SYMFONY6 – PHP8 LES BASES

PRÉSENTATION DE SYMFONY

Symfony est un **framework** qui utilise le langage **Php**, il est orienté côté serveur et permet de générer du HTML grâce par défaut à son moteur de template **Twig**. Il a été créé en **2005** par l'agence web SensioLabs et plus particulièrement Fabien Potencier considéré comme son créateur, au départ comme outil interne à l'entreprise.

En effet afin d'éviter de refaire les mêmes choses sur tous les projets en terme de gestion des utilisateurs, d'ORM, de sécurité, leur équipe a développé ce framework. Comme les problématiques étaient les mêmes pour tous les développeurs Php, ils ont décidé de le partager.

Symfony respecte l'architecture **MVC** (Model, View, Controller), on créé nos contrôleurs, on utilise Twig pour la vue et Doctrine pour nos modèles.

Il faut comprendre qu'au delà d'être un Framework Symfony représente des composants, on peut tout à fait utiliser des composants de Symfony dans un projet Php, Laravel emprunte lui même de nombreux composants de Symfony.

UN FRAMEWORK POPULAIRE

Symfony est un framework open-source, de plus le Php est un langage qui est utilisé dans la grande majorité des sites web actuels. Il est très populaire en France étant lui même français.

Le 5 Septembre 2017, le framework a passé la barre emblématique du milliard de téléchargements.

Il existe une très grande communauté pour ce framework, il est donc plus facile de trouver des ressources. De plus Fabien Potencier rédige des livres lors de la sortie des versions majeures de Symfony.

DEVELOPPER AVEC SYMFONY C'EST QUOI ?

Dans les faits Symfony facilite énormément le travail du développeur, tout d'abord avec la Cli de Symfony qui permet d'utiliser des lignes de commandes afin d'interagir avec le framework et utiliser notamment son maker.

Le maker de Symfony permet en ligne de commande de créer des composants Symfony complexes, comme l'authentification, la création d'une base de données, de formulaires...

En résumé développer avec Symfony c'est se concentrer sur l'essentiel, la logique en évitant les tâches répétitives grâce à un cadre complet et flexible.

SYMFONY 6, WHAT'S NEW ?

Symfony 6 n'est ni plus ni moins que la version 5 finale, en effet elle comporte tous les ajouts de fonctionnalités des versions 5 que ce soit 5.1, 5.2, 5.3 ou 5.4

On peut entre guillemet que les différentes versions de Symfony 6 prépareront au final la version 7. Outre l'utilisation de php 8 et des attributs ainsi que des nouveautés liés à php 8 comme les nouvelles fonctions, Symfony 6 propose de nouveaux composants :

- L'uid component qui facilite l'utilisation d'uuid
- Le Security qui a modifié son système de Guard et facilite ainsi la mise en place de la double authentification
- Rate Limiter qui permet de limiter le nombre de tentative de login
- PasswordHasher qui a maintenant son propre composant pour hasher les mots de passe
- Session qui n'est plus considéré comme un service car lié à l'état de l'application ce qui ne doit pas être le cas d'un service qui ne doit pas être lié à l'état de l'application
- TranslationProviders qui permettent d'interfacer les traductions avec un Saas

Voici maintenant quelques nouvelles fonctionnalités de Symfony 6 :

- Pouvoir configurer le code http qui résulte d'une exception
- La classe Path du FileSystem qui permet de gérer plus facilement le concept de chemin

- Utiliser des unions d'autowire appeler plusieurs classes dans une même variable (NormalizerInterface&DenormalizerInterface $serializer)
- Ajoute l'autocompletion sur la console

LES PRÉ-REQUIS

Symfony est un framework qui utilise le langage Php, et le moteur de template Twig pour rendre la vue qui elle-même utilisera du HTML, du CSS et un framework CSS comme Bootstrap ou Tailwindcss. Il utilise l'ORM Doctrine pour effectuer les opérations liées à la base de données et dans ce cours nous utiliserons une base de données en SQL. Symfony permet l'installation de packages qui nécessitent un logiciel permettant de les installer, on appelle ça un gestionnaire de dépendances. Enfin il nous faudra un éditeur de code permettant de saisir notre code.

On comprend rapidement que l'on va avoir besoin de compétences et de logiciels, nous allons passer en revue ces différents éléments et voir comment les mettre en place.

LES COMPÉTENCES

- ### PHP

  Symfony utilise le **Php** comme langage de programmation, il faudra donc avoir de bonnes connaissances de ce langage afin de maîtriser le framework et comprendre ce qu'il se passe quand on l'utilise ou même pour intéragir avec le framework (créer un service, rédiger les contrôleurs...). Dans ce cours nous utiliserons la **version 8 de Php**.

  Lien vers vidéo démo de Php 8 vs Php 7

- ### HTML-CSS

  Symfony permet de renvoyer une vue grâce par défaut au moteur de template Twig qui offre de nombreuses possibilités notamment avec son système de filtre. **Twig** fait donc appel au **HTML** ainsi qu'au **CSS**. À noter que bien souvent on utilise un framework CSS afin de se faciliter le travail de développement front-end. Dans ce cours nous utiliserons **Bootstrap**.

- ### SQL

  Comme tout framework back-end, Symfony permet d'effectuer des opérations sur une base de données. Dans ce cours nous utiliserons une base de données **SQL**, il

faudra donc connaître ce langage pour maîtriser Symfony. Cependant on peut tout à fait envisager de travailler avec d'autres types de base de données comme du no-sql, mais afin de tirer profit au maximum de la "magie" de Symfony il est recommandé d'utiliser une base de donnée de type SQL, le système de Documents du no-sql étant plus réservé à un usage dans le cadre de Big Data.

- POO

La programmation orientée objet en Php est un concept majeur utilisé dans Symfony et qu'il faut comprendre et maîtriser.

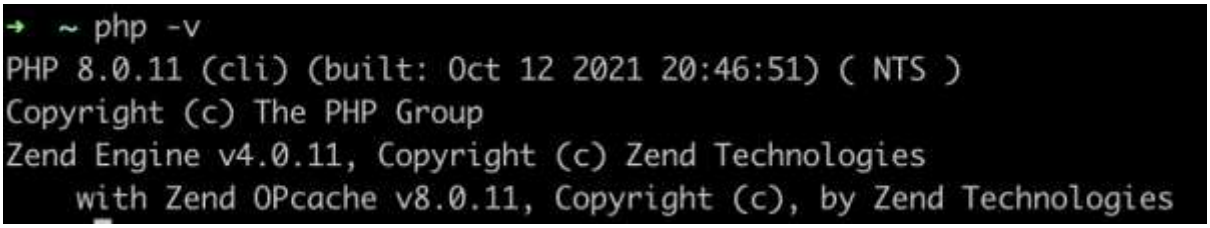
- MVC

Symfony fonctionnant sur le modèle d'un MVC, il est important de comprendre ce concept de la programmation.

LES LOGICIELS

- VERSION DE PHP :

Il va dans un premier s'assurer de la version de Php installée dans notre machine. Voici la ligne de commande à taper dans son terminal :



Lien vers le site de Php si besoin

- XAMPP OU WAMP :

Afin de pouvoir faire fonctionner un serveur local et avoir accès à une base de données et à l'interface PhpMyAdmin nous devrons installé soit Xampp compatible Mac et Windows ou Wamp qui lui n'est compatible que sur Windows.

Lien vers le site Wamp

Lien vers le site de Xampp

- COMPOSER :

Afin d'installer les différentes dépendances nous utiliserons Composer. Nous pourrons vérifier l'installation comme ceci :

```
➜  ~ composer --version
Composer version 2.0.14 2021-05-21 17:03:37
```

Lien vers le site de Composer

- COMMANDER (SI WINDOWS) :

Si votre machine fonctionne sous Windows il est recommandé d'installer le logiciel Commander afin d'avoir un terminal plus facile d'accès et plus agréable à utiliser.

Lien vers le site de Commander

- ÉDITEUR DE CODE :

Il nous faudra un éditeur de code de type IDE comme Vs-Code qui est gratuit mais vous êtes libre d'utiliser celui qui vous convient. Il faudra passer en revue les extensions et plugins afin d'optimiser notre éditeur pour Php Twig et Symfony. Pour développer en Symfony on peut utiliser PhpStrom qui est payant mais qui est extrêmement complet et orienté vers le Php.

Lien vers Visual Studio Code

LES RESSOURCES

Elle va devenir votre meilleure atout et amie, la documentation officielle Symfony. Elle couvre absolument tous les cas d'usage de Symfony, avec une interface utilisateur intuitive.

Lien vers la documentation Symfony

DÉMARRER UN PROJET SYMFONY

Maintenant que notre environnement de développement est Symfony ready, nous allons initialiser notre projet Symfony.

Pour ce faire je vous propose d'utiliser l'outil de Symfony la CLI de Symfony plutôt que de passer par Composer. Cette CLI offre de très nombreux avantages dans le développement d'une application Symfony.

DÉMARRER PROJET VIA LA CLI SYMFONY

Nous allons installer la CLI :

Lien vers la doc pour installer la CLI

Nous pourrons lancer nos lignes de commande en utilisant directement symfony. Nous pouvons aussi utiliser la CLI afin de vérifier notre environnement.

```
➜  ~ symfony check:requirements
```

ATTENTION : Il faudra dans notre terminal nous placer dans le dossier où l'on souhaite créer le projet, lui même dans le dossier htdocs.

Voici la ligne de commande afin de créer un projet Symfony afin de construire une application web ce que nous allons faire :

```
~ symfony new my_blog --full
```

Pour info voici la ligne de commande afin de créer un microservice, une API ou une application console donc plus léger :

```
~ symfony new my_blog
```

LANCER LE SERVEUR VIA LA CLI

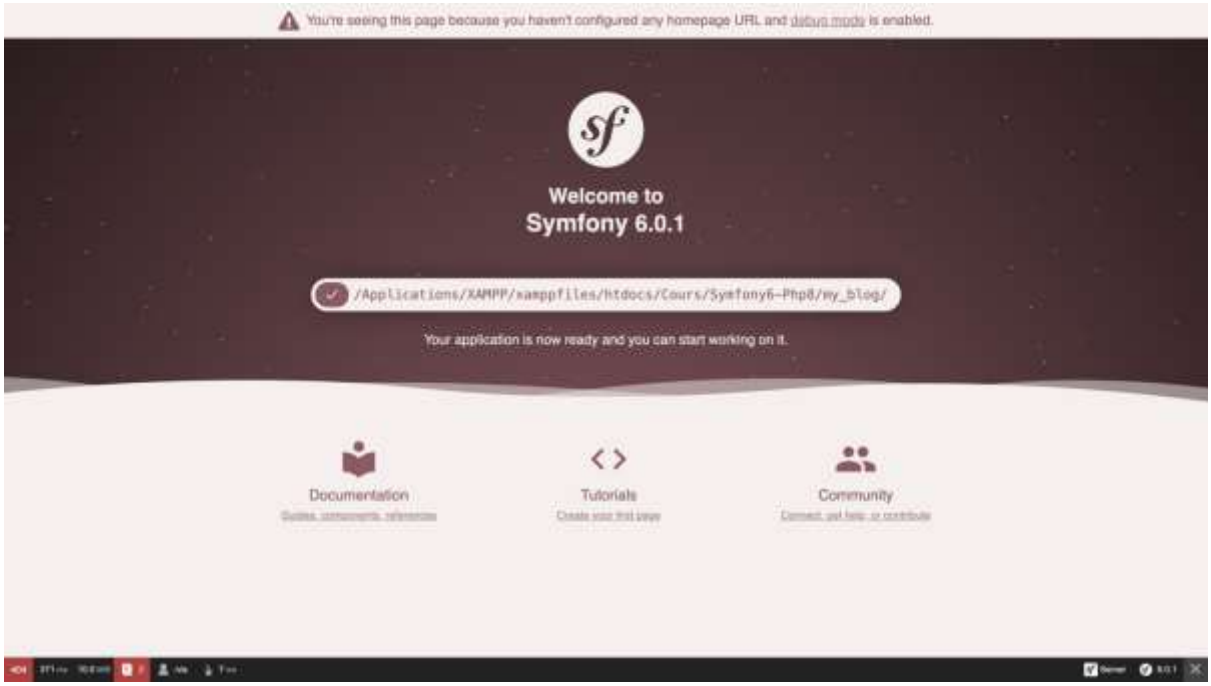
Il faut maintenant se déplacer dans nos dossiers afin de se placer à la racine de notre projet. Dans notre cas :

```
Symfony6-Php8 cd my_blog
```

Une fois dans le bon dossier on peut lancer cette commande afin de démarrer le serveur et voir notre application Symfony fonctionner :

```
my_blog git:(master) symfony server:start
```

Voici la vue que l'on obtient :



LA DEBUG TOOLBAR

Vous remarquerez en bas de page la debug toolbar de Symfony, quand on dit qu'un Framework apporte un vrai gain de productivité et de vrais outils en voici un exemple parfait. Vous retrouverez dans cette debug toolbar l'ensemble des informations liés au projet, bien souvent les erreurs qui apparaissent en rouge. Il est important d'apprendre à bien utiliser ce merveilleux outil qui peut vous permettre d'avoir toutes les informations et de comprendre les erreurs liées aux requêtes, aux performances, au routing, au Cache, à Doctrine, aux formulaires par exemple.

