



CRÉATION D'UN NOUVEAU PROJET :

```
composer create-project symfony/website-skeleton my_project_name  
cd my_project_name  
composer require webapp  
ln -s public htdocs  
composer install
```

COMMANDES UTILES:

- Générer les entités depuis la base de données :

```
php bin/console doctrine:mapping:import App\Entity annotation --path=src/Entity
```

- Générer les getters et setters des entités :

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

- Créer / modifier une entité :

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

- Générer un controller :

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

- Générer un controller CRUD et les formulaires et vues associées :

```
php bin/console make:crud
```

- Générer une migration Doctrine (lier les entités à la base de données) :

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

- Exécuter les migrations Doctrine :

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

- Créer la base de données via Doctrine :

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

- Démarrer un serveur de développement :

```
symfony server:start  
( on ne sera pas en connexion sécurisée https)
```

```
composer require symfony/apache-pack  
(nécessaire pour les redirections de page sous windows)
```

QU'EST CE QUE DOCTRINE :

Doctrine est un ORM open-source pour PHP qui permet de travailler avec des bases de données relationnelles en utilisant des objets. Doctrine a été développé pour fournir une alternative à la couche d'accès aux données traditionnelle de PHP en fournissant une solution moderne et robuste pour la gestion des données.

Doctrine utilise le modèle de conception de mappage objet-relationnel (ORM) pour faire correspondre les objets de votre application avec les tables de la base de données. Les objets représentent des enregistrements individuels dans la base de données, tandis que les propriétés de l'objet représentent les colonnes de la table de la base de données. Les méthodes de l'objet sont utilisées pour effectuer des opérations sur la base de données, comme l'ajout, la suppression ou la mise à jour d'enregistrements.

Doctrine fournit également des fonctionnalités telles que la validation de données, la gestion des relations entre tables et la gestion de la migration de base de données. Cela signifie que vous pouvez facilement créer des schémas de base de données complexes et les modifier au fil du temps sans avoir à écrire des requêtes SQL directement.

Doctrine est utilisé par de nombreux frameworks PHP populaires tels que Symfony, Laravel et Zend Framework, ainsi que dans des projets indépendants. Il est également disponible sous forme de bibliothèque autonome, ce qui signifie que vous pouvez l'utiliser dans n'importe quel projet PHP sans avoir à utiliser un framework spécifique.

Le principe de base de l'ORM est de considérer que :

- les tables de la BDD sont des classes PHP
- les lignes de ces tables sont des instances de ces classes
- les colonnes sont des propriétés de ces instances.

CRÉER UNE PREMIÈRE PAGE EN SYMFONY :

Log Messages

All messages Errors 2 Deprecations 0 Level (All) Channel (2)

Time	Message
16:53:48.259 error	Uncaught PHP Exception Symfony\Component\HttpKernel\Exception\NotFoundHttpException: "No route found for "GET http://127.0.0.1:8000/" at /home/egantois/Workspace/youtube/symrecipe/vendor/symfony/http-kernel/EventListener/RouterListener.php line 130
	request Show context Show trace
16:53:48.268 error	error while trying to collect executed migrations
	app Show context Show trace

Container Compilation Logs (703)

On va devoir corriger cette erreur que l'on trouve en bas, dans la barre de la page de notre site.

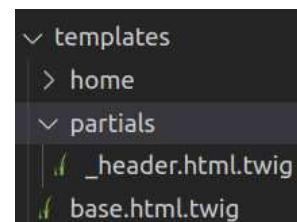
- création d'un HomeController à l'aide de la commande :
`php bin/console make:controller`
- (bien penser à supprimer le home après le \), ce qui va nous créer une première page index.html.twig dans notre template home.
- Il va falloir faire des block, notamment dans la base.html.twig, ainsi que des partials (dossier à créer dans templates), par exemple pour le header contenant la navbar (_header.html.twig) , et de le noter en include dans la base pour alléger le code :

```
<body>

    {% block header %}
        {% include "/partials/_header.html.twig" %}
    {% endblock %}

    {% block body %}{% endblock %}

</body>
```



- Nous pouvons passer par Bootswatch pour les templates, mais il faudra inclure dans le fichier base le block suivant :

```
{% block stylesheets %}
{{ encore_entry_link_tags('app') }}
<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootswatch@5.1.3/dist/united/bootstrap.min.css">
<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootswatch@5.1.3/dist/united/variables.scss">
{% endblock %}
```

CRÉER NOTRE BASE DE DONNÉES VIA LE TERMINAL:

En passant par Doctrine, on va créer nos entités, qui vont mapper directement la base de données pour l'implémenter.

On va tout d'abord créer un nouveau fichier : .env.dev.local (qui va permettre de sécuriser les mots de passe, ce fichier n'ira pas sur le repository git)

```
> vendor
⚙️ .env
$ .env.dev.local
```

qui va contenir uniquement la database suivante : (symrecipe est à remplacer par le nom de dossier)

```
DATABASE_URL="mysql://root@127.0.0.1:3306/symrecipe"
```

Et on va pouvoir créer la base de données avec le terminal sur PHPMYADMIN :

```
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console doctrine:database:create
Created database `symrecipe` for connection named default
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$
```

On peut abréger avec

```
php bin/console d:d:c
```

CRÉER UNE PREMIÈRE ENTITÉ EN SYMFONY :

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

```
egantois@egantois:~/Workspace/youtube/symrecipe$ php bin/console make:entity
Class name of the entity to create or update (e.g. FierceGnome):
> Ingredient

created: src/Entity/Ingredient.php
created: src/Repository/IngredientRepository.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):
> name
```

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

Field length [255]:
>

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

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

On remplit tout d'abord le nom de la table, puis la première colonne (property name), avec le type, longueur length, si cela peut être null en bdd, puis on ajoute la deuxième colonne (propriété).

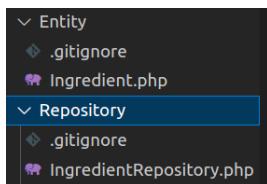
Quand on a terminé, on clique sur entrée , et on va pouvoir créer une migration.

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

Success!
I

Next: When you're ready, create a migration with php bin/console DEV
ole make:migration
```

On aura donc deux fichiers de crées , l'un dans Entity, l'autre dans Repository :



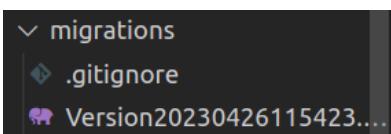
Dans Entity, on va retrouver les getters et les setters de la POO.

MIGRATION VERS LA BDD :

On va pouvoir ensuite effectuer une migration pour lier nos entités à la BDD:

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

Ce qui va générer un fichier dans le dossier migrations de notre VS code :



où l'on va retrouver du SQL classique avec trois fonctions :

- getDescription() :
retourne une courte description de la migration
- up(Schema \$schema) :
pour le code exécuté lors de la mise à jour de la BDD vers la version actuelle de la migration
(ajout de table, colonne, clé étrangère)
- down(Schema \$schema) :
pour le code exécuté lors du retour arrière de la migration vers la version précédente
(suppression)

Maintenant, il va falloir pousser toutes les migrations contenues dans le fichier présent dans le dossier migrations en BDD.

On va faire la commande suivante :

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

```

See https://symfony.com/doc/current/bundles/DoctrineMigrationsBundle/index.html
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console doctrine:migrations:migrate
WARNING! You are about to execute a migration in database "symrecipe" that could result in schema changes and data loss. Are you sure you wish to continue? (yes/no) [yes]:
>

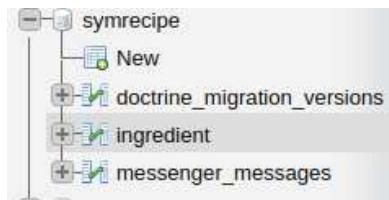
[notice] Migrating up to DoctrineMigrations\Version20230426115423
[notice] finished in 42.5ms, used 20M memory, 1 migrations executed, 2 sql queries
[OK] Successfully migrated to version : DoctrineMigrations\Version20230426115423

```

Dans notre BDD, on trouvera trois tables de créées :

- notre table Ingredient
- doctrine_migration_versions (stocke la liste des migrations effectuées avec succès sur la bdd, elle évite d'exécuter plusieurs fois la même migration)
- messenger_messages (système de messagerie asynchrone de Symfony, qui stocke les messages en attente de traitement)

Toutes les tables sont à conserver afin que l'application continue de fonctionner correctement.



VALIDATION DES ENTITÉS / LES CONTRAINTES:

<https://symfony.com/doc/current/validation.html#constraints>

Ce sont des contraintes que l'on souhaite imposer afin de limiter certaines données, comme par exemple limiter le nom à 2 caractères minimum et 50 caractères maximum, ou bien que le prix ne pourra pas être inférieur à 0 et supérieur à 200...

Voici quelques contraintes de bases :

- | | |
|------------|----------------------------------------------------------------------------------------------------|
| • NotBlank | Permet de s'assurer qu'un champ de formulaire n'est pas vide |
| • Blank | Permet qu'un champ de formulaire soit vide (ex: 2e prénom) |
| • NotNull | Permet de s'assurer qu'une valeur n'est pas nulle |
| • IsNull | Permet de s'assurer qu'une valeur est nulle |
| • IsTrue | Permet de s'assurer qu'une valeur soit vraie |
| • IsFalse | Permet de s'assurer qu'une valeur soit fausse |
| • Type | Permet de s'assurer que la valeur du champ du formulaire soit du bon type (ex : date de naissance) |

Cela va être une vérification lors de la soumission des formulaires que les données sont valables ou non pour être enregistrées dans la BDD.

Si on veut par exemple, ne pas pouvoir excéder plus de 50 caractères et au moins 2 caractères, on va chercher dans Strings, et on va trouver length :

String Constraints

- [Email](#)
- [ExpressionSyntax](#)
- [Length](#)

Et on aura un exemple :

Basic Usage

To verify that the `firstName` field length of a class is between 2 and 50, you might add the following:

```
Attributes | YAML | XML | PHP
Copy

1 // src/Entity/Participant.php
2 namespace App\Entity;
3
4 use Symfony\Component\Validator\Constraints as Assert;
5
6 class Participant
7 {
8     #[Assert\Length(
9         min: 2,
10        max: 50,
11        minMessage: 'Your first name must be at least {{ limit }} characters long',
12        maxMessage: 'Your first name cannot be longer than {{ limit }} characters',
13    )]
14     protected $firstName;
15 }
```

Il va donc falloir copier le lien use et l'ajouter à notre Entity :

```
src > Entity > Ingredient.php > ...
1  <?php
2
3  namespace App\Entity;
4
5  use App\Repository\IngredientRepository;
6  use Doctrine\ORM\Mapping as ORM;
7
8  use Symfony\Component\Validator\Constraints as Assert;|
```

Ce lien use est le même quel que soit la contrainte, une fois qu'on l'a mis dans notre code, on peut apporter n'importe quelle contrainte à nos colonnes.

On va pouvoir ajouter les Asserts correspondants :

- limitation des noms minimum 2 caractères et maximum 50 caractères :

```
#ORM\Column(length: 50)
#[Assert\Length(min :2, max:50)]
private ?string $name = null;
```

- limitation du prix qui ne peut pas être inférieur à 1 et supérieur à 200 :

```
#ORM\Column
#[Assert\Positive()]
#[Assert\LessThan(200)]
private ?float $price = null;
```

- Si on ne veut pas que la donnée soit vide, on ajoute :

```
#[Assert\NotBlank()]
```

- Si on ne veut pas que la donnée soit nulle, on ajoute :

```
#[Assert\NotNull()]
```

LES FIXTURES :

<https://symfony.com/doc/current/testing.html#load-dummy-data-fixtures>

C'est un fichier de configuration qui fournit un ensemble de données de test pour simuler une requête HTTP et vérifier la réponse du serveur. Les fixtures peuvent également inclure des scripts de test, des fichiers de données ou des configurations de système. C'est un jeu de fausses données.

Pour installer les Fixtures, voici la ligne de commande :

```
› maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ composer require --dev orm-fixtures
```

Ce qui va nous permettre de ne l'installer que dans l'environnement de développement, et pas en production car nous n'en aurons pas besoin.

On peut également installer Faker, qui va générer des faux noms, des fausses adresses mail, de fausses images, fausses dates....

On aura donc un nouveau dossier avec un nouveau fichier :

```
✓ DataFixtures  
  🐘 AppFixtures.php
```

On pourra donc remplacer l'exemple par la variable de notre choix :

```
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager): void
    {
        $ingredient = new Ingredient();
        $ingredient -> setName ('Ingredient #1')
        -> setPrice (3.0);

        $manager -> persist($ingredient);

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

On va également pouvoir faire en sorte que le createdAt se crée automatiquement lors de la création d'un élément . Dans notre Entity.php, on va ajouter un constructeur avant le getId :

```
// MISE EN PLACE DU CONSTRUCTEUR :

public function __construct()
{
    $this -> createdAt = new DateTimeImmutable();
```

ce qui va importer le use également en haut de l'entité :

```
namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;
use App\Repository\IngredientRepository;
use DateTimeImmutable;
use Symfony\Component\Validator\Constraints as Assert;
```

Le DateTimeImmutable est une classe PHP qui représente une date et une heure immuables, ce qui signifie que cela ne peut pas être modifié après leur création. La date de création ne sera donc jamais modifiée une fois qu'elle a été définie.

Qu'est ce qu'un bundle ? :

Dans le contexte de Symfony, un bundle est un ensemble de fichiers qui contient des fonctionnalités spécifiques et qui peut être intégré à une application Symfony existante. Les bundles sont des composants réutilisables qui peuvent être ajoutés à un projet Symfony pour ajouter de nouvelles fonctionnalités ou améliorer celles qui existent déjà.

Dans Doctrine, on va avoir une notion de persistance. La persistance, c'est dire à l'objet qu'il est prêt à aller en base de données avec les informations qu'il contient, et donc flush via le manager que l'on récupère via `use Doctrine\Persistence\ObjectManager;`

Si on clique sur `ObjectManager`, on va pouvoir voir les différentes méthodes proposées :

- find : récupère un objet de la BDD en fonction de son ID
 - remove : supprimer un objet
 - persist : ajouter un objet
 - clear : supprime tous les objets persistants de l'unité de travail
 - detach : détache un objets persistant de l'unité de travail
 - refresh : met à jour les données à partir de la BDD
 - flush : enregistre les modifications

Revenons à notre code dans AppFixtures.php :

```
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager): void
    {
        $ingredient = new Ingredient();
        $ingredient -> setName ('Ingredient #1')
        -> setPrice (3.0);

        $manager -> persist($ingredient);

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

On va pouvoir flush les éléments que l'on a persist dans la BDD. Pour cela, on va exécuter la ligne de commande :

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

```
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console doctrine:fixtures:load
Careful, database "symrecipe" will be purged. Do you want to continue? (yes/no) [no]: > yes
> purging database
> loading App\DataFixtures\AppFixtures
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipes$
```

Et on retrouve bien notre fixture dans la BDD :

Server: mysql > Database: symtree > Table: ingredient

[Browse](#) [Structure](#) [SQL](#) [Search](#) [Insert](#) [Export](#) [Import](#)

Showing rows 0 - 0 (1 total). Query took 0.00004 seconds.

```
SELECT * FROM `ingredient`
```

Profiling [Edit inline] [Edit] [Explain SQL] [Create PHP code] [Refresh]

<input type="checkbox"/> Show all	Number of rows:	25	Filter rows:	Search this table
+ Options				
<input type="button" value="←"/> <input type="button" value="→"/> ▾ id name price created_at <small>(DC2TYPE:datetime_immutable)</small>				
<input type="checkbox"/>	Edit	Copy	Delete	1 Ingredient #1 3 2023-04-26 16:15:53
◀ ▶ First Last Previous Next Last ▶				

Le created-at a bien été mis à jour automatiquement, grâce au construct dans notre Entity.php.

Afin de générer plusieurs fausses données, il suffit de faire une boucle et de relancer :

```
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager): void
    {
        for ($i=1; $i <= 50 ; $i++) {
            $ingredient = new Ingredient();
            $ingredient -> setName ('Ingredient' . $i)
            -> setPrice (mt_rand(0, 100));
            $manager -> persist($ingredient);
        }
    }

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

mt-rand est une fonction PHP qui génère un nombre aléatoire, prenant deux paramètres, le premier étant la valeur minimale et le second la valeur maximale.

LE CRUD (Create, Read, Update, Delete):

Création d'un Controller dans le terminal :

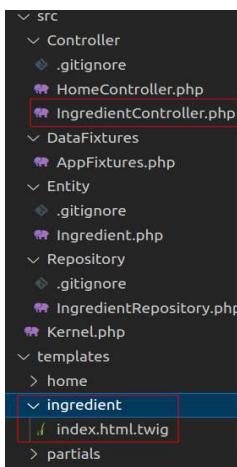
```
php bin/console make:controller nom du controller
```

```
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console make:controller IngredientController
created: src/Controller/IngredientController.php
created: templates/ingredient/index.html.twig

Success!

Next: Open your new controller class and add some pages!
```

Un fichier a été crée dans Controller, et également dans Templates.



On va maintenant mettre en place le Repository, qui va nous permettre de récupérer des informations de la BDD.

L'injection de dépendances et le Repository :

L'injection de dépendances est une technique de programmation qui consiste à fournir à un objet les objets dont il a besoin pour fonctionner, plutôt que de les créer directement dans l'objet lui-même.

Dans le cas de l'injection de dépendances via le repository, cela signifie que l'objet qui utilise une base de données n'interagit pas directement avec la base de données, mais plutôt avec un objet appelé "repository". Ce repository contient les méthodes pour accéder à la base de données et récupérer ou stocker des données.

En injectant le repository dans l'objet qui a besoin d'accéder à la base de données, l'objet n'a pas besoin de connaître les détails de la façon dont les données sont stockées ou récupérées. Il peut simplement appeler les méthodes du repository pour accomplir les tâches nécessaires.

Cela rend le code plus modulaire, plus facile à tester et plus facile à maintenir. Si jamais il est nécessaire de changer la façon dont les données sont stockées ou récupérées, il suffit de changer le code du repository, sans avoir à modifier l'objet qui utilise le repository.

On peut constater que dans chaque fichier Repository, nous disposons de quatre méthodes principales :

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

- Find :
méthode qui permet de récupérer une entrée spécifique de la table de la BDD en fonction de son ID
- FindOneBy :
méthode qui permet de récupérer une entrée spécifique de la BDD en fonction de certains critères de recherche, tels que des colonnes précises de la table.
- FindAll :
méthode qui permet de récupérer toutes les entrées de la table de la BDD liée au repository , et de les afficher.
- FindBy :
méthode qui permet de récupérer une ou plusieurs entrées de la table de la BDD en fonction de certains critères de recherche, tels que des colonnes spécifiques, et les retourne sous forme d'un tableau.

