Découvrir Symfony 7

Grafikart

*Documentation faite par Annaig en juin 2024*

Playlist de tutoriels en libre accès (les numéros des chapitres ci-dessous correspondent aux vidéos) : <https://www.youtube.com/playlist?list=PLjwdMgw5TTLXuvlGqP18gbJCYVg7y6Fig>

Exemple de mise en oeuvre (Symfony 7.1) :

<https://github.com/AnnaigJegourel/Sf7-Recettes-de-cuisine>

Table des matières

[01 02 : Installation et structure 3](#_Toc171579143)

[03 : Premiers contrôleurs 4](#_Toc171579144)

[04 : Twig 6](#_Toc171579145)

[05 : Doctrine 8](#_Toc171579146)

[06 : Les formulaires 11](#_Toc171579147)

[07 : Valider les données 13](#_Toc171579148)

[08 : Comprendre les services 16](#_Toc171579149)

[09 : TP : Formulaire avec envoi d’e-mail 18](#_Toc171579150)

[10 : TP : Catégories, interface admin, service FormListenerFactory 20](#_Toc171579151)

[11 : ORM relation ManyToOne 22](#_Toc171579152)

[12 : Envoi de fichiers 25](#_Toc171579153)

[13 : Le composant Security 27](#_Toc171579154)

[14 : Paginer les données 31](#_Toc171579155)

[15 : Doctrine avancé 35](#_Toc171579156)

[16 : AssetMapper 36](#_Toc171579157)

[17 : API – Serializer 39](#_Toc171579158)

[18 : API – Désérialisation et MapRequestPayload 40](#_Toc171579159)

[19 : API – Authenticator stateless 43](#_Toc171579160)

[20 – Les fixtures 45](#_Toc171579161)

[21 – Les Voters 46](#_Toc171579162)

[22 – Les événements 46](#_Toc171579163)

[23 – Symfony UX 46](#_Toc171579164)

[24 – Formulaires imbriqués 46](#_Toc171579165)

[25 – Messenger 46](#_Toc171579166)

[26 – Internationalisation 46](#_Toc171579167)

[27 – Hébergement Infomaniak 46](#_Toc171579168)

[28 – Hébergement O2Switch 46](#_Toc171579169)

# 01 02 : Installation et structure

Prérequis

Nécessite PHP >= 8.2.\*

Installation

1. Installer le squelette minimal avec le CLI :

symfony new <nom du projet> 🡪

1. Installer les modules pour une application web :

composer require webapp

(garder les réponses par défaut aux questions posées)

Structure

Fichiers à la racine :

* composer.json
* .env pour piloter les variables d’environnement (connexion bdd..)

assets/ 🡪 fichiers JS et CSS

bin/ 🡪 2 exécutables :

* console pour interagir avec le framework

php bin/console 🡪 liste toutes les interactions possibles

* phpunit = sorte de pont pour interagir avec l’outil de test

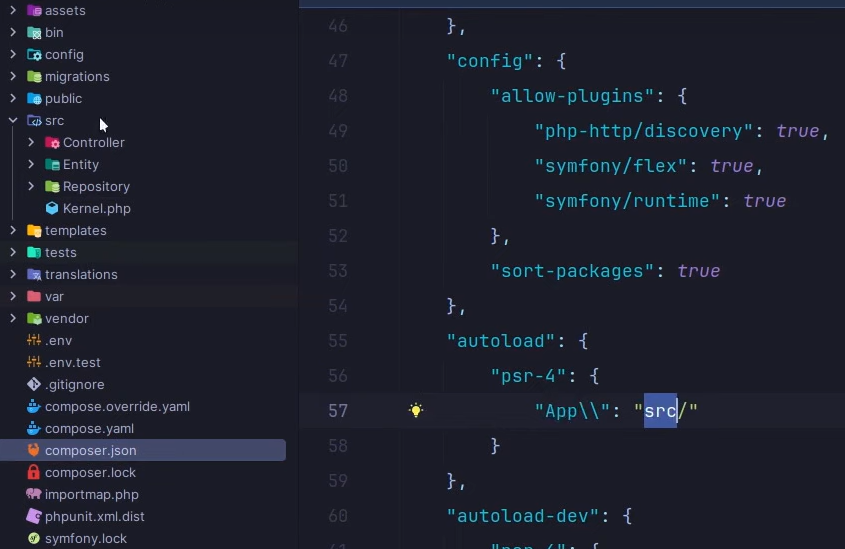
config/ 🡪 configuration du framework :

* organisée selon les différents modules
* syntaxe yaml, basée sur les indentations

migrations/ 🡪 fichiers définissant la structure de la base de données

public/ est la racine lorsque l’app sera hébergée en ligne

src/ contient toutes les classes 🡪 est dans le namespace App, que l’on retrouve au niveau de l’auto load dans composer.json :



templates/ contient les vues HTML

tests/ où on écrit les tests

translation/ contient les traductions

var/ contient des logs & dossiers temporaires auxquels on touche peu

🡪 Ceux qui nous intéressent le plus sont : public, src et templates.

Démarrer le serveur

Avec la commande Symfony : symfony serve

# 03 : Premiers contrôleurs

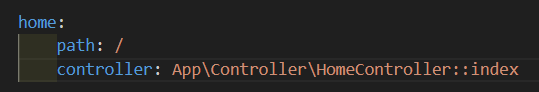
1. Créer un contrôleur

php bin/console make :controller HomeController

* 1 controleur = 1 classe avec des méthodes
* 1 méthode 🡪 1 URL, 1 réponse (HttpFoundation)

1. Configurer l’url racine dans config/routes.yaml

on crée la route racine & on indique quelle méthode de quel contrôleur correspond :



1. 2e façon de gérer les route : avec un attribut dans le contrôleur

 La route et l’action sont au même endroit ! pratique !

1. Utilisation du AbstractController

* Avec le mot clé « extends »
* On peut en utiliser les méthodes sans tout réécrire à la main, par exemple

return $this🡪redirect()

1. Utilisation des fonctions de dump

var\_dump()

dump() = var dump améliorée

dd() = dump & die (l’exécution du script s’arrête)

Récupérer un attribut de la requête :

dd($request->attributes->get(‘slug’), $request->attributes->get(‘id’)) ;

ou getInt(‘id’) pour spécifier qu’on veut récupérer un entier

Bcp d’objets de la request comme attributes sont de type « parameter bag » = collection de paramètres

1. Utilisation de l’Objet Request

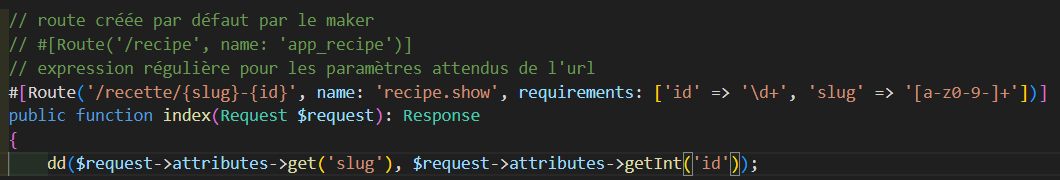
Pour définir un paramètre d’url par défaut (ici, si le nom est vide dans l’url) :

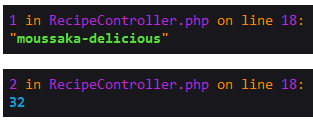
function index(Request $request): Response {

   return new Response ('Hallo ' . $request->query->get('name', 'Janosch'));

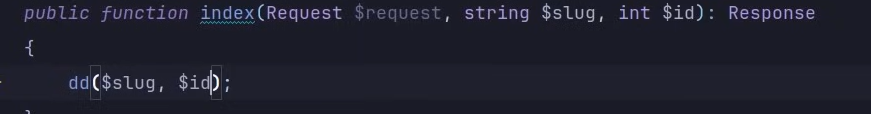
}

Pour définir des url spécifiques (ici, une recette en particulier) :

* Les attributs (ici le slug et l’id)
* Requirements = définition du format attendu pour les paramètres de l’url (ici, id = simplement des nombres avec l’expression régulière)
* Id rendu sous forme d’entier avec la méthode getInt()

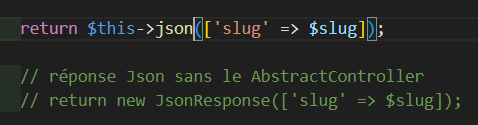
**

Autre possibilité : préciser les paramètres dans la méthode

**

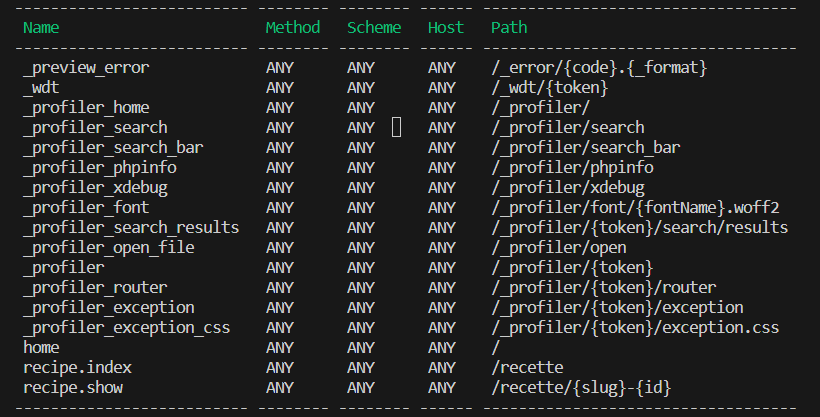
1. Utilisation de l’objet Response

Retour classique : return new Response(‘Recette ‘ . $slug) ;

On peut aussi utiliser la JsonResponse, avec ou sans le AbstractController**

1. Voir toutes les routes de l’app : php bin/console debug :router

On voit celles qu’on a créées mais aussi celles définies dans config/routes/



# **04 : Twig**

Cf. [documentation](https://twig.symfony.com/doc/3.x/)

Etendre le fichier de base : {% extends "base.html.twig" %}

Dans le contrôleur, rendre le template : return $this->render('recipe/index.html.twig');

Ecraser le contenu par défaut du base.html.twig, par exemple pour définir le titre de la page :

{% block title %}Toutes les recettes{% endblock %}

Syntaxe raccourcie pour les blocs si on n’a que du texte à mettre :

{% block title "Toutes les recettes"%}

Ajouter du CSS avec Bootstrap

Commenter le code de assets/styles/app.css pour éviter les conflits

<https://getbootstrap.com/> 🡪 copier le link de « Include via CDN »

Dans base.html.twig :

* Coller le lien du cdn de Bootstrap au-dessus du block stylesheet
* Dans le body, ajouter une div container :

    <body>

        <div class="container my-4">

            {% block body %}{% endblock %}

        </div>

    </body>

* On peut aussi copier-coller le code d’une navbar bootstrap et l’ajouter au-dessus de la div container

Afficher le slug sur ma page en le récupérant dans l’url

Dans le contrôleur, on passe à la fonction render en 2e paramètre un tableau des variables à envoyer à la vue :

render(

1e paramètre 🡪 ‘fichier-vue.html.twig’,

2e paramètre 🡪 [tableau des variables à y envoyer]

)

    return $this->render('recipe/show.html.twig', [

        'slug' => $slug,

        'id' => $id

    ]);

Créer le template recipe/show.html.twig ; grâce au contrôleur, j’ai accès aux variables et je peux les afficher :     {{ slug }} (le html est automatiquement échappé par Twig)

Accéder à des éléments dans un tableau

Contrôleur :      'person' => [

            'firstname' => 'Jane',

            'lastname' => 'Doe'

        ],

Vue : {{ person.firstname }}

Concaténer : {{ person.firstname ~ person.lastname }}

{% block title "Recette : " ~ slug %}

Appliquer des filtres

Afficher en majuscules : {{ slug |upper }}

Afficher le HTML (inverser l'échappement) : {{ echappee |raw }}

Echapper manuellement : {{ echappee |e }}

Les URL / lier deux pages

Utiliser la fonction Twig URL

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

On peut utiliser le tag include pour séparer le code en deux fichier et intégrer l’un à l’autre.

Url d’une recette particulière (avec slug et id)

* 1e paramètre = nom de l’url
* 2e paramètre = tableau associatif en Twig (même syntaxe que JS) :

<a href={{ url('recipe.show', {id:32, slug:'pate-bol'}) }}>Pâtes bolo</a>

Utiliser la fonction path() pour générer une url qui ne contient que le chemin (pas tout le domaine) :

<a href={{ path('recipe.show', {id:32, slug:'pate-bol'}) }}>Pâtes bolo</a>

Voir l’ensemble des fonctions que Symfony ajoute dans Twig : documentation [sur la configuration](https://symfony.com/doc/current/reference/configuration/twig.html) et [sur les extensions Twig](https://symfony.com/doc/current/reference/twig_reference.html).

La méthode dump() de Twig

<li>Dump de la variable globale app : {{ dump(app) }} </li>

<li>Dump de app.current\_route : {{ dump(app.current\_route) }} </li>

<li>Dump de la request : {{ dump(app.request) }} </li>

Utilisation :

{# si le lien courant est 'home', alors faire apparaître le lien comme actif, sinon ne rien faire #}

<a class="nav-link {{ app.current\_route == "home" ? 'active' : '' }}" href={{ url("home") }}>Accueil</a>

(remarque : pas de « === » en Twig)

Variante :

<a class="nav-link {{ app.current\_route starts with "recipe." ? 'active' : '' }}" href={{ url("recipe.index") }}>Recettes</a>

On a vu le système Twig qui permet de générer de jolies pages HTML et de structurer notre code avec un layout : on va mettre tout ce qui est commun entre les pages dans la partie base.html.twig, on peut aussi se créer des fichiers que l'on pourra inclure si c'est nécessaire et dans nos vues ; on fait un extend, on choisit le titre que l'on veut mettre, on définit un bloc body et on met le bloc à l'intérieur.

Et au niveau de notre contrôleu,r souvent on va avoir à la fin de notre code une méthode render, on lui passera en premier paramètre le nom du fichier twig que l'on souhaite utiliser et en second paramètre un tableau contenant tout ce que l'on souhaite envoyer à notre vue.

Au niveau de notre vu,e on a accès à une variable globale ‘app’ qui nous permet de récupérer des informations sur notre application et ensuite toutes les variables qui sont envoyées par le contrôleur.

# 05 : Doctrine

 Object Relational Mapping : Système pour communiquer avec la base de données avec des objets qui représentent les données présentes dans les tables.

Configurer et connecter à la bdd

* dans .env.local (copie de .env, entrer ses propres paramètres)
* on peut télécharger [Adminer](https://www.adminer.org/en/) et l’enregistrer dans le dossier public/ et l’ouvrir dans le navigateur. Là on saisit ses paramètres et on a une interface pour communiquer avec la bdd.