LA STRUCTURE DES DOSSIERS

Nous allons maintenant lancer notre éditeur de code et ouvrir le dossier de notre projet pour passer en revue les différents dossiers qui composent un projet Symfony.

- Le dossier **bin** : Lié à la console et les commandes associées
- Le dossier **config** : Lié à la configuration de Symfony, des packages (routes, services...)
- Le dossier **migration** : Lié à nos créations de structure de la base de données, contient du SQL lié aux modifications apportées à la base
- Le dossier **public** : Il est la racine du projet, ce qu'il contient est donc publique
- Le dossier **src** : Un des dossiers les plus importants, il contient nos Contrôleurs et nos Modèles mais aussi nos services, nos classes. En bref notre code Php
- Le dossier **template** : Un des dossier principaux, c'est là que l'on range nos Vues, des fichiers html.twig
- Le dossier **test** : Lié aux tests comme son nom l'indique
- Le dossier **translations** : Lié aux traductions du contenu du site
- Le dossier **var** : Lié aux fichiers de cache et de log
- Le dossier **vendor** : Lié aux librairies installées via composer, dossier dont on ne modifie pas le contenu afin d'assurer la maintenabilité du projet
- Les fichiers **.env** : Contient nos variables d'environnement (base de données, service de mail etc...)
- Les fichiers **composer** : Lié aux librairies installées dans le projet et leur version, permet d'installer le projet avec les versions souhaitées.

En option à cette étape, nous pouvons initialiser un nouveau dépôt git.

NOTRE PREMIER CONTROLLER

Afin de bien comprendre le concept de MVC et le fonctionnement de Symfony nous allons créer notre premier controller. Pour ce faire nous utiliserons la cli et le maker de Symfony.

```
my_blog git:(master) ✗ symfony console make:controller HomePageController
```

ATTENTION : Par convention on nomme un controller en finissant par Controller, de plus il faut respecter le camel case.

Cette commande créée un nouveau fichier dans le dossier controller et un nouveau dossier, avec un nouveau fichier html.twig à l'intérieur, dans le dossier template.

Nous allons ensuite modifier le controller comme suit :

```php
<?php

namespace App\Controller;

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

class HomePageController extends AbstractController
{
    /**
     * Permet d'afficher la page d'accueil
     * @return Response
     */
    #[Route('/', name: 'homepage')]
    public function index(): Response
    {
        return $this->render( view: 'homepage/index.html.twig');
    }
}
```

Et la vue comme cela :

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

{% block title %}Accueil - My Blog{% endblock %}

{% block body %}
<h1>Page d'accueil du site</h1>
{% endblock %}
```

Allons voir le rendu.

PASSER DES INFORMATIONS Á LA VUE TWIG

Lors des précédentes modifications nous avons supprimé un tableau avec une valeur pour controller_name. Prenons un exemple comme ceci :

```php
/**
 * Permet d'afficher la page d'accueil
 * @return Response
 */
#[Route('/', name: 'homepage')]
public function index(): Response
{
    return $this->render( view: 'homepage/index.html.twig', [
        'page_name' => 'HomePage'
    ]);
}
```

LA PUISSANCE DU DUMP DE TWIG

Dans notre fichier de vue on va ajouter :

```twig
{{ dump(page_name) }}
```

Voilà comment passer des informations à la vue, gardez en tête que celle-ci doivent être contenu dans un tableau []. Nous pouvons effacer ces modifications c'était juste pour la compréhension.

INSTALLER BOOTSTRAP

Nous allons avoir besoin d'un framework **CSS** afin de styliser notre projet et gagner du temps dans le développement. Ici nous utiliserons **Bootstrap** car il est très répandu et utilisé. Nous aurions pu faire le choix de **Tailwindcss** qui aurait nécessité plus de mise en place et l'installation de **Webpack** mais qui offre plus de possibilités pour le front-end.

Lien vers Bootstrap

Pour varier un peu on peut utiliser aussi un de ces thèmes Bootstrap :

Lien vers Bootswatch

Dans le dossier public nous allons créer un dossier assets qui contiendra lui-même un dossier css et un dossier js. Nous rangerons donc les fichiers minifiés de Bootstrap dans les dossiers correspondants.

Il faudra ensuite modifier notre template base.html.twig qui représente le gabarit du site afin d'intégrer nos fichiers Bootstrap au projet et pouvoir l'utiliser sur l'ensemble des vues. Nous utiliserons ici le système d'asset de Symfony.

Pour le CSS:

```
<!-- Bootstrap core CSS -->
<link href="{{ asset('./assets/css/bootstrap.min.css') }}" rel="stylesheet">
```

Pour le Javascript :

```
<!-- Bootstrap core Js -->
<script src="{{ asset('assets/js/bootstrap.bundle.min.js') }}"></script>
```

Votre dossier public doit donc être comme ça :

Mettez à jour la page pour voir que Bootstrap fonctionne sur le `<h1>`.

LE FICHIER BASE.HTML.TWIG

Ce fichier correspond au gabarit de notre site web, bien souvent il contiendra le header, la nav et le footer. Nous allons découvrir le **concept de bloc** dans **Twig** et mettre en place le gabarit de notre projet. Il sera ensuite appelé dans tous les templates où il est nécessaire avec la méthode extends comme suit :

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

Tout d'abord nous allons ajouter les meta viewport et description, puis créer un fichier CSS que l'on nomme base.css et que l'on range dans le dossier assets puis css. On l'appelle dans *base.html.twig* comme nous l'avons fait pour les fichiers Bootstrap.

Ensuite nous modifions le contenu du titre du site dans la balise `<title>`, vous remarquerez que le contenu est dans un **block** qui est nommé title, cela va permettre ensuite dans chacune de vos vues d'appeler ce bloc et d'en changer le contenu. Cela vous donnera une balise `<title>` adaptée et qui peut même être dynamique, SEO friendly.

LES PARTIALS

Afin de découper notre code et respecter les bonnes pratiques en vue d'une bonne maintenabilité et évolution future possible, nous allons créer un dossier partials dans le dossier template. On mettra dedans ni plus ni moins que des bouts de code html que l'on pourra ensuite appeler dans Twig.

Il va donc falloir créer un fichier header.html.twig puis un pour la nav et un pour le footer. Pour le moment ils doivent juste contenir un titre h2 avec leur nom Nav pour la nav ainsi de suite...

Nous allons ensuite mettre en place l'appel de ces fichiers dans Twig grâce au include comme suit :

```
<body>
    <!-- Le header du site qui contient la navigation-->
    {% include './partials/header.html.twig' %}
    <main>
        <!-- Block content avec le contenu du main -->
        {% block content %}{% endblock %}
    </main>
    <!-- Le footer du site -->
    {% include './partials/footer.html.twig' %}
    <!-- Bootstrap core Js -->
    <script src="{{ asset('assets/js/bootstrap.bundle.min.js') }}"></script>
</body>
```

Il faut faire la même chose dans le fichier *header.html.twig* afin d'inclure le fichier *nav.html.twig.* À ce stade tout est en place il n'y a plus qu'à mettre en place le html de chaque fichier et le css si nécessaire mais on va éviter au maximum le but ici n'étant pas d'approfondir le CSS.

Notez qu'il faudra renommer le bloc body de *index.html.twig* du dossier homepage afin de mettre block content pour coller à notre *base.html.twig.*

MISE EN PLACE DE LA NAVIGATION DU HEADER ET DU FOOTER

Vous pouvez prendre directement l'exemple de Bootstrap, votre navigation doit juste comporter trois items Blog About Contact. En termes de design vous êtes libre. Elle doit être inclue dans votre header.

TP 1 CONTROLLERS SYMFONY

Nous venons d'en finir avec la mise en place de notre projet, nous avons créer une première route qui permet d'arriver sur la page d'accueil du site. Vous allez devoir créer les autres routes et controllers associés aux éléments de la barre de navigation.

- 1 er élément :
  Nom du controller : BlogController,
  Nom de la route : blog,
  Url : /blog

- 2 ème élément :
  Nom du controller : AboutController
  Nom de la route : about
  Url : /about

- 3 ème élément :
  Nom du controller : ContactController
  Nom de la route :
  Url : /contact

Vos templates devront à l'heure actuelle n'afficher qu'un titre `<h2>` avec comme contenu le nom de la page. Ils devront étendre de *base.html.twig*, pensez au bloc *content*.

MISE EN PLACE BASE DE DONNEES AVEC CLI

Nous venons de mettre en place notre projet et les différents controllers liés au front de notre projet. Nous allons maintenant mettre en place notre base données. Nous allons commencer à utiliser l'ORM (Object Relationnel Mapping) **Doctrine**.

Lien vers la documentation Doctrine

CRÉATION DE LA BASE

LE FICHIER ENV

Afin de paramétrer notre base de données, nous allons intervenir sur le fichier *.env* du projet. Ce fichier je le rappelle concerne les variables d'environnement. Nous allons travailler avec une base données SQL, il va donc falloir préciser à Symfony **l'adresse** de cette base, **l'identifiant** et le **mot de passe** pour s'y connecter, le **nom de la base**, le **type** et la **version** de base de données utilisée.

UN FICHIER ENV.LOCAL

Puisque nous travaillons actuellement en local, nous allons préciser des variables d'environnement pour le développement. Il faudra donc créer un fichier *env.local* à la racine de notre projet, si l'on travaille avec git il faudra veiller à ce que le fichier soit dans le .gitignore pour ne pas le versionner.

Une fois en production, il faudra veiller à modifier notre fichier *.env* afin de lui affecter les valeurs de notre base données hébergée sur le serveur.

Voici la ligne sur laquelle appliquer nos valeurs :

```
DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=5.7"
```

Une fois la variable d'environnement saisie, nous allons pouvoir interagir avec **Doctrine** via la CLI de Symfony afin de lui demander de créer notre base. Nous pourrons ensuite aller dans php MyAdmin afin de vérifier la création de celle-ci.

Voici la commande à utiliser :

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

Vous pouvez également afficher la liste des commandes liées à **Doctrine** avec cette commande :

```
symfony console list doctrine
```

CRÉER LES ENTITÉS

Nous avons maintenant une base de données paramétrée, il va falloir créer les tables de celle-ci.

NOTRE PROJET :

Nous allons créer un blog, il nous faudra donc une table themes qui représentera et contiendra les thèmes de nos articles, et évidemment une table articles qui représentera et contiendra les articles du blog.

Nous allons donc devoir créer deux entités, vous l'aurez compris une entité représente ni plus ni moins qu'une table de la base de données.

NOTRE PREMIÈRE ENTITÉ :

C'est là que la magie de Symfony intervient, nous allons pouvoir, via la CLI et au maker de Symfony, créer nos tables mais aussi les champs de la table avec leurs types, s'ils peuvent être null ou non et même créer des relations entre les tables.

Voici la commande qui permet de lancer la création d'une entité, ici l'entité Themes :