On va modifier le fichier Controller, et on fera un FindAll pour tout importer :

```
use App\Repository\IngredientRepository;  
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;  
use Symfony\Component\HttpFoundation\Response;  
use Symfony\Component\Routing\Annotation\Route;  
  
class IngredientController extends AbstractController  
{  
    #[Route('/ingredient', name: 'app_ingredient')]  
    public function index(IngredientRepository $repository): Response  
    {  
        $ingredients = $repository -> findAll();  
  
        return $this->render('ingredient/index.html.twig', [  
            ]);  
    }  
}
```

Si on fait un dd(\$ingredients), on constate que l'on récupère bien toute la liste des ingrédients présents dans la BDD :

```
IngredientController.php on line 16:
array:50 [▼
  0 => App\Ent_\Ingredient {#1361 ▶}
  1 => App\Ent_\Ingredient {#1563 ▶}
  2 => App\Ent_\Ingredient {#6110 ▶}
  3 => App\Ent_\Ingredient {#1551 ▶}
  4 => App\Ent_\Ingredient {#4472 ▶}
  5 => App\Ent_\Ingredient {#1011 ▶}
  6 => App\Ent_\Ingredient {#4422 ▶}
  7 => App\Ent_\Ingredient {#6101 ▶}
  8 => App\Ent_\Ingredient {#2098 ▶}
  9 => App\Ent_\Ingredient {#4571 ▶}
  10 => App\Ent_\Ingredient {#3342 ▶}
  11 => App\Ent_\Ingredient {#1242 ▶}
  12 => App\Ent_\Ingredient {#913 ▶}
  13 => App\Ent_\Ingredient {#2635 ▶}
  14 => App\Ent_\Ingredient {#1414 ▶}
  15 => App\Ent_\Ingredient {#6466 ▶}
  16 => App\Ent_\Ingredient {#2138 ▶}
  17 => App\Ent_\Ingredient {#5708 ▶}
  18 => App\Ent_\Ingredient {#4448 ▶}
  19 => App\Ent_\Ingredient {#3322 ▶}
  20 => App\Ent_\Ingredient {#4502 ▶}
  21 => App\Ent_\Ingredient {#4486 ▶}
  22 => App\Ent_\Ingredient {#4457 ▶}
  23 => App\Ent_\Ingredient {#4459 ▶}
  24 => App\Ent_\Ingredient {#4455 ▶}
  25 => App\Ent_\Ingredient {#4454 ▶}
  26 => App\Ent_\Ingredient {#4443 ▶}
  27 => App\Ent_\Ingredient {#4842 ▶}
  28 => App\Ent_\Ingredient {#6097 ▶}
  29 => App\Ent_\Ingredient {#6129 ▶}
  30 => App\Ent_\Ingredient {#1267 ▶}
  31 => App\Ent_\Ingredient {#5216 ▶}
  32 => App\Ent_\Ingredient {#1547 ▶}
  33 => App\Ent_\Ingredient {#1803 ▶}
  34 => App\Ent_\Ingredient {#1202 ▶}
  35 => App\Ent_\Ingredient {#1119 ▶}
  36 => App\Ent_\Ingredient {#4429 ▶}
  37 => App\Ent_\Ingredient {#6218 ▶}
  38 => App\Ent_\Ingredient {#2401 ▶}
  39 => App\Ent_\Ingredient {#2630 ▶}
  40 => App\Ent_\Ingredient {#6110 ▶}
  41 => App\Ent_\Ingredient {#4663 ▶}
  42 => App\Ent_\Ingredient {#1225 ▶}
  43 => App\Ent_\Ingredient {#932 ▶}
  44 => App\Ent_\Ingredient {#754 ▶}
  45 => App\Ent_\Ingredient {#4773 ▶}
  46 => App\Ent_\Ingredient {#4515 ▶}
]
```

On va donc pouvoir passer une variable à notre vue, en ajoutant la ligne suivante :

```
$ingredients = $repository -> findAll();
return $this->render('ingredient/index.html.twig', [
    'ingredients' => $ingredients
]);
```

Dans notre fichier index.html.twig, on va pouvoir afficher l'intégralité des données présentes dans la base de données, en fonction de ce que l'on souhaite afficher, à l'aide d'une boucle.

Par exemple, ici, nous souhaitons afficher pour chaque ingrédient de la liste ingrédients, leur nom :

```
{% for ingredient in ingredients %}
<tr class="table-primary">
    <th scope="row">{{ingredient.id}}</th>
    <td>{{ingredient.name}}</td>
    <td>{{ingredient.price}}</td>
    <td>{{ingredient.createdAt}}</td>
</tr>
{% endfor %}
```

La liste « ingredients » correspond à la variable \$ingredients que nous avons créée précédemment, dans l'IngredientController, avec le findAll, dans la fonction index.

On va obligatoirement avoir un message d'erreur :

```
Error
Object of class DateTimeImmutable could not be converted to string
```

Car nous demandons d'afficher une chaîne de caractère, alors que createdAt est un Objet. On va donc devoir utiliser un filtre PHP en plus de notre createdAt : {{ "now" | date("m/d/Y") }}

```
<td>{{ingredient.createdAt|date("d/m/Y") }}</td>
```

Ce qui nous donnera ce tableau :

Mes ingrédients

Numéro	Nom	Prix	Date de création
3	Ingredient1	43	26/04/2023
4	Ingredient2	74	26/04/2023
5	Ingredient3	63	26/04/2023
6	Ingredient4	67	26/04/2023
7	Ingredient5	91	26/04/2023

En résumé :

La liste présente dans la table de notre base de données, va donc être récupérée via la fonction index, et être appelée par la boucle « for in » présente dans notre index.html.twig.

Grâce au Repository dans notre Controller, fonction index, nous avons fait une injection de dépendance (on injecte un service dans les paramètres de la fonction du Controller).

LA PAGINATION :

On va utiliser la pagination afin de limiter le nombre d'affichage . Pour cela, on va utiliser un bundle : PaginatorBundle : <https://github.com/KnpLabs/KnpPaginatorBundle>

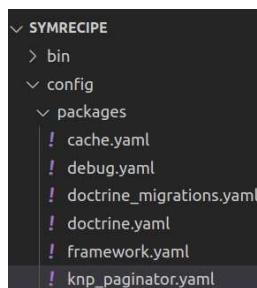
On va taper la ligne de commande suivante dans notre terminal :

```
composer require knplabs/knp-paginator-bundle
```

Et on peut constater dans notre fichier compose.json qu'il a bien été installé :



```
ingredientController.php index.html.twig composer.json
composer.json > ...
1  {
2      "type": "project",
3      "license": "proprietary",
4      "minimum-stability": "stable",
5      "prefer-stable": true,
6      "require": {
7          "php": ">=8.1",
8          "ext-ctype": "*",
9          "ext-iconv": "*",
10         "doctrine/annotations": "^2.0",
11         "doctrine/doctrine-bundle": "^2.9",
12         "doctrine/doctrine-migrations-bundle": "^3.2",
13         "doctrine/orm": "^2.14",
14         "knplabs/knp-paginator-bundle": "^6.2",
```



```
SYMRECIPE
> bin
< config
  < packages
    ! cache.yaml
    ! debug.yaml
    ! doctrine_migrations.yaml
    ! doctrine.yaml
    ! framework.yaml
    ! knp_paginator.yaml
```

On va devoir créer un fichier yaml dans config/packages :

où l'on va pouvoir copier le code suivant :

knp paginator:

```
page_range: 5          # number of links shown in the pagination menu (e.g: you have 10
pages, a page_range of 3, on the 5th page you'll see links to page 4, 5, 6)
default_options:
  page_name: page      # page query parameter name
  sort_field_name: sort # sort field query parameter name
  sort_direction_name: direction # sort direction query parameter name
```

```

distinct: true          # ensure distinct results, useful when ORM queries are using GROUP
BY statements
filter_field_name: filterField # filter field query parameter name
filter_value_name: filterValue # filter value query parameter name
template:
    pagination: '@KnpPaginator/Pagination/sliding.html.twig' # sliding pagination controls
template
    sortable: '@KnpPaginator/Pagination/sortable_link.html.twig' # sort link template
filtration: '@KnpPaginator/Pagination/filtration.html.twig' # filters template
-----
```

Ensute, on va devoir ajouter du code dans notre Controller :

```

namespace App\Controller;

use App\Repository\IngredientRepository;
use Knp\Component\Pager\PaginatorInterface;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

class IngredientController extends AbstractController
{
    #[Route('/ingredient', name: 'app_ingredient')]
    public function index(IngredientRepository $repository, PaginatorInterface $paginator, Request $request): Response
    {
        $ingredients = $paginator->paginate(
            $repository -> findAll(),
            $request->query->getInt('page', 1),
            10
        );
        return $this->render('ingredient/index.html.twig', [
            'ingredients' => $ingredients
        ]);
    }
}
```

Si PaginatorInterface et Request sont en rouge, clic droit et import class (choisir HTTP FOUNDATION).

On peut constater que dans notre vue, le nombre d'affichage s'est réduit à 10. Mais il manque la fonctionnalité qui permet de changer de page, que nous allons afficher dans notre index.html.twig :

```

<table class="table table-hover">
    <thead>
        <tr>
            <th scope="col">Numéro</th>
            <th scope="col">Nom</th>
            <th scope="col">Prix</th>
            <th scope="col">Date de création</th>
        </tr>
    </thead>
    <tbody>
        {% for ingredient in ingredients %}
        <tr class="table-primary">
            <th scope="row">{{ingredient.id}}</th>
            <td>{{ingredient.name}}</td>
            <td>{{ingredient.price}}</td>
            <td>{{ingredient.createdAt|date("d/m/Y") }}</td>
        </tr>
        {% endfor %}

    </tbody>
</table>
<div class="navigation">
    {{ knp_pagination_render(ingredients) }}
</div>
```

Ce qui va nous créer une possibilité de changer de page d'affichage :

Mes ingrédients

Numéro	Nom	Prix	Date de création
3	Ingredient1	43	26/04/2023
4	Ingredient2	74	26/04/2023
5	Ingredient3	63	26/04/2023
6	Ingredient4	67	26/04/2023
7	Ingredient5	91	26/04/2023
8	Ingredient6	64	26/04/2023
9	Ingredient7	89	26/04/2023
10	Ingredient8	67	26/04/2023
11	Ingredient9	81	26/04/2023
12	Ingredient10	41	26/04/2023

1 2 3 4 5 >>>

On peut changer le front de cette pagination via Bootstrap . Dans notre fichier knpPaginator.yaml, on peut remplacer le lien de pagination par ce lien :

@KnpPaginator/Pagination/bootstrap_v5_pagination.html.twig

```
pagination: '@KnpPaginator/Pagination/bootstrap_v5_pagination.html.twig'
```

« Previous 1 2 3 4 5 Next »

On peut ajouter une div également dans notre vue , afin d'afficher le nombre de données total que contient la table de notre BDD :

```
<small>
    <div class="count">
        Il y a {{ ingredients.getTotalItemCount }} ingrédients au total.
    </div>
</small>
```

Il est bien d'installer l'extension PHP DocBlocker, qui va permettre d'afficher des informations utiles sur l'utilisation et le fonctionnement de notre code , rien qu'en tapant /** :

```
/**
 * Cette fonction permet d'afficher tous les ingrédients :
 *
 * @param IngredientRepository $repository
 * @param PaginatorInterface $paginator
 * @param Request $request
 * @return Response
 */

#[Route('/ingredient', name: 'app_ingredient', methods:['GET'])]
```

PARTIE CREATE :

Pour remettre à zéro notre fixture dans la BDD , dans la console, taper la ligne suivante :

```
php bin/console d:f:l
```

```
● maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console d:f:l
Careful, database "symrecipe" will be purged. Do you want to continue? (yes/no) [no]:
> yes
> purging database
> loading App\DataFixtures\AppFixtures
```

On va créer des données à l'aide d'un formulaire :
<https://symfony.com/doc/current/forms.html>

Dans notre terminal, on va taper la commande suivante :

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

```
● maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console make:form

The name of the form class (e.g. FiercePopsicleType):
> IngredientType

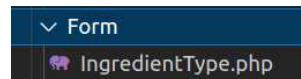
The name of Entity or fully qualified model class name that the new form will be bound to (empty for none):
> Ingredient

created: src/Form/IngredientType.php

Success!

Next: Add fields to your form and start using it.
Find the documentation at https://symfony.com/doc/current/forms.html
```

Un fichier Type a été créé dans le dossier Form



Dans notre fichier Type, nous avons une fonction buildForm qui va reprendre nos colonnes de la table dans la BDD :

```
public function buildForm(FormBuilderInterface $builder, array $options): void
{
    $builder
        ->add('name')
        ->add('price')
        ->add('createdAt')
}
```

On supprime createdAt car ce dernier s'auto-incrémente.

Mais il faut également renseigner ce formulaire dans le Controller . Cela va permettre de gérer la création d'un nouvel ingrédient dans la BDD :

```
#[Route('/ingredient/new', name: 'ingredient.new', methods:[GET, POST])]
public function new(): Response
{
    $ingredient = new Ingredient();
    $form = $this->createForm(IngredientType::class, $ingredient);

    return $this -> render('ingredient/new.html.twig', [
        'form' => $form->createView()
    ]);
}
```

Et il faut bien penser à rajouter le use en haut du Controller:

```
use App\Form\IngredientType;
```

Et aller créer un fichier dans le dossier templates/ingredient qui s'appellera new.html.twig.
Afin d'afficher le formulaire dans la vue new.html.twig, on va coder comme cela :

```
<div class="container">
    <h1> Création d'un ingrédient :</h1>

    {{ form(form) }}
</div>
```

PERSONNALISATION DU FORMULAIRE :

On va devoir ensuite apporter des modifications à notre formulaire afin de le styliser .

Dans un premier temps, on va s'occuper du contenu :

<https://symfony.com/doc/current/reference/forms/types/text.html>

Cette modification de style de formulaire va se faire dans notre buildform, présent dans notre fichier Type.

```
public function buildForm(FormBuilderInterface $builder, array $options): void
{
    $builder
        ->add('name', TextType::class, [
            'attr' => [
                'class' => 'form-control', // classe de Bootswatch pour le formulaire
                'minlength' => '2', // longueur minimale de la chaîne de caractère
                'maxlength' => '50', // longueur maximale de la chaîne de caractère
            ],
            'label' => 'Nom', // ce qui va remplacer le 'name' de notre BDD pour l'affichage
            'label_attr' => [
                'class' => 'form-label mt-4'
            ],
            'constraints' => [ // bien penser à ajouter le use Assert
                new Assert\Length(['min' => 2, 'max' => 50]), // contrainte de longueur minimale et maximale des mots
                new Assert\NotBlank() // contrainte qui s'assure que le champ de formulaire n'est pas vide
            ]
        ])
        ->add('price', MoneyType::class, [
            'attr' => [
                'class' => 'form-control', // classe de Bootswatch pour le formulaire
            ],
            'label' => 'Prix', // ce qui va remplacer le 'price' de notre BDD pour l'affichage
            'label_attr' => [
                'class' => 'form-label mt-4'
            ],
            'constraints' => [ // bien penser à ajouter le use Assert
                new Assert\Positive(), // contrainte que le prix soit supérieur à 1
                new Assert\LessThan(200) // contrainte qui s'assure que le prix ne sera pas supérieur à 200
            ]
        ])
}
```

Il faut bien penser à vérifier que les use se sont bien affichés :

```
Symfony\Component\Form\Extension\Core\Type\SubmitType;
Symfony\Component\Form\Extension\Core\Type\TextType;
Symfony\Component\Form\Extension\Core\Type\MoneyType;

Symfony\Component\Validator\Constraints as Assert;
Webmozart\Assert\Assert as AssertAssert;
```

Concernant le contenant, on va pouvoir personnaliser le rendu du formulaire dans la vue (new.html.twig) :

https://symfony.com/doc/current/form/form_customization.html

Une fois que le formulaire est soumis, on va devoir process le formulaire (traiter ou gérer le formulaire afin d'en extraire les informations nécessaires et les utiliser, ainsi que d'effectuer les vérifications de sécurité et de validité des données soumises) et l'envoyer dans la BDD :

```

#[Route('/ingredient/new', name: 'ingredient.new', methods:['GET','POST'])]
public function new(Request $request, EntityManagerInterface $manager) : Response
{
    $ingredient = new Ingredient();
    $form = $this->createForm(IngredientType::class, $ingredient);

    $form->handleRequest($request); // mise à jour du formulaire avec les données soumises

    if($form->isSubmitted() && $form->isValid()) // si le formulaire a été soumis et est valide (contraintes
        $ingredient = $form->getData();

        $manager -> persist($ingredient); // donner l'ordre d'ajouter la donnée dans la BDD
        $manager -> flush(); // pousser la donnée en BDD

        return $this->redirectToRoute('app_ingredient'); // Redirection vers la liste globale des ingrédients
    }

    return $this -> render('ingredient/new.html.twig', [
        'form' => $form->createView()
    ]);
}

```

On peut vérifier que notre donnée a bien été transmise et prise en compte par la BDD :

Edit Copy Delete 103 Jambon 3 2023-04-28 11:34:09

LIMITER LA CRÉATION A UN SEUL NOM :

Pour limiter la création d'un élément à un seul par nom (par exemple, un seul citron comme ingrédient, un seul jambon ...), on va ajouter la ligne suivante dans notre fichier dans le dossier Entity :

`#[UniqueEntity('name')]`

```

#[ORM\Entity(repositoryClass: IngredientRepository::class)]
#[UniqueEntity('name')]

```

Cela va donc générer un message d'erreur, si on essaye de créer un ingrédient avec un nom déjà existant dans la BDD :

Nom

- This value is already used.

MISE EN PLACE DE MESSAGE FLASH (messages de confirmation)

```

$manager -> persist($ingredient); // donner l'ordre d'ajouter la donnée dans la BDD
$manager -> flush(); // pousser la donnée en BDD

$this->addFlash(
    'success',
    'Votre ingrédient a été crée avec succès !'
);
return $this->redirectToRoute('app_ingredient');

```

On ajoute cette variable dans notre controller, et on va l'appeler dans notre vue.

Dans notre fichier index.html.twig, on va ajouter les lignes suivantes :

```

<div class="container mt-4">
    {% for flash_message in app.session.flashbag.get('success') %}
        <div class="alert alert-success mt-4">{{ flash_message }}</div>
    {% endfor %}

    <h1> Mes ingrédients </h1>

```

Ce qui va nous donner le message suivant :

```
Votre ingrédient a été créé avec succès !
```

Pour un message d'erreur, on va utiliser :

```
} else {
    $this->addFlash(
        'warning',
        'Il y a des erreurs, votre ingrédient n a pas pu être créé !'
    );
}
```

et on affichera dans notre vue :

```
<div class="container">

    {% for flash_message in app.session.flashbag.get('warning') %}
        <div class="alert alert-danger mt-4">{{ flash_message }}</div>
    {% endfor %}
```

FAIRE UN BOUTON DE REDIRECTION DE PAGE :

```
<a href="{{ path('ingredient.new') }}" class="btn btn-primary">Créer un
ingrédient</a>
```

PARTIE UPDATE / MISE A JOUR DES ÉLÉMENTS EXISTANTS :

On va pouvoir créer un formulaire de modification d'un élément déjà existant dans notre BDD, et pour cela, il faut récupérer l'id de l'élément, ainsi que les informations associées.

Tout d'abord, on va créer une fonction dans le Controller appelée edit, qui vont afficher les informations relatives à l'id de l'élément sélectionné :

```
#[Route('/ingredient/edit/{id}', name: 'ingredient.edit', methods:[ 'GET', 'POST'])]
public function edit(IngredientRepository $repository, int $id) : Response{
    $ingredient = $repository->findOneBy(['id' => $id]);
    $form = $this->createForm(IngredientType::class, $ingredient);

    return $this->render('ingredient/edit.html.twig', [
        'form' => $form ->createView()
    ]);
}
```

Et on va pouvoir créer une autre page appelée edit.html.twig, contenant un formulaire relativement similaire avec celui de la création d'un élément.

