Cours Symfony :

Le but de ce cours est de découvrir le framework Symfony est de pouvoir développer des applications web.

Nous verrons pendant la formation :

- L'initialisation d'un projet Symfony,
- Faire la configuration de la base de donnée avec l'ORM Doctrine,
- La configuration de la partie backend de l'application avec la gestion des controllers ainsi que les différentes routes de l'application,
- La partie développement du backend ainsi que la gestion de l'administration des données.

Prérequis pour la formation :

Il est important d'avoir certaines base de connaissance avant de débuter la formation Symfony :

- Connaissance en HTML/CSS/JS
- Connaissance en PHP
- La connaissance du SQL est un plus
- Savoir utiliser un terminal de commande

Configurer votre environnement :

VsCode :

Pour développer, je vous conseille l'IDE VsCode qui permet d'avoir une bonne vision sur l'arboresence dossier dans un projet, et qui permet également d'installer des plugin d'aide au développement qui nous seront utiles pour la formations. Pour télécharger le logiciel, cliquez ici

Github :

Nous allons travailler avec Github, ce qui va permettre de versionner notre code et pouvoir ainsi avoir une vision claire sur l'historique des modifications.

Vous devez dans un premier temps configurer votre environnement avec Git :

- **Pour les MACs** :
    - Ouvrez un terminal de commande et entrez la commande :

```
xcode-select —install
```

Une fenêtre va s'ouvrir vous demandant si vous voulez installer des outils de ligne de commande, cliquez sur installer

- Ensuite, vous devez configurer vos informations Git : *(Ne pas oublier les guillemets)*

```
git config --global user.name "Votre Nom et Prénom"
```

Ensuite : *(Ne pas mettre les guillemets)*

```
git config --global user.email votre email
```

Pour vérifier les informations, entrez la commande :

```
git config —list
```

Yarn

Yarn est un manager de paquets en ligne de commande qui va nous permettre d'installer et de gérer des composants de notre application symfony.

Pour l'installer sur vos postes, vous devez :

- Pour les macs
  - Ouvrir un terminal de commande
  - Entrez la commande :

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

    - Si vous avez une erreur, Attendre que Hombrew soit installer pour lancer la commande suivante et mettre le mot de passe de votre poste :

```
sudo chown -R $(whoami) /usr/local/share/zsh /usr/local/share/zsh/site-
founctions
```

  - Entrez la commande :

```
brew install yarn
```

- Pour les windows :
  - Vous devez avoir installé au préalable **node** pour avoir **npm** lien de la documentation.
  - Ensuite rentrez la commande dans un terminal :

```
npm install -g yarn
```

