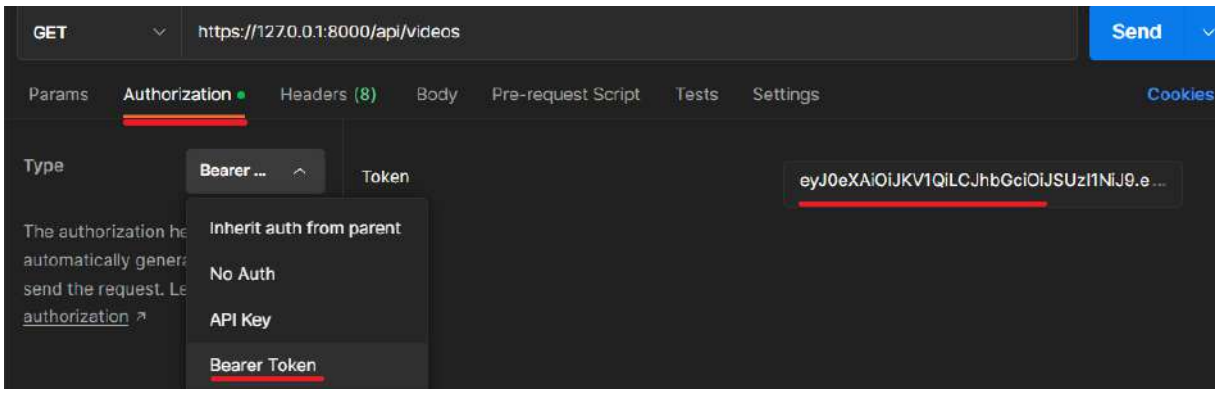
On peut aller voir ce que représente ce token JWT sur internet, il suffit de taper « **token JWT decode** » ou directement aller sur le site https://jwt.io/ . On voit ici que la première partie c'est le type de token et l'algorithme de cryptage utilisé, la deuxième partie représente les informations sur mon utilisateur :

- iat (Issued At) en gros la date d'émission du token
- exp la date d'expiration du token
- le rôle de l'utilisateur (Admin ou User)
- Et son username (email) qui l'identifie

La troisième partie représente la signature numérique.



Maintenant si vous voulez, vous pouvez utiliser ce token, il suffit d'aller dans **Authorization**, sélectionner dans **Type** : **Bearer Token** et coller votre token :



Utilisez {{url}}/api/recipes

Passons au deuxième scénario, on va essayer de sauvegarder notre username et token dans nos variables d'environnement. On va aussi tester nos requêtes avec du code JSON.

La partie **Test** est très importante, je vous montrerai comment on peut tester plusieurs requêtes en une seule commande.

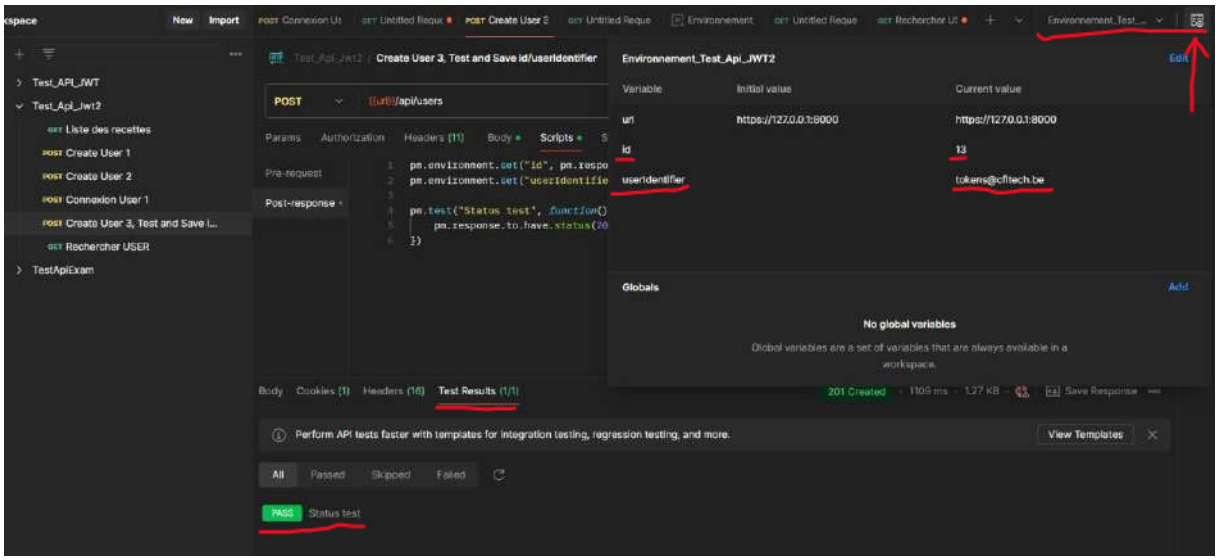
2ème scénario :