Créer une entité

php bin/console make:entity 🡪 nom et propriétés

🡪 dans le fichier Entity/Recipe.php :

* Les getters & setters sont automatiquement créés
* Les attributs des propriétés permettent à Doctrine de savoir comment sauvegarder dans la bdd

🡪 dans le Repository : des méthodes pour aller récupérer des enregistrements en bdd

Créer les tables dans la bdd

php bin/console make:migration

php bin/console doctrine:migrations:migrate

Récupérer des recettes enregistrées

Dans le contrôleur, utiliser le RecipeRepository en argument et appeler ses méthodes (find, findAll, findBy, findOneBy) :

public function index(Request $request, RecipeRepository $repository): Response

{

$recipes = $repository->findAll();

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

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

$recipe = $repository->find($id);

$recipe = $repository->findOneBy(['slug' => $slug]);

Dans la vue, créer une boucle :

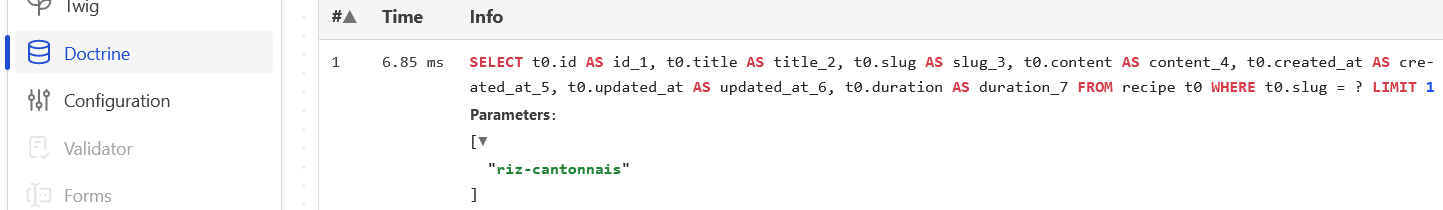
* en appelant le getter (peu utilisé)

<a href={{ url('recipe.show', {id: recipe.getId, slug: recipe.getSlug}) }}>{{ recipe.getTitle() }}</a>

* sans appeler le getter (le système cherche le getter)

<a href={{ url('recipe.show', {id: recipe.id, slug: recipe.slug}) }}>{{ recipe.title() }}</a>

On peut voir les requêtes SQL générées grâce à la debug bar :  



{# filtre pour que les sauts de ligne soient transférés #}

<p>{{ recipe.content |nl2br}}</p>

Faire une requête personnalisée

Afficher toutes les recettes qui ont une durée inférieure à 10 min :

Dans le RecipeRepository, créer une méthode personnalisée en utilisant le QueryBuilder de Doctrine : il génère les requêtes SQL en fonction du système de bdd utilisé. Il a des méthodes équivalentes aux requêtes SQL (where, orderBy, etc.)

    /\*\*

     \* @return Recipe[]

     \*/

    public function findWithDurationLowerThan(int $duration) : array

    {

        // 'r' est un alias, comme en SQL

        return $this->createQueryBuilder('r')

            ->where('r.duration < :duration')

            ->orderBy('r.duration', 'ASC')

            // prendre un seul résultat

            ->setMaxResults(1)

            ->setParameter('duration', $duration)

            //générer l'objet Query

            ->getQuery()

            ->getResult();

    }

Comment modifier un enregistrement ?

Il faut utiliser le EntityManagerInterface, classe qui mémorise toutes les entités de l’application, dans le contrôleur : $recipes[0]->setTitle(‘Pâtes boloniaises’) ;

$em->flush() ; // pour persister dans la bdd

(il peut y avoir des soucis, liés à UX turbo, qu’on peut supprimer : composer remove…)

Créer un nouvel enregistrement

1. Créer l’objet : $recipe = new Recipe;
2. Paramétrer ses propriétés

        $recipe->setTitle('Barbe à papa')

            ->setSlug('barbe-a-papa')

            ->setContent('Mettez du sucre')

            ->setDuration(2)

            ->setCreatedAt(new \DateTimeImmutable())

            ->setUpdatedAt(new \DateTimeImmutable());

1. Persister en base de données

Flush() ne suffit pas, car l’entity manager ne connaît pas l’existence de l’objet  :

        // l'entity manager doit enregistrer la présente de cet objet

        $em->persist($recipe);

        // pour que Doctrine compare avec la bdd et modifie:

        $em->flush() ;

Supprimer un enregistrement

       $em->remove($recipes[0]);

Astuce

L’entitty manager peut récupérer un repository : $em->getRepository(Recipe ::class) ;

Réucpéer la durée totale de mes recettes :

    public function findTotalDuration()

    {

        return $this->createQueryBuilder('r')

            ->select('SUM(r.duration) as total')

            ->getQuery()

            ->getSingleResult();

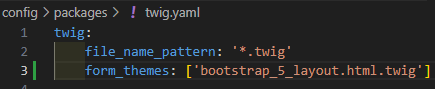
    }

A retenir : lorsqu'on a besoin de stocker des informations, on crée une nouvelle entité : on peut le faire en ligne de commande pour nous simplifier la vie, ensuite une fois qu'on a ces entités on va pouvoir les récupérer grâce au repository qui comprend des méthodes de base que l'on a vu comme find, findOneBy, findBy et findAll, mais on peut aussi créer nos propres méthodes en fonction des besoins, méthodes dans lesquelles on pourra utiliser le query Builder pour construire une requête SQL permettant de récupérer les données que l'on souhaite. Ensuite, si on souhaite faire des modifications, il nous suffit de modifier nos objets et d'utiliser l'entity manager et la méthode flush pour persister toutes les modifications qui ont été faites. De la même manière, si on souhaite créer un nouvel objet, on commence par créer une nouvelle entité en l’initialisant classiquement et ensuite on utilise la méthode persist pour dire à l'entity manager de suivre cet objet là. Une fois que l'on a persisté on peut utiliser la méthode flush pour mémoriser les informations en base de données ; enfin c'est un peu la même chose pour la suppression : on peut faire un remove et ensuite on flush pour enregistrer la suppression. On n’a plus forcément besoin d’écrire nous-mêmes les requêtes SQL, elles se génèrent pour nous ; la seule chose qu'on a besoin de faire, c'est expliquer comment récupèrer les donnée,s et après tout le reste se fait de manière assez « naturelle » : la modification la suppression et la création est beaucoup plus pratique que du SQL.

# 06 : Les formulaires

* make:form 🡪 src/form/RecipeType
* modifier contrôleur, créer la vue

**Utiliser un thème de mise en forme**



fihcier twig pilotant l’affichage des différents champs.  
On peut s’en inspirer et créer son propre thème.

**Mise en forme dans Twig**

{{ form\_start(form) }}

    {# je veux mettre des champs côte à côte #}

    <div class="d-flex">

        {{ form\_row(form.title)}}

        {{ form\_row(form.duration)}}

    </div>

{# j'affiche tous les autres champs #}

{{ form\_rest (form) }}

{#tous les champs non gérés ci-dessus apparaîtront là (redondant avec ci-dessus)#}

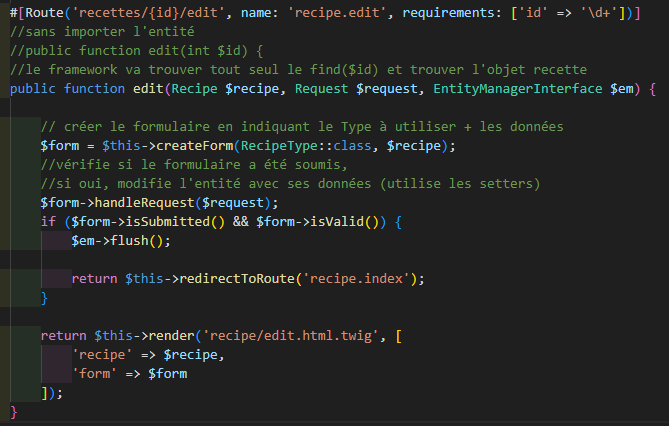
{{ form\_end(form) }}

**Ajouter un bouton de validation**

Note : opur « created at », le framework a détecté que c’était un DateTime et créé automatiquement un chmap de type date (avec un date picker).

Voir l’ensemble des chmaps disponibles dans la [documentation sur les formulaires](https://symfony.com/doc/current/forms.html) et dans la [partie références](https://symfony.com/doc/current/reference/forms/types.html).

**Gérer le traitement des données**



**Ajouter un message de succès**

message flash : sauvegarder un message en session, utilisé pour la prochaine requête puis effacée

* Contrôleur :           $this->addFlash('success', "La recette a bien été modifiée");
* Vue de base :         {% include 'partials/flash.html.twig' %}
* Créer ce fichier (les partials = bouts de vues twig) :



**Ajouter la fonctionnalité de création d’une nouvelle recette**

Contrôleur : créer une fonction comme edit, mais :

* On crée un nouvel objet Recipe vide à envoyer dans le formulaire
* On n’oublie pas le persist($recipe) pour enregistrer le nouvel objet et le setCreatedAT()

Créer un template + le bouton « créer » dans l’index

**Ajouter la fonctionnalité de suppression d’une recette**

Contrôleur : la fonction est très simple – attention : ajouter la méthode DELETE :

#[Route('recettes/{id}', name: 'recipe.delete', methods: ['DELETE'])]

Dans le template index des recettes, ajouter un bouton mais pour le sécuriser, ajouter un champ caché (nécessite un ajout de clé dans framework.yaml)

<form action="{{ path('recipe.delete', {id: recipe.id}) }}" method="post">

<input type="hidden" name="\_method" value="DELETE">

<button type="submit" class="btn btn-danger btn-sm">Supprimer</a>

</form>

**Ajouter un EventListener pour créer le slug automatiquement si le champ n’est pas rempli**

Dans RecipeType, créer une fonction qui crée le slug en utilisant le Slugger de Symfony

    public function autoSlug(PreSubmitEvent $event): void

    {

        $data = $event->getData();

        if(empty($data['slug'])) {

            $slugger = new AsciiSlugger();

            $data['slug'] = strtolower($slugger->slug($data['title']));

            $event->setData($data);

        }

    }

Dans le builder, ajouter l’appel de cette fonction et le listener :

            ->addEventListener(FormEvents::PRE\_SUBMIT, $this->autoSlug(...))

On peut Faire de même pour modifier / ajouter la date automatiquement

On a la possibilité en ligne de commande de **générer un formulaire** à partir d'une entité, ce qui aura pour effet de créer un formulaire avec directement les différents champs. Sur ces champs, on peut préciser le type que l'on veut à chaque fois et on peut préciser des attributs supplémentaires comme le libellé, required, etc. (on pourra rajouter des contraintes aussi cf. chapitre suivant).

Une fois qu'on a ce formulaire dans notre **contrôleur**, on peut l'initialiser en lui passant en premier paramètre le type de formulaire que l'on veut mettre et en second paramètre les données initiales. Ensuite, on utilisera la méthode handleRequest pour que notre formulaire traite la requête et on utilisera le is submitted et le is valide pour voir si notre formulaire a été rempli et est valide. Ensuite on définit le traitement qu’on veut, on peut ajouter un message flash.

Au niveau de **l'affichage** il nous suffit d'utiliser la fonction form pour afficher un formulaire complet ; sinon on peut utiliser des fonctions form start, form end, form row si on veut un petit peu plus de contrôle sur comment sont affichés les différents champs.

Enfin, on a vu la partie **événement** qui permet de rajouter de la logique lorsqu'un formulaire est soumis ou lorsque les données sont remplies : par exemple, ajouter de manière dynamique un Slug s'il est vide ou attacher les dates de modification et de création.

# 07 : Valider les données

Ajouter des contraintes au niveau des formulaires

Dans RecipeType :

* Attribut : par ex. ‘required’
* Clé ‘constraints’ -> Cf. [documentation](https://symfony.com/doc/current/reference/constraints.html)
* On peut définir un tableau de contraintes
* On peut ajouter un message pour chaque contrainte

            ->add('slug', TextType::class, [

                'required' => false,

                'constraints' => [

                    new Length(min: 10),

                    new Regex(

                        "/ ^[a-z0-9]+(?:-[a-z0-9]+)\*$ /",

                        message: "Ceci n'est pas un slug valide"

                    )

                ]

            ])

Autres contraintes :

* « All » -> à tous les éléments d’un tableau
* « AtLeastOneOf »
* « Sequentially » encadrant le tableau de contraintes : il s’arrête à la première contrainte qui échoue

                'constraints' => new Sequentially ([

                    new Length(min: 10),

                    new Regex("/ ^[a-z0-9]+(?:-[a-z0-9]+)\*$ /")

                ]

Ajouter des contraintes au niveau des entités

…

Alias pour plus de lisibilité

use Symfony\Component\Validator\Constraints as Assert;

…

class Recipe

{

…

    #[ORM\Column(length: 255)]

    //contrainte de validation

    #[Assert\Length(min: 5)]

    #[Assert\Regex(

        "/ ^[a-z0-9]+(?:-[a-z0-9]+)\*$ /",

        message: "Certains caractères ne sont pas acceptés."

    )]

    private ?string $slug = null;

…

    #[ORM\Column(nullable: true)]

    //Valeur non négative (contrainte uniquement pour des entiers) et pas zéro

    #[Assert\Positive()]

    #[Assert\NotBlank()]

    private ?int $duration;

Une contrainte *required* a été automatiquement mise par le composant formulaire dès qu'il voit que quelque chose a la règle de validation NotBlank.

On met les règles de validation très souvent plutôt sur les entités ou sur les classes qui vont représenter les données associées au formulaire parce que ça permet de décrire directement là ce qu’on considère comme une valeur valide ou non. L'avantage c'est qu’on pourra plus tard valider l’entité même si elle a été hydratée par d'autres moyens que le formulaire.

Les contraintes sont intéressantes au niveau du formulaire si on a des champs qui ne correspondent à rien au niveau de l’entité.

Contraintes agissant directement sur les entités

[Contrainte UniqueEntity](https://symfony.com/doc/current/reference/constraints/UniqueEntity.html) se met au niveau de la classe en précisant sur quel(s) champs elle porte (attention, c’est dans le namespace Doctrine et non dans Assertion classique) :

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

use Symfony\Component\Validator\Constraints as Assert;

#[ORM\Entity(repositoryClass: RecipeRepository::class)]

#[UniqueEntity('title')]

#[UniqueEntity('slug')]

class Recipe {…}

On peut aussi y passer en paramètre un tableau de champs : dans ce cas, c’est la combinaison des champs qui devra être valide.

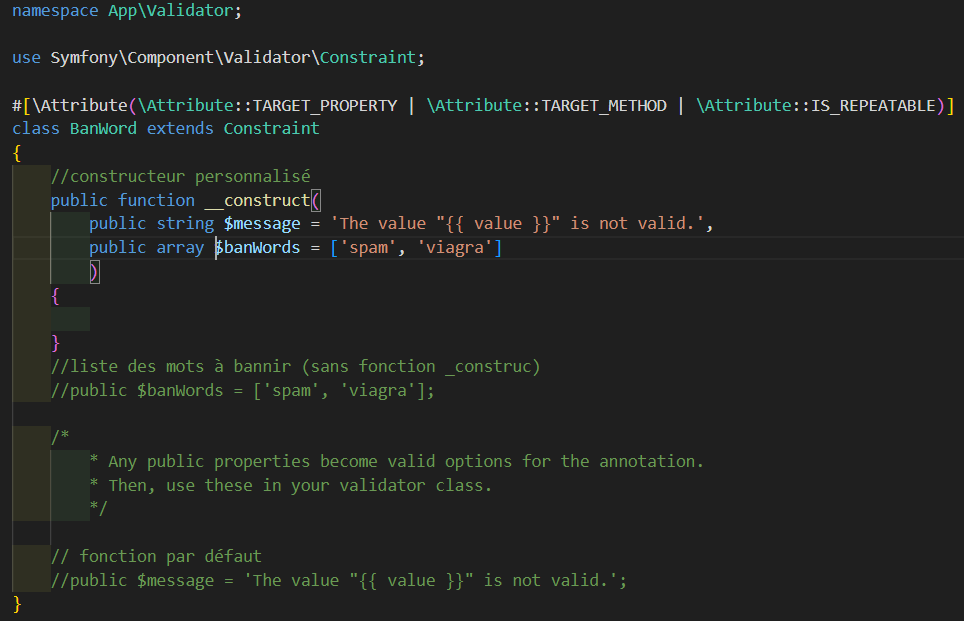
[Contraintes de comparaison](https://symfony.com/doc/current/reference/constraints/LessThan.html) : LessThan…

**Contraintes plus poussées : par exemple, mots bannis du titre**

bin/console make :validator (on l’appelle BanWordValidator)

🡪 ***validator/BanWord.php*** : définit les contraintes ; on peut le customiser avec un constructeur.