Installer votre stack

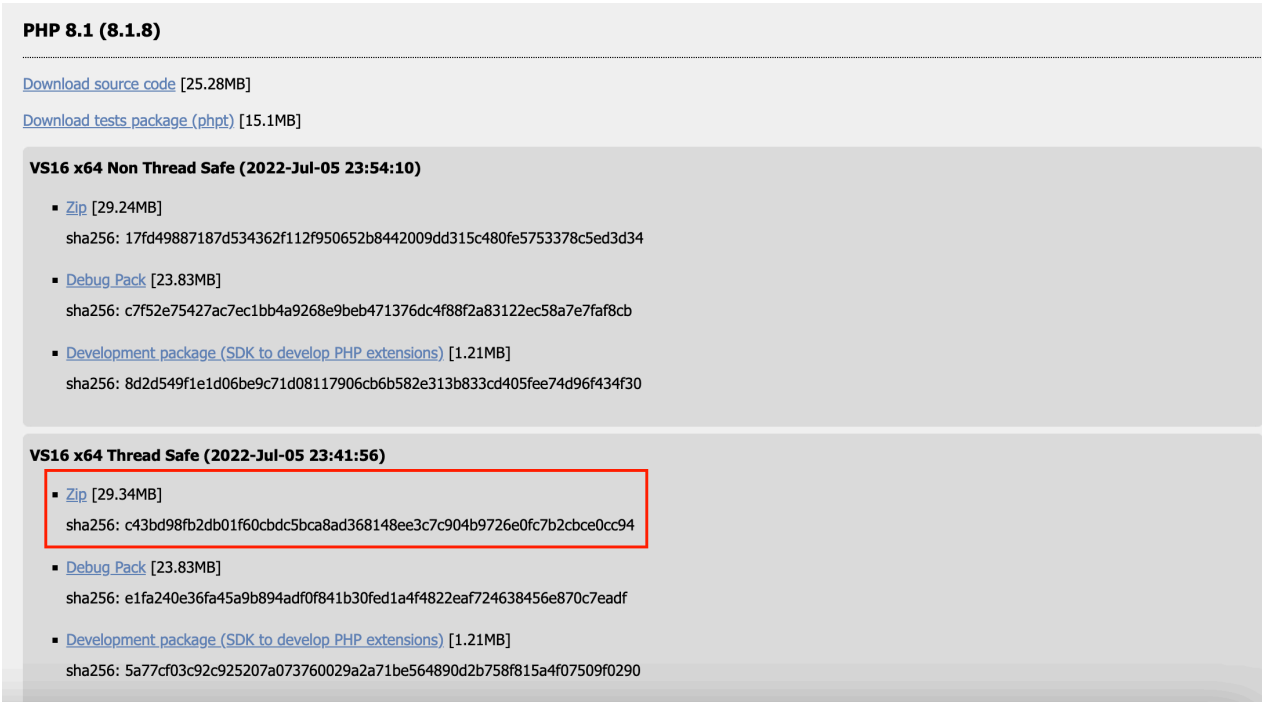
Une stack est un environnement de développement pour votre application, votre stack se compose de tout les composants indispensables pour faire tourner votre application.

Avec Symonfy 6, vous devez obligatoirement avec sur votre environnement PHP 8 au minimum.

PHP

Pour installer Php 8 avec un windows :

- Rendez-vous sur le site de Php pour télécharger le dossier
    - Vous devez télécharger la dernière version de Php en **Thread Safe** :



    - Un fois le dossier zip téléchargé vous allez devoir l'extraire à la racine de votre environnement **(dossier `C:\`)** dans un dossier que vous allez nommer `php` :

    - Ensuite vous allez devoir configurer le **php.ini**, rendez-vous dans le nouveau dossier php que vous avez créé et dupliquez le fichier `C:\php\php.ini-development` que vous allez renommer en `C:\php\php.ini`.

    - Maintenant vous allez devoir ouvrir ce nouveau fichier et ajouter les extensions indispensable pour utiliser Symfony, faite un ctrl + f pour chercher les lignes `Dynamic Extensions`, vous allez devoir décommenter certaines lignes pour ajouter les extensions utiles pour votre serveur PHP :

```
extension=curl
extension=fileinfo
extension=gd
extension=intl
extension=mbstring
extension=openssl
extension=pdo_mysql
```

      Cherchez ces lignes et enlevez le `;` en début de ligne pour activer les extensions, puis enregistrez vos modifications.

- Ajouter php dans vos variables d'environnement système, chercher dans la barre de recherche windows `Modifier les variables d'environnement systèmes`, ouvrez la page, cliquez sur le bouton `Variables d'environnement`, ensuite trouvez la section `Variables système`, dans la liste cliquez sur la ligne qui commence par `Path` et ensuite sur le bouton `Modifier`. Maintenant cliquez sur le bouton `nouveau` et ajouter la ligne `C:\php`.
- Maintenant vous pouvez ouvrir un CMD et lancer la commande `php -v` pour vérifier votre version de php, si vous avez bien ce résultat :

  ```
  PHP 8.1.8 (cli) (built: Jul  7 2022 03:11:30) (NTS)
  ```

  C'est que Php est bien installé sur votre système. Sinon essayez de fermer votre CMD et de le rouvrir en lançant la même commande.

Composer

Composer est un gestionnaire de dépendance pour PHP qui est indispensable quand vous utilisez un projet symfony, pour l'installer rendez-vous sur la documentation composer et installé l'installer de composer pour windows, vous n'aurez plus qu'à le lancer et suivre les indications d'installations.