Créez une nouvelle requête qui aura comme nom « **Create User, Test and Save id/username** ». Créez cet utilisateur :

Email : tokens@cfitech.be
Mot de passe : tokens (haché)
Prénom : Tokens
Nom : Tokens
Est Vérifié : Vrai

Avant de soumettre on va tout d'abord tester la requête et sauvegarder cet User en prenant son ID dans une variable d'environnement locale. Pour se faire il faut aller dans l'onglet « **Tests** », puis sur open puis sélectionner post-response. On va écrire en **JAVASCRIPT.** La première ligne récupère l'ID, rappelez-vous quand on crée un utilisateur il nous donne toutes ses données en bas. Le deuxième à récupérer l'useridentifier. Et le code d'après va tester si on a bien le Status de Created (201). (pm = Postman). userIdentifier.

Une fois finit lancer la requête et vous obtiendrez ce résultat :



On voit bien que si vous cliquez sur « **Test Results** », notre test qu'on a créé, est passé. On a bien eu un code 201 donc Created.

On voit aussi que si on clique pour voir nos variables d'environnement on voit qu'il a bien sauvé l'id et userIdentifier de l'user qui vient d'être créé.

Voici comment maintenant, vous pourrez tester vos requêtes et comment sauvegarder des variables.

On peut passer au scénario 3.

3ème scénario :

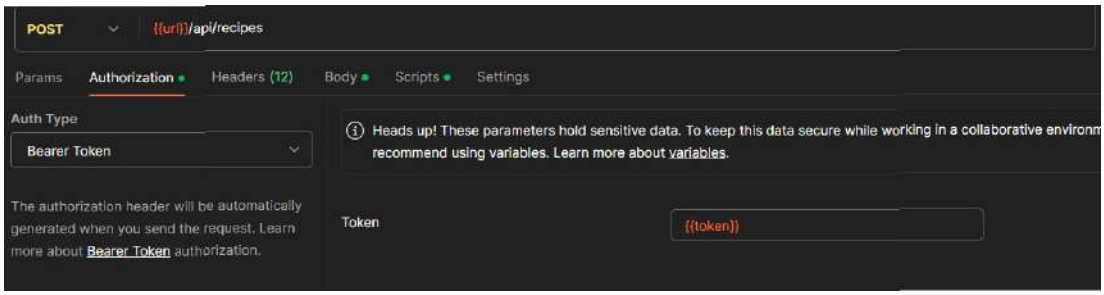
On va maintenant créer une nouvelle requête qui s'appellera « **Login with Environment Variable** » et on va essayer de se loguer avec l'username qu'on a sauvé dans nos variables d'environnement :

```
{
  "username": "{{userIdentifier}}",
  "password": "tokens"
}
```

Exos

1) **Avant de vous loguer, je vais vous demander de créer un test qui validera si on s'est bien logué et aussi de sauvegarder ce token dans une variable d'environnement qui s'appellera « token ».**
2) **Maintenant que vous avez fait ça, on va essayer de créer 2 recettes en utilisant ce token et en utilisant aussi l'id du user. Ajoutons d'abord une nouvelle requête qu'on va appeler « Create Recipe with environment variable ».**
3) **Créez maintenant une autre recette dont vous sauvegarderez l'id dans « idLastRecipe ».**

Il faut qu'au moment d'introduire le user vous mettez ce qu'il faut en utilisant la variable d'environnement. N'oubliez pas aussi de tester si on a bien le Status de création. Pour finir n'oublier pas d'aller mettre le token, en utilisant la variable d'environnement :



Allez voir sur votre site si la recette a bien été créée par l'utilisateur que vous avez créé.

Maintenant retournez dans Postman et créez une nouvelle requête qui va s'appeler « **Delete last recipe with environment variable** ».

(Exos)

1) **Elle va donc supprimer la dernière recette qu'on a créée. Attention il faudra bien entendu utiliser le token, ne pas oublier de créer le test et utiliser la variable d'environnement. Essayer de faire en sorte que quand vous supprimer la dernière recette, il met « idLastRecipe » à null.**
2) **On va maintenant faire la modification de la dernière recette créée. Attention pour faire la modification, il faut avoir la dernière recette (idLastRecipe). Créez une nouvelle requête que vous appellerez « Update a Recipe with environment variable ». On va modifier le titre et la description. N'oubliez pas de créer un test (utilisez un PATCH).**

Nous allons maintenant passer à d'autres scénarios, en définissant cette fois ci les rôles en tant qu'admin. Lui donnez toutes les permissions etc.