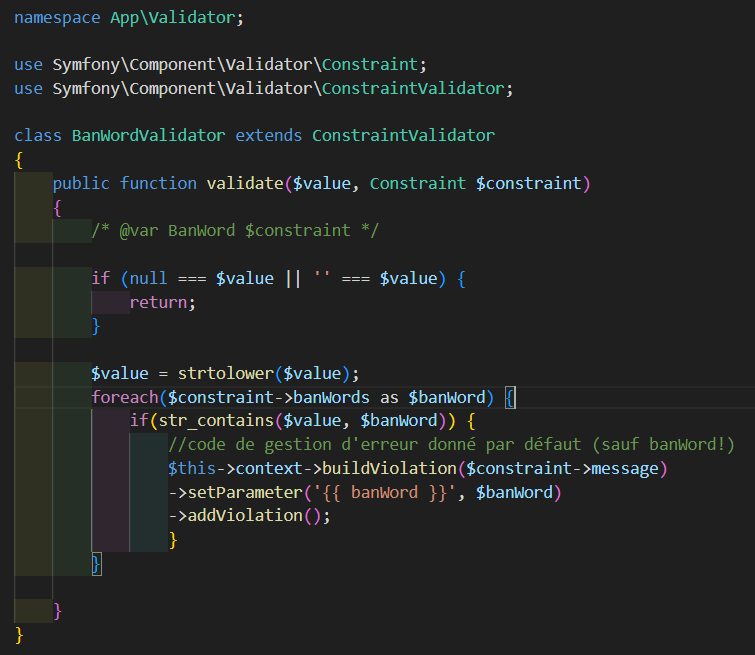
On peut retirer le PHPDoc (lié aux anciennes annotations, déprécié).



*elles s’appliquent aux méthodes et aux propriétés*

Ensuite on utilise cette nouvelle contrainte dans l’entité #[BanWord()]

🡪 ***validator/BanWordValidator.php*** : le validateur associé (vérifie les valeurs vis-à-vis des contraintes)



On peut aussi adapter le message dans le fichier de contraintes :

        public string $message = 'This contains a banned word: "{{ banWord }}".',

On peut aussi ajouter des [groupes de validation](https://symfony.com/doc/current/reference/forms/types/form.html#validation-groups) pour regrouper les contraintes., ce qui permet d’activer ou désactiver des règles dans chaque formulaire (Type -> dans les options). On les ajoute dans les attributs des entités.

C’est directement au niveau de l'entité que l'on va définir ce que l'on considère comme une entité valide et le système de formulaire va détecter ces attributs et faire en sorte de valider les données lorsqu'elles sont soumises.

De plus, automatiquement il modifie le formulaire HTML pour prendre en compte ses validations et fait en sorte qu’on ait des validations HTML 5, ce qui permet d'éviter que l'utilisateur soumette des informations en amont (il va quand même faire la validation côté serveur aussi bien sûr !)

De plus il y a une gestion automatique de l’affichage de messages au niveau du formulaire.

Si on le souhaite pour des cas plus compliqué on a la possibilité de créer des contraintes de validation personnalisées grâce au système de contrainte et de Validator.

Enfin on notera que certains champs mettent des contraintes par défaut typiquement lorsqu'on a un champ de type number automatiquement il spécifie que ça doit être un nombre donc c'est géré en interne.

**Champ vide / champ nul**

Spécifier qu’un champ peut être nul ne suffit pas : par exemple, si j’autorise title à être nul mais que j’ai défini qu’il doit être une chaîne de caractère, j’ai une erreur si je ne précise par une valeur par défaut :

            ->add('title', TextType::class, [

                'empty\_data' => ''

            ])

Ce qui suppose par cohérence de ne pas laisser la possibilité « null » dans l’entité (au niveau des propriétés, getters et setters)

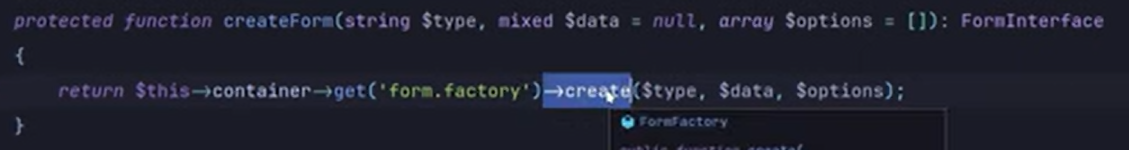
    private string $title = ''; // et non : private ?string $title = null;

# 08 : Comprendre les services

Lorsqu'on a eu besoin de communiquer avec la base de données ou de certains éléments on a directement mis dans les actions de nos contrôleurs les classes dont on avait besoin : on a utilisé le Recipe repository et comme par magie on se retrouvait avec une instance de notre Recipe repository ; de la même manière on lui a dit j'ai besoin d'un élément qui implémente l'entity manager interface et le framework a été capable de nous fournir automatiquement une classe qui correspond à cette interface. **Comment ça fonctionne ?**

Au cœur du framework on a un système de **container** : c'est une sorte de gros objet qui contient l'ensemble des classes dont on a besoin et aussi les méthodes qui permettre de les construire.

Un container c'est une classe qui contient des **clés** et pour chaque clé on a une fonction qui permettre de construire le service dont on a besoin d'instancier la classe : lorsque je fais un create form il va demander au container est-ce que tu peux me donner la classe qui a la clé form Factory et automatiquement ça va nous donner une instance de ce form Factory, instance sur laquelle il appelle ensuite la méthode Create



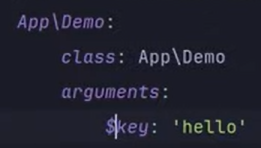
php bin/console debug :autowiring

🡪 toutes les classes disponibles dans le système de conainter de symfony

Idem + nom de la classe pour trouver spécifiquement :

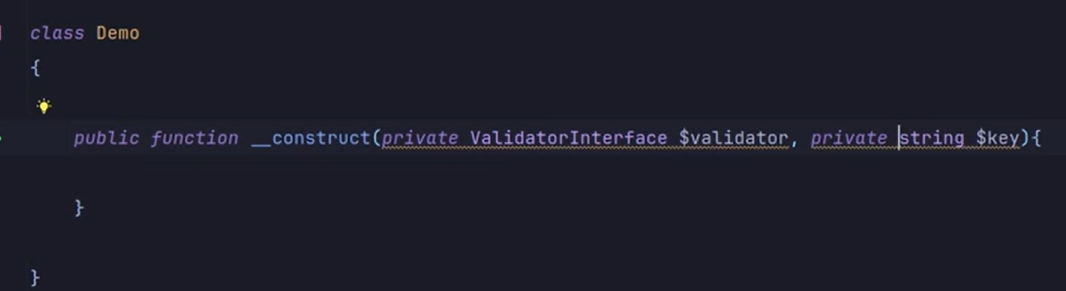
* Par exemple, je tabe php bin/console debug :autowiring mail pour trouver ce qu’il existe comme service de mail
* Ça marche aussi avec les namespaces de mon app (les classes que j’ai créées moi-même)

**Configuré dans config/services.yaml :**

* Autowiring et autoconfig à true,
* App/ est incluse dans les services
* On peut ajouter les classes à construire de manière particulière : 

Un service est une classe qu’on peut brancher de manière automatique dans nos contrôleurs.

Si j’ai besoin du validateur dans ma classe, je vais faire de l’injection de dépendance :



C’est exactement ce qui se passe quand on injecte un repository.

**Le câblage automatique (auto wiring) fonctionne donc à 2 endroits :**

* Pour un service injecté sur le container (dans la fonction \_\_construct() uniquement)
* Pour tous les contrôleurs (dans toutes leurs méthodes ; on peut aussi créer un constructeur avec le service comme propriété private afin de l’appeler avec $this dans tout le contrôleur, sans avoir à le réinjecter)

**Autre manière d’obtenir un service (moins recommandé) : accéder directement au container dans les contrôleurs qui étendent du AbstractController :**

$this->container->get(‘form’) ; // par l’alias du service !

Valable pour les services publics ; privés ne sont pas accessibles par le container.

***Symfony = un gros container plein de services préconfigurés !***

*(pour la version webapp, pas la version squelette)*

# 09 : TP : Formulaire avec envoi d’e-mail

créer un formulaire de contact où

* l'utilisateur peut rentrer son nom son email et en dessous un message
* lorsqu'il clique sur Envoyer il faudra automatiquement envoyer un email
* on ne sauvegardera pas les demandes de contact dans notre base de données

comme on l'a vu l'objet formulaire permet de recevoir un objet qui représente les données nous on lui avait donné une entité dans le cadre de nos recettes de cuisine mais on peut lui donner un objet classique

on va créer un objet que l'on va appeler contact forme DTO ou contact DTO qui va contenir les différentes propriétés

DTO : data transfer object ; c'est un objet qui permet de représenter des données qui sont transférées on pourrait utiliser simplement un tableau mais l'avantage des objet c'est que ça nous permet d'avoir de bons types

**Modifier le fichier de configuration messenger.yaml**

Messenger c'est un système qui permet de gérer des files d'attente

* Activer pour créer une ligne synchrone = qui ne passe pas par Messenger

sync: 'sync://'

* Ajouter ces éléments dans la file d’attente synchrone

        routing:

            Symfony\Component\Mailer\Messenger\SendEmailMessage: ~~async~~ sync

            Symfony\Component\Notifier\Message\ChatMessage: ~~async~~ sync

            Symfony\Component\Notifier\Message\SmsMessage: ~~async~~ sync

**Créer un faux serveur mail pour le développement**

Car l’ordi n’est pas configuré pour ça. Plusieurs possibilités :

🡪 [mailpit](https://github.com/axllent/mailpit/releases/tag/v1.18.5) (simle exécutable à mettre dans bin/)

🡪 maildev (il faut node.js)

🡪 utiliser le serveur SMTP de l’ufa ; configuration dans .env :

MAILER\_DSN=smtp://mail.dfh-ufa.org:25

**Créer l’objet représentant les données du formulaire**

src/DTO/ContactDTO.php avec les propriétés $name, $email et $message

**Créer le formulaire ContactType**

Comme nom d’ « entité », donner : \App\DTO\ContactDTO

Y ajouter

* les types de champs et empty\_data
* un bouton de soumission :

            ->add('submit', SubmitType::class, [

                'label' => 'Envoyer'

            ])

**Créer le contrôleur**

Avec une instance de ContactDTO et un formulaire, à afficher dans le template correspondant.

**Gérer la validation**

Attributs au niveau de l’objet : #[Assert\NotBlank]

#[Assert\Email] //Voir la [documentation](https://symfony.com/doc/current/reference/constraints/Email.html)

 #[Assert\Length(min: 3, max: 200)]

**L’envoi d’e-mails**

[Documentation](https://symfony.com/doc/current/mailer.html) sur Mailer

Comme chaque package, il a sa propre configuraiton dans config\packages\mailer.yaml, qui fait simplement appel à la variable d’environnement DSN\_MAILER

Utiliser le [code d’exemple](https://symfony.com/doc/current/mailer.html#creating-sending-messages) de la documentation pour le contrôleur :

* injection de la MailerInterface
* pour représenter l’e-mail, création d’une instance de
  + Email (du namespace  Symfony\Component\Mime\Email) pour construire directement un contenu en html
  + TemplatedEmail pour un contenu en twig

<?php

namespace App\Controller;

use App\DTO\ContactDTO;

use App\Form\ContactType;

use Symfony\Bridge\Twig\Mime\TemplatedEmail;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

use Symfony\Component\HttpFoundation\Request;

use Symfony\Component\HttpFoundation\Response;

use Symfony\Component\Mailer\MailerInterface;

use Symfony\Component\Routing\Attribute\Route;

class ContactController extends AbstractController

{

    #[Route('/contact', name: 'contact')]

    public function contact(Request $request, MailerInterface $mailer): Response

    {

        $data = new ContactDTO;

        $form = $this->createForm(ContactType::class, $data);

        $form->handleRequest($request);

        if($form->isSubmitted() && $form->isValid()) {

            //Création de l'email avec un template Twig

            $mail = (new TemplatedEmail())

                ->to('jegourel@dfh-ufa.org')

                ->from($data->email)    //l'email saisi par l'U

                ->subject('Demande de contact')

                ->htmlTemplate('emails/contact.html.twig')

                ->context(['data' => $data]);

            //envoi de l'e-mail

            $mailer->send($mail);

            $this->addFlash('success', "Votre e-mail a bien été envoyé");

            return $this->redirectToRoute('contact');

        }

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

            'form' => $form,

        ]);

    }

}

Créer le template pour le message.

Outil & package pour formater les e-mails : [Inky](https://get.foundation/emails/docs/inky.html) (cf. documentation de Symfony Mailer)

**Ajouter un select pour contacter différentes personnes au choix**

* Dans ContactDTO, ajouter la propriété string $service
* Dans ContactType, ajouter cette propriété en ChoiceType\*
* Dans le template, ajouter la ligne {{ form.service}}
* Dans le contrôleur, ajouter l’email sélectionné $mail->to($data->service)

\*ChoiceType permet de créer un select ou des radios

EnumType permet de créer un select basé sur un énumérateur

**Gérer le cas où l’e-mail ne s’envoie pas correctement**

En ajoutant un try catch à la partie création & envoi d’e-mail du contrôleur.

# 10 : TP : Catégories, interface admin, service FormListenerFactory

**Amélioration des routes et création d’un namespace Admin**

Créer un sous-dossier src/Controller/Admin/

Mettre une url générique pour la classe & modifier les url des méthodes

#[Route('/admin/recettes', name: 'admin.recipe.')]

class RecipeController extends AbstractController

{…

#[Route('/{id}', name: 'delete', methods: ['DELETE'], requirements:

[Requirement::DIGITS])]

…}

Ajouter des requirements en utilisant l’énumérateur de Symfony (DIGITS ci-dessus = chiffres)

Créer un sous-dossier templates/recipe/admin & modifier les url des render() et redirections dans le contrôleur, les path dans la vue.