```
symfony console make:entity Themes
```

Cette entité Themes devra comporter un champ title de type string qui ne pourra pas être null et qui représentera le nom du thème ainsi qu'un champs illustration toujours de type string et non nullable qui représentera l'image illustrant ce thème. Pour saisir les informations il suffira de répondre aux questions que le maker va nous poser tout simplement. Ici nous prenons parti d'imposer une image à la création d'un thème il aurait été possible de ne pas l'imposer et mettre nullable à true et mettre en place un système d'image par défaut.

NOTRE DEUXIÈME ENTITÉ

L'entité Articles qui représentera les articles de notre blog devra contenir un champ title de type string non nullable qui sera le titre de l'article et un champ content de type text qui représentera donc le contenu de l'article.

Nous avons donc maintenant deux entités qui représentent deux tables dans la base donnée. Vous noterez que le maker de Symfony est sympa après chaque création d'une entité il vous indique quoi faire ensuite.

On vous invite ici à lancer les migrations, nous attendrons d'en avoir fini avec nos tables avant de lancer la migration. Nous pouvons nous permettre de faire ainsi car la structure de notre base est simple. Dans le cas de grosses structures de base de données, il est recommandé de traiter les entités les unes après les autres et de créer une migration pour chacune d'elles.

Nous allons travailler une base de données relationnelle, nous allons donc devoir établir cette relation entre nos thèmes et nos articles.

LES RELATIONS AVEC DOCTRINE

Grâce au maker et à Doctrine nous allons pouvoir créer une relation entre nos deux tables très facilement, une relation sera créée grâce à un nouveau champs auquel on attribuera le type relation.

Nous allons à nouveau demander au maker de créer une entité Articles comme elle existe déjà on va nous proposer d'ajouter des champs à celle-ci. Nous allons créer un champ themes de type relation.

Le maker va nous demander sur quelle classe la relation va se faire, ici l'entité Themes. Il nous demandera ensuite le type de relation.

Il faut prendre le temps de bien lire chaque relation pour en comprendre le sens et choisir celle que notre cas nécessite. Ici vous l'aurez compris, nous utiliserons une relation de type ManyToOne.

En effet, chaque article sera relié à un thème et chaque thème sera relié ou pourra avoir plusieurs articles.

La console ensuite nous demande si le champ peut être null ici non un article doit avoir un thème. Ensuite on nous demande si on souhaite implémenter une méthode au sein de notre entité Themes qui ira chercher tous les Articles de ce thème. Nous choisirons oui, autant utiliser toute la magie du framework. Nous laisserons le nom choisit par défaut.

Enfin la console va nous demander si nous souhaitons automatiquement supprimer les articles orphelins (non reliés à une entité Themes), nous choisirons non dans ce cas.

Nous en avons fini avec nos entités, nous allons pouvoir lancer nos migrations. Tout d'abord nous allons devoir créer notre migration avec cette commande :

```
symfony console make:migration
```

La console nous invite à exécuter la commande *doctrine:migrations:migrate*

Voici la commande raccourcie :

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

Nous pouvons aller sur Php MyAdmin et constater l'état de notre base de données.

LE CYCLE DE VIE D'UNE ENTITÉ

Nous allons souhaiter ajouter des champs à nos entités, un champ createdAt sur nos articles qui représentera la date de création de l'article, ainsi qu'un champs slug de type string non nullable. Nous ajouterons également un slug à notre entité Themes qui aura les mêmes attributs que pour les articles.

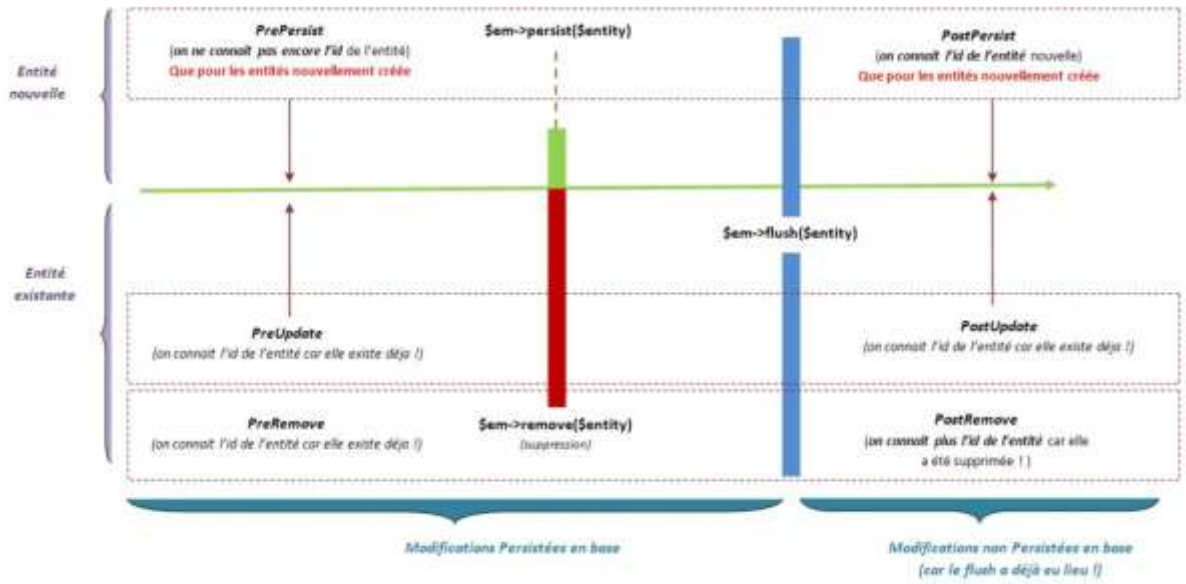
Un **slug** est un code qui permet la transformation d'une chaîne de caractère en supprimant les espaces, les caractères spéciaux, les majuscules afin d'en faire une chaîne de caractère utilisable comme **url**.

Que ce soit pour la date de création ou le slug il va falloir automatiser l'utilisation de ces champs car on ne va pas demander à l'utilisateur de les saisir. Pour la date de création nous lui attribuerons la date du jour au moment du traitement du formulaire en utilisant l'objet DateTime de Php.

Pour le slug afin d'assurer un bon niveau de performance et éviter la redondance de code nous devrons créer un service qui permettra de réaliser le Slug et utiliser ce service juste avant de persister les données en base.

Il va donc falloir agir sur le cycle de vie de l'entité. Lorsque nous allons créer nos articles ou thèmes en base de données, nous allons persister les données en base lors de la validation d'un formulaire. Nous allons donc devoir utiliser un ensemble de méthodes comme **PrePersist** pour agir sur le cycle de vie et ces nouveaux champs afin de les automatiser.

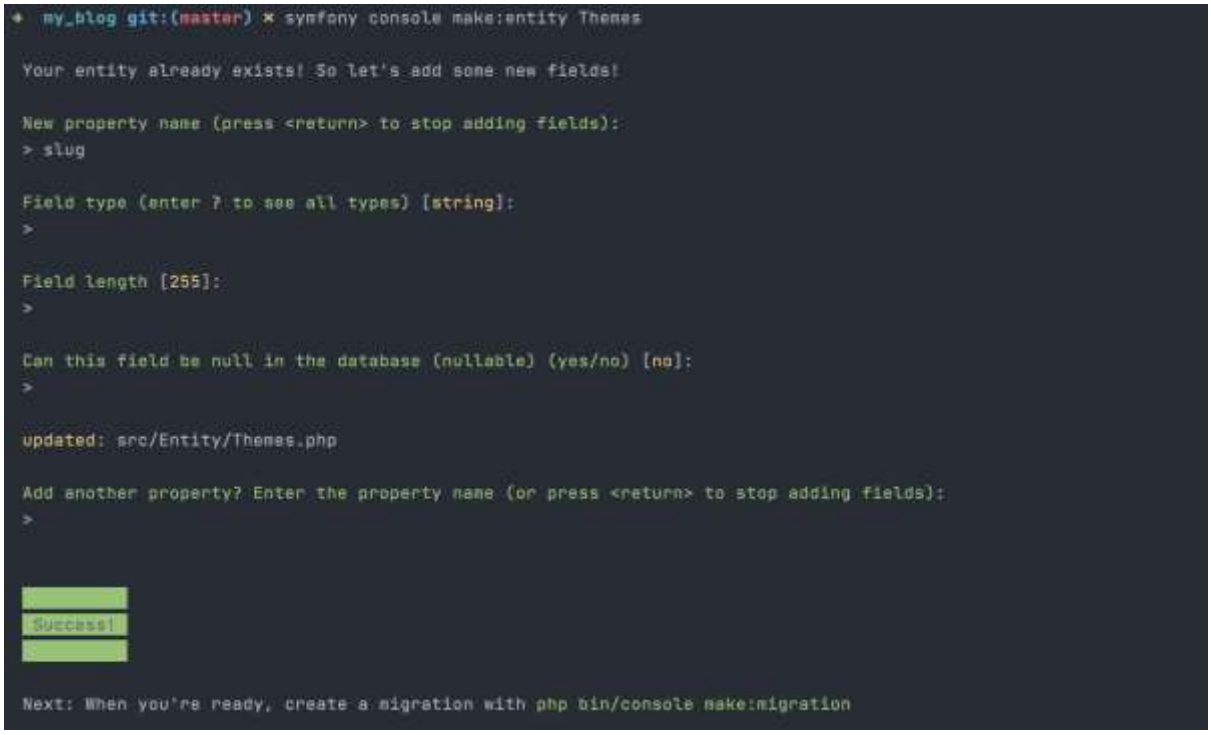
Schéma représentant le cycle de vie d'une entité et les méthodes associées :



Commençons par ajouter les champs à nos entités et créer la migration associée et l'appliquer à notre base de données.

Pour l'entité Articles :

Pour l'entité Themes :



Nous pourrons ensuite effectuer notre migration.

Création du service de Slug :

Voir TP 2

Maintenant que notre service de Slug est fonctionnel, nous allons voir comment générer le slug en utilisant le cycle de vie d'une entité.

1. **Préciser à l'entité que l'on va agir sur le cycle de vie de l'entité :**

   Nous allons utiliser les annotations afin de préciser que l'entité a des méthodes qui agissent sur son cycle de vie, pour cela nous devons coder ceci :

```
/**
 * @ORM\Entity(repositoryClass=ThemesRepository::class)
 * @ORM\HasLifecycleCallbacks()

 */
```

2. **Rédiger la méthode permettant d'intialiser le slug :**

   Nous allons écrire la méthode initializeSlug, nous devrons à nouveau utiliser les annotations afin de lui préciser que cette méthode doit être appelée à certains moments du cycle de vie de l'entité. Il ne faudra pas oublier le use afin de pouvoir utiliser notre service Slugify.