Le Symfony CLI

Pour faciliter l'installation et le démarrage d'un serveur symfony nous allons utiliser le CLI Symfony, pour l'installer suivez ces étapes :

- Ouvrez un terminal PowerShell
- Entrez la commande

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

- Puis :

  ```
  irm get.scoop.sh | iex
  ```

- Et enfin :

  ```
  scoop install symfony-cli
  ```

Installer un projet Symfony

Dans un premier temps, vous allez avoir besoin du client Symfony pour pouvoir utiliser des lignes de commandes Symfony sur votre environnement, vous pouvez retrouver la documentation d'installation sur ce lien

Ensuite, vous devez vérifier que votre environnement peut faire tourner une application symfony, pour ça, vous avez une nouvelle commande à entrer dans un terminal (grâce au client Symfony) :

```
symfony check:requirements
```

Une fois que vous savez que votre environnement peut faire tourner une application Symfony, vous devez rentrer la commande qui va permettre d'installer un projet vierge Symfony dans un terminal :

```
symfony new my_project_directory --version="6.*" --webapp
```

Cette commande va vous installer un projet Symfony vierge avec l'installation des dépendances.

Versions multiple de php sur votre environnement, Si vous utilisez WAMP ou que vous avez plusieurs version de Php sur votre système, il se peut que l'installation de Symfony soit en erreur si Symfony n'utilise pas la version de Php 8.1, pour corriger se problème, vous devez simplement entrez une commande à l'emplacement où vous voulez installer votre nouveau projet :

```
echo 8.1 > .php-version
```

Cette commande devrait vous créer un fichier .php-version avec seulement écrit à l'intérieur 8.1, ce fichier va stipuler à Symfony quelle version de php il doit utiliser pour faire l'installation, une fois le fichier créé, vous pouvez relancer la commande ci-dessus pour installer un nouveau projet.

Dernière chose à faire, installer une dépendance à notre projet PHPStan, qui va permettre d'analyser votre code PHP pendant le développement.

```
composer require --dev phpstan/phpstan
```

Pour lancer les tests vous aurez simplement à lancer la commande :

```
vendor/bin/phpstan analyse -l 0 src
```

Installer webpack :

Pour compiler vos assets (css et js) vous allez devoir utiliser Webpack Encore sur votre projet, pour ça, vous pouvez lancer la commande :

```
composer require symfony/webpack-encore-bundle
```

Puis vous devez installer les dépendances JS et css avec yarn ou npm :

```
yarn install
# OU
npm install
```

Pour plus d'information, voici le lien de la documentation.

Intégration de bootstrap

Si vous souhaitez utiliser bootstrap sur votre projet, vous pouvez également l'installer, retrouver la documentation en cliquant ici.

Symfony :

Pour installer les dépendances de Symfony et lancemer l'environnement de développement, vous devez :

- Dans le terminal de VsCode (terminal Docker), entrez la commande

```
composer install
```

- Ensuite, vous avez simplement à vous rendre sur votre navigateur sur l'url `localhost`, vous verrez la page d'accueil de l'application Symfony.

Configuration de la connexion en base de donnée :

Pour que le framework Symfony puisse communiquer avec votre base de donnée, nous allons devoir modifier l'url vers la database pour que Symfony se connecte et interagisse avec elle via Doctrine ORM.

Aller dans le fichier `.env` et cherchez la ligne commençant par `DATABASE_URL=...`.

Nous allons de voir modifier l'url pour ajouter celle de notre base de donnée que nous avons grâce à l'environnement docker.