**Séparation du front-end et du back-end**

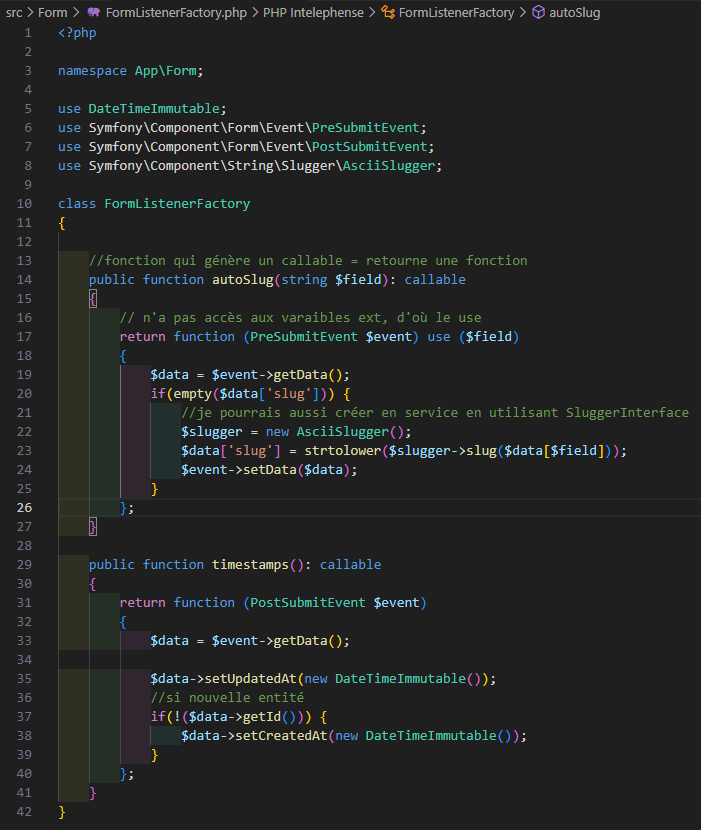
Créer un template admin/admin.html.twig

* avec le contenu de base.html.twig
* en modifiant les url
* modifier les templates avec extends admin/admin.html.twig

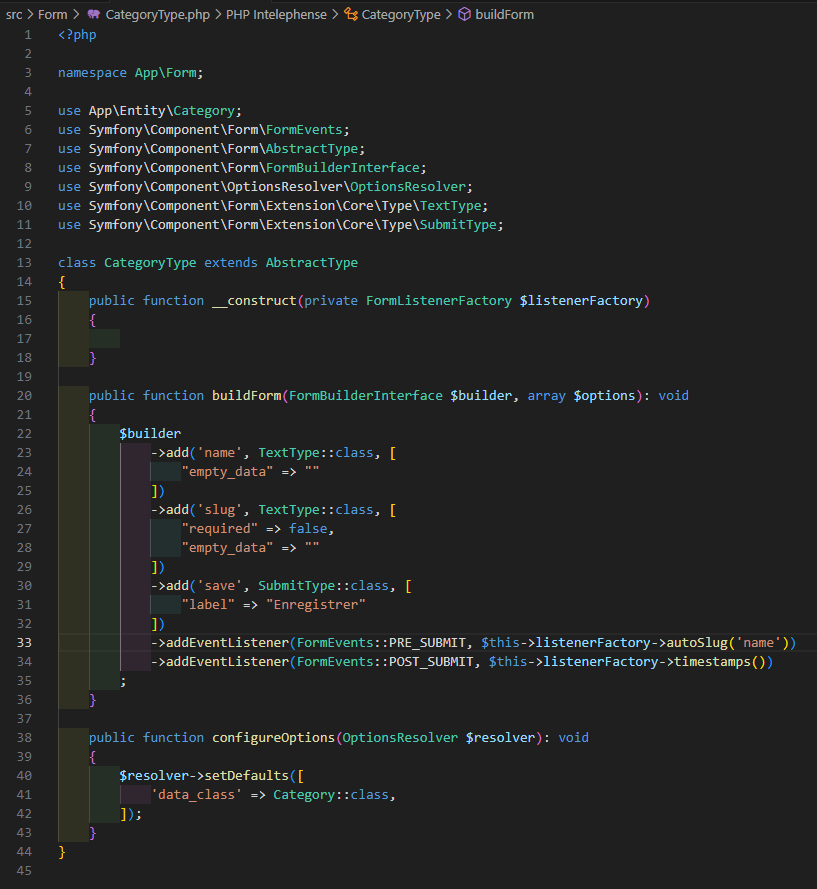
**Créer des catégories de recettes**

= créer un crud

**Créer un service de listener = une classe permettant de générer des listeners**



L’utiliser au niveau des formulaires :



# 11 : ORM relation ManyToOne

Relier les recettes et leurs categories = une clé étrangère en base de données ; l’orm permet de le faire de façon automatique.

**Créer la relation**

Modifier l’entité Recipe : php bin/console make:entity Recipe

🡪 on ajoute un champ « category » de type ManyToOne

(on peut taper « relation » pour avoir les différents types de relations)

    /\*\*

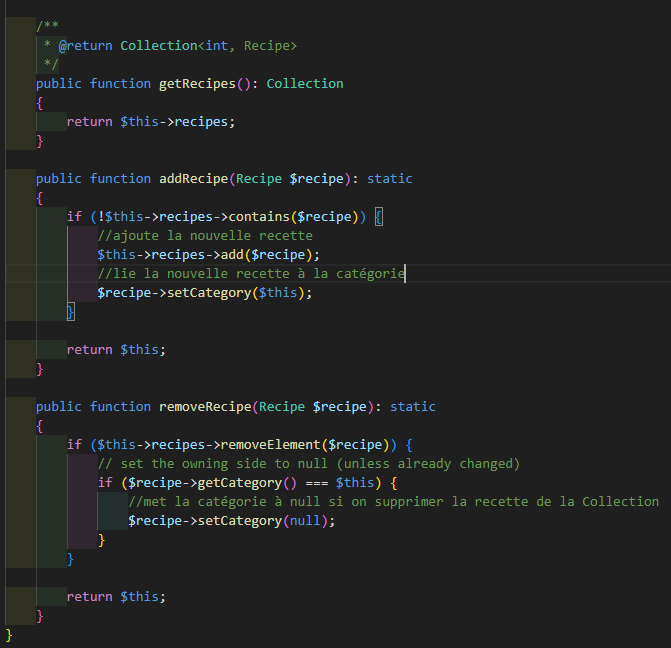
     \* @var Collection<int, Recipe>

     \*/

    #[ORM\OneToMany(targetEntity: Recipe::class, mappedBy: 'category')]

    private Collection $recipes;

Une Collection est un tableau amélioré avec des méthodes intéressantes.



Php bin/console make : migration -> ajoute clé étrangère

**Afficher les catégories**

Dans l’index des recettes :

            <th>Catégorie</th>

…

            <td>{{ recipe.category.name | default('') }}</td>

**Récupérer les catégories de recettes au niveau du Query Builder (repository)**

return $this->createQueryBuilder('r')

    //récupérer les infos concernant les recettes, mais aussi les catégories

    ->select("r", "c")

    //faire la liaison pour récupérer les catégories

    ->leftJoin("r.category", "c")

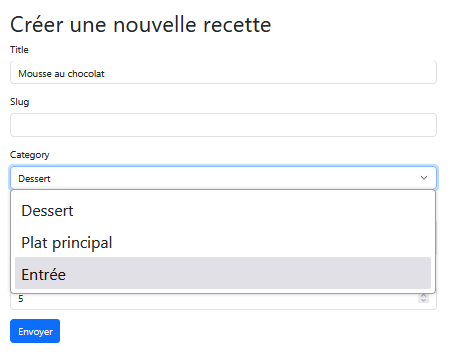
    //filtrer seulement les plats principaux

    ->andWhere("c.slug = 'plat-principal'")

On ne peut pas faire ça :    ->andWhere("r.category\_id = 1")

…car Doctrine ne prend en compte que les champs existant dans l’entité.

Pour faire une sélection sur l’id : ->andWhere("r.category = 1")

**Modifier la catégorie d’une recette depuis le formulaire**

Choicefield [EntityType](https://symfony.com/doc/current/reference/forms/types/entity.html): pour spécifier l’entité associée

->add('category', EntityType::class, [

'class' => Category::class,

//menu de labels corresopndant

au champ name

'choice\_label' => 'name'

])

**Modifier les recettes liées à une catégorie depuis le formulaire**

****

->add('recipes', EntityType::class, [

     'class' => Recipe::class,

     'choice\_label' => 'title',

     //pour pouvoir en sélectionner plusieurs

     'multiple' => true,

     'by\_reference' => false

Il faut utiliser l’option [by\_reference](https://symfony.com/doc/current/reference/forms/types/entity.html#by-reference)pour que ça soit

enregistré en bdd :

* true par défaut : il cherchera le setter
* false : cherchera add() et remove()

Pour avoir des checkbox, il faut ajouter

     expanded => true,

(si je mets cette option dans le RecipeType, j’aurai un bouton radio,

car il ne peut pas y avoir plusieurs catégories associées à une recette.

**La persistance en cascade**

Dans l’entité Recipe , attribut pour qu’il la persiste automatiquement lorsq’une novuelle catégorie est créée :

    #[ORM\ManyToOne(inversedBy: 'recipes', cascade: ['persist'])]

    private ?Category $category = null;

Dans l’entité Category, attribut pour qu’il supprime les recettes associées quand je supprimer une catégorie :

#[ORM\OneToMany(targetEntity: Recipe::class, mappedBy: 'category', cascade:

['remove'])]

private Collection $recipes;

Voir la [documentation](https://www.doctrine-project.org/projects/doctrine-orm/en/3.2/reference/working-with-associations.html#transitive-persistence-cascade-operations) de doctrine (voir aussi orpheanRemoval pour supprimer un élément automatiquement quand sa relation à un autre est supprimée).

# 12 : Envoi de fichiers

**Dans l’entité Recipe**

Ajouter un champ pour sauvegarder le chemin (string thumbnail)

**Dans le RecipeType**

ajouter Un champ fichier [de type FileType](https://symfony.com/doc/current/reference/forms/types/file.html#basic-usage) dans le champ formulaire, champ qui n’existe pas vraiment en bdd :

            ->add('thumbnailFile', FileType::class, [

                //ne sera pas mappé vers un champ de la bdd

(pas de recherche des getters et setters)

                'mapped' => false,

                //il faut donc ajouter les contraintes car pas d'entité

                'constraints' => [

                    //il faut que ce soit une image

                    new Image()

                ]

            ])

**Dans le RecipeController**

if ($form->isSubmitted() && $form->isValid()) {

/\*\* @var UploadedFile $file \*/

$file = $form->get('thumbnailFile')->getData();

//on crée le nom du fichier en récupérant l'extesion du fichier d'origine

$filename = $recipe->getId() . '.' . $file->getClientOriginalExtension();

//un objet de type UploadedFile dispose de cette méthode

//on déplace le 2e élément dans le 1e élément

$file->move($this->getParameter('kernel.project\_dir') .

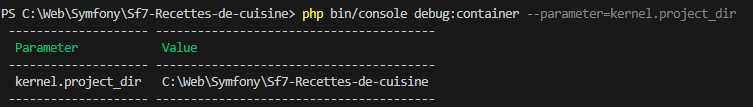
'/public/recettes/images', $filename);

$recipe->setThumbnail($filename);

Pour définir le chemin du dossier :

* on accède aux paramètres du framework par $this->getParameter()
* Pour trouver le nom du paramètre : php bin/console debug :container –parameters

Cela donne plein d’alias et leus valeurs associées, ainsi :



**Dans la vue edit**

        <img src="/recettes/images/{{ recipe.thumbnail }}" alt="">

Mais il faut aussi persister en cascade, etc.

**Vich Uploader Bundle**

Voir la [documentation](https://github.com/dustin10/VichUploaderBundle)

composer require vich/uploader-bundle 🡪 exécuter la recette ? – oui (packages/yaml, bundles…)

**Définir un maapping**

= créer un alias pour gérer les fichiers : dans config/packages/vich\_uploader.yaml

* Uri = l’endroit où c’est dans nos dossiers du projet
* On retrouve le paramètre kernel.project\_dir, entouré de % qui permettent à Symfony d’automatiquement remplacer
* Namer = classe qui a pour fonction de nommer un fichier. Elle implémente la NamerInterface qui a une seule méthode : name().

**Dans l’entité Recipe**

Créer une privée qui représente un fichier (+ getter et setter)

Gérer l’upload de fichier et mettre son nom dans cette propriété : ajouter des attributs

use Vich\UploaderBundle\Mapping\Annotation as Vich;

…

#[Vich\Uploadable]

class Recipe

…

    #[Vich\UploadableField(mapping: 'recipes', fileNameProperty: 'thumbnail')]

    #[Assert\Image()]

    private ?File $thumbnailFile = null;

…

* Signaler que cette classe contiendra des fichiers qui peuvent être uploadés
* Relier la propriété au mapping défini dans le yaml + récupérer les infos du fichier envoyé et sauvegarder le nom créé dans la propriété ‘thumbnail’
* Contraindre le type de propriété : Image

🡪On peut supprimer

* dans le RecipeType, contrainte et mapping
* dans le contrôleur, toute la logique perosnnalisée
* dans le template, l’affichage de l’image( ?!)

Lorsqu’on supprimer une recette, son image disparaît.  
Lorsqu’on la modifie et choisit une nouvelle image, celle-ci est supprimée.

**Récupérer l’url de l’image avec un helper**

Dans la vue – 2 paramètres :

* entité sur laquelle récupérer l’image
* nom du champ sur lequel j’ai mis mon uploaded file

        <img src="{{ vich\_uploader\_asset(recipe, 'thumbnailFile') }}" alt="">

🡪 génère l’url dynamiquement

Il existe aussi un helper à injecter dans le contrôleur si c’est là qu’on veut récupérer l’url de l’image

* UploaderHelper de Vich
* Utiliser la méthode asset() avec la même signature que ci-dessus

    public function edit(…, UploaderHelper $helper) {

        $fileUrl = $helper->asset($recipe, 'thumbnailFile');

**2 remarques**

1. Le sustème d’upload de fichier ne se déclenche que si au moment du flush il y a une modification de l’entité : pas que envoyer l’image, mais changement d’un autre champ (ici on a les événements de post submit)
2. Sur le [système de fichiers : celui de symfony](https://symfony.com/doc/current/components/filesystem.html) est essentiellement basé sur le local ; il faut passer par une librairie tiers si on veut utiliser s3 ou autre 🡪 Gaufrette, Flysystem : on peut brancher tout système de stocakge hors Symfony. Par exemple pour redimensionner les images ([Glide](https://www.glideapps.com/?utm_source=community&utm_medium=referral&utm_campaign=forum-header)).

# 13 : Le composant Security

Objectif : que seule l’admin puisse gérer les contenus & que les utilisateurs puissent créer leur propres recettes sur une interface dédiée.

* Système d’authentification
* Enregistrer les utilisateurs en bdd
* Formulaire de connection
* Formulaire d’inscription

**Créer l’entité utilisateur**

Php bin/console make :user

🡪 nom de l’entité : User ; propriété d’identification : username, hacher mdp : oui

l’entité User (propriétés : id, username, role) implémente

* UserInteface -> getRoles(), getUserIdentifier()
* PasswordAuthenticatedUserInterface -> getPassword()

UserRepository : upgradePassword() permet de re-hacher le mot de passe

Security.yaml définit la configuration du composant

* Clé password hashers définit les classes à utiliser pour hacher le mdp ; utilisation du hacher automatique
* Provider explique comment récupéerre des utilisateurs (quelle classe et quelle propriété)
* Firewalls : de dev et principal = lazy (ne se charge que qd on accède à l’utilisateur, en utilisatn le provider)

**Protéger une action dans un contrôleur**

AbstractController ::denyAccessUnlessGranted(//permission)-> sinon, Exception

public function index(…): Response {

        $this->denyAccessUnlessGranted('ROLE\_USER');

…}

**Créer un système d’authentification**

Avec la base donnée par le Maker Bundle

Php bin/console make :auth marche toujours mais est deprecated, il faut préférer la commande :

Php bin/console make :security

Et on choisit ici 1 : login form authenticator 🡪 AppAuthenticator ; SecurityController ; url logout : yes ; se souvenir de moi : oui, utilisateur doit cocher une case pour l’activer

Cf [documentation](https://symfony.com/doc/current/security.html#authenticating-users)

**AppAuthenticator** :

* Authenticate() return Passport avec : UserBadge ; PasswordCredentials qui vérifie mot de passe haché (bdd) ; vérifie csrf token ; rememberMe (crée cookie)
* Le composant regarde le Passport ->onAuthenticationSuccess :
  + Par défaut, redirige l’U vers la page cherchée au début (géré par TargetPathTrait);
  + On peut ajouter une redirection en utilisant UrlGeneratorInterface (injecté dans le constructeur) :

return new RedirectResponse($this->urlGenerator->generate('admin.recipe.index'));

étend **AbstractLoginFormAuthenticator**,

* ***supports***(Request)//gère la requête si dans url login + méthode post
* ou ->onAuthenticationFailure : sauvegarde erreurs d’auth en session + redirige vers login
* ***createToken(****Passport $passport, string firewallName****) : TokenInterface***

qui étend **AbstractAuthenticator**,

qui implément **AuthenticatorInterface**

* ***supports***(Request) :bool // l’auth gère la requête si true
* ***authenticate***(Request) :Passport // se déclenche si supports() true
* ***createToken(****Passport $passport, string firewallName****) : TokenInterface***