```
/**
 * Permet de créer le slug avec le titre
 * @ORM\PrePersist
 * @ORM\PreUpdate
 */
public function intializeSlug():void
{
    if(empty($this->slug)){
        $this->slug = Slugify::slugify($this->title);
    }
}
```

Ici nous précisons deux méthodes du cycle de vie de l'entité, la méthode **PrePersist** qui veut dire avant de persister les données dans la base et **PreUpdate** qui veut dire avant de mettre à jour les données en base.

3. **Réaliser la même implémentation pour l'entité Articles.**

TP2 CRÉER UN PREMIER SERVICE

Nous allons créer un service afin de réaliser le slug sur une chaîne de caractères. Il faudra créer dans le dossier src un dossier Services avec à l'intérieur une classe Slugify qui devra contenir une méthode slugify.

Elle prendra en argument une string et aura pour fonction de modifier cette chaîne de caractères en supprimant les espaces et en les remplaçant par des - ainsi que de supprimer les majuscules et les caractères spéciaux.

SYSTÈME D'AUTHENTIFICATION

Maintenant que nos deux entités principales sont en place nous serions tentés de créer une entité pour les utilisateurs du site. C'est là que la magie de Symfony intervient, nous allons utiliser le maker afin de générer un système d'authentification.

Pour rappel, nous réalisons un blog, nous n'avons donc besoin que peu de propriétés pour nos utilisateurs afin de ne pas faire une étape bloquante lors de l'inscription. Un utilisateur aura un **email** et un **mot de passe**. Nous pourrons facilement de toute façon changer d'avis et ajouter des propriétés comme une image d'avatar si on met un système de commentaires en place par exemple.

Nous allons donc une fois de plus utiliser le terminal afin de générer via des lignes de commandes notre système d'authentification.

Symfony va nous poser des questions afin de générer un système d'authentification conforme à nos besoins. Ici il suffira de valider chaque étape.

Ici notre entité représentant les utilisateurs se nommera **User,** ils seront **stockés en base** avec Doctrine, l'identifiant sera **l'email** et sera **unique** à chaque utilisateur et enfin nous demandons à ce que les mots de passe soient **cryptés** (hasher).

Symfony nous affiche ensuite les prochaines étapes à réaliser.

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

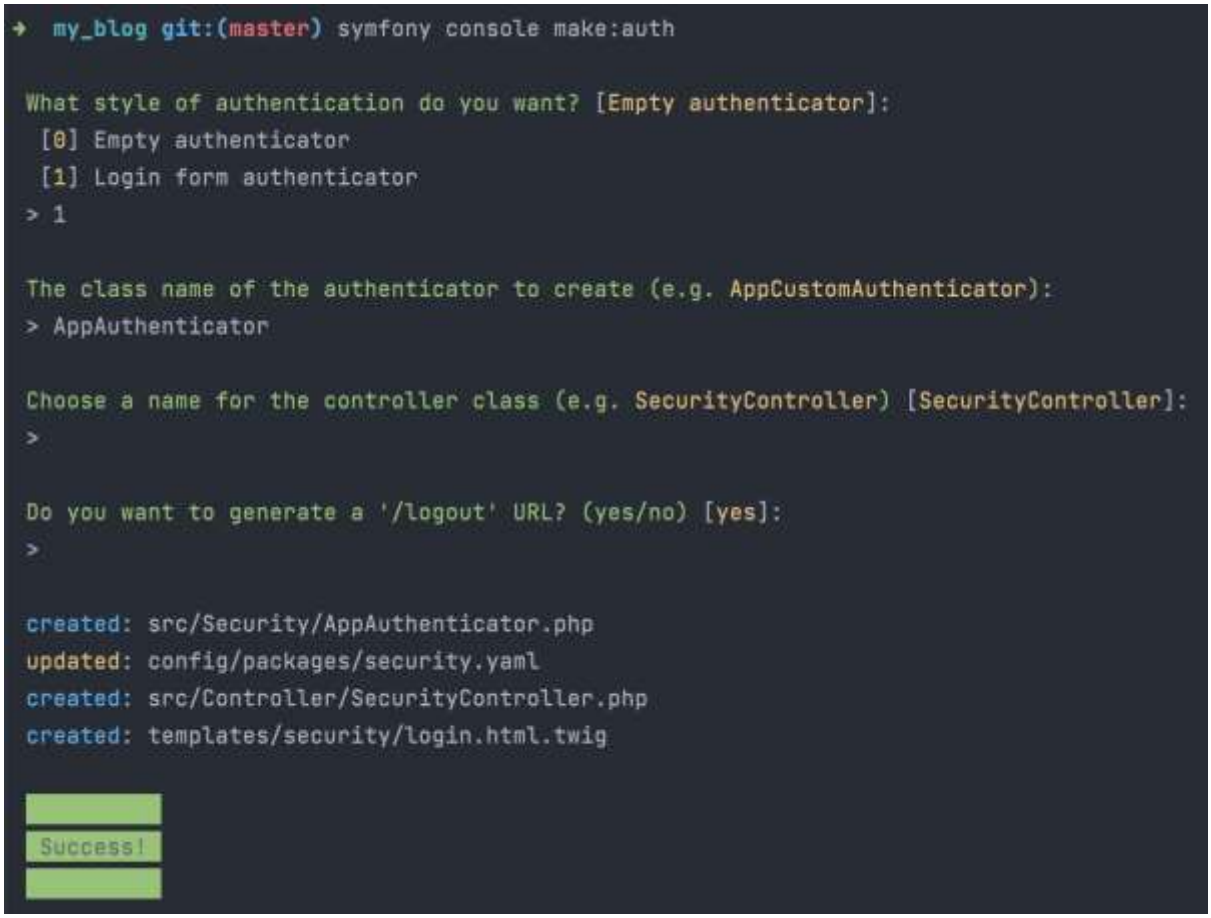
Allons voir ce que Symfony a créé pour nous en commençant par l'entité User, et effectuer la migration pour cette entité. Ici nous n'ajouterons pas de nouvelles propriétés mais nous aurions pu décider d'en ajouter à ce moment-là.

LE FICHIER SECURITY.YAML

Ici nous pouvons voir le système mit en place par Symfony, avec le provider ainsi que le système de hashage de mots de passe choisit, ici auto (brcypt).

Jusque-là on vient de gagner beaucoup de temps afin de réaliser un système d'authentification, mais pour le moment on n'a rien pour se connecter ou déconnecter. On va à nouveau utiliser le maker afin d'obtenir un authentificateur.

Nous allons donc utiliser la commande `symfony console make:auth` , comme ceci :



On va juste modifier le nom de l'authentificateur par AppAuthenticator, pour le reste nous validons le choix par défaut. Allons à nouveau voir le fichier security.yaml, nous allons aussi jeter un œil sur le fichier AppAuthenticator que Symfony nous a généré dans le dossier Security. Nous pouvons y voir un *TODO*, en effet il faudra préciser la route vers laquelle nous souhaitons rediriger notre utilisateur une fois connecté.

Enfin jetons un œil au template créé dans le dossier security, *login.html.twig* , nous pouvons le mettre à jour afin d'avoir un template conforme à notre système de bloc.

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

{% block title %}Connexion - My Blog{% endblock %}

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

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

    <h1 class="h3 mb-3 font-weight-normal">Connectez-vous</h1>
    <label for="inputEmail">Email</label>
    <input type="email" value="{{ last_username }}" name="email" id="inputEmail" class="form-control" autocomplete="email" required autofocus>
    <label for="inputPassword">Mot de passe</label>
    <input type="password" name="password" id="inputPassword" class="form-control" autocomplete="current-password" required>

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

    {#        Uncomment this section and add a remember_me option below your firewall to activate remember me functionality.....    #}

    <button class="btn btn-lg btn-primary" type="submit">
        Se connecter
    </button>
</form>
{% endblock %}
```

Enfin allons voir le controller qui a été créé et modifions-le afin de rester dans notre optique de travailler en php 8. En effet php 8 demande d'utiliser le système d'attribut plutôt que le système d'annotations afin de paramétrer nos routes.

```php
class SecurityController extends AbstractController
{
    /**
     * Permet de se connecter
     * @param AuthenticationUtils $authenticationUtils
     * @return Response
     */
    #[Route('/connexion', name: 'app_login')]
    public function login(AuthenticationUtils $authenticationUtils): Response
    {
        // if ($this->getUser()) {
        //     return $this->redirectToRoute('target_path');
        // }

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

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

    /**
     * Permet de se déconnecter
     * @Route("/logout", name="app_logout")
     */
    #[Route('/logout', name: 'app_logout')]
    public function logout(): void
    {
        throw new \LogicException( message: 'This method can be blank - it will be intercepted by the logout key on your firewall.');
    }
}
```

Nous pouvons maintenant aller sur l'onglet qui fait tourner notre projet et aller sur l'url /connexion. Nous voyons notre formulaire de login s'afficher. Il demande un peu de mise en forme grâce à Bootstrap mais il est en place. Nous allons ensuite voire comment s'inscrire, puis mettre en place un système de fixtured afin de paramétrer un compte administrateur.

GÉRER L'INSCRIPTION AU SITE

Nous venons de mettre en place notre système d'authentification, il va maintenant falloir permettre à un utilisateur anonyme de s'inscrire sur le site.

Une fois n'est pas coutume, nous allons utiliser le **maker** de Symfony afin de créer notre **controller**, mais aussi créer le **formulaire d'inscription**.

CRÉATION CONTROLLER

Nous allons créer le controller qui va se nommer RegisterController qui se nommera register et dont la route aura pour url /inscription. Une fois ce controller généré il va falloir mettre en place le template associé et ensuite aller sur notre page web /inscription pour voir s'afficher notre page d'inscription.

Voici le controller :

```php
<?php

namespace App\Controller;

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

class RegisterController extends AbstractController
{
    /**
     * Permet à un utilisateur anonyme de s'inscrire
     * @return Response
     */
    #[Route('/inscription', name: 'register')]
    public function index(): Response
    {
        return $this->render( view: 'register/index.html.twig');
    }
}
```

Voici la mise en place du template :

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

{% block title %}Inscription - My Blog{% endblock %}

{% block content %}
    <div class="container-md">
        <div class="jumbotron bg-primary text-light">
            <h2>Vous souhaitez vous inscrire !</h2>
            <p>
                Être inscrit vous permet d'être alerté lors de la sortie d'un nouvel article et de commenter les articles.
                De plus certains articles ne sont accessibles qu'aux membres enregistrés sur le blog.
            </p>
        </div>
        <section>
            <!-- Formulaire d'inscription -->
        </section>
    </div>
{% endblock %}
```

MISE EN PLACE DU FORMULAIRE

Nous allons donc utiliser une nouvelle commande du maker de Symfony `make:form`, tout simplement. Par convention les formulaires dans un projet Symfony sont placés dans le dossier `Form`, leur nom se termine toujours par **Type**, mais le **maker** l'ajoute automatiquement.

Le maker va commencer par nous demander le nom de notre formulaire, ici **Register** ensuite sur quelle entité ce formulaire se base, ici User car ce sont des User que l'on souhaite pouvoir créer via le formulaire.

```
➜  my_blog git:(master) ✗ symfony console make:form

 The name of the form class (e.g. DeliciousGnomeType):
 > Register

 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/RegisterType.php

  Success!
```

LE FORMBUILDER DE SYMFONY

Maintenant, nous pouvons aller voir le fichier RegisterType qui a été créé dans le dossier Form. Il contient actuellement trois champs qui correspondent aux 3 propriétés de notre entité User. Nous allons supprimer le champ lié au roles car on ne souhaite pas que l'utilisateur puisse choisir son rôle. Nous reviendrons dans une prochaine section sur cette propriété roles.

Le FormBuilder de Symfony auquel on a accès grâce à la FormBuilderInterface. Il va nous permettre de mettre en place notre formulaire (label, placeholder, classes Bootstrap), le sécuriser en typant les champs et afficher les erreurs. Il faudra cependant ajouter des contraintes de validation sur nos entités afin d'assurer la sécurité d'un formulaire côté serveur, encore plus lorsque ce formulaire est accessible au public.

Lien vers la documentation Symfony les Formulaires

Lien vers la documentation Symfony les Types

LE CHAMP EMAIL

```
->add( child: 'email',  type: EmailType::class, [
    'label' => 'Votre email',
    'attr' => [
        'placeholder' => 'Merci de saisir votre email',
        'class' => 'form-control my-2'
    ],
    'label_attr' => [
        'class' => 'text-info'
    ],
    'required' => true,
    'invalid_message' => 'Le mot de passe et la confirmation doivent être identiques',
])
```

Le premier argument 'email', correspond au nom de la propriété à laquelle le champ que l'on créé se réfère. Le second EmailType::class, correspond au type de champs supporté par Symfony (voir la doc). Dans la documentation nous voyons toutes les options possibles sur ce type de champs. Ici nous modifions le **label** associé et grâce à attr qui représente les attributs, nous lui ajoutons les classes **Bootstrap** et le **placeholder**. label_attr représente les attributs du label, ici nous ajoutons des classes **Bootstrap** afin de modifier la couleur du texte.

Enfin nous mettons l'option 'required' à true pour que le champs email soit obligatoire, et nous customisons le message en cas d'erreur avec le 'invalid_message' .

LE CHAMP PASSWORD

Ici nous allons utiliser deux autres types de Form Field de Symfony, en effet pour le mot de passe il est recommandé de doubler le champ afin de s'assurer de la bonne saisie.

Nous allons donc avoir besoin d'un second champ qui répétera le champs lié au mot de passe et sur lequel on devra s'assurer que ce qui y est saisit soit identique au premier champs.

Nous utiliserons donc le RepeatedType et le PasswordType ainsi que les options utilisées pour le champ email.