La database URL se décompose comme suit :

- `myql://` -> qui définit le type de base de données, ici nous utilisons une base de donnée mysql.
- `db_user` -> qui définit le user de la base de données, ici nous avons paramétré le user `root`
- `db_password` -> qui définit le mot de passe pour accéder à la base de données, il faut laisser vide si pas de mot de passe, ici nous n'avons pas de mot de passe de paramétré, donc on passe directement à la suite de l'url.
- `@127.0.0.1` -> qui définit le serveur host de la base donnée, ici nous avons le host `@db_symfony` car l'host est configuré par notre image Docker db_symfony.
- `:3306` -> qui définit le port à utiliser pour la connexion.
- `/db_name` -> qui définit le nom de la base de donnée, ici nous avons une database qui se nomme `db_cours`
- `?serverVersion=5.7` -> qui définit la version du serveur que nous utilisons, ici nous utilisons un serveur Mysql `version 5.7`.

Ce qui fait que notre database URL est :

```
mysql://root:@db_symfony:3306/db_cours?serverVersion=5.7
```

Voici à quoi ressemble une URL de base de donnée.

Lancement du server

Maintenant que nous avons installé notre projet Syfmony et configuré notre base de données, nous pouvons lancer le server local Symfony.

Symfony va lancer un server grâce à php installé en local sur votre système.

Pour lancer le serveur, rentrer la commande dans un terminal :

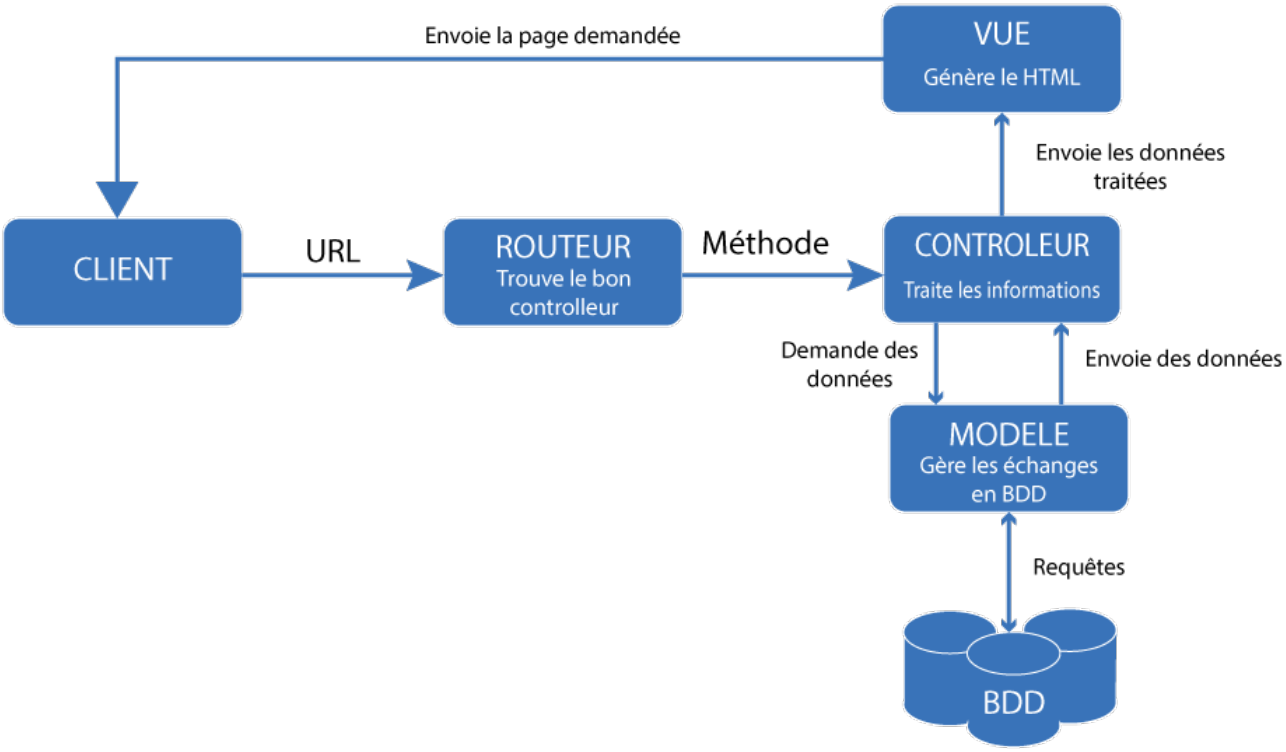
```
symfony server:start
```

Une fois la commande lancée, vous devriez voir une url sur votre terminal, c'est l'url sur laquelle vous allez pouvoir voir le rendu de votre application Symfony.