ET **AuthenticationEntrypointInterface**,

* start() // indique comment démarrer l’authentification : récupère l’url de login et redirige l’utilisateur (c’est ça qui permet la redirection vers la page login lors d’une erreur de connection)

**SecurityController** :

* Méthode login() : Récupère le service AuthenticationUtils
  + Où elle récupère les éventuelles erreurs,
  + Le dernier nom d’utilisateur saisi,
  + Et les renvoie à la vue
  + Méthode logout() : aucune logique, action jamais appelée car le système de sécurité intercepte (Par les options du pare-feu dans le security.yaml qui court-circuite)

**templates/security/login.html.twig**

* Si erreur : l’affiche
* app.user = récupère l’U actuellement connecté
* puis formulaire
  + classique avec les différentes champs
  + un champ caché qui permet d’avoir un token csrf (celui qui est validé lorsque dans le AppAuthenticator on lui passe le CsrfTokenBadge)

**Personnaliser le système d’authentification**

* Ajouter le champ email dans User & migrer
* Tester en créant un utilisateur par un contrôleur

    function index(…, UserPasswordHasherInterface $hasher): Response

    {

        $user = new User;

        $user->setEmail('jane@doe.be')

            ->setUsername('Jaja')

            ->setPassword($hasher->hashPassword($user, '0000'))

            ->setRoles([]);

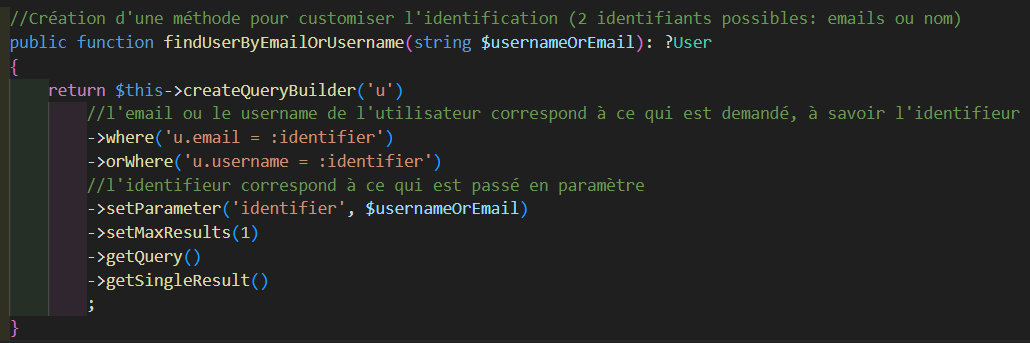
            $em->persist($user);

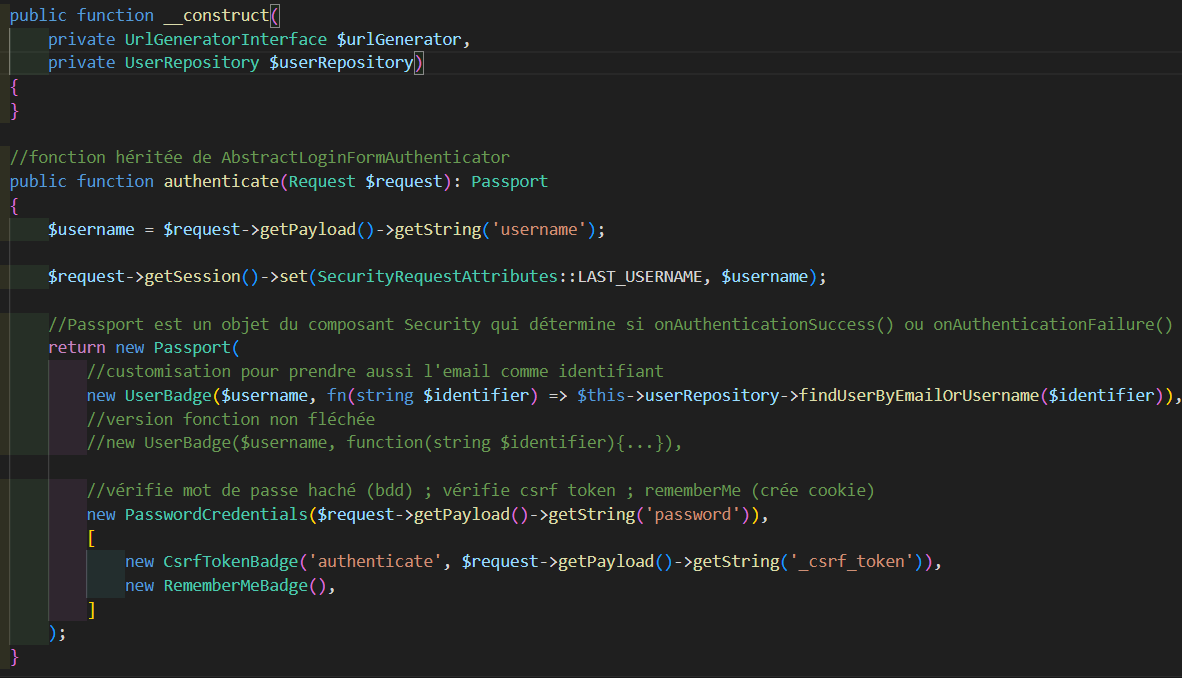
            $em->flush();

}

* Ajouter la possibilité de s’identifier avec son e-mail aussi ; pour cela, modifier UserBadge :
  + son constructeur reçoit le user identifier (ce qui est saisi dans ce champ) et un callable qui dit comment charger l’utilisateur.
  + Par défaut, il fait find() et trouve l’U par le username
  + Je peux le customiser en injectant le userRepository où j’aurai créé une méthode permettant de récupéer l’U par l’email ou par le champ username

UserRepository :





AppAuthenticator

**22’ Fonction Remember me**

Si on le désactive dans le security.yaml, il ne s’activera que si dans la requête on a une activité particulière

**Affiner les autorisations**

On peut remplacer denyAccessUnlessGranted() par l’attribut     #[IsGranted('ROLE\_USER')] au niveau de la méthode ou de la classe.

Ajouter un rôle dans l’entité (on peut aussi les sauvegarder en bdd, faire un crud pour les rôles…) :

    public function getRoles(): array

    {

        $roles = $this->roles;

        // guarantee every user at least has ROLE\_USER

        $roles[] = 'ROLE\_USER';

        if($this->email === 'jane@doe.be') {

            $roles[] = 'ROLE\_ADMIN';

        }

        return array\_unique($roles);

    }

**(25’) Connexion des utilisateurs**

Php bin/console make :registration-form

* uniqentity : yes ;
* verification email : yes ;
* ajouter l’id de l’U dans l’e-mail de confirmation (donc connexion automatique si l’U clique sur le lien reçu) : non ;
* choix de l’adresse d’envoi : support[at]demo.fr ;
* et le nom : Support ;
* auth automatique quand identifié : oui

Il faut aussi :

* Ajouter ce bundle : composer require symfonycasts/verify-email-bundle
* Faire une migration pour que s’ajoute le champ is\_verified dans User

RegisterController :

register()

* Crée un User et un formulaire
* Si le formulaire est valide, on hache le mdp
* Puis on persiste le User & on flush
* Utilise emailverifyer pour envoyer l’email de vérification
* Après quoi, avec UserAuthenticatorUtil, il authentifie l’utilisateur

2e action vérifie que l’email fonctionne bien.

**Customisation**

* On peut ajouter un r^le IS\_VERIFIED, et sélectionner seulement les U vérifiés dans le Repo… et autoriser certaines pages seulement avec vérification.
* Changer la vue en fonciton de l’état d’autthentification de l’utilisateur

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

        <div class="mb-3">

            Tu vois ce lien en plus :

<a href="{{ path('app\_logout') }}">veinard\*e!</a>

        </div>

    {% endif %}

* Récupérer des informations sur l’utilisateur
  + au niveau du cont^roleur : $this->getUser()
  + dans un service : Symfony/Bundle/SecurityBundle->getUser()
  + idem : Symfony/Bundle/SecurityBundle->getToken(), dans lequel je vois les roles, l’objet user associé, et d’autres attribut
* le token est sauvegardé dans la session, et à la session suivante, l’utiliser pour dire que l’U est authentifié (visible dans le profiler)

# 14 : Paginer les données

***En utilisant l’objet pagination de Doctrine***

Cf [documentation](https://www.doctrine-project.org/projects/doctrine-orm/en/3.2/tutorials/pagination.html) – cet objet :

* prend en paramètre un objet de type query
* implémente automatiquement implémenter ‘contable’ et ‘itérator aggregate interface’ qui sont deux permettant respectivement de compter et de boucler sur les résultats.

**Dans RecipeRepository, créer une nouvelle fonction :**

* renvoie objet Paginator du namespace de Doctrine

    public function paginateRecipes(Request $request): Paginator

    {

        return new Paginator($this

            ->createQueryBuilder('r')

            ->setFirstResult(0)

            ->setMaxResults(2)

        );

    }

**Dans RecipeController, utiliser cette fonction :**

        $recipes = $repository->paginateRecipes($request);

**Optimiser**

en désactivant la requête DISTINCT – dans le Repository :

    public function paginateRecipes(Request $request): Paginator

    {

        return new Paginator($this

            ->createQueryBuilder('r')

            ->setFirstResult(0)

            ->setMaxResults(2)

            //transforme objet QueryBuilder en Query

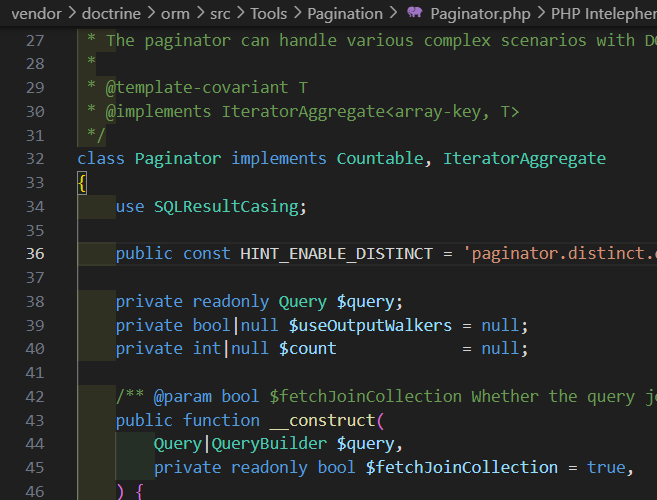
            ->getQuery()

            //passe petites infos pour mieux gérer requêtes SQL

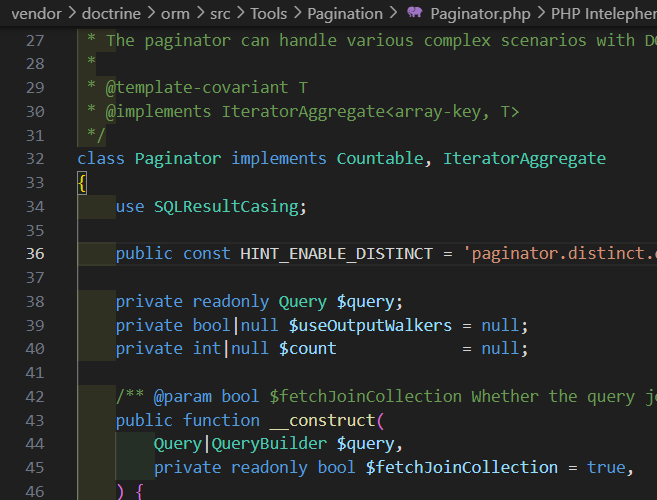
            ->setHint(Paginator::HINT\_ENABLE\_DISTINCT, false)

        );

    }

Dans l’objet lui-même Paginator : passer ce paramètre à false

Ainsi, il ne récupère pas les LEFTJOIN donc on peut désactiver si on n’en a pas.



**Affichage**

Dans le contrôleur, envoyer maxPage et page à la vue :

        //récupérer la page courante (valeur 1 par défaut)

        $page = $request->query->getInt('page', 1);

        //définir le nombre de recette par page

        $limit = 2;

        //avec pagination

        $recipes = $repository->paginateRecipes($page, $limit);

//récupérer le nombre total de recettes & le diviser par le nb de recettes

par page, arrondi à virgule sup

        $maxPage = ceil($recipes->count() / 2);

Dans le repo, ajouter la page courante et le nombre de recettes par page :

    public function paginateRecipes(int $**page**, int $**limit**): Paginator

    {

        return new Paginator($this

            ->createQueryBuilder('r')

            //si page 1, commencer à la recette 0...

            ->setFirstResult(($page - 1) \* $limit)

            ->setMaxResults($limit)

            //transforme objet QueryBuilder en Query

            ->getQuery()

            //passe petites infos pour mieux gérer requêtes SQL

            ->setHint(Paginator::HINT\_ENABLE\_DISTINCT, false)

        );

    }

Dans la vue, ajouter le bouton suivante/précédente :

    <div class="d-flex">

        {% if page > 1 %}

            <a class="btn btn-secondary" href="{{ path('admin.recipe.index', {page: page - 1}) }}">page précédente</a>

        {% endif %}

        {% if page < maxPage %}

            <a class="btn btn-secondary" href="{{ path('admin.recipe.index', {page: page + 1}) }}">page suivante</a>

        {% endif %}

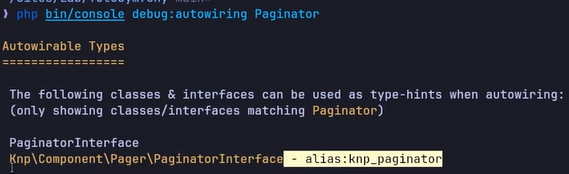
    </div>

***8’ Complexifier : Knp Paginator Bundle***

[Documentation](https://github.com/KnpLabs/KnpPaginatorBundle)

**Installation**

composer require knplabs/knp-paginator-bundle

on a alors un nouveau service

dans l’autowiring :

On l’injecte dans le constructeur du repository & on adapte le code de pagination :

    public function paginateRecipes(int $page, int $limit): PaginationInterface

    {

        //fonction prédéfinie dans le paginator knp

        return $this->paginator->paginate(

            $this->createQueryBuilder('r'),

            $page,

            $limit

        );

    }

Dans le contrôleur, remplacer count() par getTotalItemCount()

**11’30’’ On peut générer directement une structure HTML pour la pagination :**

Plutôt que la div et les boutons écrits à la main :

    {{ knp\_pagination\_render(recipes) }}

Le bundle propose des templates ! préfixés par @KnpPaginator , qui est le nouveau namespace.

Pour configurer:

* copier le code yaml donné dans la partie configuration de la doc -> config/packages/knp\_paginator.yaml (par convention on l’appelle comme la clé).
* J’y colle le code des templates que je veux utiliser (voir la liste, toujours dans la partie config)

knp\_paginator:

    page\_range: 5

    default\_options:

        …

    template:

        # templates bootstrap choisis dans la doc

        pagination: '@KnpPaginator/Pagination/bootstrap\_v5\_pagination.html.twig'

        rel\_links: '@KnpPaginator/Pagination/rel\_links.html.twig'

        sortable: '@KnpPagintor/Pagintion/bootstrap\_v5\_bi\_sortable\_link.html.twig'

        filtration: '@KnpPagator/Pagintion/botstrap\_v5\_fltraion.html.twig'

**Débuguer les config :**

php bin/console debug :config

🡪 liste des bundles & leurs alias !!

php bin/console debug :config knp\_paginator

**On peut simplifier le code**

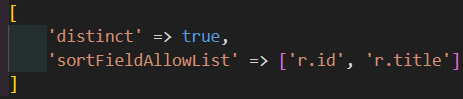
Dans le contrôleur, plus besoin de maxPages etc.