4ème Scénario : On va essayer de créer un utilisateur admin (en donnant comme rôle « **ROLE_ADMIN** »).

Créez une nouvelle requête que vous appellerez « **Create admin user** ». Il faudra bien entendu faire un test si ça a bien fonctionné.

Email : thanos@cfitech.be

Prenom : Thanos

Nom : Thanos

Mot de passe : thanos (n'oubliez le hachage)

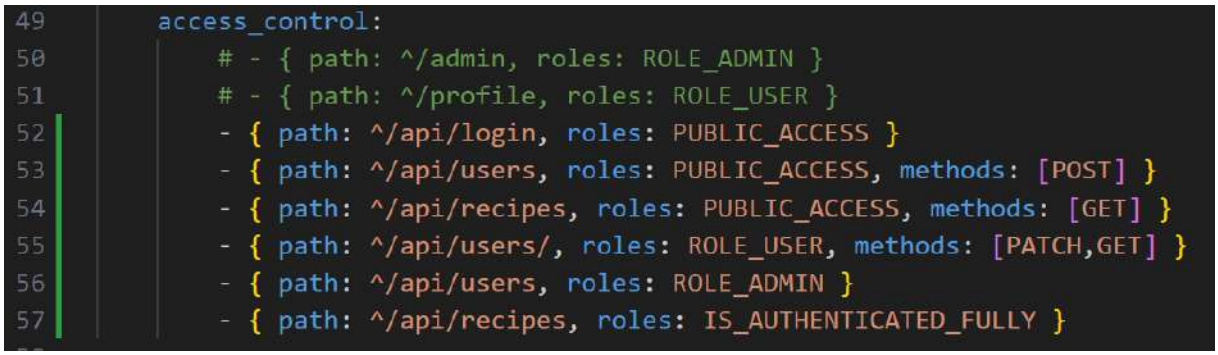
Est vérifié : vrai

Ici je vous donne la partie ou vous allez définir son rôle :

```
"roles": [
    "ROLE_ADMIN"
]
```

Il faudra aussi sauvegarder son id en donnant comme nom de variable d'environnement « **idAdmin** » et son **userIdentifier** en donnant comme nom « **admin** ».

Il faudra ensuite allez rajouter dans le fichier de « **security.yaml »** dans access_control :



```
49      access_control:
50          # - { path: ^/admin, roles: ROLE_ADMIN }
51          # - { path: ^/profile, roles: ROLE_USER }
52          - { path: ^/api/login, roles: PUBLIC_ACCESS }
53          - { path: ^/api/users, roles: PUBLIC_ACCESS, methods: [POST] }
54          - { path: ^/api/recipes, roles: PUBLIC_ACCESS, methods: [GET] }
55          - { path: ^/api/users/, roles: ROLE_USER, methods: [PATCH,GET] }
56          - { path: ^/api/users, roles: ROLE_ADMIN }
57          - { path: ^/api/recipes, roles: IS_AUTHENTICATED_FULLY }
```

Si vous essayez de voir la liste des users avec un token simple, ça ne va plus fonctionner, il faudra être connecté dorénavant avec un compte admin.

(Exos)

1) **Créez une requête "Login with admin and save adminToken" qui comme son nom l'indique va loguer un admin (admin) en sauvant son token dans une variable d'environnement adminToken. N'oubliez pas le test du status.**
2) **Créez ensuite une requête « List of users » qui va afficher tous les utilisateurs. N'oubliez pas le test du status.**
3) **Créez une requête « Update last user » qui modifie le prénom du dernier utilisateur enregistré attention pas l'admin. Oubliez pas de tester le status.**
4) **Créez une requête qui supprime le dernier utilisateur enregistré « Delete last user ». N'oubliez pas le test et de remettre id à null, token à null ainsi que userIdentifier à null.**
5) **Créez une requête qui créé un nouvel utilisateur en sauvant id et userIdentifier.**
6) **Créez une requête qui modifie le dernier utilisateur enregistré en lui donnant une photo de profile « Update user with a picture ». Réfléchissez à comment faire cela.**

Une fois finit, vous pouvez vous entrainer et tester ce que vous voulez avec les API, je vous ai montré une petite introduction aux APIs avec Postman.