On va pouvoir faire cela plus rapidement à l'aide d'un paramconverter Symfony :

<https://symfony.com/bundles/SensioFrameworkExtraBundle/current/annotations/converters.html>

```

/**
 * Cette fonction permet de modifier un ingrédient avec un ParamConverter Symfony:
 */

#[Route('/ingredient/edit/{id}', name: 'ingredient.edit', methods:['GET','POST'])]
public function edit(Ingredient $ingredient) : Response{

    $form = $this->createForm(IngredientType::class, $ingredient);

    return $this->render('ingredient/edit.html.twig', [
        'form' => $form ->createView()
    ]);
}

```

Il faudra bien penser à ajouter le use suivant :

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\ParamConverter;
```

Et enfin, on va ajouter la soumission du formulaire à notre requête, en modifiant également le message de succès :

```

#[Route('/ingredient/edit/{id}', name: 'ingredient.edit', methods:['GET','POST'])]
public function edit(Ingredient $ingredient, Request $request, EntityManagerInterface $manager) : Response{

    $form = $this->createForm(IngredientType::class, $ingredient);

    $form->handleRequest($request); // mise à jour du formulaire avec les données soumises

    if($form->isSubmitted() && $form->isValid()){ // si le formulaire a été soumis et est valide (contraintes

        $ingredient = $form->getData();

        $manager -> persist($ingredient); // donner l'ordre d'ajouter la donnée dans la BDD
        $manager -> flush(); // pousser la donnée en BDD

        $this->addFlash([
            'success',
            'Votre ingrédient a été modifié avec succès !'
        ]);
        return $this->redirectToRoute('app_ingredient');
    }
    return $this->render('ingredient/edit.html.twig', [
        'form' => $form ->createView()
    ]);
}

```

Cette requête est relativement similaire à celle de la création, sauf que l'on va préremplir le formulaire avec les informations initialement enregistrées dans la BDD.

PARTIE DELETE :

```

#[Route('/ingredient/suppression/{id}', name: 'ingredient.delete', methods:['GET'])]
public function delete(EntityManagerInterface $manager, Ingredient $ingredient) : Response{
    if(!$ingredient) { // si l'ingrédient n'existe pas, redirection vers l'index
        return $this->redirectToRoute('app_ingredient');
    }

    $manager -> remove($ingredient);
    $manager -> flush();

    $this->addFlash(
        'success',
        'Votre ingrédient a été supprimé avec succès !'
    );

    return $this->redirectToRoute('app_ingredient');
}

```

Cette fonction est très rapide, on se sert de l'ID de l'élément, et on utilise la fonction remove. On a également ajouté un message de confirmation de suppression grâce à la fonction addFlash.

Dans notre twig, nous pouvons ajouter des boutons de modification et de suppression , qui reprendront bien l'ID de l'élément que nous souhaitons modifier ou supprimer :

```
<td>{{ ingredient.id }} </td>
{{ ingredient.createdAt|date("d/m/Y") }}</td>
<a href="{{ path('ingredient.edit', {id : ingredient.id})}}>Modification </a></td>
<a href="{{ path('ingredient.delete', {id : ingredient.id})}}>Suppression</a> </td>
```

LES RELATIONS ENTRE LES ENTITÉS

Ce sont les Foreign Key et Primary Key de notre BDD.

Lors de la création de notre BDD via le terminal de VS Code, quand nous avons tapé la commande

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

Nous allons pouvoir donner un nom, un type et un isNull ou non à notre nouvelle entité.

```
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console make:entity

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

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):
> name

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

Lorsque nous avons un lien à créer entre deux tables de la BDD, nous allons non pas donner le type string/float/integer... mais le type relation.

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

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

Main Types
* string
* text
* boolean
* integer (or smallint, bigint)
* float

Relationships/Associations
* relation (a wizard 🧙 will help you build the relation)
* ManyToOne
* OneToMany
* ManyToMany
* OneToOne
```

Il va ensuite nous demander à quelle entité nous souhaitons être reliée (attention, c'est bien la même écriture de saisie que l'entité, et non la table de la BDD)

```
What class should this entity be related to?:
> Ingredient
```

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

Il va ensuite nous demander le type de relation que nous souhaitons créer entre les deux entités :

Type	Description
ManyToOne	Each <code>Recipe</code> relates to (has) <code>one Ingredient</code> . Each <code>Ingredient</code> can relate to (can have) <code>many Recipe</code> objects.
OneToMany	Each <code>Recipe</code> can relate to (can have) <code>many Ingredient</code> objects. Each <code>Ingredient</code> relates to (has) <code>one Recipe</code> .
ManyToMany	Each <code>Recipe</code> can relate to (can have) <code>many Ingredient</code> objects. Each <code>Ingredient</code> can also relate to (can also have) <code>many Recipe</code> objects.
OneToOne	Each <code>Recipe</code> relates to (has) exactly <code>one Ingredient</code> . Each <code>Ingredient</code> also relates to (has) exactly <code>one Recipe</code> .

On va choisir la méthode que nous préférons entre les quatre en analysant notre demande :

```
Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]:  
> ManyToMany
```

Ici, nous voulons que chaque recette puisse contenir plusieurs ingrédients, mais également que chaque ingrédient puisse être appelé dans plusieurs recettes. Nous utilisons donc le ManyToMany.

Ensuite, il va nous demander si depuis l'entité Ingrédient, nous souhaitons récupérer les recettes :

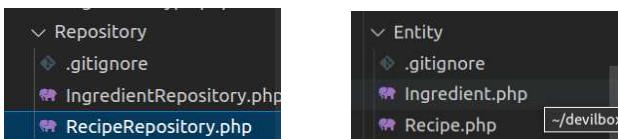
```
Do you want to add a new property to Ingredient so that you can access/update Recipe objects from it -  
e.g. $ingredient->getRecipes()? (yes/no) [yes]:  
> no  
  
updated: src/Entity/Recipe.php
```

Et enfin, il va nous demander si on souhaite une autre propriété :

```
Add another property? Enter the property name (or press <return> to stop adding fields):  
>  
  
Success!  
  
Next: When you're ready, create a migration with php bin/console make:migration
```

si oui, on note directement le nom, sinon on appuie sur entrée.

Il va bien nous avoir créer deux fichiers :



On va ensuite s'occuper des contraintes dans l'entité, en ajoutant les uses nécessaires :

```
namespace App\Entity;  
  
use App\Repository\RecipeRepository;  
use Doctrine\Common\Collections\ArrayCollection;  
use Doctrine\Common\Collections\Collection;  
use Doctrine\DBAL\Types\Types;  
use Doctrine\ORM\Mapping as ORM;  
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;  
use Symfony\Component\Validator\Constraints as Assert;
```

et on rajoute notre contrainte d'entité unique par rapport au nom :

```
#[ORM\Entity(repositoryClass: RecipeRepository::class)]  
#[UniqueEntity('name')]
```

Bien penser, quand une colonne est obligatoire, de rajouter la contrainte, pour ne pas que le champ de formulaire soit vide :

```
# [Assert\NotBlank() ]
```

Exemple de contraintes/asserts :

```
#[ORM\Column(length: 50)]
#[Assert\NotBlank()]
#[Assert\Length(min :2, max:50)]
private ?string $name = null;

#[ORM\Column(nullable: true)]
#[Assert\Positive()]
#[Assert\LessThan(1441)]
private ?int $time = null;

#[ORM\Column(nullable: true)]
#[Assert\Positive()]
#[Assert\LessThan(51)]
private ?int $nbPeople = null;

#[ORM\Column(nullable: true)]
#[Assert\Positive()]
#[Assert\LessThan(6)]
private ?int $difficulty = null;

#[ORM\Column(type: Types::TEXT)]
#[Assert\NotBlank()]
private ?string $description = null;
```

Pour les dates immuables, comme createdAt et updatedAt, il faut bien ajouter la contrainte qui vérifiera que ces données ne soient pas nulles :

```
#[ORM\Column]
#[Assert\NotNull()]
private ?\DateTimeImmutable $createdAt = null;

#[ORM\Column]
#[Assert\NotNull()]
private ?\DateTimeImmutable $updatedAt = null;
```

On va modifier quelques éléments de code dans le construct notamment pour la date de création et la date de modification :

```
public function __construct()
{
    $this->ingredients = new ArrayCollection();
    $this->createdAt = new DateTimeImmutable();
    $this->updatedAt = new DateTimeImmutable();
}

#[ORM\PrePersist()]
public function setUpdatedAtValue()
{
    $this->updatedAt = new DateTimeImmutable();
```

En ajoutant un cycle de vie en ORM :

```
#[UniqueEntity('name')]
#[ORM\HasLifecycleCallbacks]
```

Puis nous n'oublierons pas la migration :

php bin/console make:migration
php bin/console d:m:m

```

maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console make:migration

Success!

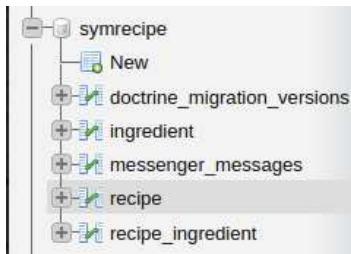
Next: Review the new migration "migrations/Version20230501134208.php"
Then: Run the migration with php bin/console doctrine:migrations:migrate
See https://symfony.com/doc/current/bundles/DoctrineMigrationsBundle/index.html
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console doctrine:migrations:migrate

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

[notice] Migrating up to DoctrineMigrations\Version20230501134208
[notice] finished in 111.2ms, used 20M memory, 1 migrations executed, 5 sql queries
[OK] Successfully migrated to version : DoctrineMigrations\Version20230501134208

```

On peut constater dans notre BDD que nous avons bien nos deux tables distinctes (ingredient et recipe) et une table de jointure qui va faire le lien entre les ID des ingrédients et les ID des recettes (recipe_ingredient)



On peut toujours effectuer des fixtures :

```

maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console d:f:l

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

> purging database
> loading App\DataFixtures\AppFixtures

```

Sans oublier de modifier dans le fichier AppFixtures :

```

$recipes = [];
for ($j=1; $j <= 25 ; $j++) {
    $recipe = new Recipe();
    $recipe
        -> setName ('Recipe' . $j)
        -> setTime (mt_rand(0, 1)== 1 ? mt_rand(1,1440) :null)
        -> setNbPeople (mt_rand(0, 1)== 1 ? mt_rand(1,50) :null)
        -> setDifficulty (mt_rand(0, 1)== 1 ? mt_rand(1,5) :null)
        -> setDescription ('Description' . $j)
        -> setPrice (mt_rand(0, 1)== 1 ? mt_rand(1,1000) :null)
        -> setIsFavorite (mt_rand(0, 1)== 1 ? true : false);

    for ($k=1; $k < mt_rand(5,15); $k++) {
        $recipe -> addIngredient($ingredients[mt_rand(0, count($ingredients) -1)]);
    }
    $manager -> persist($recipe);
}

$manager->flush();
}

```

Cela va également nous relier de façon aléatoire les ingrédients et les recettes entre eux dans la table de jointure :

	Edit	Copy	Delete	recipe_id	ingredient_id
				1	2
				1	9
				1	10
				1	14
				1	27
				1	41
				1	50
				2	2
				2	8
				2	10
				2	20
				2	22
				2	26
				2	32
				2	36
				3	14
				3	17
				3	22

Pour pouvoir ôter toutes les fixtures et insérer de vraies données (remettre la base de données propre une fois les tests effectués) :

```
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console d:d:d --force
Dropped database 'symrecipe' for connection named default
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console d:d:c
Created database 'symrecipe' for connection named default
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console d:m:m

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

[notice] Migrating up to DoctrineMigrations\Version20230426115423
[notice] finished in 53.6ms, used 20M memory, 1 migrations executed, 2 sql queries

[OK] Successfully migrated to version : DoctrineMigrations\Version20230426115423
```

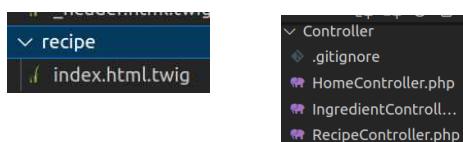
On crée ensuite le controller :

```
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console make:controller RecipeController
created: src/Controller/RecipeController.php
created: templates/recipe/index.html.twig

Success!

Next: Open your new controller class and add some pages!
```

On aura donc toujours deux fichiers de créés, un template et un controller :



On modifie le controller pour les routes et l'ajout de la méthode :

```
#[Route('/recipe', name: 'recipe.index', methods:['GET'])]
public function index(RecipeRepository $repository, PaginatorInterface $paginator, Request $request): Response
{
    $recipes = $paginator->paginate(
        $repository -> findAll(),
        $request->query->getInt('page', 1),
        10
    );
    return $this->render('recipe/index.html.twig', [
        'recipes' => $recipes
    ]);
}
```

LES TERNAIRES TWIG

<https://stackoverflow.com/questions/11820297/twig-ternary-operator-shorthand-if-then-else>

Un ternaire est une autre manière d'exprimer une condition, de façon plus simple et plus concise en terme de code.

Si la condition est vraie, alors renvoie la valeur à gauche du ?, sinon, renvoie la valeur à droite du ?

Exemple :

```
twig  
  
{{ age >= 18 ? 'Majeur' : 'Mineur' }}
```

Si la variable ‘age’ est supérieure ou égale à 18, affiche ‘Majeur’, sinon, affiche ‘Mineur’

Voilà un exemple pour l'affichage de donnée lié à la BDD :

```
{% for recipe in recipes %}  
<tr class="table-primary">  
    <th scope="row">{{recipe.id}}</th>  
    <td>{{recipe.name}}</td>  
    <td>{{ (recipe.price is same as (null) ) ? 'non renseigné' : recipe.price }}</td>  
    <td>{{ (recipe.difficulty is same as (null) ) ? 'non renseigné' : recipe.difficulty }}</td>  
    <td>{{recipe.createdAt|date("d/m/Y") }}</td>
```

Si le prix est nul, on affiche « non renseigné », sinon, on affiche le prix de la BDD.

Si la difficulté est nulle, on affiche « non renseignée », sinon, on affiche la difficulté de la BDD.

Si nous avons ce message d'erreur :

```
Object of class App\Entity\Ingredient could not be converted to string
```

Il faut simplement ajouter dans l'Entité concernée (ingrédient dans ce cas là) une public function :

```
public function __toString()  
{  
    return $this->name;  
}
```

Cette méthode va retourner l'objet sous forme de chaîne de caractères.

Documentation sur l'entity type :

<https://symfony.com/doc/current/reference/forms/types/entity.html>

SECURITE ET COMPTE UTILISATEUR

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

On va commencer par installer le bundle security :

```
composer require symfony/security-bundle
```

```

maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ composer require symfony/security-bundle
Info from https://repo.packagist.org: #StandWithUkraine
./composer.json has been updated
Running composer update symfony/security-bundle
Loading composer repositories with package information
Updating dependencies
Nothing to modify in lock file
Installing dependencies from lock file (including require-dev)
Nothing to install, update or remove
Package sensio/framework-extra-bundle is abandoned, you should avoid using it. Use Symfony instead.
Generating optimized autoload files
108 packages you are using are looking for funding.
Use the `composer fund` command to find out more!

Run composer recipes at any time to see the status of your Symfony recipes.

Executing script cache:clear [OK]
Executing script assets:install public [OK]

```

Ce qui va créer un dossier yaml :

! security.yaml

qui va contenir :

- le password hasher permettant de hasher les mots de passe,
- les providers nous fournissent les utilisateurs présents dans la BDD via un username
- les firewalls (pare feux)
- l'access control qui va contrôler les permissions requises

CREATION D'UN USER :

On va commencer par taper dans le terminal la commande suivante :

php bin/console make:user

```

• maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console make:user
The name of the security user class (e.g. User) [User]:
>

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

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

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

Does this app need to hash/check user passwords? (yes/no) [yes]:
>

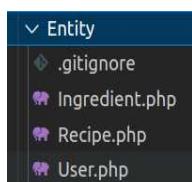
created: src/Entity/User.php
created: src/Repository/UserRepository.php
updated: src/Entity/User.php
updated: config/packages/security.yaml

Success!

```

Next Steps:
- Review your new `App\Entity\User` class.

On peut donc constater que l'entité User a été créée



On va ensuite créer nos colonnes (propriétés) :

```
● maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console make:entity User
Your entity already exists! So let's add some new fields!
New property name (press <return> to stop adding fields):
> fullName

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

Field length [255]:
> 50

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

updated: src/Entity/User.php

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

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

Field length [255]:
> 50

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

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

On peut également ajouter le createdAt par exemple...

Dans le User.php, on va ajouter la méthode suivante (sous le tableau des rôles) :

```
private ?string $plainPassword = null;
```

À laquelle on va ajouter un getter et un setter :

```
public function getPlainPassword()
{
    return $this->plainPassword;
}

/**
 * @return self
 */
public function setPlainPassword(?string $plainPassword)
{
    $this->plainPassword = $plainPassword;
    return $this;
}
```

Et on oublie pas de créer le construct pour la date createdAt , afin qu'il se complète tout seul :

```
#[ORM\Column]
private ?\DateTimeImmutable $createdAt = null;

public function __construct()
{
    $this->createdAt = new \DateTimeImmutable();
```

On va également s'occuper des contraintes , en oubliant pas d'ajouter les uses :

```
use Symfony\Component\Validator\Constraints as Assert;
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
```

Et la unique entity, par l'email cette fois :

```
#UniqueEntity('email')
```

On complète les Asserts dans notre entité :

```
#[ORM\Column(length: 50)]
#[Assert\NotBlank()]
#[Assert\Length(min :2, max:50)]
private ?string $fullName = null;

#[ORM\Column(length: 50, nullable: true)]
#[Assert\Length(min :2, max:50)]
private ?string $pseudo = null;

#[ORM\Column(length: 180, unique: true)]
#[Assert\Email()]
#[Assert\Length(min :2, max:180)]
private ?string $email = null;

#[ORM\Column]
#[Assert\NotNull()]
private array $roles = [];
```

Et on va pouvoir passer à la migration vers la BDD :

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

```
php bin/console d:m:m
```

Une fois que la BDD est prête, on va ajouter un User Provider , qui va permettre de récupérer un utilisateur depuis une zone de stockage (BDD).

On va donc retourner dans notre security.yaml et on va vérifier que l'on ai bien le code suivant :

```
app_user_provider:
    entity:
        class: App\Entity\User
        property: email
```

HASHING DE MOT DE PASSE :

On remarque que dans notre security.yaml, Symfony a déjà défini le hashage :

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

On va pouvoir créer des fixtures d'utilisateurs afin de gagner du temps . Dans notre dossier AppFixtures, on va ajouter le code suivant :

```
// Users
for ($k=0; $k < 10 ; $k++) {
    $user = new User();
    $user
        -> setFullName ('FullName' . $k)
        -> setPseudo ('Pseudo' . $k)
        -> setEmail ('Email'.'@' . $k)
        -> setRoles(['ROLE_USER'])
        -> setPassword ('Password' . $k);
    $manager -> persist($user);
}
```

On va load les fixtures :

```
● maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console d:f:l  
Careful, database "symrecipe" will be purged. Do you want to continue? (yes/no) [no]:  
> yes  
  
> purging database  
> loading App\\DataFixtures\\AppFixtures
```

Et notre BDD sera incrémentée des fixtures :

On va maintenant encoder le password . On va appeler le UserPasswordInterface qui a nous permettre , grâce à la méthode hashPassword , en lui passant le user et le plainTextPassword, d'encoder le mot de passe.

On va donc l'ajouter à notre code, en mettant une private et un construct :

```
use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;

class AppFixtures extends Fixture
{
    private UserPasswordHasherInterface $hasher;

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

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

Ainsi, on va pouvoir modifier notre Fixture User :

```
// Users
for ($k=0; $k < 10 ; $k++) {
    $user = new User();
    $user
        -> setFullName ('FullName' . $k)
        -> setPseudo ('Pseudo' . $k)
        -> setEmail ('Email' . '@' . $k)
        -> setRoles(['ROLE_USER']);
}

$hashPassword = $this->hasher->hashPassword(
    $user,
    'password'
);

$user->setPassword($hashPassword);

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

Et après avoir reload nos fixtures, on va constater que la table user, colonne password aura changée :

password
\$2y\$13\$BwVqnutVZWLIO0wGeXzkuHun8S4LBuC1D9x/wvMovi...
\$2y\$13\$6MKdaoTzJu5iXoDYLTYiYesI/8HWV1QvCdM64DS5o...
\$2y\$13\$2mmIMD8CkImo78z8nwhO0brdQsrVYG/O0ClowGN9ak...
\$2y\$13\$M4tzwHwlX9mShrdzb4KCeQh9/m3ysL4YzONnddU0DF...
\$2y\$13\$nxgjEXEKU.Ey0wfhntGfTjv.U24ta21sicbracBMbrsAH...
\$2y\$13\$emE6ggJLdftrRbb4rNxroU772/TxKfUhpdtnqN0J...
\$2y\$13\$oePv8PUlPfZgFWFwTTu7DKLsdPPvIKJ2WFfJAUo...
\$2y\$13\$LlwOmnuNTuU/lTrLBQaPU8wJ6LW.bcxDLnJLhTQdQEGK...
\$2y\$13\$nhnRMKA06hNcSBSHgUu1oPo.sgc1VygfsDKIG1Q9G...
\$2y\$13\$36d4f4B3wB1JiifwOnI52e9u068R7zHvYQH1oEw...