**On peut ajouter un tri par colonne**

Avec la fonction knp\_pagination\_sortable(notre objet paginé, titre, quel param réorganiser\*)

\*tel que dans le QueryBuilder

<th>{{ knp\_pagination\_sortable(recipes, 'Titre', 'r.title') }}</th>

Sécuriser dans le repository  en utilisant le dernier paramètres de paginate() : options

# 15 : Doctrine avancé

***Faire un select partiel***

Par défaut, doctrine hydrate l’entité complète lorsque l’on fait une requête SQL ( = convertit les résultats de la requête en objets PHP : les objets sont "remplis" ou "hydratés" avec des données provenant de la base de données = chaque colonne de la ligne correspondante dans le résultat de la requête est mappée à une propriété de l'objet).

C’est dommage quand on veut juste certains champs ou quand on veut relier à d’autres objets.

Par exemple ici, Doctrine sélectionne aussi createdAt, updatedAt… :

    public function findAllWithCount(): array

    {

        return $this->createQueryBuilder('c')

            ->select('c as category', 'COUNT(c.id) as total'

            ->leftJoin('c.recipes', 'r'

            ->groupBy('c.id'

            ->getQuery()

            ->getResult();

    }

La meilleure solution consiste à créer un objet représentant les données de mon tableau : DTO

class CategoryWithCountDTO

{

    //constructeur avec seulement ce qui nous est nécessaire

    public function \_\_construct(

        //readonly car initialisées que à la construction et jamais modifiées

        public readonly int $id,

        public readonly string $name,

        public readonly int $count

    ){…}

}

Dans mon repository, j’utilise mon DTO : Construire l’objet à la volée à partir des données de la bdd me permet de préciser que je retroune un tableau dont les éléments sont des CategoryWithcountDTO

/\*\*

\* @return CategoryWithCountDTO

\*/

public function findAllWithCount(): array

{

    return $this->createQueryBuilder('c')

//mot-clé pour utiliser un DTO

        ->select('NEW App\DTO\CategoryWithCountDTO(c.id, c.name, COUNT(c.id))')

        ->leftJoin('c.recipes', 'r')

        ->groupBy('c.id'

        ->getQuery()

        ->getResult();

 }

***7’30’’ Syntaxe alternative au QueryBuilder : DQL***

createQueryBuilder->getDQL()…

renvoie une requête écrite en DQL : ressemble au SQL mais avec nom des entités plutôt que des tables



Syntaxe utilisée en interne par Doctrine ; convertit cette syntaxe en requête SQL valide

            $this->getEntityManager()->createQuery(

                'SELECT c

                FROM App\Entity\Category c'

            )->getResult()

Que choisir : DQL ou QueryBuilder ? en effet ça donne le même résultat.

DQL seulement si la requête est complexe. Je ne peux pas changer ma requête, contrariement à avec le QueryBuilder. Choix personnel.

On peut même écrire du SQL brut en utilisant la méthode createNativeQuery().

# 16 : AssetMapper

**Comment intégrer le CSS et le JS au sein d’une application Symfony ?**

* Dans le dossier public/ de façon classique
* Dans le dossier assets/ de façon préinstallée dans Symfony : [AssetMapper](https://symfony.com/doc/current/frontend/asset_mapper.html) = copie des fichier quand on construit (build) 🡪chargement des fichiers grâce au {% block javascripts %} dans base.html.twig
* En utilisant Webpack Encore = un bundler (plus avancé, plus front-end) ( ?!\*)

\*dans Symfony 7, **Webpack Encore est dépassé et remplacé par AssetMapper**. L'introduction d'AssetMapper en Symfony **6.3** et ses améliorations ultérieures ont marqué un tournant significatif dans la gestion des actifs dans les applications Symfony. AssetMapper offre une approche simplifiée pour gérer les actifs sans nécessiter un système de construction comme Webpack Encore, ce qui réduit considérablement la complexité associée à la configuration et au maintien d'un environnement de développement JavaScript avancé.

* **Simplicité et efficacité** : AssetMapper permet d'écrire du JavaScript moderne et du CSS sans la complexité d'utiliser un bundler. Cela se fait grâce à la prise en charge par les navigateurs de nombreuses fonctionnalités JavaScript modernes, telles que l'instruction import et les classes ES6, éliminant ainsi le besoin de systèmes de construction traditionnels pour gérer les actifs.
* **Performance et espace disque** : L'utilisation d'AssetMapper entraîne une meilleure performance des sites web et libère un espace disque important en supprimant le besoin de node\_modules/ et de fichiers tels que package.json. Ces éléments sont souvent présents lors de l'utilisation de Webpack Encore mais ne sont pas nécessaires avec AssetMapper.
* **Intégration dans Symfony** : La mise en œuvre d'AssetMapper dans un projet Symfony est simple. Il suffit d'activer AssetMapper en spécifiant le chemin pour servir les actifs dans le fichier config/packages/framework.yaml, puis de placer les actifs compilés dans le répertoire assets/ de votre projet. Les actifs peuvent ensuite être référencés dans les templates à l'aide de la fonction asset()

**Comment marche AssetMapper ?**

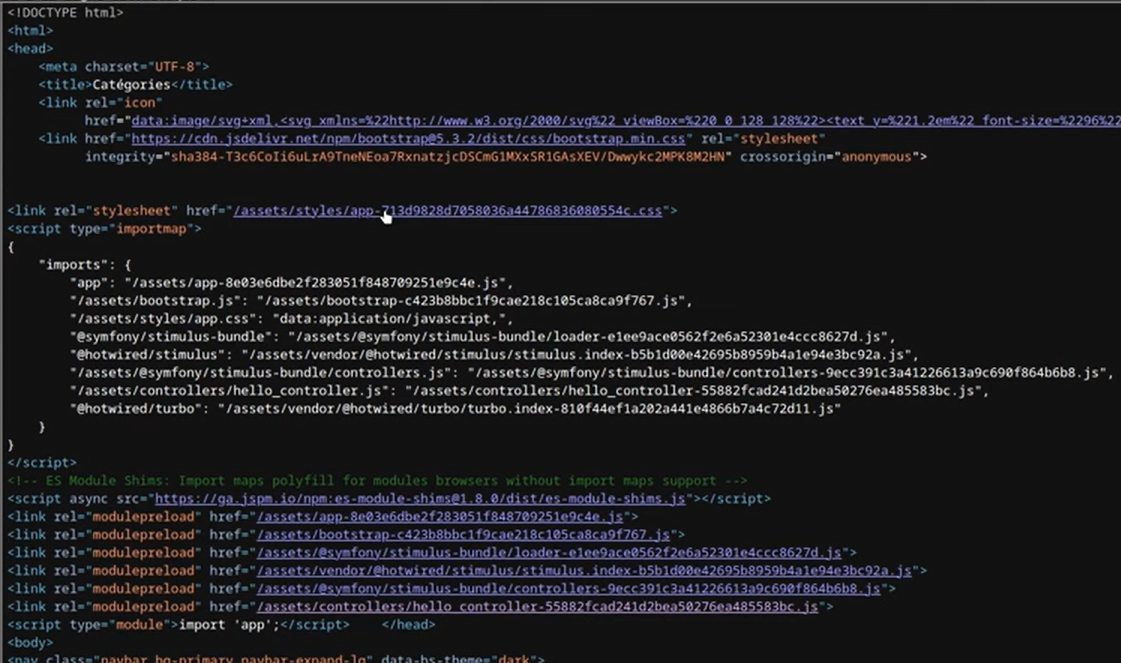
On voit ce qui est chargé (block javascripts) en regardant le code source d’une app qui l’utilise :

1. **<link rel=’’stylesheet’’>** Le CSS avec un code dynamique pour le cache (chiffre qui change si on change le fichier css associé)
2. **<script type=’’importmap’’>** : expliquer au navigateur quel fichier charger avec quel « imports »

Ex. : si je fais « imports app » 🡪 charger le fichier /assets/app-8e0…

Il crée des alias poru les différents imports

1. **<link rel=’’modulepreload’’>** dit au navigateur de précharger ces modules 🡪 le navigateur n’a pas besoin de parser le premier fichier pour détecter les dépendances ; il peut directrement charger tous ces fichiers js en parallèle (pas plus lent qu’un seul gros fichier désormais avec http2 et le navigateur peut mettre à jour seulement certains de ces fichiers /dépendances en gardant les autres en cache)
2. **<script type=’’module’’>** l’app, qui grâce au système d’alias charge le fichier présent dans imports : app (1)



Ces fichiers n’existent pas dans le dossier assets/. Pour que ça fonctionne, il faut compiler les assets :

* En phase de développement, utiliser le serveur proposé par Symfony (symfony serve)
* En production, php bin/console asset-map :compile 🡪
  + - copie les ressources de assets/ et colle dans public/
    - créer manifest.json qui permet à PHP de connaître le chemin des différentes ressources

**assets/app.js**

l.1 :import './bootstrap.js';

**🡪 assets/bootstrap.js**

n’a rien à voir avec le framework : fichier qui permet juste de dire comment démarrer l’application = en utilisant Stimulus, qui est une librairie front-end.

import { startStimulusApp } from '@symfony/stimulus-bundle';

const app = startStimulusApp();

l.8 : import './styles/app.css';

bizarre ! normalement en js on importe du js mais pas du css…

en fait le système d'Asset mapper, lorsqu'on compile, parse le fichier, voit qu'on importe un fichier CSS et génére une balise link, que l’on a vu dans le code source (1).

**🡪 ./styles/app.css'\***

Néanmoins dans le importmaps, cet import n’est câblé à rien = comme s’il n’y avait pas d’import (screenshot, encadré jaune).

Si j’ai besoin de faire du CSS je peux le faire dans ce fichier.

Ex. : si je veux mettre une image de fond sur ma page,

* je crée assets/images/ où j’importe mon jpeg
* j’ajoute dans app.css ‘background : url(../images/photo.jpeg) no-repeat…’

**Avec ce système, on peut importer des librairies tiers de npm.js**

… sans utiliser npm ! par exemple, l’animation confetti que j’importe normalement avec une commande npm

1. php bin/console importmap:require canvas-confetti🡪 ajouté à importmap.php
2. ajouter le package à app.js :

import canvasConfetti from 'canvas-confetti';

document.body.addEventListener('click', () => {

    canvasConfetti()

})

🡪 dans le code source, il est ajouté aux imports : la librairie a été automatiquement importée, et son js a été mis dans vendor/

**La fonction asset pour passer un chemin dans twig**

            <img src="{{ asset('images/ganapathy.jpg') }}">

**🡪 approche intéressante pour des besoins de base (plus compliqué avec Vue.js, des frameworks ou plus de JS : dans ce cas, préférer Webpack Encore).**

Plus gérable avec tailwind, typescript… cf. la doc pour voir comment implémenter.

*si vous voulez utiliser des assets qui sont versionné avec un système de suffixe et que vous voulez que ça soit bien mis en cache par votre navigateur vous pouvez placer les fichiers dans le dossier assets et ensuite lorsque vous allez avoir besoin de les utiliser*

* *si c'est dans vos vues vous pouvez utiliser la fonction assets()*
* *si c'est du JS ou du CSS c'est automatiquement géré lorsque vous allez faire la commande que l'on a vu (... compile)*

# 17 : API – Serializer

But : lorsque je fais localhost/api/recipes, ça me renvoie un json avec la liste des recettes

**Créer Controler/API/RecipesController.php**

    #[Route("/api/recipes")]

    public function index(RecipeRepository $repository)

    {

        $recipes = $repository->findAll();

        return $this->json($recipes);

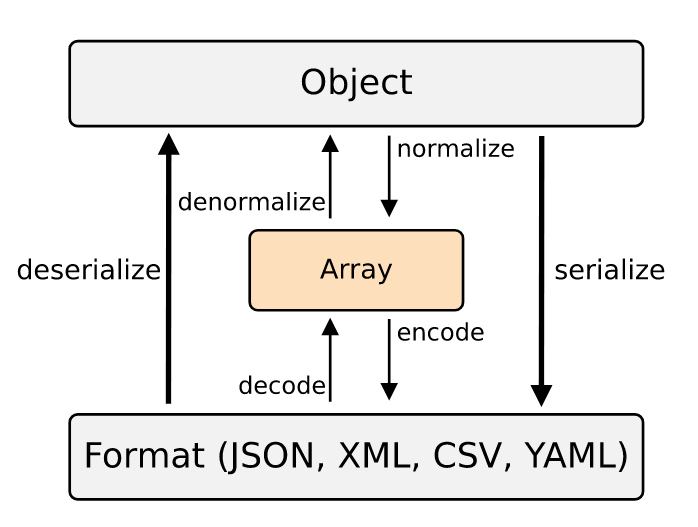
    }

Erreur de référence circulaire (1 recette a 1 catégorie qui ont des recettes qui ont des catégories…)

Entité Category : dans un premier temps, commenter la liste des recettes getRecipes()

(ou l’inverse : commenter getCategory() dans Recipe).

**2’50’’ Comment ça marche ?**

Json() appelle conainter où il récupère le service Serializer : capapble de convertir les données sous n’importe quel format.

Ce service est morcelé cf [documentation](https://symfony.com/doc/current/components/serializer.html):

Serialisation : convertir un objet en json etc

Désérialisation : l’inverse (vers un objet)

1e étape : normalisaiton des données 🡪 tableau php

2e: encodage = transformation du tableau php en json, xml…

**On peut controler la normalisation avec des groupes :**

🡪 Dernier paramètre de la méthode json(*données, statut, entête, contexte*) = le contexte : expliquer au système certaines opérations, par ex : ne pas renover d’exception en cas de dépendances circulaires

🡪 La clé ‘groups’ permet de spécifier les champs qu’on souhaite convertir

        return $this->json($recipes, 200, [], [

            'groups' => ['recipes.index']

        ]);

Dans l’entité, on précise ces groupes par l’attribut auquel on passe un tableau avec les groupes dont fait partie cette entité (et on peut décommenter le getter car il n’y aura plus de ref circulaire)

    #[Groups(['recipes.index'])]

On peut bien sûr créer plusieurs groupes, et les utiliser sur plusieurs entités (ici aussi Category).

**Comment ça marche ?**

Dans le Profiler, je peux voir l’onglet Serializer : entité sérialisée avec groupes, normailseur, encodeur…

**Pagination**

Dans le contrôleur, remplacer findAll() par paginateRecipes(//récupérer numéro de page).