```
->add( child: 'password',  type: RepeatedType::class, [
    'type' => PasswordType::class,
    'invalid_message' => 'Le mot de passe et la confirmation doivent être identiques',
    'required' => true,
    'first_options' => [
        'label' => 'Votre mot de passe',
        'attr' => [
            'placeholder' => 'Merci de saisir votre mot de passe',
            'class' => 'form-control my-2'
        ]
    ],
    'second_options' => [
        'label' => 'Confirmez votre mot de passe',
        'attr' => [
            'placeholder' => 'Merci de confirmer votre mot de passe',
            'class' => 'form-control'
        ]
    ],
])
```

Ce qu'il faut comprendre ici c'est que 'first_options' représente le premier champs et 'second_options' représente le deuxième champ, celui répété.

LE BOUTON SUBMIT

Pour le bouton de soumission du formulaire nous utiliserons un nouveau Form Field Type de Symfony, le `SubmitType`.

```
->add( child: 'submit',  type: SubmitType::class, [
    'label' => 'S\'inscrire',
    'attr' => [
        'class' => 'btn btn-lg btn-outline-success mt-5'
    ]
])
```

CRÉER LE FORMULAIRE ET LE PASSER Á LA VUE

Maintenant que nous avons créer le formulaire, il est temps de l'afficher à notre utilisateur. Comme nous travaillons en MVC, il va falloir utiliser le controller afin de passer le formulaire à la vue.

CREATION DU FORMULAIRE ET GESTION DE LA REQUETE :

```
/**
 * Permet à un utilisateur anonyme de s'inscrire
 * @param Request $request
 * @param UserPasswordHasherInterface $encoder
 * @return Response
 * @throws Exception
 */
#[Route('/inscription', name: 'register')]
public function index(Request $request, UserPasswordHasherInterface $encoder): Response
{
    // On initialise une variable notification à null
    $notification = null;
    // On créé un nouvel utilisateur
    $user = new User();

    // On créé le formulaire
    $form = $this->createForm( type: RegisterType::class, $user);

    // On récupère la requête
    $form->handleRequest($request);
```

VERIFICATION ET TRAITEMENT DU FORMULAIRE :

```php
// Ici on effectue le traitement du formulaire si il
// est soumit et si il est valide
if($form->isSubmitted() && $form->isValid()){
    // On récupère les données saisies dans le formulaire
    $user = $form->getData();

    // On va vérifier que l'email saisit n'existe pas en base de données
    $search_email = $this->entityManager->getRepository(User::class)->findOneByEmail($user->getEmail());

    // Si l'email n'existe pas
    if(!$search_email){
        // On gère la sécurité avec l'encodage des mots de passe
        $password = $encoder->hashPassword( $user, $user->getPassword() );

        // On ré injecte le mot de passe crypté
        $user->setPassword($password);

        // On persiste les données en base(uniquement lors de la création)
        $this->entityManager->persist($user);
        // On enregistre les données en base
        $this->entityManager->flush();

        // Ici on ajoute un flash pour confirmer la création du compte.
        $this->addFlash( type: 'success', message: 'Vous êtes désormais inscrit vous pouvez vous connecter' );
        return $this->redirectToRoute( route: 'app_login');
    }else {
        // Si l'email existe déjà nous enregistrons le message dans la variable notification
        $notification = 'L\'email que vous avez renseigné existe déjà';
    }
}

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

Vous avez le détail de chaque étape, la finalité ce sont les variables que nous passons à notre vue. Les variables doivent être dans un tableau. Ici on passe à la vue, la variable form qui contient notre formulaire et notre variable notification.

AFFICHER LE FORMULAIRE DANS LA VUE

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

{% block title %}Inscription - My Blog{% endblock %}

{% block content %}
    <div class="container-md mt-5 pt-5">
        <div class="jumbotron bg-primary text-light">
            <h2>Vous souhaitez vous inscrire !</h2>
            <p>
                Être inscrit vous permet d'être alerté lors de la sortie d'un nouvel article et de commenter les articles.
                De plus certains articles ne sont accessibles qu'aux membres enregistrés sur le blog.
            </p>
        </div>
        <section class="d-flex justify-content-center align-items-center">
            <!-- Formulaire d'inscription -->
            <div class="w-75 mx-auto">
                {{ form(form) }}
            </div>
        </section>
    </div>
{% endblock %}
```

Nous pouvons maintenant tester notre formulaire, vérifier que l'utilisateur est bien enregistré avec PhpMyAdmin. Cependant lorsque vous allez tenter de vous connecter avec ce nouveau compte vous recevrez cette erreur de Symfony :

TODO: provide a valid redirect inside /Applications/XAMPP/xamppfiles/htdocs/Cours/Symfony6-Php8/my_blog/src/Security/AppAuthenticator.php

Pas de panique on ne peut pas faire plus explicite comme message d'erreur, il faut juste préciser à Symfony vers quelle url on redirige l'utilisateur en cas de succès de connexion. Ici nous redirigerons sur la homepage dans le fichier AppAuthenticator.

```php
public function onAuthenticationSuccess(Request $request, TokenInterface $token, string $firewallName): ?Response
{
    if ($targetPath = $this->getTargetPath($request->getSession(), $firewallName)) {
        return new RedirectResponse($targetPath);
    }

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

Voilà nous avons mis en place un système d'authentification, certes en l'état il est basique en termes de possibilités mais en quelques lignes de commande et de codes nous avons une authentification sécurisée. On pourrait aller beaucoup plus loin, activer le remember me, ajouter une image d'avatar possible, ajouter un captcha, ajouter la possibilité de se

connecter via réseaux sociaux. Tout est possible, mais vous avez ici une base d'implémentation de l'authentification avec Symfony.

UTILISATION DES FIXTURES

Á cette étape de notre projet nous allons souhaiter mettre en place l'administration du site afin de pouvoir créer le contenu du site. Pour se faire nous allons avoir besoin d'un compte administrateur.

LE SYSTÈME DE ROLES

Lorsque nous avons créé notre système **d'authentification** et l'entité **User,** nous avons vu une propriété `roles` pour cette entité. En effet Symfony embarque un **système de rôles** qui permet de rendre des parties du site visible selon le rôle par exemple. Ici nous aurons besoin d'un rôle administrateur ADMIN. Mais vous pourriez tout à fait imaginer un système de roles complexe, comme MANAGER, ADMIN, STAFF dans le cadre d'un site pour un restaurant de type fast-food.

ATTENTION : la propriété role correspond à un tableau de roles.

UTILISATION DU BUNDLE DES FIXTURES DE DOCTRINE

Afin de facilement créer un utilisateur ayant le role ADMIN, nous allons utiliser un package qui permet de créer des entrées dans la base de données, c'est donc un bundle Doctrine. Les fixtures sont aussi utilisées lors de la réalisation des tests. Nous allons installer le package dans l'environnement dev de travail. Notez qu'il est possible d'installer ce package en production et ainsi charger des comptes établis lors de la mise en ligne du projet en production.

Lien vers la documentation du bundle

Installation du package, --dev veut dire dans l'environnement de travail de développement :

```
composer require --dev orm-fixtures
```

Nous pouvons maintenant aller voir le fichier créé dans le dossier DataFixtures. Nous allons créer un utilisateur qui sera notre administrateur.

Vous noterez que tout comme lors de l'inscription il faut s'occuper de hasher le mot de passe. Nous faisons à nouveau appel à la `UserPaswwordHasherInterface`

```php
private UserPasswordHasherInterface $encoder;

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

public function load(ObjectManager $manager): void
{
    $adminUser = new User();

    $adminUser->setEmail( email: 'admin@gmail.com')
        ->setPassword($this->encoder->hashPassword($adminUser, plainPassword: 'admin'))
        ->setRoles(["ROLE_ADMIN"]);
    $manager->persist($adminUser);

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

Ensuite nous utiliserons une ligne de commande liée au bundle et nous répondrons oui à la mise en garde :

```
➜  my_blog git:(master) ✗ symfony console doctrine:fixtures:load
```

On peut aller constater dans PhpMyAdmin que notre administrateur est créé.

CRÉATION DE LA PARTIE ADMINISTRATION

CRÉATION DU CONTROLLER

Nous allons une fois de plus créer un controller, il se nommera `AdminDashBoardController`, la route sera sur `/admin` et s'apellera `admin_dashboard`.

Afin de bien segmenter notre code et avoir une architecture strcturée, nous rangerons les controllers liés à l'administration dans un dossier Admin, et les templates liés à l'administration dans un dossier Admin également.

ATTENTION aux namespaces et au chemin du render pour le fichier html.twig.

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

{% block title %}Adminisitration - My Blog{% endblock %}

{% block content %}
    <h2>Dashboard Administration</h2>
{% endblock %}
```

```php
<?php

namespace App\Controller\Admin;

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

class AdminDashBoardController extends AbstractController
{
    /**
     * Permet d'afficher le dashboard de l'espace administration
     * @return Response
     */
    #[Route('/admin', name: 'admin_dashboard')]
    public function index(): Response
    {
        return $this->render( view: 'Admin/dashboard/index.html.twig');
    }
}
```

Allons sur */admin*, sans surprise notre page s'affiche. Cependant on se rend bien compte que l'on ne va plus pouvoir étendre de base.html.twig car il contient le header et le footer côté utilisateur. Nous allons devoir créer un nouveau fichier `base.html.twig` dans le dossier Admin et étendre de ce fichier dans nos templates liés à l'administration.

Il faudra que cette fois nous ayons un menu composé de Dashboard, Thèmes, Articles. Il est aussi intéressant de changer le thème de couleur afin de bien identifier où l'on se trouve, côté utilisateur et admin. On ajoutera les partials liés à l'administration dans un dossier

Admin qui sera lui-même dans le dossier partials.Ici nous n'utiliserons qu'une nav en partials, pour la partie administration nous nous passerons de footer et de hero section.

Voici la navigation mise en place :

```
<header>
    <nav class="navbar navbar-expand-lg navbar-dark bg-primary">
        <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarTogglerDemo01" aria-controls
            <span class="navbar-toggler-icon"></span>
        </button>
        <div class="collapse navbar-collapse" id="navbarTogglerDemo01">
            <a class="navbar-brand ml-md-5 text-white" href="#"><h1>Administration My Blog</h1></a>
            <ul class="navbar-nav mr-auto ml-5 mt-2 mt-lg-0">
                <li class="nav-item active">
                    <a class="nav-link" href="#">Dashboard <span class="sr-only">(current)</span></a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="#">Thèmes</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="#">Articles</a>
                </li>
            </ul>
        </div>
    </nav>
</header>
```

Voici le fichier admin_base.html.twig :

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <meta name="description" content="Site de type blog pour découvrir et apprendre Symfony 6 avec Php 8">
    <!-- Bootstrap core CSS -->
    <link href="{{ asset('./assets/css/bootstrap.min.css') }}" rel="stylesheet">
    <!-- Fichier css custom -->
    <link rel="stylesheet" href="{{ asset('./assets/css/base.css') }}"/>
    <title>{% block title %}Administration - My Blog{% endblock %}</title>
    <link rel="icon" href="data:image/svg+xml,<svg xmlns=%22http://www.w3.org/2000/svg%22 viewBox=%220 0 128 12
    {# Run `composer require symfony/webpack-encore-bundle` to start using Symfony UX #}
    {% block stylesheets %}
        {{ encore_entry_link_tags('app') }}
    {% endblock %}

    {% block javascripts %}
        {{ encore_entry_script_tags('app') }}
    {% endblock %}
</head>
<body>
<!-- La navigation administration -->
{% include './partials/Admin/admin_nav.html.twig' %}
<main>
    <!-- Block content avec le contenu du main -->
    {% block content %}{% endblock %}
</main>
<!-- Bootstrap core Js -->
<script src="{{ asset('assets/js/bootstrap.bundle.min.js') }}"></script>
</body>
</html>
```

SÉCURISER L'ACCÈS Á L'ADMINISTRATION

Il est évident que l'on va devoir **sécuriser l'accès** aux pages d'administration, et que ces pages ne soient accessibles **qu'aux** utilisateurs ayant le rôle **ADMIN**. Nous allons donc utiliser les access-control du fichier `security.yaml` qui se trouve dans le dossier `config` puis `packages`.

Lien vers la documentation du access-control

Nous allons juste le modifier comme cela :

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

Symfony avait tout prévu pour nous et nous n'avons qu'une ligne à dé-commenter afin d'obtenir le résultat souhaité. En actualisant la page /admin vous constaterez que l'on est redirigé vers notre formulaire de login. Nous allons mettre en place ici un formulaire de login propre à l'administration.

Nous allons donc devoir mettre en place un nouveau controller afin de gérer la connexion à l'administration. Nous appellerons ce fichier AdminAccountController, nous organiserons à nouveau nos fichiers ainsi créés comme précédemment.

Ce nouveau controller aura **deux fonctions** une **login** et une **logout** à l'image de ce que nous avons déjà dans le fichier `SecurityController`.

Voici le controller modifié et la fonction login initialisée, cependant on va se rendre compte que les modifications sur le `access-control` fait que l'on ne peut pas accéder à la route pour se connecter. Nous allons donc ensuite modifier ces acces-control afin de pouvoir accéder à `/admin/connexion`.