LES ENTITY LISTENERS

On a une méthode plus simple dans Symfony, qui s'appellent les Entity Listeners :
<https://symfony.com/bundles/DoctrineBundle/current/entity-listeners.html>

Ce sont des fichiers qui vont écouter ce qui se passe au niveau des Entités, et faire plusieurs actions, que ce soit au niveau de la persistance, de la mise à jour...

On modifie donc notre AppFixtures :

```
// Users
for ($k=0; $k < 10 ; $k++) {
    $user = new User();
    $user
        -> setFullName ('FullName' . $k)
        -> setPseudo ('Pseudo' . $k)
        -> setEmail ('Email'.'@' . $k)
        -> setRoles(['ROLE_USER'])
        -> setPlainPassword('password');

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

Et dans le fichier service.yaml, on va aller ajouter ce code :

```
App\:
    resource: '../src/'
    exclude:
        - '../src/DependencyInjection/'
        - '../src/Entity/'
        - '../src/Kernel.php'

App\EntityListener\UserListener:
    arguments:
        $hasher: '@security.password_hasher'
    tags: ['doctrine.orm.entity_listener']
```

Dans le fichier User.php, on va aller ajouter un #ORM :

```
#[ORM\Entity(repositoryClass: UserRepository::class)]
#[UniqueEntity('email')]
#[ORM\EntityListeners('App\EntityListener\UserListener')]
```

Dans le dossier SRC, on va ensuite créer un sous-dossier EntityListener, qui contiendra un fichier UserListener.php :

```
src
> Controller
< DataFixtures
  AppFixtures.php
> Entity
< EntityListener
  UserListener.php
```

Dans lequel on va ajouter le code suivant :

```
<?php

namespace App\EntityListener;

use App\Entity\User;
use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;

class UserListener
{
    private UserPasswordHasherInterface $hasher;

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

    public function prePersist(User $user) // fonction qui va prendre un user courant et
    //écouter les actions faites dessus pour le récupérer
    {
        $this->encodePassword($user);
    }

    public function preUpdate(User $user) // à chaque fois que l'utilisateur modifie son MDP,
    //on va vouloir que son MDP soit encodé
    {
        $this->encodePassword($user);
    }
}
```

```

    /**
     * Encodage du mot de passe en fonction de plainPassword
     *
     * @param User $user
     * @return void
     */
    public function encodePassword(User $user)
    {
        if($user->getPlainPassword() === null) // on va vérifier si le plainPassword est vide
        {
            return; // s'il est vide, on revient en arrière
        };

        $user-> setPassword(
            $this->hasher->hashPassword(
                $user,
                $user -> getPlainPassword()
            )
        );
    }
}

```

FIREWALL

<https://symfony.com/doc/current/security.html#the-firewall>

Le pare-feu va définir quelles parties de l'application sont sécurisées et comment les utilisateurs pourront s'authentifier (formulaire de connexion, jeton API ...)

Dans notre fichier security.yaml, on en retrouve deux :

- main :
ce firewall définit les règles de sécurité dans l'environnement de production
(authentification utilisateurs,gestion des rôles et permissions, et protection contre les attaques type CSRF. Il utilise le provider d'authentification ‘database’ pour vérifier les informations d'identification des utilisateurs dans la BDD.
- dev :
ce firewall définit les règles de sécurité dans l'environnement de développement. IL utilise le provider d'authentification ‘anonymous’ pour permettre un accès non authentifié aux ressources de l'application.

```

firewalls:
  dev:
    pattern: ^/(_(profiler|wdt)|css|images|js)/
    security: false
  main:
    lazy: true
    provider: app_user_provider

```

Pour rappel, un provider est un composant de sécurité responsable de charger et authentifier l'utilisateur. IL agit comme une passerelle entre l'utilisateur et les données d'authentification stockées, telles que la BDD ou un serveur d'authentification externe.

Il va vérifier si les informations d'identification de l'utilisateur sont valides et va fournir un objet utilisateur authentifié à Symfony.

Symfony va fournir des providers prédéfinis pour l'authentification basée sur :

- le formulaire
- les jetons
- les certificats

FORMULAIRE DE LOGIN

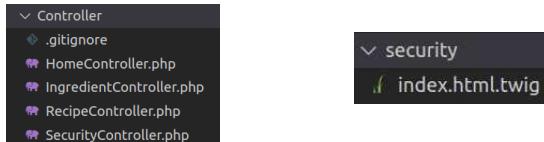
<https://symfony.com/doc/current/security.html#authenticating-users>

On va créer un SecurityController qui va s'occuper de la gestion du login utilisateur.

```
• maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console make:controller SecurityController
  r
  created: src/Controller/SecurityController.php
  created: templates/security/index.html.twig

  Success!
Next: Open your new controller class and add some pages!
```

On aura donc bien la création des fichiers suivants :



On va renommer le index.html.twig en login.html.twig

On va modifier le Controller :

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

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

On va ensuite modifier le fichier security.yaml :

```
main:
    lazy: true
    provider: app_user_provider
    form_login:
        login_path: security.login
        check_path: security.login
```

On va maintenant construire notre formulaire de connexion. Nous allons dans le fichier login.html.twig :

```
{% block body %}



# Formulaire de connexion </h1> <form action="{{path('security.login')}}" method="post"> <div class="form-group"> <label for="username" class="form-label mt-4">Adresse Email</label> <input type="email" class="form-control" id="username" name="_username" placeholder="exemple@symrecipe.com"> <small id="emailHelp" class="form-text text-muted">We'll never share your email with anyone else.</small> </div> <div class="form-group"> <label for="password" class="form-label mt-4">Mot de passe</label> <input type="password" class="form-control" id="password" name="_password" placeholder="*****"> </div> <button type="submit" class="btn btn-primary mt-4"> Se connecter </button> </form> </div> {% endblock %}


```

DECONNEXION

<https://symfony.com/doc/current/security.html#logging-out>

On va ajouter cette fonction vide dans notre SecurityController :

```
/**  
 * Cette fonction permet à l'utilisateur de se déconnecter  
 *  
 * @return void  
 */  
#[Route('/deconnexion', name: 'security.logout')]  
public function logout()  
{  
  
}
```

Et on va ajouter ce code à notre security.yaml :

```
form_login:  
    login_path: security.login  
    check_path: security.login  
    logout:  
        path: security.logout
```

GESTION DES ERREURS

On va modifier notre Controller afin d'avoir des messages d'erreur :

```
/**  
 * Cette fonction permet à l'utilisateur de se connecter  
 *  
 * @return Response  
 */  
#[Route('/connexion', name: 'security.login', methods:['GET', 'POST'])]  
public function login(AuthenticationUtils $authenticationUtils): Response  
{  
    return $this->render('security/login.html.twig', [  
        'last_username' => $authenticationUtils->getLastUsername(),  
        'error' => $authenticationUtils->getLastAuthenticationError(),  
    ]);  
}
```

Et on va modifier notre login.html.twig :

```
<div class="container">  
    <h1 class="mt-4"> Formulaire de connexion </h1>  
  
    {% if error %}  
        <div class="alert alert-danger mt-4">  
            {{ error.messageKey|trans(error.messageData, 'security') }}  
        </div>  
    {% endif %}  
  
    <form action="{{path('security.login')}}" method="post">  
        <div class="form-group">  
            <label for="username" class="form-label mt-4">Adresse Email</label>  
            <input type="email" class="form-control" id="username"  
                  name="username" placeholder="exemple@symrecipe.com" value="{{last_username}}>  
            <small id="emailHelp" class="form-text text-muted">We'll never share your email with anyone else.</small>  
        </div>  
        <div class="form-group">  
            <label for="password" class="form-label mt-4">Mot de passe</label>  
            <input type="password" class="form-control" id="password" name="_password" placeholder="*****">  
        </div>  
  
        <button type="submit" class="btn btn-primary mt-4"> Se connecter </button>  
    </form>  
</div>
```

On a ajouté la value à notre input email, et un message d'erreur type alerte selon bootswatch.

On peut constater sur notre vue que nous avons bien l'apparition d'un message d'erreur si les identifiants de connexion sont erronés.

Formulaire de connexion



Si on entre un bon utilisateur avec le bon mot de passe, on pourra voir en bas de la page que nous sommes bien sur une session privée :

```
| ↗ @ app_home 18 ms 2.0 MB 🚶 test@gmail.com 🔍 2 ms 1 in 0.74 ms
```

on pourra se déconnecter en cliquant sur l'adresse mail de l'utilisateur dans cette barre en bas de la page.

FORMULAIRE D'INSCRIPTION

Dans le terminal, on va saisir la commande suivante afin de créer un formulaire :

```
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console make:form
The name of the form class (e.g. GentlePizzaType):
> RegistrationType

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

created: src/Form/RegistrationType.php

Success!
```

Next: Add fields to your form and start using it.
Find the documentation at <https://symfony.com/doc/current/forms.html>

Il va nous avoir créé un fichier RegistrationType dans le dossier Form :

```
✓ Form
  ↗ IngredientType.php
  ↗ RecipeType.php
  ↗ RegistrationType.php
```

Et on va pouvoir aller dans ce fichier modifier notre buildForm :

```
class RegistrationType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('fullName', TextType::class, [
                'attr'=> [
                    'class' => 'form-control',
                    'minlength' => '2',
                    'maxlength'=> '50',
                ],
                'label' => 'Nom / Prénom',
                'label_attr' => [
                    'class' => 'form_label'
                ],
                'constraints' => [
                    new Assert\NotBlank(),
                    new Assert\Length(['min'=> 2, 'max'=> 50])
                ]
            ])
            ->add('pseudo', TextType::class, [
                'attr'=> [
                    'class' => 'form-control',
                    'minlength' => '2',
                    'maxlength'=> '50',
                ],
                'label' => 'Pseudo',
                'label_attr' => [
                    'class' => 'form_label'
                ],
                'constraints' => [
                    new Assert\NotBlank(),
                    new Assert\Length(['min'=> 2, 'max'=> 50])
                ]
            ])
    }
}
```

```

        ],
        'label' => 'Pseudo Facultatif',
        'label_attr' => [
            'class' => 'form-label'
        ],
        'constraints' => [
            new Assert\Length(['min'=> 2, 'max'=> 50])
        ]
    ])
->add('email', EmailType::class, [
    'attr' => [
        'class' => 'form-control',
        'minlength' => '2',
        'maxlength'=> '180',
    ],
    'label' => 'Adresse Email',
    'label_attr' => [
        'class' => 'form-label'
    ],
    'constraints' => [
        new Assert\NotBlank(),
        new Assert\Email(),
        new Assert\Length(['min'=> 2, 'max'=> 180])
    ]
])
->add('plainPassword', RepeatedType::class, [
    'type' => PasswordType::class,
    'first_options' => [
        'label' => 'Mot de passe',
    ],
    'second_options' => [
]
}

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

```

On va ensuite devoir créer notre Controller :

```

#[Route('/inscription', name: 'security.registration', methods:[GET, POST])]
public function registration(Request $request, EntityManagerInterface $manager) : Response
{
    $user = new User();
    $user->setRoles(['ROLE_USER']);
    $form = $this->createForm(RegistrationType::class, $user);

    $form->handleRequest($request);

    if($form->isSubmitted() && $form->isValid()) {
        $user = $form->getData();

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

        $this->addFlash(
            'success',
            'Votre compte utilisateur a été créé avec succès !'
        );
        return $this->redirectToRoute('security.login');
    }

    return $this->render('security/registration.html.twig', [
        'form' => $form->createView(),
    ]);
}

```

On va rajouter le message flash dans login.html.twig :

```

<div class="container">
    <h1 class="mt-4"> Formulaire de connexion </h1>

    {% for flash_message in app.session.flashbag.get('success') %}
        <div class="alert alert-success mt-4">{{ flash_message }}</div>
    {% endfor %}

```

On va ensuite aller dans notre dossier security, créer un fichier registration.html.twig



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

{% block title %} SymRecipe - Incription {% endblock %}

{% block body %}

<div class="container">
    <h1 class="mt-4">Formulaire d'inscription</h1>
    {{form(form)}}
</div>

{% endblock %}
```

MODIFICATION DU PROFIL UTILISATEUR (update)

Dans le terminal, on va saisir la commande suivante pour créer un nouveau formulaire de modification du profil utilisateur :

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

```
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console make:form
The name of the form class (e.g. GentleChefType):
> UserType

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

created: src/Form/UserType.php

Success!
```

Next: Add fields to your form and start using it.
Find the documentation at <https://symfony.com/doc/current/forms.html>

On va ensuite modifier notre UserType, en ne gardant que fullName, pseudo et submit :

```
public function buildForm(FormBuilderInterface $builder, array $options): void
{
    $builder
        ->add('fullName', TextType::class, [
            'attr'=> [
                'class' => 'form-control',
                'minlength' => '2',
                'maxlength'=> '50',
            ],
            'label' => 'Nom / Prénom',
            'label_attr' => [
                'class' => 'form-label mt-4'
            ],
            'constraints' => [
                new Assert\NotBlank(),
                new Assert\Length(['min'=> 2, 'max'=> 50])
            ]
        ])
        ->add('pseudo', TextType::class, [
            'attr'=> [
                'class' => 'form-control',
                'minlength' => '2',
                'maxlength'=> '50',
            ],
            'required' => false,
            'label' => 'Pseudo (facultatif)',
            'label_attr' => [
                'class' => 'form-label mt-4'
            ],
            'constraints' => [
                new Assert\Length(['min'=> 2, 'max'=> 50])
            ]
        ])
        ->add('submit', SubmitType::class, [
            'attr' => [
                'class' => 'btn btn-primary mt-4'
            ]
        ]);
}
```

Si on souhaite ajouter un contrôle supplémentaire, on peut également ajouter le plainPassword, qui va obliger l'utilisateur à re-saisir son mot de passe pour valider les modifications de son profil :

```
->add('plainPassword', PasswordType::class, [
    'attr' => [
        'class' => 'form-control',
    ],
    'label' => 'Mot de passe',
    'label_attr' => [
        'class' => 'form-label mt-4'
    ]
])
```

Et dans le terminal, nous allons créer le controller :

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

```
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console make:controller
Choose a name for your controller class (e.g. AgreeablePopsicleController):
> UserController
created: src/Controller/UserController.php
created: templates/user/index.html.twig

Success!
Next: Open your new controller class and add some pages!
```

On constate que l'on a un sous-dossier user dans le dossier templates, et nous allons renommer le fichier index.html.twig en edit.html.twig.

```
└─ user
    └── edit.html.twig
```

On va modifier notre controller :

```
#Route('/utilisateur/edit/{id}', name: 'user.edit', methods:['GET','POST'])
public function edit(User $user, Request $request, EntityManagerInterface $manager, UserPasswordHasherInterface $hasher): Response
{
    // vérification que l'utilisateur est bien connecté, sinon il est redirigé vers la page de connexion
    if(!$this->getUser())
    {
        return $this->redirectToRoute('security.login');
    }
    // vérification que l'utilisateur courant est bien le même que l'utilisateur dont on a récupéré l'ID
    if($this->getUser() !== $user)
    {
        return $this->redirectToRoute('recipe.index');
    }

    $form = $this->createForm(UserType::class, $user);
    $form->handleRequest($request);
    if($form->isSubmitted() && $form->isValid()) {

        // mise en place d'une condition, si le mot de passe saisi par l'utilisateur est le même qu'en BDD alors on autorise la modification des données :
        if($hasher->isPasswordValid($user, $form->getData()->getPlainPassword())){
            $user = $form->getData();
            $manager->persist($user);
            $manager->flush();

            $this->addFlash(
                'success',
                'Votre compte utilisateur a été modifié avec succès !'
            );

            return $this->redirectToRoute('recipe.index');
        } else {
            $this->addFlash(
                'warning',
                'Le mot de passe renseigné est incorrect'
            );
        }
    }
    return $this->render('user/edit.html.twig', [
        'form' => $form->createView(),
    ]);
}
```

On va modifier notre template twig :

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

{% block title %}SymRecipe - Modification de l'utilisateur{% endblock %}

{% block body %}

<div class="container">
    <h1 class="mt-4">Formulaire de modification des informations de l'utilisateur</h1>

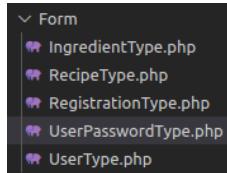
    {% for flash_message in app.session.flashbag.get('warning') %}
    <div class="alert alert-danger mt-4">{{ flash_message }}</div>
    {% endfor %}

    {{ form(form) }}
</div>

{% endblock %}
```

MODIFICATION DU MOT DE PASSE (update)

On va directement créer un formulaire de modification de mot de passe dans Type sans passer par une entité. Dans le dossier Form, on va créer un fichier UserPasswordType.php :



Dans ce fichier, nous allons créer le builder :

```
class UserPasswordType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('plainPassword', RepeatedType::class, [
                'type' => PasswordType::class,
                'first_options' => [
                    'attr' => [
                        'class' => 'form-control',
                    ],
                    'label' => 'Mot de passe',
                    'label_attr' => [
                        'class' => 'form-label mt-4'
                    ]
                ],
                'second_options' => [
                    'attr' => [
                        'class' => 'form-control',
                    ],
                    'label' => 'Confirmation du mot de passe',
                    'label_attr' => [
                        'class' => 'form-label mt-4'
                    ]
                ],
                'invalid_message' => 'Les mots de passe ne correspondent pas.'
            ])
            ->add('newPassword', PasswordType::class, [
                'attr' => [
                    'class' => 'form-control',
                ],
                'label' => 'Nouveau mot de passe',
                'label_attr' => [
                    'class' => 'form-label mt-4'
                ],
                'constraints'=> [new Assert\NotBlank()]
            ])

            ->add('submit', SubmitType::class, [
                'attr' => [
                    'class' => 'btn btn-primary mt-4'
                ]
            ]);
    }
}
```

Ce formulaire va nous demander de saisir notre mot de passe existant, de le confirmer et de saisir un nouveau mot de passe.

Nous allons ensuite retourner dans le UserController et nous allons ajouter la public function suivante :

```
#[Route('/utilisateur/edition-mot-de-passe/{id}', name: 'user.edit.password', methods:['GET','POST'])]
public function editPassword(User $user, Request $request, UserPasswordEncoderInterface $hasher, EntityManagerInterface $manager) : Response
{
    $form = $this->createForm(UserPasswordType::class);

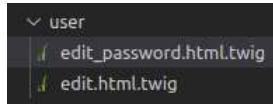
    $form->handleRequest($request);
    if($form->isSubmitted() && $form->isValid())
    {
        if($hasher->isPasswordValid($user,$form->getData()['plainPassword']))
        {
            $user->setPassword(
                $hasher->hashPassword(
                    $user,
                    $form->getData()['newPassword']
                )
            );
        }

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

        $this->addFlash(
            'success',
            'Votre mot de passe a été modifié avec succès !'
        );
        return $this->redirectToRoute('recipe.index');
    }
    else {
        $this->addFlash(
            'warning',
            'Le mot de passe renseigné est incorrect'
        );
    }
}

return $this->render('user/edit_password.html.twig', [
    'form' => $form->createView()
]);
}
```

Dans le template user, on va créer un fichier edit_password.html.twig



où nous allons intégrer le code suivant, similaire à celui de notre edit.html.twig :

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

{% block title %}SymRecipe - Modification du mot de passe{% endblock %}

{% block body %}


# Formulaire de modification du mot de passe de l'utilisateur



    {% for flash_message in app.session.flashbag.get('warning') %}
    <div class="alert alert-danger mt-4">{{ flash_message }}</div>
    {% endfor %}

    {{ form(form) }}


{% endblock %}
```

Il y a également une deuxième méthode, où nous allons créer une autre colonne dans la base de données, appelée updated_at, et qui va venir enregistrer les informations temporelles de la modification, et prendre en compte la fonction preUpdate présente dans le UserListener . C'est cette dernière méthode que nous allons privilégier.