Expliquer comment normaliser le type pagination = créer un normaliseur personnalisé :

Src/Normalizer/PaginationNormalizer.php ; en implémentant NormalizerInterface, comme services autoconfigure : true, il sera enregistré dans les normaliseurs. Déclarer ses méthodes…

…

Ensuite je l’appelle : $serializer->serialize($recipes, ‘csv’, [‘groups’ => [‘recipes.index’]]) ;

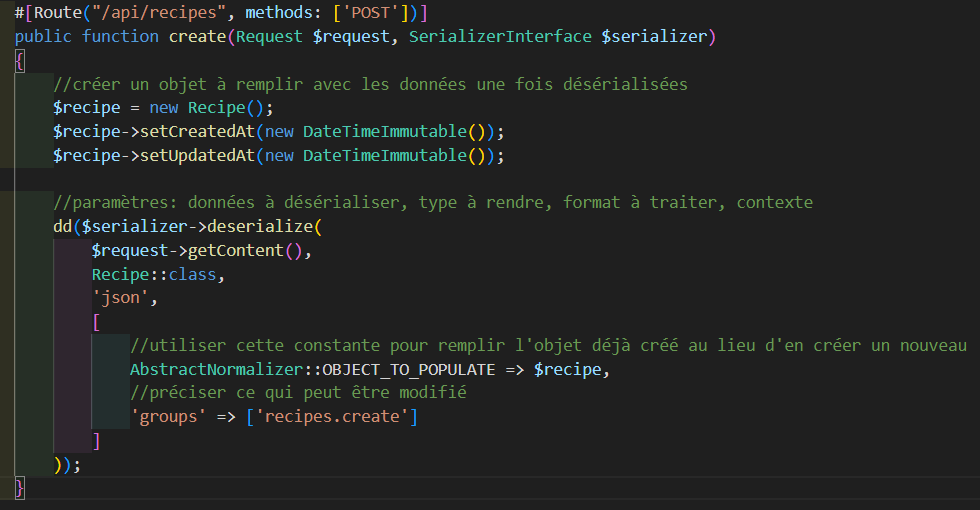
# 18 : API – Désérialisation et MapRequestPayload

Processus inverse du chapitre précédent : objectif = prendre le contenu d’une requête et obtenir un objet en sortie = **désérialisation** :

1. **décoder** : prendre le json et le convertir en tableau php,
2. **dénormaliser** : convertir le tableau PHP en objet

**Créer la méthode create() dans le contrôleur :**

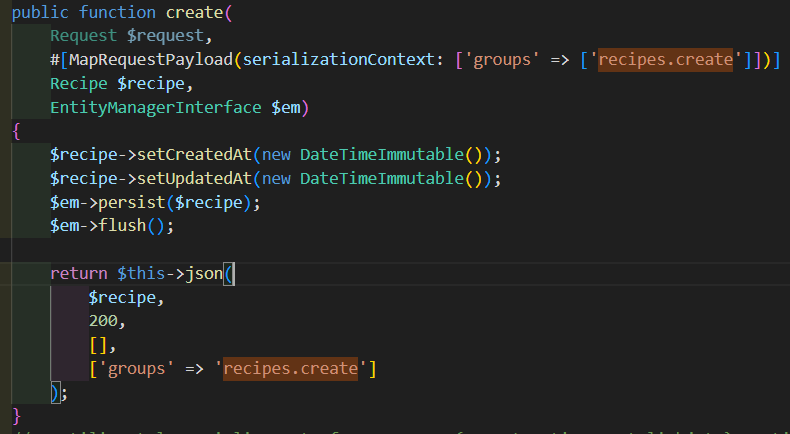
* même route que index mais méthode POST
* récupérer le contenu dans la requête
* injecter le sérializeur et j’appelle la méthdoe deserialize() dessus



**Encore plus rapide : avec l’attribut MapRequestPayload**

Avoir un objet recette automatiquement créé à partir des données provenant de l’API

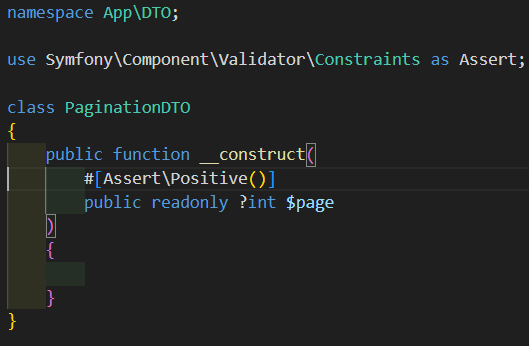
* on passe un objet Recipe en paramètre ;
* comme les entités ne sont pas en auto wiring, on ajoute l’attribut MapRequestPayload à ce paramètre ! **cet attribut permet de dire que le payload de la requête[[1]](#footnote-1), ici le json, va être injecté dans cet objet**. Cet attribut prend en paramètres :
  + 1 : le format qu’on accepte (facultatif)
  + 2 : le contexte de sérialisation (les groupes…)
* quand on utilise cet attribut, le processus de validation est mis automatiquement en place (des contraintes sont créées et appliquées automatiquement), ce qui peut poser des problèmes à résoudres de diverses façons (le plus simple : groups).



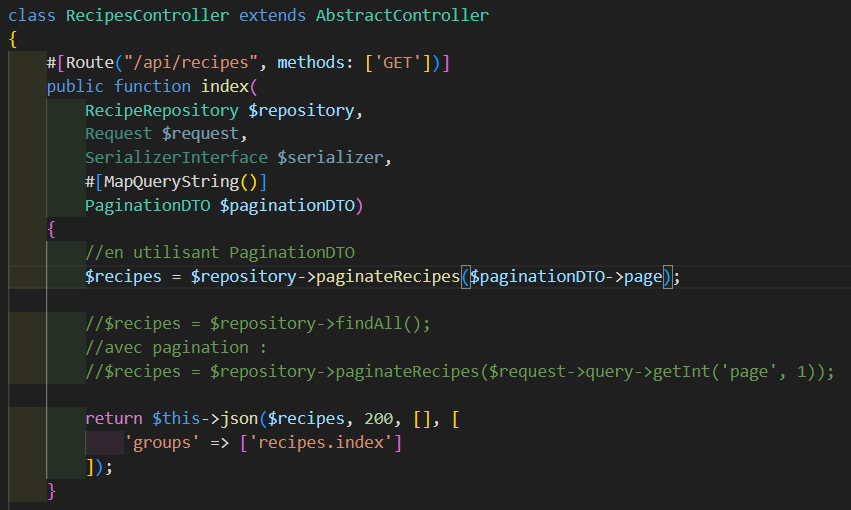
**Cet attribut MapRequestPayLoad peut être aussi utilisé par exemple pour créer des DTO à la volée**

Ex. avec la fonction index()

* d’abord créer PaginationDTO()  avec constructeur = je m’attends à avoir une propriété publique « page », entier ou nul, en readonly (non modifiable une fois construite)



* dans la fonction du contrôleur , prendre en paramètre DTO avec l’attribut MapQueryString : automatiquement, il va prendre tous les paramètres



#[MapQueryString()] indique que les paramètres de la requête seront automatiquement mappés à cet objet. Par exemple, si la requête contient un paramètre page, sa valeur sera assignée à la propriété page de l'objet $paginationDTO. La page actuelle est déterminée par la propriété page de l'objet $paginationDTO, qui a été remplie grâce à l'annotation #[MapQueryString()]. Cela signifie que la pagination est basée sur les paramètres de la requête.

Attention, il existe encore des issues sur cet attribut.

Conseil : créer un DTO pour create(), un pour update() séparément pour sécuriser.

Si vous avez besoin de valider des données qui rentrent par l'URL ou par le contenu de la requête, ces deux attributs sont super intéressants : ils vont se câbler par-dessus le système de sérialisation pour ajouter la logique de validation, permettant de grandement économiser de code et de garder une sûreté, parce que là contrairement à tout à l'heure où je pouvais un peu avoir n'importe quoi je sais que j'ai un objet qui représente les données de pagination objet que je peux peut-être transférer et envoyer à mes différents services.

Si vous avez besoin de valider des données qui rentrent par l'URL ou par le contenu de la requête, ces deux attributs sont super intéressants : ils vont se câbler par-dessus le système de sérialisation pour ajouter la logique de validation, permettant de grandement économiser de code et de garder une sûreté, parce que là contrairement à tout à l'heure où je pouvais un peu avoir n'importe quoi je sais que j'ai un objet qui représente les données de pagination objet que je peux peut-être transférer et envoyer à mes différents services.

La différence principale entre MapQueryString et MapRequestPayload réside dans la manière dont ils traitent les données de la requête HTTP, notamment en ce qui concerne les requêtes GET et POST.

* **MapQueryString** est utilisé pour mapper les paramètres de la chaîne de requête (query string) à des arguments de méthode dans un contrôleur. Cela signifie qu'il lit les valeurs passées dans l'URL après le signe ? lorsqu'une requête GET est faite. Par exemple, si votre URL est /api/items?color=blue&size=L, MapQueryString peut être utilisé pour extraire ces paramètres et les passer à votre contrôleur sous forme d'objets typés. Si aucun paramètre n'est fourni dans l'URL, cela peut entraîner une exception 404, car aucune donnée n'est présente pour être mappée [**2**](https://github.com/symfony/symfony/issues/50690).
* **MapRequestPayload**, quant à lui, est utilisé pour mapper le corps de la requête (payload) lorsqu'une requête POST est faite. Cela implique généralement de lire les données envoyées dans le corps de la requête, souvent sous forme de JSON. Contrairement à MapQueryString, MapRequestPayload est mieux adapté pour traiter les données complexes envoyées dans le corps d'une requête POST plutôt que dans l'URL d'une requête GET. Si le corps de la requête est vide, MapRequestPayload retournera null pour les valeurs par défaut, ce qui peut être géré différemment selon la logique métier de votre application [**2**](https://github.com/symfony/symfony/issues/50690).

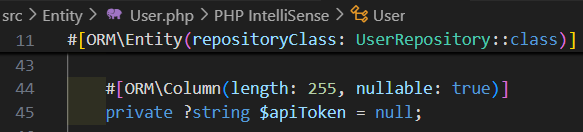
Un point important à noter est que ces deux attributs permettent de personnaliser le contexte de sérialisation et la classe utilisée pour mapper les données de la requête à vos objets. Cela offre une grande flexibilité pour adapter le traitement des données de la requête à vos besoins spécifiques [**1**](https://symfony.com/blog/new-in-symfony-6-3-mapping-request-data-to-typed-objects).

En résumé, MapQueryString est utilisé pour mapper les paramètres de l'URL dans les requêtes GET, tandis que MapRequestPayload est utilisé pour mapper le corps des requêtes POST. Chacun a ses propres comportements et utilisations optimales en fonction du type de requête et des données attendues.

# 19 : API – Authenticator stateless

L'authentification stateless, ou **sans état**, signifie qu'il n'y a **pas de système de session avec un point d'entrée spécifique pour le login**. Dans cette approche, **chaque requête** envoyée à l'API doit inclure toutes les informations nécessaires pour l'authentification, telles que les jetons d'accès. Cette méthode permet de garder chaque connexion autonome et sécurisée, sans dépendre d'un état global sur le serveur. En pratique, cela signifie que pour effectuer une action via l'API après s'être connecté, **le client doit envoyer deux requêtes** : une pour **obtenir un jeton** d'accès (token) et une autre pour **effectuer l'action** souhaitée, en incluant le jeton dans l'en-tête de la requête.

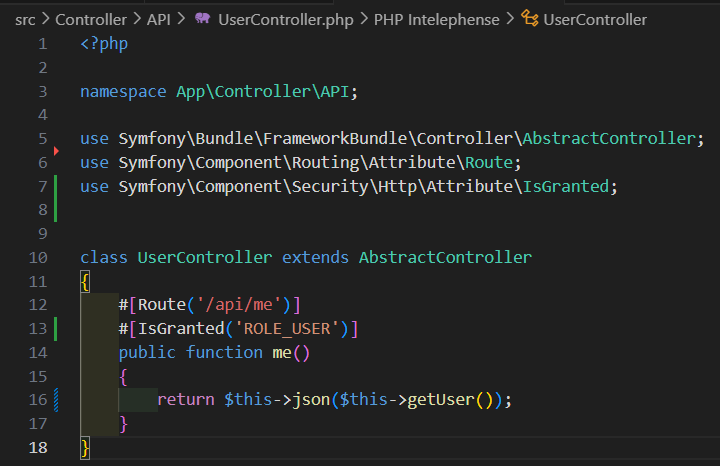
* Dans l’entité **User**, ajouter une propriété **$apiToken** (string nullable)



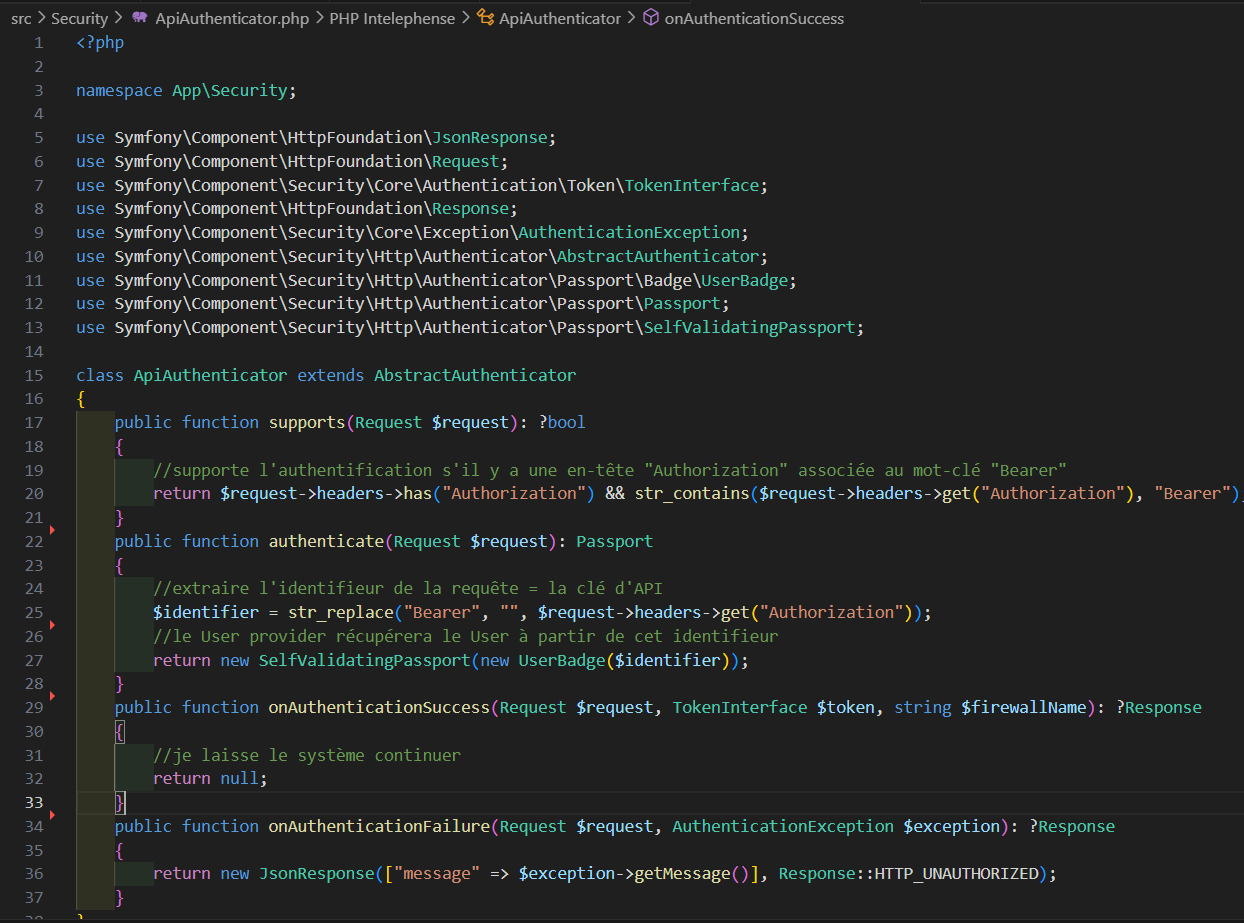
* Créer un point d’entrée **/api/me** renvoyant les données de l’utilisateur connecté
  + Pour tester :
    - En bdd, ajouter à la main un token à une utilisatrice
    - Dans postman ou autre outil de test api, dans l’en-tête, ajouter

**Authorization 🡪 Bearer [valeur du token]**

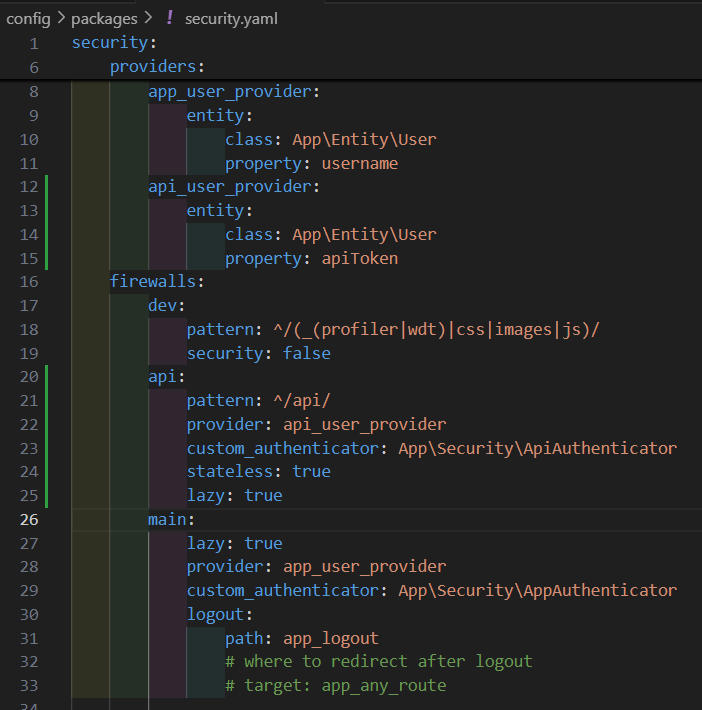
* + Créer **Api / UserController ::me()** sur api/me qui retourne json()



* Mettre en place le système d’authentification qui se déclenche lorsqu’il y a une entête Authorization avec le mot clé Bearer
  + Créer **Security/ApiAuthenticator** qui étend de AbstractAuthenticator & implémenter ses différentes méthodes, entre autres, mettre en place un passport mais d’un autre type que dans l’authentification pour l’app car pas vraiment de validation.