Framework MVC

Symfony utilise le design pattern MVC, c'est à dire un patron de conception qui désigne une certaine architecture dans une application web.

Voici un petit schéma pour mieux comprendre :



Concrètement, comment ça marche ?

Le client (le navigateur) va envoyer une URL à notre **Routeur**, le routeur à pour but de trouver le bon contrôleur (il cherche une méthode dans un contrôleur en particulier suivant l'url envoyée).

Ensuite le **contrôleur** va traiter les informations : aller chercher dans les modèles dans informations en base de données (les modèles vont exécuter les requêtes et renvoyer les données).

Ensuite le contrôleur va envoyer les informations à une **vue** qui doit générer le HTML, et rendre la page au client (navigateur).

Les dossiers

Une fois que vous avez installé un nouveau projet symfony, vous pourrez voir qu'il y a plusieurs dossiers qui viennent avec lac structure de base, passons en revu ces dossiers et leurs intérêt.

Assets

Comme son nom l'indique, ce dossier contient tous les assets nécessaires pour le front de notre application Symfony.

Les assets sont principalement des fichiers **.css** et **.js** qui seront par la suite compilés à l'aide du package **Webpack Encore**.

Bin

Ce dossier contient les fichiers de commandes permettant, par exemple, de vider le cache Symfony, mettre à jour la base de données ou encore lancer nos tests unitaires.

On utilise généralement la commande `php bin/console` qui affiche toutes les commande Symfony disponibles.

Config

Toute la configuration des packages, services et routes se fera dans ce dossier.

Cela permettra, entre autre, de configurer notre connexion à la base de données, mettre en place tout un système de sécurité, ou encore personnaliser les services que nous développerons.

Les fichiers de configuration sont par défaut en **YAML**, même s'il est tout à fait possible d'utiliser **PHP** ou **XML**.

Public

C'est le point d'entrée de l'application : chaque requête / demande passe forcément par ce dossier et le fichier `index.php`.

Étant accessible par tous, il est généralement utilisé pour mettre à disposition des fichiers de ressources, principalement des images.

Src

C'est le cœur du projet ! L'endroit où vous passerez le plus de temps à coder. Il regroupe tout le code PHP de votre application, c'est ici que vous mettrez en place toute la logique de votre application.

Les dossiers qui seront obligatoires à utiliser pour le fonctionnement de l'application sont :

- **[Controller]** : Définition des points d'entrée de votre application. Il se charge de rediriger vers les Manager / Service / Repository. Aucun traitement de données, accès à la BDD (base de données) ne doit se faire depuis un Controller (très important). Possibilité de choisir les méthodes d'entrée (GET, POST, PUT, DELETE, ...) ainsi que le type de réponse retournée (JSON, XML, ...).
- **[Entity]** : Définition de la structure de votre BDD (base de données) au travers de classes. Chaque

Entity représente généralement une table en BDD. La commande `php bin/console doctrine:migrations` nous permettra de mettre à jour notre BDD à chaque modification de l'Entity.

- **[Repository]** : Un Repository est toujours rattaché à une Entity, il nous permet de créer nos fonctions qui iront requêter la table de notre Entity (ainsi que les tables liées). Symfony utilise l'ORM **Doctrine** qui permet de créer nos requêtes SQL à travers les **queryBuilder** (très utile si l'on déteste faire du SQL).

Templates

Symfony utilise depuis ses débuts le moteur de templates **Twig**.

Les fichiers de template Twig ont comme format `monfichier.html.twig` et viennent rajouter quelques fonctionnalités au HTML classique :

- **{{ ... }}** : appel à une variable ou une fonction PHP, ou un template Twig parent.
- **{% ... %}** : commande, comme une affectation, une condition, une boucle ou un bloc HTML.
- **{# ... #}** : commentaires.

Pour avoir plus d'infos et en apprendre plus sur Twig, vous trouverez ici une documentation assez complète.