```php
namespace App\Controller\Admin;

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

class AdminAccountController extends AbstractController
{
    /**
     * Permet de se connecter à l'administration
     * @return Response
     */
    #[Route('/admin/connexion', name: 'admin_account_login')]
    public function login(): Response
    {
        return $this->render( view: 'Admin/account/index.html.twig');
    }
}
```

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

CRÉER UN FIREWALL

Á cette étape nous avons mis en place une page qui va permettre d'afficher un simple formulaire de connexion à l'administration sur /admin/connexion. Mais nous avons un problème car à l'heure actuelle quand on essaie de se rendre sur /admin on est redirigé vers le formulaire pour les utilisateurs du site.

Nous allons pouvoir mettre ça en place assez facilement en créant un nouveau **firewall** (pare-feu) dans le fichier security.yaml. Un firewall délimite des emplacements du site et la façon de les sécuriser, ici nous allons créer le firewall admin.

[Lien vers la doc Symfony les firewalls]

Nous devons donner à notre firewall un **pattern,** ici tout ce qui commence par /admin**,** ensuite le **provider** d'où proviennent les utilisateurs ici le app_user_provider qui définit que les utilisateurs sont liés à **l'entité User** et que la propriété pour se loguer est **l'email**.

Enfin nous devons lui préciser les **chemins** pour accéder au **formulaire de login,** et pour se **déconnecter** ainsi que **L'URL de redirection** après la déconnexion.

ATTENTION IL FAUT BIEN COMPRENDRE QU'ICI L'ORDRE DES FIREWALLS EST TRES IMPORTANT.

Ici, si l'on avait mit le main avant le admin, nous aurions fait en sorte que peu importe L'URL on est dans le firewall main et notre firewall admin n'aurait jamais été pris en compte.

MISE EN PLACE DU FORMULAIRE DE LOGIN

Nous allons commencer par renommer le fichier `Admin/account/index.html.twig` en `login.html.twig` et nous allons effacer tout le contenu du template. En effet ici l'idée est de mettre en place une structure HTML en embarquant Bootstrap voir Font-Awesome en CDN. Ce template devra afficher le formulaire de connexion.

Il faudra également mettre à jour notre controller associé à la vue afin de créer la fonction de **login** correspondant à la route `/admin/connexion`. Nous prendrons comme modèle le `SecurityController`.

Voici le **DOCTYPE** de la vue :

```
<!doctype html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
          content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-scale=1.0, minimum-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Connexion Administration - My Blog</title>
    <!-- Fav icon -->
    <link rel="icon" href="{{ asset('assets/icons/favicon.ico') }}"/>
    <!-- Fichier Bootstrap -->
    <link href="{{ asset('./assets/css/bootstrap.min.css') }}" rel="stylesheet">
    <!-- Font awesome cdn -->
    <script src="https://kit.fontawesome.com/4e44dc3df4.js" crossorigin="anonymous"></script>
    <!-- Fichier css custom -->
    <link rel="stylesheet" href="{{ asset('assets/css/base.css') }}"/>
</head>
```

Le contenu du body :

```twig
<div class="row center mt-md-5">
    <div class="col-md-6">
        <div class="card bg-light">
            <div class="card-header">
                <h4 class="card-title text-center">
                    <i class="fas fa-lock"></i>
                    Connexion à l'administration My Blog
                </h4>
            </div>
            <div class="card-body">
                {% if hasError %}
                    <div class="alert-warning">
                        <p>Les informations saisies ne correspondent pas</p>
                    </div>
                {% endif %}
                <form method="post">
                    <div class="form-group">
                        <label for="_username">Email</label>
                        <input type="text" class="form-control" placeholder="Adresse email" name="_username" value="{{ username }}"/>
                    </div>
                    <div class="form-group">
                        <label for="_password">Mot de passe</label>
                        <input type="password" class="form-control" placeholder="Mot de passe" name="_password" />
                    </div>
                    <div class="form-group mt-3">
                        <button class="btn my-2 btn-primary">
                            <i class="fas fa-lock-open"></i>
                            Connexion
                        </button>
                        <a href="{{ path('homepage') }}" class="btn my-2 btn-link">
                            <i class="fas fa-arrow-circle-left"></i>
                            Retour au site
                        </a>
                    </div>
                </form>
            </div>
        </div>
    </div>
</div>
```

Le controller et la fonction login :

```php
namespace App\Controller\Admin;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\Security\Http\Authentication\AuthenticationUtils;

class AdminAccountController extends AbstractController
{
    /**
     * Permet de se connecter à l'administration
     * @param AuthenticationUtils $utils
     * @return Response
     */
    #[Route('/admin/connexion', name: 'admin_account_login')]
    public function login(AuthenticationUtils $utils): Response
    {
        $error = $utils->getLastAuthenticationError();
        $username = $utils->getLastUsername();

        return $this->render( view: 'Admin/Account/login.html.twig', [
            'hasError' => $error !== null,
            'username' => $username
        ]);
    }
}
```

PERMETTRE LA DÉCONNEXION

Il ne nous reste plus qu'à implémenter notre route et la fonction associée afin de se déconnecter en tant qu'admin sur /admin/deconnexion. Une fois de plus nous prendrons comme modèle le SecurityController. Cette méthode ne doit implémenter aucune logique mais doit juste exister afin de coller à notre fichier security.yaml.

```php
/**
 * Permet de déconnecter l'administrateur
 */
#[Route('/admin/deconnexion', name: 'admin_account_logout')]
public function logout(): void
{
    throw new LogicException( message: 'This method can be blank - it will be intercepted by the logout key on your firewall.');
}
```

Nous pouvons dès maintenant tester notre connexion à l'administration, tout fonctionne comme souhaité.

ALLER PLUS LOIN

La mise en place que nous venons de réaliser reste basique et nécessiterait d'être approfondie, mettre en place un système anti-robot sur le formulaire, verrouiller les données saisies par l'utilisateur en effectuant des contraintes de validation. Nous verrons ce sujet lors de chapitres suivants.

NOTRE PREMIER CRUD

LE MAKER

Comme à son habitude Symfony fournit grâce à son **maker** un outil permettant de **créer un crud sur une entité**. Cela vous créé les templates et les méthodes nécessaires.

Cependant parce qu'il est important de comprendre comment fonctionne le framework et aussi afin de garder la main, par exemple ne pas laisser la possibilité de supprimer un thème, nous n'allons cette fois pas utiliser le maker de Symfony.

CRUD THEMES

Nous allons donc créer un crud pour les thèmes dans la partie administration, il nous faudra donc une route pour afficher les thèmes, une pour les créer, une pour les éditer et enfin une pour les modifier.
Nous n'aurons pas ici besoin d'une route afin d'afficher les détails d'un thème car un thème ne comporte qu'un titre et une illustration.

LA ROUTE DU READ

Nous allons utiliser cette route index afin d'afficher les thèmes présents en base de données afin de pouvoir proposer ensuite à l'administrateur d'agir dessus (update et delete).

Il faudra donc rédiger le code qui permettra de récupérer nos thèmes, on utilisera **l'EntityManagerInterface** de Symfony qui permet de récupérer les données d'une entité. Afin d'éviter de répéter l'injection de dépendance dans chaque fonction du controller nous utiliserons ici un constructeur afin de rendre disponible notre entitymanager utilisable dans toutes nos fonctions.

Le template lui n'affichera pour le moment qu'un titre "Les thèmes". Nous allons utiliser le **maker** afin de créer le `AdminThemesController`, il faudra penser à réorganiser les dossier pour coller à ce que nous avons mis en place en terme d'architecture.

Voici le controller et la fonction associée au READ page suivante.

```php
namespace App\Controller\Admin;

use App\Entity\Themes;
use Doctrine\ORM\EntityManagerInterface;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class AdminThemesController extends AbstractController
{
    private EntityManagerInterface $entityManager;

    /**
     * AdminThemesController constructor.
     * @param EntityManagerInterface $entityManager
     */
    public function __construct(EntityManagerInterface $entityManager)
    {
        $this->entityManager = $entityManager;
    }

    /**
     * Permet d'afficher les thèmes dans la partie Administration
     * @return Response
     */
    #[Route('/admin/themes', name: 'admin_themes')]
    public function index(): Response
    {
        // Nous utilisons l'entityManager pour récupérer tous les thèmes en base
        $themes = $this->entityManager->getRepository(Themes::class)->findAll();

        return $this->render( view: 'Admin/themes/index.html.twig', [
            'themes' => $themes,
        ]);
    }
}
```

Voici le template associé à la route :

```twig
{% extends 'Admin/admin_base.html.twig' %}

{% block title %}Administration Thèmes  - My Blog{% endblock %}

{% block content %}
<h2>Les thèmes</h2>
{% endblock %}
```

Nous allons maintenant nous occuper du CREATE afin de pouvoir saisir nos premiers thèmes et ainsi pouvoir travailler l'affichage de ceux-ci sur la route du READ.

LE ROUTE DU CREATE

Il va donc falloir dans un premier temps s'occuper du formulaire associé à l'entité Themes, nous utiliserons le maker afin de le générer. Nous considérerons que l'illustration n'est ni plus ni moins qu'une url menant à une image et donc de type text Field. Nous verrons l'upload d'images par la suite.

Rappel, le champs slug se remplira automatiquement lors de la création d'un Themes grâce à notre service Slugify.

Voici le formulaire initialisé grâce au maker et customisé page suivante.