Pour cela, on commence par modifier la BDD en ajoutant une colonne updated_at, après la colonne created_at :

<input type="checkbox"/> 7 <u>created_at</u> datetime	No	None	(DC2Type:datetime_immutable)				
<input type="checkbox"/> 8 <u>updated_at</u> datetime	No	None	(DC2Type:datetime_immutable)				

Ensuite, on va aller déclarer cette colonne dans notre fichier User, de la même manière que le created_at , en effectuant les getter et setter:

```

#[ORM\Column]
#[Assert\NotNull()]
private ?\DateTimeImmutable $createdAt = null;

#[ORM\Column]
#[Assert\NotNull()]
private ?\DateTimeImmutable $updatedAt = null;

public function __construct()
{
    $this -> createdAt = new DateTimeImmutable();
    $this -> updatedAt = new DateTimeImmutable();
}

```

```

public function getUpdatedAt(): ?\DateTimeImmutable
{
    return $this->updatedAt;
}

public function setUpdatedAt(\DateTimeImmutable $updatedAt): self
{
    $this->updatedAt = $updatedAt;

    return $this;
}

```

Enfin, nous allons modifier notre UserController :

```

#[Route('/utilisateur/edition-mot-de-passe/{id}', name: 'user.edit.password', methods:[GET,POST])
public function editPassword(User $user, Request $request, UserPasswordEncoderInterface $hasher, EntityManagerInterface $manager) : Response
{
    $form = $this->createForm(UserPasswordType::class);

    $form->handleRequest($request);
    if($form->isSubmitted() && $form->isValid())
    {
        if($hasher->isPasswordValid($user,$form->getData()['plainPassword']))
        {
            $user->setUpdatedAt(new DateTimeImmutable());
            $user->setPlainPassword(
                $form->getData()['newPassword']
            );

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

            $this->addFlash(
                'success',
                'Votre mot de passe a été modifié avec succès !'
            );
            return $this->redirectToRoute('recipe.index');
        }
        else {
            $this->addFlash(
                'warning',
                'Le mot de passe renseigné est incorrect'
            );
        }
    }

    return $this->render('user/edit_password.html.twig', [
        'form' => $form->createView()
    ]);
}

```

MISE EN RELATION ENTRE UNE ENTITÉ USER ET UNE ENTITÉ OBJETS

Dans cette partie, nous allons voir comment rattacher des objets (recettes de cuisine, ingrédients...) à un utilisateur.

On va commencer par mettre en place cette relation entre l'entité User et l'entité concernée (ici, ce sera l'entité Ingrédient .

```

maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipes$ php bin/console make:entity User
Your entity already exists! So let's add some new fields!
New property name (press <return> to stop adding fields):
> ingredients

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

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

What type of relationship is this?
-----
Type      Description
-----
ManyToOne  Each User relates to (has) one Ingredient.
           Each Ingredient can relate to (can have) many User objects.

OneToMany  Each User can relate to (can have) many Ingredient objects.
           Each Ingredient relates to (has) one User.

ManyToMany Each User can relate to (can have) many Ingredient objects.
           Each Ingredient can also relate to (can also have) many User objects.

OneToOne   Each User relates to (has) exactly one Ingredient.
           Each Ingredient also relates to (has) exactly one User.
-----
Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]:
> OneToMany

A new property will also be added to the Ingredient class so that you can access and set the related User object from it.

New field name inside Ingredient [user]:
> yes

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

Do you want to activate orphanRemoval on your relationship?
A Ingredient is "orphaned" when it is removed from its related User.
e.g. $user->removeIngredient($ingredient)

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

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

updated: src/Entity/User.php
updated: src/Entity/Ingredient.php

```

On va modifier les fixtures dans notre AppFixtures :

```

// Users
$users = [];
for ($k=0; $k < 10 ; $k++) {
    $user = new User();
    $user
        -> setFullName ('FullName' . $k)
        -> setPseudo ('Pseudo' . $k)
        -> setEmail ('Email'.'@' . $k)
        -> setRoles(['ROLE_USER'])
        -> setPlainPassword('password');

    $users[] = $user; // on range les utilisateurs dans le tableau avant qu'ils ne soient persistés
    $manager -> persist($user);
}

// Ingrédients

$ingredients = [];
for ($i=1; $i <= 50 ; $i++) {
    $ingredient = new Ingredient();
    $ingredient
        -> setName ('Ingredient' . $i)
        -> setPrice (mt_rand(0, 100))
        -> setUser ($users[mt_rand(0, count($users) -1)]);

    $ingredients[] = $ingredient;
    $manager -> persist($ingredient);
}

```

On va pouvoir effectuer la migration :

php bin/console make:migration
php bin/console d:m:m

On va modifier deux fonctions dans notre Controller, en utilisant findBy plutôt que findAll :

```

#[Route('/ingredient', name: 'app_ingredient', methods:['GET'])]
public function index(IngredientRepository $repository, PaginatorInterface $paginator, Request $request): Response
{
    $ingredients = $paginator->paginate(
        $repository -> findBy(['user' => $this->getUser()]),
        $request->query->getInt('page', 1),
        10
    );
    return $this->render('ingredient/index.html.twig', [
        'ingredients' => $ingredients
    ]);
}

```

et également la requête liée à l'enregistrement de l'objet (ingrédient) en BDD pour le lier au User :

```
#Route('/ingredient/new', name: 'ingredient.new', methods: ['GET', 'POST'])
public function new(Request $request, EntityManagerInterface $manager, Security $security): Response
{
    $ingredient = new Ingredient();
    $form = $this->createForm(IngredientType::class, $ingredient);

    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $user = $security->getUser();
        $ingredient->setUser($user);

        $manager->persist($ingredient);
        $manager->flush();

        $this->addFlash(
            'success',
            'Votre ingrédient a été créé avec succès !'
        );

        return $this->redirectToRoute('app_ingredient');
    } else {
        $this->addFlash(
            'warning',
            'Il y a des erreurs, votre ingrédient n\'a pas pu être créé !'
        );
    }

    return $this->render('ingredient/new.html.twig', [
        'form' => $form->createView()
    ]);
}
```

Dans l' entité de l'objet à lier à l'utilisateur (ici, ingrédient), on va ajouter les lignes suivantes :

```
#ORM\ManyToOne(inversedBy: 'ingredients')
#[ORM\JoinColumn(name: 'user_id', referencedColumnName: 'id', nullable: false)]
private ?User $user = null;

public function getUser(): ?User
{
    return $this->user;
}

public function setUser(?User $user): self
{
    $this->user = $user;
    return $this;
}
```

Dans l' entité User, on va ajouter ceci :

```
#ORM\OneToMany(mappedBy: 'yes', targetEntity: Ingredient::class, orphanRemoval: true)
private Collection $ingredients;

public function __construct()
{
    $this->createdAt = new DateTimeImmutable();
    $this->updatedAt = new DateTimeImmutable();
    $this->ingredients = new ArrayCollection();
}

/**
 * @return Collection<int, Ingredient>
 */
public function getIngredients(): Collection
{
    return $this->ingredients;
}

public function addIngredient(Ingredient $ingredient): self
{
    if (!$this->ingredients->contains($ingredient)) {
        $this->ingredients->add($ingredient);
        $ingredient->setUser($this);
    }

    return $this;
}

public function removeIngredient(Ingredient $ingredient): self
{
    if ($this->ingredients->removeElement($ingredient)) {
        // set the owning side to null (unless already changed)
        if ($ingredient->getUser() === $this) {
            $ingredient->setUser(null);
        }
    }

    return $this;
}
```

Deuxième exemple :

Ici, on veux créer des recettes avec les ingrédients qui sont uniquement présents dans le compte de l' utilisateur, et donc lier ces recettes à ce compte ;

On va donc créer la relation entre User et Recipe :

```
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console make:entity
Class name of the entity to create or update (e.g. OrangeChef):
> User
Your entity already exists! So let's add some new fields!
New property name (press <return> to stop adding fields):
> recipes
Field type (enter ? to see all types) [string]:
> OneToMany
What class should this entity be related to:
> Recipe
A new property will also be added to the Recipe class so that you can access and set the related User object from it.
New field name inside Recipe [user]:
>
Is the Recipe.user property allowed to be null (nullable)? (yes/no) [yes]:
> no
Do you want to activate orphanRemoval on your relationship?
A Recipe is "orphaned" when it is removed from its related User.
e.g. $user->removeRecipe($recipe)
NOTE: If a Recipe may *change* from one User to another, answer "no".
Do you want to automatically delete orphaned App\Entity\Recipe objects (orphanRemoval)? (yes/no) [no]:
> yes
updated: src/Entity/User.php
updated: src/Entity/Recipe.php
Add another property? Enter the property name (or press <return> to stop adding fields):
>

Success!
```

Next: When you're ready, create a migration with `php bin/console make:migration`

Pour rappel, on a effectué une table de jointure, appelée `recipe_ingredient` dans ma BDD, qui reprend les id des recettes et les id des ingrédients.

Dans l' Entité User, on a donc eu ce code qui s'est ajouté :

```
#[ORM\OneToMany(mappedBy: 'user', targetEntity: Recipe::class, orphanRemoval: true)]
private Collection $recipes;

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

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

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

    return $this;
}

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

    return $this;
}
```

Et dans l' Entité Recipe, on a eu ce code qui s'est ajouté :

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

qu'il faudra remplacer par :

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

On va ajouter une colonne user_id dans notre BDD , dans la table recipe avec les FK.

Normalement, on devrait pouvoir créer cette clé étrangère grâce à la migration, mais si ce n'est pas le cas, il faut récupérer le fichier de migration, noter le code de la clé étrangère et la créer manuellement dans la BDD :

The screenshot shows two windows from MySQL Workbench. On the left, the 'Indexes' tab of the 'Edit Table' dialog for the 'recipe' table is visible, showing two existing indexes: 'PRIMARY' (BTREE, Yes, id) and 'IDX_DA88B137A76ED395' (BTREE, No, user_id). On the right, the 'Edit Index' dialog is open, showing the configuration for the new index. The 'Index name' field contains 'IDX_DA88B137A76ED395'. The 'Index choice' dropdown is set to 'INDEX'. Under 'Advanced options', there is a 'Column' section with 'user_id [int(11)]' selected. At the bottom, there are 'Go', 'Preview SQL', and 'Cancel' buttons.

Ensuite, on va pouvoir modifier notre controller :

Pour l'affichage uniquement des recettes de l'utilisateur :

```
#[Route('/recipe', name: 'recipe.index', methods:['GET'])]
public function index(RecipeRepository $repository, PaginatorInterface $paginator, Request $request): Response
{
    $recipes = $paginator->paginate(
        $repository -> findBy(['user' => $this->getUser()]),
        $request->query->getInt('page', 1),
        10
    );
    return $this->render('recipe/index.html.twig', [
        'recipes' => $recipes
    ]);
}
```

Et pour l'enregistrement des recettes par utilisateur:

```
#[Route('/recipe/new', name: 'recipe.new', methods:['GET', 'POST'])]
public function new(Request $request, EntityManagerInterface $manager) : Response
{
    $recipe = new Recipe();
    $form = $this->createForm(RecipeType::class, $recipe);

    $form->handleRequest($request); // mise à jour du formulaire avec les données soumises

    if($form->isSubmitted() && $form->isValid()){ // si le formulaire a été soumis et est valide (contraintes)

        $recipe = $form->getData();
        $recipe->setUser($this->getUser());

        $manager -> persist($recipe); // donner l'ordre d'ajouter la donnée dans la BDD
        $manager -> flush(); // pousser la donnée en BDD

        $this->addFlash(
            'success',
            'Votre recette a été créée avec succès !'
        );
        return $this->redirectToRoute('recipe.index'); // Redirection vers la liste globale des ingrédients
    } else {
        $this->addFlash(
            'warning',
            'Il y a des erreurs, votre recette n a pas pu être créée !'
        );
    }

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

Pour afficher uniquement les ingrédients relatifs à l'utilisateur, qui pourront donc être sélectionné pour la création de recettes, nous allons modifier le code dans le RecipeType:

<https://stackoverflow.com/questions/57844345/symfony-4-get-current-user-in-form-type>

1e méthode :

Symfony\Component\Security\Core\Security

```
private $security;

public function __construct(Security $security)
{
    $this->security = $security;
}

->add('ingredients', EntityType::class, [
    'class' => Ingredient::class,
    'query_builder' => function (IngredientRepository $repository) {
        return $repository->createQueryBuilder('i')
            ->where('i.user = :user') // i qui est notre ingrédient, est égal à user
            ->orderBy('i.name', 'ASC')
            ->setParameter('user', $this->security->getUser());
    },
],
```

2e méthode :

Dans Symfony 5, avec utilisation

```
 Symfony\Component\Security\Core\Authentication\Token\Storage\TokenStorageInterface

vous pouvez obtenir l'utilisateur où vous voulez. Injectez ceci dans FormType et utilisez

$user = $this->token->getToken()->getUser();

private $token;
public function __construct(TokenStorageInterface $token)
{
    $this->token = $token;
}
```

AFFICHAGE PERSONNALISÉ : APP.USER

Nous allons voir comment avoir un affichage différent en fonction du user connecté.

https://symfony.com/doc/2.8/templating/app_variable.html

Si l'utilisateur est connecté, app.user représentera l'utilisateur, sinon , s'il est déconnecté, il sera null.

Si l'utilisateur est connecté, il aura accès à ces items :

```
{% if app.user %}
<li class="nav-item">
    <a class="nav-link active" href="{{ path('app_ingredient') }}">Mes ingrédients
    | <span class="visually-hidden">(current)</span>
    </a>
</li>
<li class="nav-item">
    <a class="nav-link active" href="{{ path('recipe.index') }}">Mes recettes
    | <span class="visually-hidden">(current)</span>
    </a>
</li>
{% endif %}
```

On va pouvoir vraiment, à l'aide de conditions, afficher certaines informations si l'utilisateur est connecté ou non, et afficher également son pseudo ou nom/prénom :

```

<div class="d-flex">
  <ul class="navbar-nav me-auto">
    {% if app.user %}
      <li class="nav-item dropdown" style="margin-right:5rem;">
        <a class="nav-link dropdown-toggle" data-bs-toggle="dropdown" href="#" role="button" aria-haspopup="true" aria-expanded="false">{{ app.user.fullName }}</a>
        <div class="dropdown-menu">
          <a class="dropdown-item" href="{{ path('user.edit', {id : app.user.id})}}>Modifier mon profil</a>
          <a class="dropdown-item" href="{{ path('user.edit.password', {id : app.user.id})}}>Modifier mon mot de passe</a>
          <div class="dropdown-divider"></div>
          <a class="dropdown-item" href="{{ path('security.logout')}}>Déconnexion</a>
        </div>
      </li>
    {% else %}
      <li class="nav-item">
        <a class="nav-link active" href="{{ path('security.login')}}> Connexion
          <span class="visually-hidden">(current)</span>
        </a>
      </li>
      <li class="nav-item">
        <a class="nav-link active" href="{{ path('security.registration')}}> Inscription
          <span class="visually-hidden">(current)</span>
        </a>
      </li>
    {% endif %}
  </ul>
</div>

```

RESTRICTION D'ACCÈS AUX PAGES : @ISGRANTED / @SECURITY

<https://symfony.com/doc/current/reference/attributes.html#security>

On a deux annotations à notre disposition :

- **IsGranted** :
Il va vérifier si l'utilisateur possède un certain rôle ou une certaine autorisation. IL est utilisé pour protéger l'accès à une ressource ou une action. Par exemple : pour vérifier si un utilisateur peut accéder à une page admin, on peut utiliser ‘isGranted('ROLE_ADMIN')’.
- **Security** :
C'est une méthode plus globale, qui gère l'ensemble de la sécurité de l'application (authentification, autorisation, gestion des accès, cryptage des mots de passe...)

En général, on va utiliser isGranted pour les autorisation d'un utilisateur dans un contrôleur, service ou template, et security sera plutôt utilisé pour configurer et paramétrier la sécurité de l'application.

@ISGRANTED :

Dans notre Controller, on va simplement ajouter une ligne de code qui va nous permettre de restreindre l'accès de telle ou telle page à un utilisateur connecté :

```

#[Route('/ingredient', name: 'app_ingredient', methods:['GET'])]
#[IsGranted('ROLE_USER')]
public function index(IngredientRepository $repository, PaginatorInterface $paginator, Request $request): Response
{
    $ingredients = $paginator->paginate(
        $repository -> findBy(['user' => $this->getUser()]),
        $request->query->getInt('page', 1),
        10
    );
    return $this->render('ingredient/index.html.twig', [
        'ingredients' => $ingredients
    ]);
}

```

Lorsqu'un utilisateur non connecté veut accéder à la page concernée, à savoir ici la page app_ingredient, il sera automatiquement redirigé vers le formulaire de connexion par Symfony (redirection naturelle). Lorsque l'utilisateur se sera connecté après cette redirection naturelle, il sera redirigé vers la page qu'il aura demandé (page app_ingredient)

@SECURITY :

Dans notre cas, on va utiliser cette méthode car nous voulons que ce soit l'utilisateur connecté qui puisse modifier les ingrédients, mais plus précisément que les ingrédients qui lui appartiennent :

```
#[Security("is_granted('ROLE_USER') and user === ingredient.getUser()")]
#[Route('/ingredient/edit/{id}', name: 'ingredient.edit', methods:['GET','POST'])]
```

Effectivement , si l'on souhaite aller sur un ingrédient que nous n'avons pas crée dans notre compte utilisateur, nous aurons donc, grâce à cette méthode, un accès refusé :

Access denied.

On va pouvoir également bloquer le formulaire de modification d'un profil utilisateur au seul utilisateur concerné, dans le User Controller :

```
#[Security("is_granted('ROLE_USER') and user === choosenUser")]
#[Route('/utilisateur/edit/{id}', name: 'user.edit', methods:['GET','POST'])]
public function edit(User $choosenUser, Request $request, EntityManagerInterface $manager, UserPasswordHasherInterface $hasher): Response
{
```

En pensant bien à changer le nom de la variable \$user en \$choosenUser.

On aura bien un accès refusé en cas de tentative de connexion sur un compte utilisateur qui n'est pas le nôtre :

Access denied.

Il faut également penser à faire la même logique sur le formulaire de modification de mot de passe :

```
#[Security("is_granted('ROLE_USER') and user === choosenUser")]
#[Route('/utilisateur/edit-mot-de-passe/{id}', name: 'user.edit.password', methods:['GET','POST'])]
public function editPassword(User $choosenUser, Request $request, UserPasswordHasherInterface $hasher, EntityManagerInterface $manager) : Response
{
```

MODE PRIVE / PUBLIC :

Dans l'exemple, on va mettre en place un champ dans un formulaire, permettant à l'utilisateur de partager ou non sa recette . C'est seulement lorsque la recette sera rendue publique qu'elle pourra être consultée .

Dans un premier temps, on va modifier notre Entité pour lui ajouter une colonne booléenne isPublic :