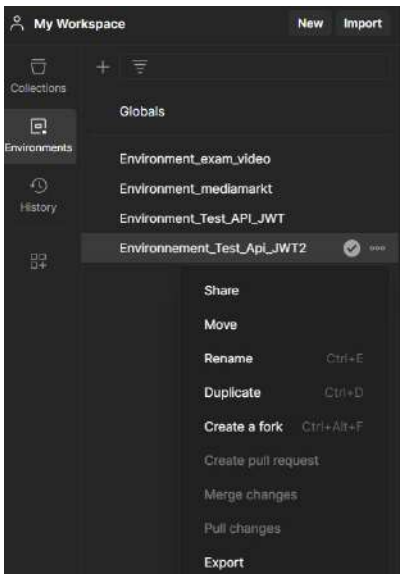
Newman

On va maintenant faire en sorte que tous nos tests des scénarios soient faits en même temps, et montre que ça peut être fonctionnelle. Allez sur le site pour suivre la démarche https://learning.postman.com/docs/running-collections/using-newman-cli/installing-running-newman/ . Installez npm si ce n'est pas déjà fait ensuite allez dans votre projet et tapez :

- **npm install -g newman**

On va ensuite allez créer un dossier « postman » à la racine de notre projet Symfony. Dans ce dossier on va mettre notre collection « **TEST_API_JWT** » qu'on aura exporter en cliquant droit dessus ensuite export. On mettra le fichier « .json » dans ce dossier.

On va aussi exporter nos variables d'environnement, pour ce faire il faut cliquer sur environnement ensuite cliquer sur « **Environment_test_API** », suivez l'image ci-dessous :



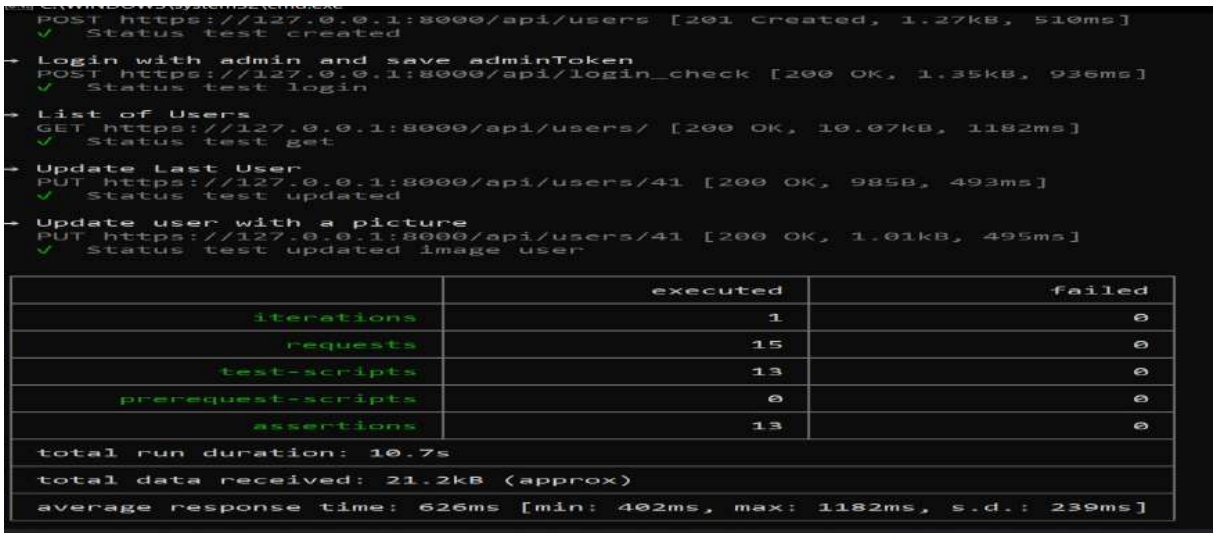
Vous mettrez le fichier dans le dossier Postman de votre projet.

Attention pas d'espace dans les noms des 2 fichier et supprimez tous vos recettes et utilisateurs qui ont été créé dans votre base de données avec Postman !!

Ensuite il suffira de lancer la commande :

- **newman run ./postman/Test_Api_Jwt2.postman_collection.json -k -e ./postman/Environment_Test_Api_Jwt2.postman_environment.json**

Non seulement il fera les tests mais en plus de cela, il ajoutera les utilisateurs et recettes que vous avez mis dans Postman.



Ceci clôture mon cours de Symfony. N'oubliez pas que c'est à vous d'approfondir les matières qu'on vous donne. On vous ouvre les portes de la programmation mais c'est à vous de les franchir ensuite pour aller en profondeur. Ce fut un plaisir et bonne continuation à tous.