```php
public function buildForm(FormBuilderInterface $builder, array $options): void
{
    $builder
        ->add( child: 'title', type: TextType::class, [
            'label' => 'Nom du thème',
            'attr' => [
                'placeholder' => 'Nom du thème',
                'class' => 'form-control my-2'
            ],
            'label_attr' => [
                'class' => 'text-info'
            ]
        ])
        ->add( child: 'illustration', type: TextType::class, [
            'label' => 'URL de l\'image',
            'attr' => [
                'placeholder' => 'URL menant à l\'image',
                'class' => 'form-control my-2'
            ],
            'label_attr' => [
                'class' => 'text-info'
            ]
        ])
        ->add( child: 'submit', type: SubmitType::class, [
            'label' => 'Valider',
            'attr' => [
                'class' => 'btn btn-lg btn-outline-success mt-5'
            ]
        ])
    ;
}
```

Voici la fonction dans le controller :

```php
/**
 * Permet la création d'un thème dans l'administration
 * @param Request $request
 * @return Response
 */
#[Route('/admin/themes/ajout', name: 'admin_themes_create')]
public function create(Request $request): Response
{
    $theme = new Themes();
    $form = $this->createForm( type: ThemesType::class, $theme);
    $form->handleRequest($request);

    if($form->isSubmitted() && $form->isValid()){
        $this->entityManager->persist($theme);
        $this->entityManager->flush();

        $this->addFlash(
            type: 'success_admin',
            message: "Le thème <strong>{$theme->getTitle()}</strong> a bien été enregistré ! "
        );

        return $this->redirectToRoute( route: 'admin_themes');
    }
    return $this->render( view: '/Admin/themes/new.html.twig', [
        'form' => $form->createView(),
    ]);
}
```

Enfin le template qui affiche le formulaire :

```twig
{% extends 'Admin/admin_base.html.twig' %}

{% block title %}Administration Créer Thème  - My Blog{% endblock %}

{% block content %}
    <div class="container-md mt-5">
        <h2>Création d'un nouveau thème</h2>
        <div class="mt-5">
            {{ form(form) }}
        </div>
    </div>
{% endblock %}
```

Vous pourrez noter la mise en place d'un Flash dans le controller avec la méthode AddFlash, cela va permettre de notifier à l'utilisateur le succès de la création. Nous utiliserons donc un rendu conditionnel dans notre template twig associé. Nous afficherons les thèmes dans un tableau avec le champ titre et les actions associées, UPDATE et DELETE.

Nous allons pouvoir créer nos premiers thèmes en allant sur l'URL `/admin/themes/ajout`, et ensuite nous occuper de notre template.

Gérer le flash grâce au composant Alert de Bootstrap

Nous allons activer jQuery afin de pouvoir l'utiliser avec Bootstrap, c'est nécessaire pour gérer la fermeture des alertes.

Il va falloir aussi mettre en place un affichage conditionnel avec Twig.

[Lien vers la doc pour le CDN de jQuery]

Nous allons sur la doc de Bootstrap afin de récupérer un template d'alerte. Nous allons ensuite l'afficher sous condition comme suit :

```
{% for message in app.flashes('success_admin') %}
    <div class="alert alert-success alert-dismissible fade show" role="alert">
        <button type="button" class="close" data-dismiss="alert" aria-label="Close">
            <span aria-hidden="true">&times;</span>
        </button>
        {{ message }}
    </div>
{% endfor %}
```

Voilà un affichage conditionnel couplé à une boucle. Notez bien ici la syntaxe car nous l'utiliserons tout au long de ce cours. Nous avons ici appelé ce message `'success_admin'`, il est dans les bonnes pratiques de donner un nom explicite à son message d'erreur afin d'optimiser ensuite leurs affichages. Nos messages flash ici sont contenus dans la variable **flashes** du **app**. Nous pouvons utiliser la fonction **dump** de twig afin d'afficher l'ensemble des variables auxquelles nous pouvons accéder.

Nous pouvons ensuite afficher un tableau Bootstrap en bouclant sur notre variable `themes`.

```
<h2 class="my-5">Les thèmes du blog</h2>
<div class="table-responsive">
    <table class="table table-striped">
        <thead>
        <tr>
            <th class="text-center" scope="col">Titre</th>
            <th class="text-center" scope="col">Actions</th>
        </tr>
        </thead>
        <tbody>
        {% for theme in themes %}
        <tr>
            <td class="text-center">{{ theme.title }}</td>
            <td>
                <div class="row flex justify-content-center">
                    <div class="mx-2">
                        <a href="#" class="btn btn-warning"><i class="fas fa-edit mr-1"></i>Editer</a>
                    </div>
                    <div class="mx-2">
                        <a href="#" class="btn btn-danger"><i class="fas fa-trash mr-1"></i>Supprimer</a>
                    </div>
                </div>
            </td>
        </tr>
        {% endfor %}
        </tbody>
    </table>
</div>
```

J'ai ajouté les icônes FontAwesome afin de styliser mes boutons action.

Notez ici la syntaxe permettant de boucler et celle permettant d'afficher une variable.

Nous allons maintenant devoir mettre en place le UPDATE et le DELETE, nous allons devoir créer nos premières routes avec paramètre.

LA ROUTE DU UPDATE

Nous allons donc devoir créer notre première route avec paramètre, ici nous utiliserons le slug car même si nous sommes dans le back-office, il est dans les bonnes pratiques de rendre des url propres et explicites, SEO friendly.

Nous récupérerons également les données du thème modifié afin de personnaliser l'affichage dans le template. Le template d'ailleurs sera très similaire à celui du create et la fonction associée dans le controller également.

Nous mettrons en place une nouvelle alerte en choisissant la couleur warning cette fois et en choisissant un autre nom pour le message d'erreur.

```php
/**
 * Permet de modifier un thème dans l'administration
 * @param Themes $themes
 * @param Request $request
 * @param $slug
 * @return Response|RedirectResponse
 */
#[Route('/admin/themes/modifier/{slug}', name: 'admin_themes_edit')]
public function edit(Themes $themes, Request $request, $slug): RedirectResponse|Response
{
    $theme = $this->entityManager->getRepository(Themes::class)->findOneBySlug($slug);

    $form = $this->createForm( type: ThemesType::class, $themes);
    $form->handleRequest($request);

    if($form->isSubmitted() && $form->isValid()){
        $this->entityManager->flush();

        $this->addFlash(
            type: 'admin_edit_success',
            message: "Le thème {$themes->getTitle()} a bien été modifiée ! "
        );

        return $this->redirectToRoute( route: 'admin_themes');
    }

    return $this->render( view: '/Admin/themes/edit.html.twig', [
        'form' => $form->createView(),
        'theme' => $theme
    ]);
}
```

Voici l'affichage du message flash dans le fichier index.html.twig :

```twig
{% for message in app.flashes('admin_edit_success') %}
    <div class="alert alert-warning alert-dismissible fade show" role="alert">
        <button type="button" class="close" data-dismiss="alert" aria-label="Close">
            <span aria-hidden="true">&times;</span>
        </button>
        {{ message }}
    </div>
{% endfor %}
```

Et enfin le template edit.html.twig :

```twig
{% extends 'Admin/admin_base.html.twig' %}

{% block title %}Administration Modifier Thème  - My Blog{% endblock %}

{% block content %}
    <div class="container-md mt-5">
        <h2>Modification du thème : {{ theme.title }}</h2>
        <div class="mt-5">
            {{ form(form) }}
        </div>
    </div>
{% endblock %}
```

Voilà nous avons mis en place le UPDATE il ne reste plus qu'à s'occuper du delete, ce sera plus simple car ce sera juste une route qui ne renverra rien hormis le message flash de succès ainsi qu'une redirection.

LA ROUTE DU DELETE

Nous mettrons en place un nouveau flash cette fois en danger pour le thème afin d'alerter de la suppression effectuée.

Voici la route et la fonction associée :

```php
/**
 * Permet de supprimer un thème dans l'administration
 * @param Themes $themes
 * @return RedirectResponse
 */
#[Route('/admin/themes/supprimer/{slug}', name: 'admin_themes_delete')]
public function delete(Themes $themes): RedirectResponse
{
    $this->entityManager->remove($themes);
    $this->entityManager->flush();

    $this->addFlash(
        type: 'admin_delete_success',
        message: 'Le thème a bien été supprimé'
    );

    return $this->redirectToRoute( route: 'admin_themes');
}
```

Nous n'avons plus qu'à mettre à jour les liens de nos boutons actions comme suit :

```
<div class="row flex justify-content-center">
    <div class="mx-2">
        <a href="{{ path('admin_themes_edit', {'slug': theme.slug}) }}" class="btn btn-warning"><i class="fas fa-edit mr-1"></i>Editer</a>
    </div>
    <div class="mx-2">
        <a href="{{ path('admin_themes_delete', {'slug': theme.slug}) }}" class="btn btn-danger"><i class="fas fa-trash mr-1"></i>Supprimer</a>
    </div>
</div>
```

Notre système de slug comporte en l'état un gros problème, que se passera-t-il lorsque deux thèmes porteront le même nom ?

Nous allons résoudre ce problème et aborder le concept de contrainte de validation dans Symfony lors du prochain chapitre.

UPDATE DU SYSTEME DE SLUG

Nous nous rendons vite compte que notre système de slug va rencontrer un problème.

Nous allons donc devoir faire en sorte que chaque titre de thème soit unique. Pour se faire nous allons ajouter une contrainte de validation sur le champ.

Les contraintes de validation sont très importantes en termes de sécurité, nous aborderons cette notion en détail lors d'un prochain chapitre. Pour le moment nous allons juste mettre en place une **contrainte** appelée **UniqueEntity**.

Nous en profiterons également pour **passer les annotations en attributs** afin de respecter les bonnes pratiques de **Php8**, le maker nous créé toujours les fichiers sous forme d'annotations, cependant la doc recommande bien l'usage des attributs.

```
/**
 * @ORM\Entity(repositoryClass=ThemesRepository::class)
 */
#[UniqueEntity('title', message: 'un thème porte déjà ce titre !')]
#[ORM\HasLifecycleCallbacks()]
```

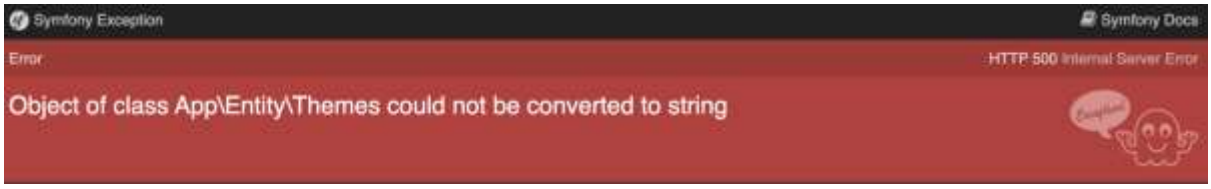
Voilà avec cette simple ligne nous venons de rendre le champs title unique.

TP3 CRÉER LE CRUD DES ARTICLES

Vous allez devoir, à l'image de ce que nous avons fait pour les thèmes, implanter un controller qui se nommera AdminArticlesController pour les **Articles** et créer le CRUD correspondant. Une fois n'est pas coutume il faudra respecter la structure de nos dossiers.

Il faudra utiliser la documentation sur le **FieldType** de symfony afin de réaliser le formulaire grâce au maker sur l'entité Articles. Il y a une subtilité pour le champ qui représentera le choix du thème car les thèmes sont une entité mais vous trouverez la solution facilement dans la documentation. De plus il faudra, grâce à la méthode __toString au sein de l'entité Themes, rendre le champs title du thème lisible dans le formulaire de l'article.

Sinon vous aurez cette erreur :



Nous aurons donc 4 routes et donc 4 fonctions associées :

- route : /admin/articles , nom : admin_articles , fonction index qui affichera le tableau d'articles avec le titre et le nom du thème ainsi que les actions
- route : /admin/articles/ajout, nom : admin_articles_create, fonction create qui permettra de créer un article
- route : /admin/articles/modifier/{slug}, nom : admin_articles_edit, fonction edit qui permettra de modifier un article déjà existant.
- route : /admin/articles/supprimer/{slug}, nom : admin_articles_delete, fonction delete qui permettra de supprimer un article de la base de donnée.

ATTENTION : comme pour le crud des thèmes il va falloir régler le problème du slug unique donc du titre.

DIFFICULTÉ :

Il va falloir traiter le champs createdAt lors du traitement du formulaire. Regardez du côté de la méthode `setCreatedAt()`

MISE EN PLACE UI

Il est temps maintenant de finaliser notre partie utilisateur. Nous allons mettre en place le contenu pour la homepage, créer un controller pour afficher le blog qui aura deux routes, une pour afficher les articles et une pour afficher un article en détail.

Dans la partie précédente nousa vons juste ajouter des liens dans la barre de navigation en utilisant l'affichage conditionnel et la variable app.

Nous allons commencer par mettre en place le contenu de la homepage.

LA HOMEPAGE

Elle va devoir contenir une section pour le dernier article créé, une pour le dernier thème créé et enfin une dernière section pour lister les thèmes sous forme de bouton.

Nous pouvons imaginer afficher aussi une section about et un lien menant à cette page, une section newsletter qui est de coutume dans un blog.

Pour le moment nous remplirons notre design avec des données en dur, image random et lorem ipsum. Une fois le design mit en place avec Bootstrap nous nous occuperons de fournir de vraies données au template.

FOURNIR A LA PAGE LES VRAIES DONNEES EN DEUX REQUETES ET GRACE AUX FILTRES TWIG.

Nous allons utiliser dans notre HomePageController l'injection de dépendances car nous n'avons qu'une seule fonction au sein de ce controller. Rien de neuf, voici le controller avec les deux requêtes afin de récupérer nos articles et nos thèmes. Ici le but est de donner accès au front à tous les thèmes et articles afin de pouvoir tout envisager en terme d'affichage.