```
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console make:entity
Class name of the entity to create or update (e.g. DeliciousPopsicle):
> Recipe
Your entity already exists! So let's add some new fields!
New property name (press <return> to stop adding fields):
> isPublic
Field type (enter ? to see all types) [boolean]:
>
Can this field be null in the database (nullable) (yes/no) [no]:
>
updated: src/Entity/Recipe.php
Add another property? Enter the property name (or press <return> to stop adding fields):
>

Success!
```

Next: When you're ready, create a migration with `php bin/console make:migration`

Dans notre entité Recipe, nous aurons bien l'ORM et les getter/setter correspondants:

```
#[ORM\Column]
private ?bool $isPublic = null;
```

```

public function isIsPublic(): ?bool
{
    return $this->isPublic;
}

public function setIsPublic(bool $isPublic): self
{
    $this->isPublic = $isPublic;
    return $this;
}

```

Et on va procéder à la migration :

php bin/console make:migration
php bin/console d:m:m

PAGE SHOW QUI NE CONTIENT QU'UN SEUL ELEMENT PUBLIC

On va créer une nouvelle page permettant d'afficher l'une des recettes rendues publiques.

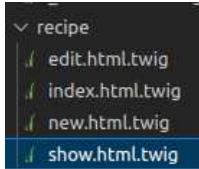
Dans notre Controller des recettes, on va créer une nouvelle fonction publique , en dessous de l'index :

```

#[Route('/recette/{id}', name: 'recipe.show', methods:['GET'])]
public function show(Recipe $recipe) : Response
{
    return $this->render('recipe/show.html.twig', [
        'recipe' => $recipe
    ]);
}

```

Dans notre dossier templates/show, on va créer un fichier show.html.twig



Dans lequel on va intégrer les informations relatives à notre recette sélectionnée :

```

{% block body %}
<div class="container">
{{ dump(recipe) }}
<h1 class="mt-4"> {{ recipe.name}} </h1>

<span class="badge bg-primary"> Crée le {{ recipe.createdAt|date('d/m/Y')}} </span>

<p> Temps (en minutes) : {{ recipe.time}}</p>
<p> Pour {{ recipe.nbPeople}} personne(s)</p>
<p> Difficulté : {{ recipe.difficulty}} /5</p>
<p> Prix : {{ recipe.price}} €</p>
<div> {{ recipe.description|raw}} </div>
</div>
{% endblock %}

```

Si on fait un {{dump()}} , on aura les informations affichées directement sur notre affichage :

```
App\Entity\Recipe {#491 ▾
  -id: 2
  -name: "Poulet au citron"
  -time: 34
  -nbPeople: 6
  -difficulty: 3
  -description: "super recette originale"
  -price: 12
  -isPrivate: true
  -createdAt: DateTimeImmutable @1683201681 {#494 ▶}
  -updatedAt: DateTimeImmutable @1683201681 {#487 ▶}
  -ingredients: Doctrine\PersistentCollection {#504 ▶}
  -user: Proxies\User {#545 ▶}
  -isPublic: null
}
```

Poulet au citron

Créée le 04/05/2023
Temps (en minutes) : 34

Pour 6 personne(s)

Difficulté : 3 / 5

Prix : 12 €

On peut également effectuer des conditions d'affichage :

```
{% if recipe.time %}
  <p>
    Temps (en minutes) : {{ recipe.time}}
  </p>
{% else %}
  <p> Temps non renseigné </p>
{% endif %}

{% if recipe.nbPeople %}
  <p>
    Pour {{ recipe.nbPeople}} personne(s)
  </p>
{% else %}
  <p> Nombre de personnes non renseigné </p>
{% endif %}

{% if recipe.difficulty %}
<p>
  Difficulté : {{ recipe.difficulty}} /5
</p>
{% else %}
  <p> Difficulté non renseignée </p>
{% endif %}

{% if recipe.price %}
  <p>
    Prix : {{ recipe.price}} €
  </p>
{% else %}
  <p> Prix non renseigné </p>
{% endif %}

{% if recipe.description %}
<div>
  {{ recipe.description|raw}}
</div>
{% else %}
  <p> Description non renseignée </p>
{% endif %}
```

Si on veut ajouter uniquement les ingrédients sélectionnés pour cette recette précise, on pourra également saisir le code suivant :

```
{% for ingredient in recipe.ingredients %}
  <span class="badge bg-primary mt-2"> {{ ingredient.name}} </span>
{% endfor %}
```

Maintenant, on va gérer l'accès à cette vue en fonction de son état, dans notre Controller :

```
#|Security("is_granted('ROLE_USER') and recipe.isIsPublic() === true")|
#[Route('/recette/{id}', name: 'recipe.show', methods:['GET'])]
public function show(Recipe $recipe) : Response
{
    return $this->render('recipe/show.html.twig', [
        'recipe' => $recipe
    ]);
}
```

Si la recette est publique, elle va directement s'afficher, sinon, on aura un accès refusé.

PAGE INDEX PUBLIC QUI CONTIENT TOUS LES ÉLÉMENTS PUBLICS

On va créer une nouvelle public function dans notre Controller :

```
#Route('/recette/public', name: 'recipe.index.public', methods:[ 'GET'])
public function indexPublic() : Response
{
    return $this->render('recipe/index_public.html.twig', [
        'recipes' => $recipes
    ]);
}
```

Et dans notre templates/recipe, on va créer une page index_public.html.twig, dans lequel on va reprendre les informations issues de index.html.twig :

```
└── recipe
    ├── edit.html.twig
    ├── index_public.html.twig
    ├── index.html.twig
    ├── new.html.twig
    └── show.html.twig
```

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

{% block title %}SymRecipe - Idées de recettes{% endblock %}

{% block body %}

<div class="container mt-4">
    <h1 style="text-align: center; font-size:4rem"> Recettes de la communauté</h1>
    <br>
    <br>
    {% if not recipes.items is same as([]) %}
        <div class="row card-deck">
            {% for recipe in recipes %}
                <div class="col-md-4 mb-3">
                    <div class="card text-white bg-primary mb-3">
                        <div class="card-header">Recette n°{{ recipe.id }}</div>
                        <div class="card-body">
                            <h4>{{ recipe.name }}</h4>
                            <p>{{ recipe.description|slice(0,100) ~ '...' }}</p>
                        </div>
                    </div>
                {% endif %}
            </div>
            {% else %}
                <h4> Il n'y a pas de recettes</h4>
            {% endif %}
        </div>
    {% endifblock %}
```

Le filtre que l'on va appliquer à recipe.description, qui est le | , permet de tronquer le texte s'il a plus de 100 caractères, et si c'est le cas, d'afficher ‘...’.

On va ensuite créer une méthode qui va permettre d'extraire les recettes publiques, et également le nombre de recettes . On va pouvoir se servir de cette méthode pour la page affichant l'intégralité des recettes publiques, mais également sur la page d'accueil :

```
public function findPublicRecipe(?int $nbRecipes) : array
{
    $queryBuilder = $this->createQueryBuilder('r')
        ->where('r.isPublic = 1') // tu récupères toutes les recettes publiques
        ->orderBy('r.createdAt', 'DESC');// et tu vas nous les classer de la moins récente à la plus récente

    if($nbRecipes !== 0 || $nbRecipes === null ) {
        $queryBuilder->setMaxResults($nbRecipes); // on met un nombre maximum de résultat
    }

    return $queryBuilder->getQuery() // on récupère la query
        ->getResult(); // on récupère le résultat
}
```

Dans notre Controller, nous allons donc appeler la méthode findPublicRecipe :

```
#Route('/recette/public', name: 'recipe.index.public', methods:[ 'GET'])
public function indexPublic(PaginatorInterface $paginator, RecipeRepository $repository, Request $request) : Response
{
    $recipe = $paginator->paginate(
        $repository->findPublicRecipe(null),
        $request->query->getInt('page', 1),
        10
    );
    return $this->render('recipe/index_public.html.twig', [
        'recipe' => $recipe
    ]);
}
```

Maintenant, on va afficher les trois dernières recettes publiques de la communauté sur la page d'accueil . Dans la page d'accueil home/index.html.twig, on va ajouter ce code :

```
<div class="recipes">
    <h2 class="mt-4">Recettes de la communauté</h2>
    <div class="row card-deck">
        {% for recipe in recipes %}
            <div class="col-md-4 mb-3">
                <div class="card text-white bg-primary mb-3">
                    <div class="card-header style="text-align:center" >Recette n°{{ recipe.id }}</div>
                    <div class="card-body">
                        <h4 class="card-title" style="text-align:center">{{ recipe.name }}</h4>
                        <p class="card-text">{{ recipe.description|slice(0,100) ~ '...' }}</p>
                    </div>
                </div>
            </div>
        {% endfor %}
    </div>
```

Et on va modifier le code dans le HomeController :

```
class HomeController extends AbstractController // va nous permettre d'avoir accès à render, qui renvoie vers un template twig
{
    #[Route('/', name: 'app_home')]
    public function index(RecipeRepository $recipeRepository): Response
    {
        return $this->render('home/index.html.twig', [
            'recipes' => $recipeRepository->findPublicRecipe(3),
        ]);
    }
}
```

NOTATION:

Chaque recette sera éligible à la notation d'un utilisateur connecté, limitée à une seule notation par utilisateur pour une recette, et l'utilisateur qui aura publiée sa recette ne pourra pas la noter.
On va pouvoir créer une nouvelle entité Mark :

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

```
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console make:entity Mark
created: src/Entity/Mark.php
created: src/Repository/MarkRepository.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):
> mark

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

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

updated: src/Entity/Mark.php

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

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

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

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

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

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

New field name inside User [marks]:
>

Do you want to activate orphanRemoval on your relationship?
A Mark is "orphaned" when it is removed from its related User.
e.g. $user->removeMark($mark)

NOTE: If a Mark may *change* from one User to another, answer "no".
Do you want to automatically delete orphaned App\Entity\Mark objects (orphanRemoval)? (yes/no) [no]:
> yes

updated: src/Entity/Mark.php
updated: src/Entity/User.php

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

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

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

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

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

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

New field name inside Recipe [marks]:
>

Do you want to activate orphanRemoval on your relationship?
A Mark is "orphaned" when it is removed from its related Recipe.
e.g. $recipe->removeMark($mark)

NOTE: If a Mark may *change* from one Recipe to another, answer "no".
Do you want to automatically delete orphaned App\Entity\Mark objects (orphanRemoval)? (yes/no) [no]:
> yes

updated: src/Entity/Mark.php
updated: src/Entity/Recipe.php

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

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

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

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

Dans notre Entité, on ajoute constructeur, l'entité unique et les asserts :

```
public function __construct()
{
    $this->createdAt = new DateTimeImmutable();
}

#[ORM\UniqueEntity(fields: ['user', 'recipe'],
errorPath: 'user',
message: 'Cet utilisateur a déjà noté cette recette')]
use Symfony\Component\Validator\Constraints as Assert;
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
```

Et on pourra procéder à la migration :

php bin/console make:migration
php bin/console d:m:m

AFFICHAGE DE LA MOYENNE SUR LA VUE DE LA RECETTE

Dans l'entité de la recette, à savoir Recipe, on va créer une nouvelle variable \$average:

```
#[ORM\OneToMany(mappedBy: 'recipe', targetEntity: Mark::class, orphanRemoval: true)]
private Collection $marks;

private ?float $average = null;
```

et on va faire une fonction pour la get :

```
// fonction qui va nous permettre de calculer la moyenne
public function getAverage()
{
    $marks = $this->marks;

    if($marks->toArray() === []) {
        $this->average = null;
        return $this->average;
    }

    $total = 0;

    foreach ($marks as $mark){
        $total += $mark->getMark();
    }

    $this->average = $total / count($marks);
    return $this->average;
}
```

et on va , sur le show du template recipe, et on ajouter la ligne suivante qui va afficher la note moyenne de la recette :

```
<h1 class="mt-4"> {{ recipe.name }} </h1>

<p> La moyenne de cette recette est de {{ recipe.average|number_format(2,'.',',')}} / 5 </p>
```

CRÉATION DU SYSTÈME DE NOTATION

On va commencer par créer un formulaire via le terminal :

```
mæva@mæva-ThinkPad-T460:~/devilbox/data/www/symrecipes$ php bin/console make:form
The name of the form class (e.g. DeliciousKangarooType):
> MarkType
The name of Entity or fully qualified model class name that the new form will be bound to (empty for none):
> Mark
created: src/Form/MarkType.php

Success!

Next: Add fields to your form and start using it.
Find the documentation at https://symfony.com/doc/current/forms.html
```

Dans notre MarkType, on va modifier la variable \$builder :

```
$builder
->add('mark', ChoiceType::class, [
    'choices' => [
        '1' => 1,
        '2' => 2,
        '3' => 3,
        '4' => 4,
        '5' => 5,
    ],
    'attr' => [
        'class' => 'form-select'
    ],
    'label' => 'Noter la recette',
    'label_attr' => [
        'class' => 'form-label mt-4'
    ]
])
-> add('submit', SubmitType::class, [
    'attr' => [
        'class' => 'btn btn-primary mt-4'
    ],
    'label' => 'Noter la recette'
]);
;
```

On va ensuite modifier la fonction show de notre Controller :

```
#[Security("is_granted('ROLE_USER') and recipe.isIsPublic() === true || user === recipe.getUser()")]
// on peut voir la recette si on est connecté, si la recette est publique ou si on est l'utilisateur qui l'a enregistrée
#[Route('/recette/{id}', name: 'recipe.show', methods:[ 'GET', 'POST'])]

public function show(Recipe $recipe, Request $request, MarkRepository $markRepository, EntityManagerInterface $manager) : Response
{
    $mark = new Mark();
    $form = $this->createForm(ChoiceType::class, $mark);

    $form->handleRequest($request);

    if($form->isSubmitted() && $form->isValid()) {
        if($form->isSubmitted() && $form->isValid()) {
            $mark->setUser($this->getUser())
                ->setRecipe($recipe);
        }

        // cela permet de vérifier si l'utilisateur a déjà noté la recette
        $existingMark = $markRepository->findOneBy([
            'user' => $this->getUser(),
            'recipe' => $recipe
        ]);

        if(!$existingMark) {
            $manager->persist($mark);
        } else {
            $existingMark ->setMark(
                $form->getData()->getMark()
            );
        }

        $manager->flush();
        $this->addFlash(
            'success',
            'Votre note a bien été prise en compte !'
        );
    }

    return $this->redirectToRoute('recipe.show', ['id'=> $recipe-> getId()]);
}

return $this->render('recipe/show.html.twig', [
    'recipe' => $recipe,
    'form' => $form->createView()
]);
```

Et dans notre show.html.twig, on va ajouter le code suivant pour simplement afficher le formulaire :

```
{% for ingredient in recipe.ingredients %}
    <span class="badge bg-primary mt-2"> {{ ingredient.name }} </span>
{% endfor %}

<div class="mark">
    {{ form(form)}}
</div>
```

en oubliant pas d'ajouter le code pour le message flash de succès :

```
{% for flash_message in app.session.flashbag.get('success') %}  
<div class="alert alert-success mt-4">{{ flash_message }}</div>  
{% endfor %}
```

UPLOAD UNE IMAGE

Pour cela, on va se servir d'un nouveau bundle : VichUploaderBundle

<https://github.com/dustin10/VichUploaderBundle/blob/master/docs/installation.md>

1/ configuration du mapping de téléchargement

<https://github.com/dustin10/VichUploaderBundle/blob/master/docs/usage.md>

On va l'installer via cette commande dans le terminal :

```
composer require vich/uploader-bundle
```

```
# maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipes$ composer require vich/uploader-bundle  
Info from https://repo.packagist.org: #StandWithUkraine  
Using version "2.1" for vich/uploader-bundle  
.composer.json has been updated  
Running composer update vich/uploader-bundle  
Loading composer repositories with package information  
Updating dependencies  
Lock file operations: 2 installs, 0 updates, 0 removals  
  - Locking jms/metadata (2.8.0)  
  - Locking vich/uploader-bundle (2.1.1)  
Writing lock file  
Installing dependencies from lock file (including require-dev)  
Package operations: 2 installs, 0 updates, 0 removals  
  - Downloading jms/metadata (2.8.0)  
  - Downloading vich/uploader-bundle (2.1.1)  
  - Installing jms/metadata (2.8.0): Extracting archive  
  - Installing vich/uploader-bundle (2.1.1): Extracting archive  
Package sensio/framework-extra-bundle is abandoned, you should avoid using it. Use Symfony instead.  
Generating optimized autoload files  
108 packages you are using are looking for funding.  
Use the 'composer fund' command to find out more!  
  
Symfony operations: 1 recipe (c079f5b0a6f243ce924e0075569d29a8)  
  - WARNING vich/uploader-bundle (>=1.13): From github.com/symfony/recipes-contrib:main  
    The recipe for this package comes from the "contrib" repository, which is open to community contributions.  
    Review the recipe at https://github.com/symfony/recipes-contrib/tree/main/vich/uploader-bundle/1.13  
  
Do you want to execute this recipe?  
[y] Yes  
[n] No  
[a] Yes for all packages, only for the current installation session  
[p] Yes permanently, never ask again for this project  
(defaults to n): yes  
  - Configuring vich/uploader-bundle (>=1.13): From github.com/symfony/recipes-contrib:main  
Executing script cache:clear [OK]  
Executing script assets:install public [OK]  
  
What's next?
```

Some files have been created and/or updated to configure your new packages.
Please review, edit and commit them: these files are yours.

On aura un nouveau fichier d'installé dans notre vendor :

```
✓ vich/uploader-bundle  
> config  
> src  
> templates  
> translations
```

On va également retrouver un fichier yaml dans le dossier packages :

```
✓ packages  
  ! cache.yaml  
  ! debug.yaml  
  ! doctrine_migrations.yaml  
  ! doctrine.yaml  
  ! framework.yaml  
  ! knp_paginator.yaml  
  ! mailer.yaml  
  ! messenger.yaml  
  ! monolog.yaml  
  ! notifier.yaml  
  ! routing.yaml  
  ! security.yaml  
  ! sensio_framework_extr...  
  ! translation.yaml  
  ! twig.yaml  
  ! validator.yaml  
  ! vich_uploader.yaml  
  ! web_profiler.yaml
```

Dans ce fichier, on va voir notamment la notion de mappings, c'est ce qui va indiquer au bundle où le fichier doit être téléchargé (chemin de l'image) . On va procéder à quelques modifications dans ce fichier yaml, comme le remplacement de products et l'ajout du metadata :

```
vich_uploader:
    db_driver: orm
    metadata:
        type: attribute

    mappings:
        recette_images:
            uri_prefix: /images/recette
            upload_destination: '%kernel.project_dir%/public/images/recette'
            namer: Vich\UploaderBundle\Naming\SmartUniqueNamer
```

2/ lier le mapping aux entités

On va donc aller dans l'entité qui va nécessiter l'import de photos, à savoir dans notre cas l'entité Recipe , et on va ajouter le code suivant :

```
#[Vich\Uploadable]

#[ORM\Entity(repositoryClass: RecipeRepository::class)]
#[UniqueEntity('name')]
#[ORM\HasLifecycleCallbacks]
#[Vich\Uploadable]
```

Et on va ajouter les Uses suivants :

```
use Symfony\Component\HttpFoundation\File\File;
use Vich\UploaderBundle\Mapping\Annotation as Vich;
```

Et nous allons ajouter les imageFile et imageName que l'on mettra en dessous de la variable name:

```
#[ORM\Column(length: 50)]
#[Assert\NotBlank()]
#[Assert\Length(min :2, max:50)]
private ?string $name = null;