* + Modifier **config/security.yaml** : créer un user provider pour l’api + pare-feu s’activant sur les routes /api



On pourrait faire un système plus complexe avec un token JWT, système où on doit d’abord entrypoint pour recevoir un token, puis entrer ce token 🡪 faire un User Provider personnalisé, cf. [documentation](https://symfony.com/doc/current/security/user_providers.html#creating-a-custom-user-provider)

# 20 – Les fixtures

Jeu de données fictives pour tester l’app, chargeables en une commande.

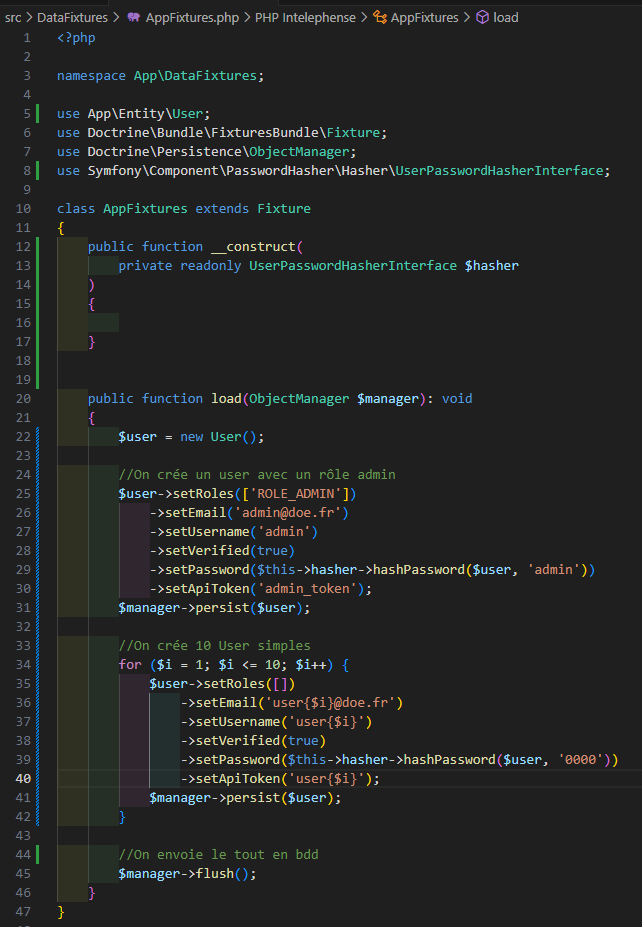
Cf. [documentation](https://symfony.com/bundles/DoctrineFixturesBundle/current/index.html) de Doctrine Fixtures Bundle – pour l’installer :

composer require --dev orm-fixtures

**🡪 création de src/DataFixtures/AppFixtures.php où je crée mes fixtures :**

Un User avec role admin 🡪 utiliser le pwd hasher

Ensuite je peux boucler pour créer autant d’éléemnts que je veux.



POur charcher les fixtures en bdd : Php bin/console doctrine :fixtures :load

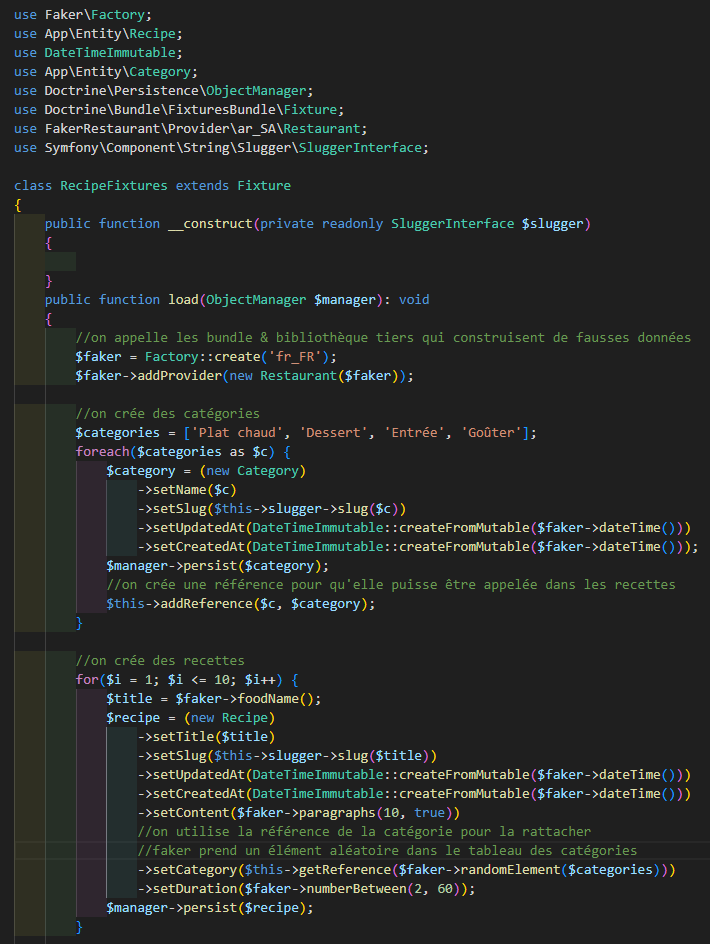
!!! Attention !!! cela peut écraser toutes les données présentes en bdd.

**Organiser les fixtures en plusieurs fichiers**

Php bin/console make :fixtures 🡪 RecipeFixtures (pour données permettant de renseigner les recettes)

On peut utiliser [le bundle Faker](https://fakerphp.org/) (indépendant de Symfony), qui permet de générer de fausses données en PHP : composer require –dev fakerphp/faker

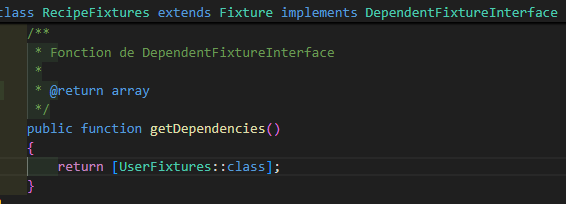
On peut y ajouter des [librairies tiers](https://fakerphp.org/third-party/) permettant de rajouter de nouveaux comportements, comme FakerRestaurant avec composer require jzonta/faker-restaurant



**Ordre de création des fixtures**

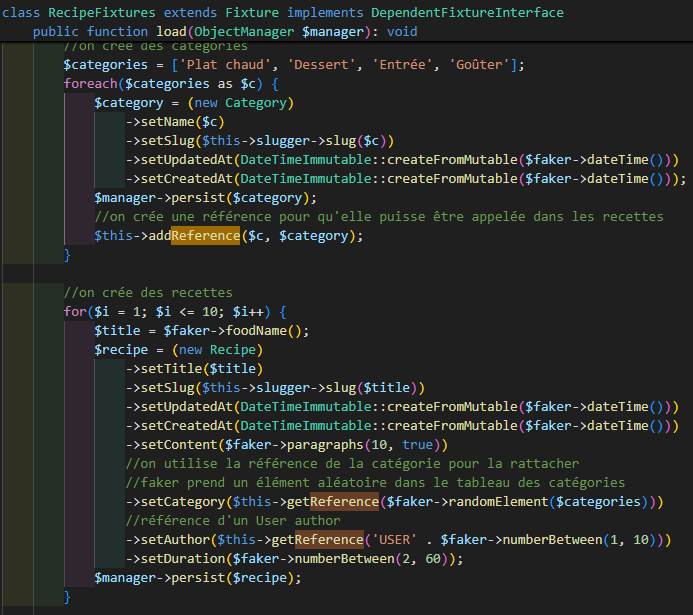
Les fixtures se lancent par ordre alphabétique : si je crée UserFixtures.php, il sera exécuté après RecipeFixtures.php, ce qui peut être gênant (relation de dépendance).

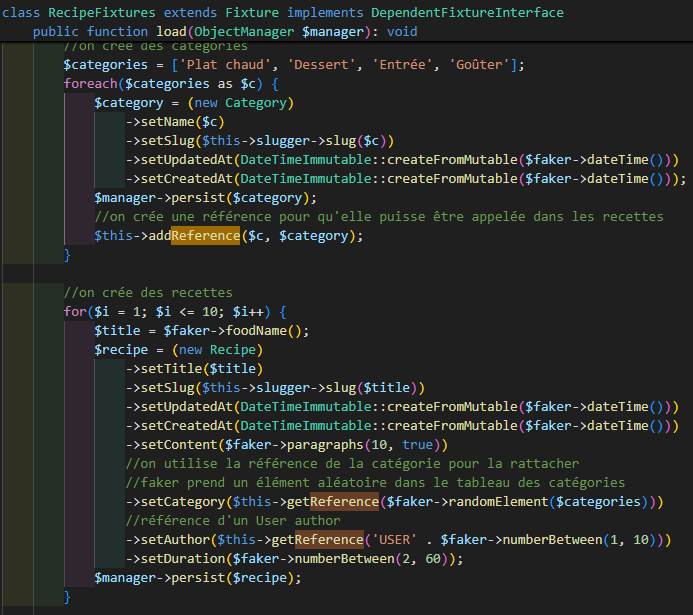
Implémenter DependentFixturesInterface pour gérer l’ordre :



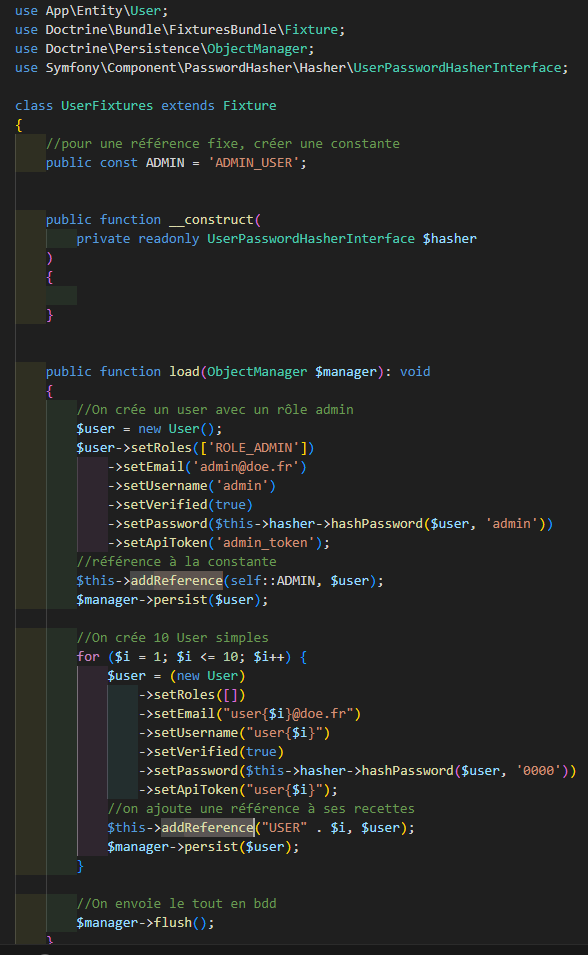
**Créer des références (addReference**(name, class)**), les utiliser (getReference**(name, class)**)**

RecipeFixture





Ci-dessus on utilise une référence créée dans la dépendance UserFixtures :



**Et on peut là aussi créer des groupes avec implémentation de FitxtureGroupInterface.**

# 21 – Les Voters

Sécurité : gérer des permissions plus finement, par exemple non seulement filtrer des routes en fonction du rôle utilisateur, mais même : ne pouvoir gérer que ses propres recettes.

Plus pratique que d’ajouter de la logique dans le contrôleur : on crée des classes qui peuvent voter pour dire si l’utilisateur a le droit de faire telle action.

Make :voter RecipeVoter & customiser

namespace App\Security\Voter;

use App\Entity\Recipe;

use App\Entity\User;

use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;

use Symfony\Component\Security\Core\Authorization\Voter\Voter;

class RecipeVoter extends Voter

{

    public const EDIT = 'RECIPE\_EDIT';

    public const VIEW = 'RECIPE\_VIEW';

    /\*\*

     \* dit si on supporte ou non cette permission

     \*

     \* @param string $attribute

     \* @param mixed $subject

     \* @return boolean

     \*/

    protected function supports(string $attribute, mixed $subject): bool

    {

        // code par défaut

        // https://symfony.com/doc/current/security/voters.html

        //si on a une des permissions définies ci-dessus ET qu'on vérifie la permission sur une instance de Recipe, renvoie true

        return in\_array($attribute, [self::EDIT, self::VIEW])

            && $subject instanceof \App\Entity\Recipe;

    }

    /\*\*

     \* faire le vote

     \*

     \* @param string $attribute nom de la permission

     \* @param mixed $subject la recette

     \* @param TokenInterface $token au niveau de la sécurité, on y récupère l'user

     \* @return boolean

     \*/

    protected function voteOnAttribute(string $attribute, mixed $subject, TokenInterface $token): bool

    {

        $user = $token->getUser();

        // if the user is anonymous, do not grant access

        if (!$user instanceof User) {

            return false;

        }

        //si le sujet n'est pas de type Recipe (ne devrait pas arriver car condition ligne 29)

        if (!$subject instanceof Recipe) {

            return false;

        }

        // ... (check conditions and return true to grant permission) ...

        switch ($attribute) {

            case self::EDIT:

                // logic to determine if the user can EDIT

                // return true or false

                //condition: si l'utilisateur auteur de la recette est l'utilisateur connecté

                return $subject->getAuthor()->getId() === $user->getId();

                break;

            case self::VIEW:

                // logic to determine if the user can VIEW

                // return true or false

                break;

        }

        return false;

    }

}

Ajouter l’attribut     #[IsGranted(RecipeVoter::EDIT, subject : ‘recipe’)] à la fonction RecipeController ::edit()… on peut aussi l’utiliser pour delete(), ou créer une condition dédiée si on préfère.

Si on a le droit de voir tout, on affiche tous les éléments, sinon seulement les miens

11’30’’

# 22 – Les événements

# 23 – Symfony UX

# 24 – Formulaires imbriqués

# 25 – Messenger

# 26 – Internationalisation

# 27 – Hébergement Infomaniak

# 28 – Hébergement O2Switch

1. En Symfony, le terme "payload" fait référence aux données envoyées dans une requête HTTP, qu'il s'agisse de données envoyées via une méthode POST ou d'autres méthodes qui incluent une corps de requête. Le payload peut contenir des données sous forme de texte brut, de JSON, ou d'autres formats. [↑](#footnote-ref-1)