```php
/**
 * Permet d'afficher la page d'accueil
 * @param EntityManagerInterface $entityManager
 * @return Response
 */
#[Route('/', name: 'homepage')]
public function index(EntityManagerInterface $entityManager): Response
{
    $articles = $entityManager->getRepository(Articles::class)->findAll();
    $themes = $entityManager->getRepository(Themes::class)->findAll();

    return $this->render( view: 'homepage/index.html.twig', [
        'articles' => $articles,
        'themes' => $themes
    ]);
}
```

Maintenant nous avons accès à nos thèmes et nos articles sur la homepage. Voici comment afficher l'article et le thème portant le dernier id en base. Notez qu'afin d'optimiser le système il serait bon de mettre en place un createdAt automatique sur chaque entité. Ici nous mettrons en place par id pour l'entité Themes et par date de création pour les Articles.

Voici la mise en place au sein du template pour le dernier article :

```twig
<!-- last article start -->
{% for last_article in articles|sort((a, b) => b.createdAt <=> a.createdAt)|slice(0,1) %}
<div class="col-lg-8 col-sm-12">
    <div class="about_img"><img alt="about me" src="{{ asset('./assets/img/about-img.png') }}"></div>
    <div class="like_icon"><img alt="like" src="{{ asset('./assets/img/like-icon.png') }}"></div>
    <p class="post_text">Posté le : {{ last_article.createdAt|date('d-m-Y') }}</p>
    <h2 class="most_text">{{ last_article.title }}</h2>
    <p class="lorem_text">{{ last_article.content }}</p>
    <div class="social_icon_main">
        <div class="social_icon">
            <ul class="list-unstyled d-flex justify-content-center">
                <li class="mx-5"><a href="#"><img src="{{ asset('./assets/img/fb-icon.png') }}"></a></li>
                <li class="mx-5"><a href="#"><img src="{{ asset('./assets/img/twitter-icon.png') }}"></a></li>
                <li class="mx-5"><a href="#"><img src="{{ asset('./assets/img/instagram-icon.png') }}"></a></li>
            </ul>
        </div>
        <div class="read_bt"><a href="#">Read More</a></div>
    </div>
</div>
{% endfor %}
<!-- last article end -->
```

Vous noterez ici l'utilisation de deux nouveaux filtres Twig, sort qui permet de trier les données en fonction d'un champ ici createdAt et d'indiquer l'ordre grâce aux repères a et b, et slice qui permet de selectionner un nombre de résultat à afficher.

Voici la mise en place pour le dernier thème :

```
<!-- last themes start -->
{% for last_theme in themes|sort((a, b) => b.id <=> a.id)|slice(0,1) %}
<div class="col-lg-8 col-sm-12">
    <div class="about_img"><img alt="about me" class="mx-auto w-50" src="{{ last_theme.illustration }}"></div>
    <div class="like_icon"><img alt="like" src="{{ asset('./assets/img/like-icon.png') }}"></div>
    <h2 class="most_text">{{ last_theme.title }}</h2>
    <p class="lorem_text">{{ last_theme.articles|length }} Articles</p>
    <div class="social_icon_main">
        <div class="social_icon">
            <ul class="list-unstyled d-flex justify-content-center">
                <li class="mx-5"><a href="#"><img src="{{ asset('./assets/img/fb-icon.png') }}"></a></li>
                <li class="mx-5"><a href="#"><img src="{{ asset('./assets/img/twitter-icon.png') }}"></a></li>
                <li class="mx-5"><a href="#"><img src="{{ asset('./assets/img/instagram-icon.png') }}"></a></li>
            </ul>
        </div>
        <div class="read_bt"><a href="#">Read More</a></div>
    </div>
</div>
{% endfor %}
<!-- last theme end -->
```

Comme nous avons accès à tous nos thèmes, nous pouvons facilement mettre en place une boucle sur cette entité Themes et afficher chaque thème sous forme de bouton. Nous nous occuperons de l'affichage des articles par thème plus tard, le lien sera un # pour le moment.

```
<div class="tag_bt">
    <ul class="list-unstyled row">
        {% for theme in themes %}
        <li class="col-4 col-md-3 col-lg-2"><a class="btn btn-primary" href="#">{{ theme.title|capitalize }}</a></li>
        {% endfor %}
    </ul>
</div>
```

Nous en avons fini pour la partie HomePage du site, libre par la suite d'ajouter des sections comme les 5 derniers articles, les thèmes avec le plus d'articles.

Nous allons maintenant mettre en place la partie blog du site qui va permettre d'afficher les articles, les thèmes, les articles par thème.

LE BLOG

Au sein de notre blog controller nous aurons besoin de deux routes :

- Une route sur /blog qui affichera tous les articles et en haut de page la liste des thèmes. Elle se nommera 'blog' et devra donc fournir au template index.html.twig les articles et les thèmes. Nous classerons les articles du plus récent au plus ancien.
- Une seconde route sur /blog/themes/{slug} qui affichera les articles d'un thème. Elle se nommera 'blog_themes' et devra fournir au template show.html.twig les articles appartenant à un thème.

Voici la route /blog :

```php
private EntityManagerInterface $entityManager;

/**
 * BlogController constructor.
 * @param EntityManagerInterface $entityManager
 */
public function __construct(EntityManagerInterface $entityManager)
{
    $this->entityManager = $entityManager;
}

/**
 * Permet d'afficher la liste des thèmes et des articles
 * @return Response
 */
#[Route('/blog', name: 'blog')]
public function index(): Response
{
    $articles = $this->entityManager->getRepository(Articles::class)->findAll();
    $themes = $this->entityManager->getRepository(Themes::class)->findAll();

    return $this->render( view: 'blog/index.html.twig', [
        'articles' => $articles,
        'themes' => $themes
    ]);
}
```

La route /blog/themes/{slug} :

```php
/**
 * Permet d'afficher les articles par thème
 */
#[Route('/blog/themes/{slug}', name: 'blog_themes')]
public function show($slug): Response
{
    $articles_themes = $this->entityManager->getRepository(Themes::class)->findOneBySlug($slug);

    return $this->render( view: 'blog/show.html.twig', [
        'articles_themes' => $articles_themes
    ]);
}
```

Voici la partie template pour la page /blog :

La liste des thèmes sous forme de liens par boutons cliquables :

```twig
<!-- themes link section start -->
<div class="jumbotron-fluid bg-light m-4 p-5">
    <div class="m-3">
        {% for theme in themes %}
            <a href="{{ path('blog_themes', {'slug': theme.slug}) }}" class="btn btn-outline-primary">{{ theme.title }}</a>
        {% endfor %}
    </div>
</div>
<!-- themes link section end -->
```

Et la boucle pour afficher les articles :

```twig
<!-- blog section start -->
<div class="container-md mt-5">
    <div class="container">
        <div class="row">
            {% for article in articles|sort((a, b) => b.createdAt <=> a.createdAt) %}
            <div class="col-12 mt-5">
                <div class="d-flex justify-content-around">
                    <div>
                        <div class="about_img "><img class="rounded-lg img-fluid" alt="{{ article.themes.title }}" src="{{ article.themes.illustration }}"></div>
                        <div class="like_icon"><img alt="like" src="{{ asset('./assets/img/like-icon.png') }}"></div>
                    </div>
                    <div class="d-flex justify-content-center align-items-center">
                        <h5 class="text-center">Thème : {{ article.themes.title }}</h5>
                    </div>
                </div>
                <p class="post_text">Posté le : {{ article.createdAt|date('d-m-Y') }}</p>
                <h2 class="most_text">{{ article.title }}</h2>
                <p class="lorem_text">{{ article.content }}</p>
                <div class="social_icon_main">
                    <div class="social_icon">
                        <ul class="list-unstyled d-flex justify-content-center">
                            <li class="mx-5"><a href="#"><img src="{{ asset('./assets/img/fb-icon.png') }}"></a></li>
                            <li class="mx-5"><a href="#"><img src="{{ asset('./assets/img/twitter-icon.png') }}"></a></li>
                            <li class="mx-5"><a href="#"><img src="{{ asset('./assets/img/instagram-icon.png') }}"></a></li>
                        </ul>
                    </div>
                    <div class="read_bt"><a href="#">Read More</a></div>
                </div>
            </div>
            {% endfor %}
        </div>
    </div>
</div>
<!-- blog section end -->
```

Pour le template show.html.twig nous devons juste modifier la boucle :

```
<div class="row">
    {% for article in articles_themes.articles|sort((a, b) => b.createdAt <=> a.createdAt) %}
        <div class="col-12 mt-5">
            <div class="d-flex justify-content-around">
                <div>
                    <div class="about_img "><img class="rounded-lg img-fluid" alt="{{ article.themes.title }}" src="{{ article.themes.illustration }}"></div>
                    <div class="like_icon"><img alt="like" src="{{ asset('./assets/img/like-icon.png') }}"></div>
                </div>
                <div class="d-flex justify-content-center align-items-center">
                    <h3 class="text-center">Thème : {{ article.themes.title }}</h3>
                </div>
            </div>
            <p class="post_text">Posté le : {{ article.createdAt|date('d-m-Y') }}</p>
            <h2 class="most_text">{{ article.title }}</h2>
            <p class="lorem_text">{{ article.content }}</p>
            <div class="social_icon_main">
                <div class="social_icon">
                    <ul class="list-unstyled d-flex justify-content-center">
                        <li class="mx-5"><a href="#"><img src="{{ asset('./assets/img/fb-icon.png') }}"></a></li>
                        <li class="mx-5"><a href="#"><img src="{{ asset('./assets/img/twitter-icon.png') }}"></a></li>
                        <li class="mx-5"><a href="#"><img src="{{ asset('./assets/img/instagram-icon.png') }}"></a></li>
                    </ul>
                </div>
                <div class="read_bt"><a href="#">Read More</a></div>
            </div>
        </div>
    {% endfor %}
</div>
```

LA PAGE ABOUT

Cette page étant non dynamique, vous êtes libre de son contenu, l'idée est de pouvoir communiquer sur vous.

LA PAGE CONTACT

Comme pour la page about, cette page est non dynamique, elle doit juste comporter un formulaire de contact simple avec email, nom, prénom, objet et message. Pour le moment nous n'utiliserons pas le maker car ce formulaire ne se base sur aucune entité.

Nous verrons la mise en place réelle dans le cours sur les notions avancées et la gestion des envois d'email.

CONCLUSION

Nous venons de voir les bases du frameworks Symfony au travers la création d'un simple blog, permettant d'être administré, à l'utilisateur de s'inscrire sur le blog. Pour se faire nous avons appréhender :

- Les bases du moteur de template **Twig** (filtres, blocs, boucles, conditions)
- Utiliser **Bootstrap** dans un projet Symfony mais aussi FontAwesome
- L'architecture d'un projet Symfony et son organisation de type **MVC**
- Nous avons appris à mettre en place **une base de données SQL**
- Utilisation du maker de symfony pour créer controller, formulaires, entités, **système d'authentification**
- Authentification, sécurisation partie administration, création de CRUD
- Création d'un **service**
- Utilisation d'un **bundle** pour créer des fixtures

Dans le prochain cours nous aborderons des notions plus avancées en continuant notre projet de blog, nous mettrons en place l'upload des images, nous rendrons notre menu dynamique, nous utiliserons du Javascript dans Symfony, nous mettrons en place une pagination, un système de filtre de recherche, une gestion de son compte par l'utilisateur et de son mot de passe et bien d'autres notions.