#[Vich\UploadableField(mapping: 'recette_images', fileNameProperty: 'imageName')]
private ?File $imageFile = null;

#[ORM\Column(nullable: true)]
private ?string $imageName = null;
```

Le imageFile va définir l'endroit de stockage où la photo va être contenue. Il faut bien penser à renommer le mapping de la manière dont il a été défini dans le fichier yaml.

Le imageName va définir le stockage du nom de l'image en BDD.

On va ensuite ajouter les getter/setter, que l'on ajoutera en dessous de ceux de name :

```
/*
 * If manually uploading a file (i.e. not using Symfony Form) ensure an instance
 * of 'UploadedFile' is injected into this setter to trigger the update. If this
 * bundle's configuration parameter 'inject_on_load' is set to 'true' this setter
 * must be able to accept an instance of 'File' as the bundle will inject one here
 * during Doctrine hydration.
 *
 * @param File|\Symfony\Component\HttpFoundation\File\UploadedFile|null $imageFile
 */
public function setImageFile(?File $imageFile = null): void
{
    $this->imageFile = $imageFile;

    if ($null !== $imageFile) {
        // It is required that at least one field changes if you are using doctrine
        // otherwise the event listeners won't be called and the file is lost
        $this->updatedAt = new \DateTimeImmutable();
    }
}

public function getImageFile(): ?File
{
    return $this->imageFile;
}

public function setImageName(?string $imageName): void
{
    $this->imageName = $imageName;
}

public function getImageName(): ?string
{
    return $this->imageName;
}
```

On va ensuite ajouter dans notre fichier Type le add correspondant à notre imageFile:

```
->add('imageFile', VichImageType::class, [
    'label' => 'Image de la recette',
    'label_attr' => [
        'class' => 'form-label mt-4'
    ],
])
```

Dans le fichier new présents dans le template concerné (dans notre exemple, recipe), on va ajouter le code suivant , à l'endroit du formulaire où l'on souhaite que l'image soit demandée :

```
 {{form_row(form.imageFile)}}
```

On procède bien à la migration afin d'envoyer nos modifications à la BDD :

php bin/console make:migration
php bin/console d:m:m

Ce qui va créer dans la BDD la ligne suivante dans notre colonne :

<input type="checkbox"/> 13 image_name varchar(255) utf8mb4_unicode_ci	Yes	NULL	Change	Drop	More
------------------------------------------------------------------------	-----	------	--------	------	------

Si l'on teste le formulaire, cela va bien enregistrer la photo dans notre BDD.

Maintenant, on va voir pour afficher l'image dans notre vue.

Dans le fichier show, on va ajouter le code suivant :

```
<div class="recipe_image">
    
</div>
```

CRÉER UN FORMULAIRE DE CONTACT

On va commencer par créer une nouvelle entité Contact :

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

```
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console make:entity Contact
created: src/Entity/Contact.php
created: src/Repository/ContactRepository.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):
> fullName

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

Field length [255]:
> 50

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

updated: src/Entity/Contact.php

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

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

Field length [255]:
> 180

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

updated: src/Entity/Contact.php

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

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

Field length [255]:
> 100

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

updated: src/Entity/Contact.php

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

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

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

updated: src/Entity/Contact.php

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

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

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

updated: src/Entity/Contact.php

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

Success!
```

Next: When you're ready, create a migration with `php bin/console make:migration`

On va ensuite dans le fichier Contact dans le dossier Entity, afin d'ajouter les contraintes :

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

```

#[ORM\Entity(repositoryClass: ContactRepository::class)]
class Contact
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    #[ORM\Column(length: 50, nullable: true)]
    #[Assert\Length(min :2, max:50)]
    private ?string $fullName = null;

    #[ORM\Column(length: 180)]
    #[Assert\Email()]
    #[Assert\Length(min :2, max:180)]
    private ?string $email;

    #[ORM\Column(length: 100, nullable: true)]
    #[Assert\Length(min :2, max:100)]
    private ?string $subject = null;

    #[ORM\Column(type: Types::TEXT)]
    #[Assert\NotBlank()]
    private ?string $message;

    #[ORM\Column]
    #[Assert\NotNull()]
    private ?\DateTimeImmutable $createdAt;

    public function __construct()
    {
        $this -> createdAt = new \DateTimeImmutable();
    }
}

```

On va pouvoir procéder à la migration afin d'exporter notre nouvelle entité vers la BDD :

php bin/console make:migration

php bin/console d:m:m

On va créer notre ContactController :

php bin/console make:controller ContactController

```

maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console make:controller ContactController
created: src/Controller/ContactController.php
created: templates/contact/index.html.twig

Success!

Next: Open your new controller class and add some pages!

```

On aura donc notre Controller de créé, mais également notre template contact.

On va pouvoir créer notre formulaire, dans notre dossier Form, appelé ContactType.

php bin/console make:form ContactType

```

maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console make:form ContactType
The name of Entity or fully qualified model class name that the new form will be bound to (empty for none):
> Contact

created: src/Form/ContactType.php

Success!

Next: Add fields to your form and start using it.
Find the documentation at https://symfony.com/doc/current/forms.html

```

Afin de remplir notre builder avec le style et les contraintes :

```
$builder
->add('fullName', TextType::class, [
    'attr'=> [
        'class' => 'form-control',
        'minlength' => '2',
        'maxlength'=> '50',
    ],
    'label' => 'Nom / Prénom',
    'label_attr' => [
        'class' => 'form-label mt-4'
    ],
    'constraints' => [
        new Assert\Length(['min'=> 2, 'max'=> 50])
    ]
])

->add('email', EmailType::class, [
    'attr' => [
        'class' => 'form-control',
        'minlength' => '2',
        'maxlength'=> '180',
    ],
    'label' => 'Adresse Email',
    'label_attr' => [
        'class' => 'form-label mt-4'
    ],
    'constraints' => [
        new Assert\NotBlank(),
        new Assert\Email(),
        new Assert\Length(['min'=> 2, 'max'=> 180])
    ]
])

->add('subject', TextType::class, [
    'attr'=> [
        'class' => 'form-control',
        'minlength' => '2',
        'maxlength'=> '50',
    ],
    'label' => 'Sujet',
    'label_attr' => [
        'class' => 'form-label mt-4'
    ],
    'constraints' => [
    ]
])

->add('message', TextareaType::class, [
    'attr' => [
        'class' => 'form-control',
    ],
    'label' => 'Message',
    'label_attr' => [
        'class' => 'form-label mt-4'
    ],
    'constraints' => [
        new Assert\NotBlank(),
    ]
])

->add('submit', SubmitType::class, [
    'attr' => [
        'class' => 'btn btn-primary mt-4'
    ],
    'label' => 'Enregistrer'
])
;
```

On va ensuite modifier le Controller :

```
#Route('/contact', name: 'contact.index')
public function index(Request $request, EntityManagerInterface $manager ): Response
{
    $contact = new Contact();

    // pré-remplir les champs si l'utilisateur est connecté
    if($this->getUser()) {
        $contact -> setFullName($this->getUser()->getFullName())
        -> setEmail($this->getUser()->getEmail());
    }

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

    $form->handleRequest($request);

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

        if($form->isSubmitted() && $form->isValid()) {
            $contact = $form -> getData();

            $manager -> persist($contact);
            $manager ->flush();
            $this->addFlash(
                'success',
                'Votre message a bien été envoyé!'
            );
        }

        return $this->redirectToRoute('contact.index');
    }
}

return $this->render('contact/index.html.twig', [
    'form' => $form ->createView(),
]);
```

Enfin, nous allons modifier notre template, en supprimant ce qui est indiqué par défaut , et en ajoutant ce code, afin d'importer notre formulaire issu de notre ContactType :

```
{% block body %}

<div class="container mt-4">
    {% for flash_message in app.session.flashbag.get('success') %}
        <div class="alert alert-success mt-4">{{ flash_message }}</div>
    {% endfor %}

    <h1 style="text-align: center;"> Formulaire de contact</h1>
    {{ form(form)}}
</div>

{% endblock %}
```

ENVOYER UN MAIL VIA UN FORMULAIRE DE CONTACT

On va se servir du composant Mailer.

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

Dans notre fichier Composer.json , on voit qu'il est déjà installé :

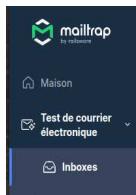
```
"symfony/mailer": "6.1.*",
```

On a également un fichier mailer.yaml dans le dossier packages :

```
! mailer.yaml
```

Afin de transporter le message par email, on va utiliser mailtrap . C'est un service qui offre un faux serveur SMTP pour tester les envois d'emails pendant la phase de développement de notre projet.
<https://mailtrap.io/>

Il suffit de créer un compte, puis d'aller dans Inboxes :



Là, on peut choisir l'intégration, pour nous ce sera Symfony 5+ :

A screenshot of the Mailtrap configuration page. It shows various tabs like 'Paramètres SMTP', 'Adresse e-mail', 'Transfert automatique', 'Renvoi manuel', and 'Des droits d'accès'. Under the 'SMTP/POP3' tab, there's a note about using parameters to send messages directly from a client or agent. A warning about revealing user names and passwords is present. Below, there's a section for 'Intégrations' with a dropdown menu set to 'Symfony 5+'.

Ce qui va nous afficher en dessous une ligne MAILER_DSN , que nous allons copier et intégrer à notre fichier .env.dev.local

```
1 DATABASE_URL="mysql://root:toor@127.0.0.1:3306/symrecipe?serverVersion=5.7"
2 MAILER_DSN=smtp://44feeb150d1935:322dc374ce5947@smtp.mailtrap.io:2525?encryption=tls&auth_mode=login
```

Dans le fichier messenger.yaml, il faut également commenter la ligne suivante :

```
routing:
    # Symfony\Component\Mailer\Messenger\SendEmailMessage: async
    # Symfony\Component\Notifier\Message\ChatMessage: async
    # Symfony\Component\Notifier\Message\SmsMessage: async
```

Ensuite, dans notre ContactController, on va ajouter le code suivant dans notre fonction index :
<https://symfony.com/doc/current/mailer.html#creating-sending-messages>

```
public function index(Request $request, EntityManagerInterface $manager, MailerInterface $mailer): Response
{
    $contact = new Contact();

    // pré-remplir les champs si l'utilisateur est connecté
    if($this->getUser()) {
        $contact -> setFullName($this->getUser()->getFullName())
        -> setEmail($this->getUser()->getEmail());
    }

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

    $form->handleRequest($request);

    if($form->isSubmitted() && $form->isValid()) {
        if($form->isSubmitted() && $form->isValid()) {
            $contact = $form -> getData();

            $manager -> persist($contact);
            $manager ->flush();

            // Email
            $email = [new Email()
                ->from($contact->getEmail())
                ->to('admin@symrecipe.com')
                ->subject($contact->getSubject())
                ->html($contact->getMessage());
            $mailer->send($email);

            $this->addFlash(
                'success',
                'Votre message a bien été envoyé!'
            );
        }
        return $this->redirectToRoute('contact.index');
    }
}

return $this->render('contact/index.html.twig', [
    'form' => $form -> createView(),
]);
```

On ajoute bien les Users suivants :

```
use Symfony\Component\Mailer\MailerInterface;
use Symfony\Component\Mime\Email;
```

Et on constate que lorsque l'on envoie un formulaire de contact via notre site, cela s'enregistre bien dans la BDD, mais également dans MailTrap :



On va pouvoir personnaliser cette réception de messages à l'aide de templates Twig :

<https://symfony.com/doc/current/mailer.html#html-content>

```
$mailer->flush();

// Email
$email = (new TemplatedEmail())
    ->from($contact->getEmail())
    ->to('admin@symrecipe.com')
    ->subject($contact->getSubject())
    ->htmlTemplate('emails/contact.html.twig')
    ->context([
        'expiration_date' => new \DateTime('+7 days'),
        'username' => 'foo',
    ]);
$mailer->send($email);
```

Dans le dossier templates, on va créer un nouveau fichier emails, contenant un sous-fichier contact.html.twig, dans lequel on va ajouter le code suivant :

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
    </head>

    <body>
        <h1> Demande de {{ contact.fullName }}</h1>

        <p>Email : {{ contact.email }}</p>
        <p>Sujet : {{ contact.subject }}</p>
        <p>Message :</p>
        <div>
            {{ contact.message|raw }}
        </div>
    </body>
</html>
```

Ce qui va nous permettre de personnaliser notre message dans MailTrap

MISE EN PLACE DU RECAPTCHA V3

<https://developers.google.com/recaptcha/docs/v3?hl=fr>

On va utiliser le bundle Karser :

<https://github.com/karser/KarserRecaptcha3Bundle>

On va commencer par saisir cette ligne de commande dans notre terminal :

```
composer require karser/karser-recaptcha3-
bundle
```

```
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ composer require karser/karser-recaptcha3-bundle
Info from https://repo.packagist.org: #StandWithUkraine
Using version ^0.1.23 for karser/karser-recaptcha3-bundle
./composer.json has been updated
Running composer update karser/karser-recaptcha3-bundle
Loading composer repositories with package information
Updating dependencies
Lock file operations: 2 installs, 0 updates, 0 removals
- Locking google/recaptcha (1.3.0)
- Locking karser/karser-recaptcha3-bundle (v0.1.23)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 2 installs, 0 updates, 0 removals
- Downloading google/recaptcha (1.3.0)
- Downloading karser/karser-recaptcha3-bundle (v0.1.23)
- Installing google/recaptcha (1.3.0): Extracting archive
- Installing karser/karser-recaptcha3-bundle (v0.1.23): Extracting archive
Package sensio/framework-extra-bundle is abandoned, you should avoid using it. Use Symfony instead.
Generating optimized autoload files
108 packages you are using are looking for funding.
Use the `composer fund` command to find out more!

Symfony operations: 2 recipes (3dc7ac53614911607bb31ba608e943)
- [WARNING] google/recaptcha (>=1.1): From github.com/symfony/recipes-contrib:main
  The recipe for this package comes from the "contrib" repository, which is open to community contributions.
  Review the recipe at https://github.com/symfony/recipes-contrib/tree/main/google/recaptcha/1.1

Do you want to execute this recipe?
[y] Yes
[n] No
[a] Yes for all packages, only for the current installation session
[p] Yes permanently, never ask again for this project
(defaults to n): no
- [WARNING] karser/karser-recaptcha3-bundle (>=0.1): From github.com/symfony/recipes-contrib:main
  The recipe for this package comes from the "contrib" repository, which is open to community contributions.
  Review the recipe at https://github.com/symfony/recipes-contrib/tree/main/karser/karser-recaptcha3-bundle/0.1

Do you want to execute this recipe?
[y] Yes
[n] No
[a] Yes for all packages, only for the current installation session
[p] Yes permanently, never ask again for this project
(defaults to n): yes
- Configuring karser/karser-recaptcha3-bundle (>=0.1): From github.com/symfony/recipes-contrib:main
Executing script cache:clear [OK]
Executing script assets:install public [OK]

What's next?
```

Some files have been created and/or updated to configure your new packages.
Please review, edit and commit them: these files are yours.

karser/karser-recaptcha3-bundle instructions:

- * Make sure you have proper values in your .env file
- * Read the documentation at <https://github.com/karser/KarserRecaptcha3Bundle>

On aura un fichier yaml dans packages :



Il va ensuite falloir créer des clefs :

The screenshot shows the 'Create API key' form on the Google Cloud Platform. The 'Label' field is set to 'symrecipe'. Under 'Type de reCAPTCHA', the radio button for 'Sur la base d'un score (v3)' is selected. In the 'Domaines' section, there is a single entry: 'localhost'. At the bottom of the form, there is a checked checkbox that says 'I accept the Conditions of Use of Google Cloud Platform, as well as those of the concerned APIs and services.' Below the checkbox are two buttons: 'ANNULER' and 'ENVOYER'.

<https://www.google.com/recaptcha/admin/create?hl=fr>

On aura deux clefs qui nous seront données, l'une appelée ‘clé du site’ , l'autre ‘clé secrète’ :

Utilisez cette clé de site dans le code HTML de votre site destiné aux utilisateurs. [Voir l'intégration côté client](#)

[COPIER LA CLÉ DU SITE](#)

6LeWxfMIAAAAACH9uaGKKOFnh_QkVVkwMouXjHvl

Utilisez cette clé secrète pour la communication entre votre site et le service reCAPTCHA. [Voir l'intégration côté serveur](#)

[COPIER LA CLÉ SECRÈTE](#)

6LeWxfMIAAAAACDNPlcwOsmtB9G2nWVdFJ_CjXpj

On va copier les deux clefs et les ajouter dans notre fichier .env :

```
###> karser/karser-recaptcha3-bundle ###
# Get your API key and secret from https://g.co/recaptcha/v3
RECAPTCHA3_KEY=6LeWxfMIAAAAACH9uaGKKOFnh_QkVVkwMouXjHvl
RECAPTCHA3_SECRET=6LeWxfMIAAAAACDNPlcwOsmtB9G2nWVdFJ_CjXpj
###< karser/karser-recaptcha3-bundle ###
```

Et enfin, on va ajouter un Captcha au sein de notre formulaire dans ContactType, sous le bouton Submit:

```
->add('captcha', Recaptcha3Type::class, [
    'constraints' => new Recaptcha3(),
    'action_name' => 'contact',
])
```

LES SERVICES

https://symfony.com/doc/current/service_container.html

Ce sont des objets PHP qui vont remplir une fonction particulière (par exemple , dans notre Controller : EntityManager pour l'enregistrement en BDD, Mailer pour la gestion des mails...) ; Tous ces services sont au sein d'une classe PHP appelée Container de service, qui va centraliser tous les services au sein de notre application Symfony.

Pour visualiser tous les services, on peut taper dans notre terminal :

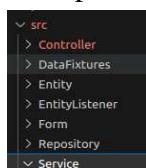
```
php bin/console debug:autowiring --all
```

Si on veut cibler un service en particulier, par exemple pour la gestion d'envoi de mail, on peut taper la commande suivante

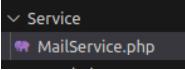
```
php bin/console debug:autowiring mail
```

Mais on peut également créer nos propres services. Par exemple, on va créer un service propre pour l'envoi d'emails.

Dans un premier temps, on va créer un nouveau dossier Service dans le dossier principal SRC :



Et on va créer à l'intérieur de ce dossier un fichier appelé MailService.php



```
class MailService {

    /**
     * @var MailerInterface
     */
    private MailerInterface $mailer;

    public function __construct(MailerInterface $mailer)
    {
        $this->mailer = $mailer;
    }

    public function sendEmail(
        string $from,
        string $subject,
        string $htmlTemplate,
        array $context,
        string $to = 'admin@symrecipe.com'
    ) : void
    {
        // Email
        $email = (new TemplatedEmail())
            ->from($from)
            ->to($to)
            ->subject($subject)
            ->htmlTemplate($htmlTemplate)
            ->context($context);
        $this->mailer->send($email);
    }
}
```

Et au sein du ContactController, on va pouvoir supprimer le MailerInterface et appeler le MailService :

```
public function index(Request $request, EntityManagerInterface $manager, MailService $mailService ): Response
```

et on va remplacer le code précédemment indiqué par celui là (toujours dans notre ContactController) :

```
public function index(Request $request, EntityManagerInterface $manager, MailService $mailService ): Response
{
    $contact = new Contact();

    // pré-remplir les champs si l'utilisateur est connecté
    if($this->getUser()) {
        $contact -> setFullName($this->getUser()->getFullName());
        $contact -> setEmail($this->getUser()->getEmail());
    }

    $form = $this->createForm(ContactType::class, $contact);
    $form->handleRequest($request);

    if($form->isSubmitted() && $form->isValid()) {
        $contact = $form->getData();
        $manager->persist($contact);
        $manager->flush();

        // Envoi de l'email
        $mailService->sendEmail(
            $contact->getEmail(),
            $contact->getSubject(),
            'emails/contact.html.twig',
            ['contact' => $contact]
        );

        $this->addFlash(
            'success',
            'Votre message a bien été envoyé!'
        );
    }

    return $this->redirectToRoute('contact.index');
}

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

ESPACE ADMINISTRATEUR AVEC EASY ADMIN

I / Installation d'EasyAdmin

<https://symfony.com/bundles/EasyAdminBundle/current/index.html>

C'est le bundle recommandé par Symfony pour créer des interfaces admin.

Dans notre terminal, on va saisir la commande suivante :

```
composer require easycorp/easyadmin-bundle
```

```
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ composer require easycorp/easyadmin-bundle
Info from https://repo.packagist.org: #StandWithUkraine
Using version ^4.6 for easycorp/easyadmin-bundle
./composer.json has been updated
Running composer update easycorp/easyadmin-bundle
Loading composer repositories with package information
Restricting packages listed in "symfony/symfony" to "6.1.*"
Updating dependencies
Lock file operations: 3 installs, 0 updates, 0 removals
- Locking easycorp/easyadmin-bundle (v4.6.5)
- Locking symfony/polyfill-uuid (v1.27.0)
- Locking symfony/uuid (v6.1.11)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 3 installs, 0 updates, 0 removals
- Downloading symfony/polyfill-uuid (v1.27.0)
- Downloading symfony/uuid (v6.1.11)
- Downloading easycorp/easyadmin-bundle (v4.6.5)
- Installing symfony/polyfill-uuid (v1.27.0): Extracting archive
- Installing symfony/uuid (v6.1.11): Extracting archive
- Installing easycorp/easyadmin-bundle (v4.6.5): Extracting archive
Package sensio/framework-extra-bundle is abandoned, you should avoid using it. Use Symfony instead.
Generating optimized autoload files
111 packages you are using are looking for funding.
Use the 'composer fund' command to find out more!

Symfony operations: 1 recipe (3dbf2f274bbd161a8da9ed112bc3d3fe)
- Configuring easycorp/easyadmin-bundle (>=3.0): From github.com/symfony/recipes:main
Executing script cache:clear [OK]
Executing script assets:install public [OK]

What's next?
```

Some files have been created and/or updated to configure your new packages.
Please review, edit and commit them: these files are yours.

II/ Crédation d'un Dashboard

C'est le point d'entrée de la partie administrateur, dans lequel on va pouvoir afficher un menu principal (Main Menu), où l'on va pouvoir naviguer entre les entités.

On va donc saisir dans notre terminal la commande suivante :

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

```
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console make:admin:dashboard
Which class name do you prefer for your Dashboard controller? [DashboardController]:
>

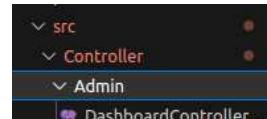
In which directory of your project do you want to generate "DashboardController"? [src/Controller/Admin/]:
>

[OK] Your dashboard class has been successfully generated.

Next steps:
* Configure your Dashboard at "src/Controller/Admin/DashboardController.php"
* Run "make:admin:crud" to generate CRUD controllers and link them from the Dashboard.

maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$
```

On a un nouveau dossier Admin dans notre Controller :



Dans le DashboardController , on va modifier un peu le code qui nous a été fourni par Symfony :

```
class DashboardController extends AbstractDashboardController
{
    #[Route('/admin', name: 'admin')]
    public function index(): Response
    {
        return $this->render('admin/dashboard.html.twig');
```

Attention, ne changer ni la Route, ni le name

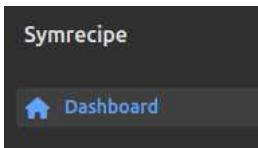
On va ensuite créer notre fichier dashboard.html.twig dans le dossier Templates, sous-dossier admin :

```
└─ templates
    └─ admin
        └─ dashboard.html.twig
```

où je vais simplement indiquer le code suivant :

```
{% extends "@EasyAdmin/page/content.html.twig" %}
```

ce qui va me rediriger vers une autre page :

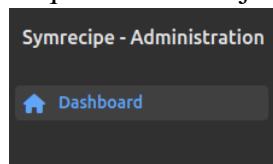


On va ensuite configurer le Dashboard dans notre Controller :

<https://symfony.com/bundles/EasyAdminBundle/current/dashboards.html#dashboard-configuration>

```
public function configureDashboard(): Dashboard
{
    return Dashboard::new()
        ->setTitle('Symrecipe - Administration')
        ->renderContentMaximized();
```

Ce qui va mettre à jour le Dashboard :



Dans mon fichier twig, si je rajoute le code suivant :

```
{% extends "@EasyAdmin/page/content.html.twig" %}

{% block content %}
    <h1> Bienvenue au sein de l'administration de SYMRECIPE</h1>
{% endblock %}
```

Cela va intégrer mon titre h1 :



III/ Création du CRUD Controller

<https://symfony.com/bundles/EasyAdminBundle/current/crud.html>

On commence par saisir la commande suivante dans notre terminal :

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

```
> maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console make:admin:crud
Which Doctrine entity are you going to manage with this CRUD controller?:
[0] App\Entity\Contact
[1] App\Entity\Ingredient
[2] App\Entity\Mark
[3] App\Entity\Recipe
[4] App\Entity\User
[5] Vich\UploaderBundle\Entity\File
> 4

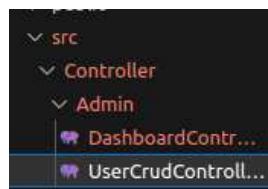
Which directory do you want to generate the CRUD controller in? [src/Controller/Admin/]:
>

Namespace of the generated CRUD controller [App\Controller\Admin]:
>

[OK] Your CRUD controller class has been successfully generated.

Next steps:
* Configure your controller at "src/Controller/Admin/UserCrudController.php"
* Read EasyAdmin docs: https://symfony.com/doc/master/bundles/EasyAdminBundle/index.html
```

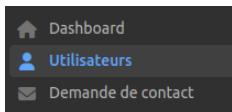
On va donc avoir un nouveau fichier : UserCrudController :



Dans notre DashboardController, on va modifier The Label , et également ajouter des icônes sur fontAwesome : <https://fontawesome.com/icons>

```
public function configureMenuItems(): iterable
{
    yield MenuItem::linkToDashboard('Dashboard', 'fa fa-home');
    yield MenuItem::linkToCrud('Utilisateurs', 'fas fa-user', User::class);
    yield MenuItem::linkToCrud('Demande de contact', 'fas fa-envelope', Contact::class);
}
```

ce qui va changer les icônes dans notre menu :



Puis dans notre UserCrudController, on va ajouter la fonction suivante :

```
class UserCrudController extends AbstractCrudController
{
    public static function getEntityFqn(): string // renseigne vers quelle entité on se base pour créer ce CRUD Controller
    {
        return User::class;
    }

    public function configureCrud(Crud $crud) :Crud
    {
        return $crud
            ->setEntityTypeInPlural('Utilisateurs')
            ->setEntityTypeInSingular('Utilisateur')
            ->setPageTitle('index', 'Symrecipe - Administration des utilisateurs')
            ->setPaginatorPageSize(10);
    }
}
```

Et on va modifier la dernière fonction, le système de champs, qui gère l'affichage des différents champs : <https://symfony.com/bundles/EasyAdminBundle/current/crud.html#fields>

```
public function configureFields(string $pageName): iterable
{
    return [
        IdField::new('id'),
        TextField::new('fullName'),
        TextField::new('pseudo'),
        TextField::new('email'),
        ArrayField::new('roles'),
        DateTimeField ::new('createdAt')
    ];
}
```

Cette fonction va nous permettre notamment de passer de cet affichage (partie edit):

The screenshot shows a dark-themed 'Edit Utilisateur' form. It contains the following fields:

- Full Name***: Ponton Maeva
- Pseudo**: Maeva
- Email***: maeva@gmail.com
- Password***: S2y\$13\$cLBKAkFd7iHUtCDgq/inV.6i3MH7f/lfQb5Fp17XWm9En7qgfId3e
- Created At***: 03/05/2023 13:40:40
- Updated At***: 03/05/2023 13:40:40

où l'on trouve le mot de passe, ce qui n'est pas très sécurisé, à ce visuel :

The screenshot shows a dark-themed 'Edit Utilisateur' form. It contains the following fields:

- ID***: 1
- Full Name***: Ponton Maeva
- Pseudo**: Maeva
- Email***: maeva@gmail.com
- Roles***: ROLE_USER
+ Add a new item
- Created At***: 03/05/2023 13:40:40

On va pouvoir ajouter des sécurités :

```
public function configureFields(string $pageName): iterable
{
    return [
        IdField::new('id')
            ->hideOnForm(), // ID ne sera pas présent sur la page de modification de formulaire
        TextField::new('fullName'),
        TextField::new('pseudo'),
        TextField::new('email')
            ->setFormTypeOption('disabled', 'disabled'), // email va s'afficher mais ne sera pas modifiable
        ArrayField::new('roles'),
        DateTimeField ::new('createdAt')
            ->hideOnForm()
    ];
}
```

ce qui nous donnera l'affichage suivant :

The screenshot shows a dark-themed form titled "Edit Utilisateur". It contains four input fields: "Full Name*" with value "Ponton Maeva", "Pseudo" with value "Maeva", "Email*" with value "maeva@gmail.com", and "Roles*" with value "ROLE_USER". A button "+ Add a new item" is visible next to the roles field.

On va ensuite faire le CRUD des contacts :

```
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console make:admin:crud
Which Doctrine entity are you going to manage with this CRUD controller?:
[0] App\Entity\Contact
[1] App\Entity\Ingredient
[2] App\Entity\Mark
[3] App\Entity\Recipe
[4] App\Entity\User
[5] Vich\UploaderBundle\Entity\File
> 0

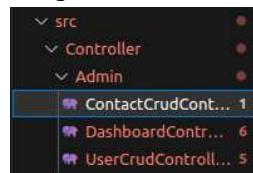
Which directory do you want to generate the CRUD controller in? [src/Controller/Admin/]:
>

Namespace of the generated CRUD controller [App\Controller\Admin]:
>

[OK] Your CRUD controller class has been successfully generated.

Next steps:
* Configure your controller at "src/Controller/Admin/ContactCrudController.php"
* Read EasyAdmin docs: https://symfony.com/doc/master/bundles/EasyAdminBundle/index.html
```

Ce qui va nous créer un fichier ContactCrudController :



Dans lequel on va créer la fonction configureCrud, et modifier la fonction configureFields

```
public function configureCrud(Crud $crud) :Crud
{
    return $crud
        ->setEntityLabelInPlural('Demandes de contact')
        ->setEntityLabelInSingular('Demande de contact')
        ->setPageTitle('index', 'Symrecipe - Administration des demandes de contact')
        ->setPaginatorPageSize(20);
}

public function configureFields(string $pageName): iterable
{
    return [
        IdField::new('id')
            ->hideOnForm(), // ID ne sera pas présent sur la page de modification de formulaire
        TextField::new('fullName'),
        TextField::new('email')
            ->setFormTypeOption('disabled', 'disabled'), // email va s'afficher mais ne sera pas modifiable
        TextareaField::new('message')
            ->hideOnIndex(),
        DateTimeField ::new('createdAt')
            ->hideOnForm()
    ];
}
```

IV/ Mise en place d'un WYSIWYG

C'est une zone de texte où tout ce qui est écrit va s'afficher à l'écran.

On va utiliser le bundle CKEDITOR :

<https://symfony.com/bundles/FOSCKEditorBundle/current/installation.html#download-the-bundle>

Dans notre terminal, on va ajouter la commande suivante :

```
composer require friendsofsymfony/ckeditor-bundle
```

Si lors de l'installation, ce message apparaît dans le terminal :

```
Package friendsofsymfony/ckeditor-bundle has requirements incompatible with your PHP version, PHP extensions and Composer version:
- friendsofsymfony/ckeditor-bundle 2.4.0 requires ext-zip * but it is not present.
```

Il suffit de télécharger l'extention ext-zip , en notant cette commande dans le terminal :

```
sudo apt-get install php-zip
```

Et ensuite, relancer la commande précédente.

```
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ composer require friendsofsymfony/ckeditor-bundle
Using version ^2.4 for friendsofsymfony/ckeditor-bundle
./composer.json has been updated
Running composer update friendsofsymfony/ckeditor-bundle
Loading composer repositories with package information
Updating dependencies
Lock file operations: 1 install, 0 updates, 0 removals
  - Locking friendsofsymfony/ckeditor-bundle (2.4.0)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
  - Downloading friendsofsymfony/ckeditor-bundle (2.4.0)
  - Installing friendsofsymfony/ckeditor-bundle (2.4.0): Extracting archive
Package sensio/framework-extra-bundle is abandoned, you should avoid using it. Use Symfony instead.
Generating optimized autoload files
111 packages you are using are looking for funding.
Use the 'composer fund' command to find out more!

Symfony operations: 1 recipe (6e9029c85148e5d5ddfb4f6d7dc881ca)
  - WARNING: friendsofsymfony/ckeditor-bundle (>=2.0): From github.com/symfony/recipes-contrib:main
    The recipe for this package comes from the "contrib" repository, which is open to community contributions.
    Review the recipe at https://github.com/symfony/recipes-contrib/tree/main/friendsofsymfony/ckeditor-bundle/2.0

Do you want to execute this recipe?
[y] Yes
[n] No
[a] Yes for all packages, only for the current installation session
[p] Yes permanently, never ask again for this project
(defaults to n): yes
  - Configuring friendsofsymfony/ckeditor-bundle (>=2.0): From github.com/symfony/recipes-contrib:main
Executing script cache:clear [OK]
Executing script assets:install public [OK]
Executing script ckeditor:install [OK]

What's next?
```

```
Some files have been created and/or updated to configure your new packages.
Please review, edit and commit them: these files are yours.
```

Puis on va continuer sur la commande suivante :

```
php bin/console ckeditor:install
```

```
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console ckeditor:install
| CKEditor Installer |

CKEditor is already installed in "/home/maeva/devilbox/data/www/symrecipe/vendor/friendsofsymfony/ckeditor-bundle/src/Resources/public"...

What do you want to do?
[drop] Drop the directory & reinstall CKEditor
[keep] Keep the directory & reinstall CKEditor by overriding files
[skip] Skip installation
> drop

// Dropping CKEditor from "/home/maeva/devilbox/data/www/symrecipe/vendor/friendsofsymfony/ckeditor-bundle/src/Resources/public"

482/482 [=====] 100%
// Downloading CKEditor ZIP archive from "https://github.com/ckeditor/ckeditor-releases/archive/full/latest.zip"
1706545/1706545 [=====] 100%
// Extracting CKEditor ZIP archive to "/home/maeva/devilbox/data/www/symrecipe/vendor/friendsofsymfony/ckeditor-bundle/src/Resources/public"
566/566 [=====] 100%
// Dropping CKEditor ZIP archive "/tmp/ckeditor-full-latest.zipJiQSkG"

[OK] - CKEditor has been successfully installed...
```

Puis on va installer les assets :

```
php bin/console assets:install public
```

```
maeva@maeva-ThinkPad-T460:~/devilbox/data/www/symrecipe$ php bin/console assets:install public
Installing assets as hard copies.

-----  
Bundle           Method / Error  
-----  
✓ EasyAdminBundle copy  
✓ FOSCKEditorBundle copy  
-----  
!  
[NOTE] Some assets were installed via copy. If you make changes to these assets you have to run this command again.  
[OK] All assets were successfully installed.
```

Enfin, on va aller dans le dossier packages, sélectionner le fichier fos_ckeditor.yaml

```
config  
└── packages  
    ├── cache.yaml  
    ├── debug.yaml  
    ├── doctrine_migrations.yaml  
    ├── doctrine.yaml  
    └── fos_ckeditor.yaml
```

```
fos_ck_editor:  
  configs:  
    main config:  
      toolbar:  
        - {  
            name: "styles",  
            items: [  
                "Bold",  
                "Italic",  
                "Underline",  
                "Strike",  
                "Blockquote",  
                "Image",  
                "Table",  
                "Styles",  
                "Format",  
                "Font",  
                "FontSize",  
                "TextColor",  
                "BGColor",  
                "Source",  
            ]  
        },  
        [
```

Dans lequel on va effacer les éléments présents et ajouter ce code :

https://gitlab.com/DeveloppeurMuscle/symrecipe/-/blob/master/config/packages/fos_ckeditor.yaml

On va retourner dans notre ContactCrudController, et on va modifier la fonction configureCrud :

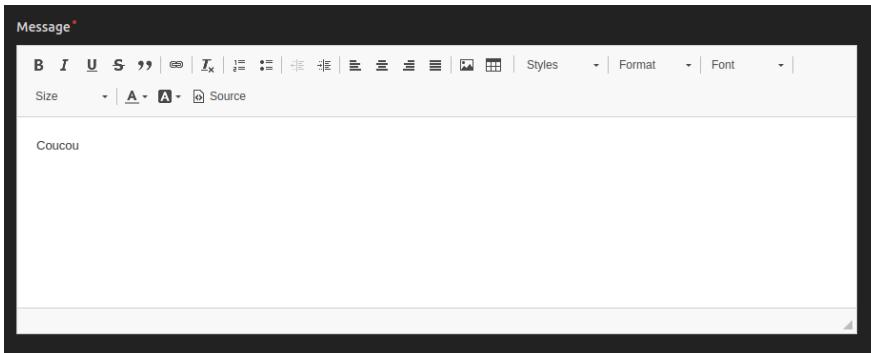
```
public function configureCrud(Crud $crud) :Crud
{
    return $crud
        ->setEntityLabelInPlural('Demandes de contact')
        ->setEntityLabelInSingular('Demande de contact')
        ->setPageTitle('index','Symrecipe - Administration des demandes de contact')
        ->setPaginatorPageSize(20)

        ->addFormTheme('@FOSCKEditor/Form/ckeditor_widget.html.twig');
}
```

Puis, modifier également la fonction configureFields, partie message :

```
TextareaField::new('message')
->setFormType(CKEditorType::class)
->hideOnIndex(),
```

La partie message sera donc modifiée dans notre Dashboard :



IV/ Restriction de l'accès administrateur

On va commencer par créer un administrateur dans notre BDD , avec un rôle Admin :

Edit Copy Delete 33 Administrateur Administrateur admin@symrecipe.fr ["ROLE_ADMIN"] \$2y\$13\$hpMKaLL4zW0hduYngTnLJenQ.o9HyIUQSd97qJ6yu3c... 2023-05-10 08:16:32 2023-05-10 08:16:32

Ensuite, on va aller dans le DashboardController , et on va ajouter la sécurité Admin à index :

```
#[Route('/admin', name: 'admin')]
#[IsGranted('ROLE_ADMIN')]
public function index(): Response
```

Et on va modifier la navbar dans le partial header, afin que le lien Admin n'apparaisse que si c'est l'administrateur qui est connecté :

```
<div class="d-flex">
<ul class="navbar-nav me-auto">
    {% if app.user %}
        <li class="nav-item dropdown" style="margin-right:5rem;">
            <a class="nav-link dropdown-toggle" data-bs-toggle="dropdown" href="#" role="button" aria-haspopup="true" aria-expanded="false">{{ app.user.fullName }}</a>
            <div class="dropdown-menu">
                <a class="dropdown-item" href="{{ path('user.edit') , {id : app.user.id} }}>Modifier mon profil</a>
                <a class="dropdown-item" href="{{ path('user.edit.password') , {id : app.user.id} }}>Modifier mon mot de passe</a>
                <div class="dropdown-divider"></div>
            {% if 'ROLE_ADMIN' in app.user.roles %}
                <a class="dropdown-item" href="{{ path('admin') }}>Espace Administrateur</a>
            {% endif %}
        </div>
    {% endif %}
</ul>

```