Tests

Les tests unitaires **PHPUnit** seront définis ici pour tester notre application.

La commande pour lancer nos tests :

```
php bin/phpunit
```

Il faut néanmoins s'assurer que le **package phpunit** soit installé en utilisant la commande suivante :

```
composer require --dev symfony/phpunit-bridge
```

Translation

L'internationalisation des applications est très importante aujourd'hui. Il est donc nécessaire de mettre en place un système de traduction dès le début du projet. Cela reste néanmoins facultatif si vous êtes certain de ne jamais avoir à traduire votre application.

Pour cela, on installe le **package translation** et on suit la documentation Symfony :

```
composer require symfony/translation
```

Var

Ici seront stockés le **cache** et les fichiers de **log**.

Il est possible dans les fichiers de config de paramétrer la mise en cache et ce que l'on écrit dans les logs.

Le fichier composer.json

Tous les **packages** sont enregistrés dans ce fichier.

Ils sont installés automatiquement dans le dossier `vendors` lors de l'initialisation du projet mais on peut utiliser la commande `composer install` pour les installer manuellement si besoin.

Pour mettre à jour les packages, on utilise la commande `composer update` et pour ajouter un package on utilise `composer require monpackage`

Les controllers :

Symfony utilise les controllers pour afficher les pages et intégrer des requêtes en base de donnée afin de pouvoir récupérer les informations, et les afficher ensuite sur la page.

Pour créer un controller, vous pouvez créer un fichier `FrontController.php` par exemple dans le dossier `src/controller`.

Notez que tous les controllers de Symfony sont écrit en PHP et qu'ils ont tous la même structure :

```
<?php // <- Ouverture de la balise PHP

namespace App\Controller; // <- Défintion du namespace (Dans quel dossier se trouve le fichier)

// La partie `use` permet d'importer des modules(classe) de symfony et d'indiquer au fichier où chercher un module quand on l'appelle dans le code. (De chercher dans le bon namepsace la bonne classe)
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

// définition de la class du fichier (la class est utiliser pour importer le controller dans un autre fichier si besoin).La fonction extends définit que le Controller étends de l'AbstractController qui est un composant de symfony et qui va aider le développement du Controller. Attention, le nom de la class doit être identique au nom du fichier (sans l'extension .php)
class FrontController extends AbstractController
{
  /* Le traitement de votre controller... */
}
```

Maintenant, nous allons écrire dans le FrontController une fonction qui va afficher 'Hello World!' sur la page d'accueil :

```php
<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response; // <- va permettre de pouvoir récupérer la reponse de la requete
use Symfony\Component\Routing\Annotation\Route; // <- va permettre de définir les routes pour les functions

class FrontController extends AbstractController
{
  #[Route('/', name: 'home')]
  public function index(): Response
    {
        return new Response("Hello World !"); // On demande au controller d'envoyer une réponse avec le contenu Hello World
    }
}
```

Maintenant, plutôt que de simplement envoyer une réponse avec hello world, nous allons vouloir rendre une vue, c'est à dire rendre un fichier html, pour ça nous allons utiliser la méthode render que nous pouvons utiliser grâce à la classe parente AbstractController.

Donc pour rendre une vous vous avez simplement à retourner la fonction render en stipulant le chemin vers votre vue en partant du dossier template :

```php
public function index(): Response
{
  return $this->render('Home/index.html.twig');
}
```

Grâce à cette méthode, nous pouvons rendre une vue twig depuis notre contrôleur, maintenant il nous reste à voir comment nous pouvons envoyer des données à notre vue :

```php
public function index(): Response
{
  $data = ['Pierre', 'Paul', 'Jacques'];
  return $this->render('Home/index.html.twig', [
    'data' => $data
  ]);
}
```

Ici, avant d'envoyer ma vue, je définit une variable PHP $data dans laquelle je stocke un tableau, pour envoyer cette information à ma vue, je vais la passer dans la fonction render, juste après avoir définit le chemin vers le fichier, j'ouvre un tableau associatif (clé/valeur) la clé que je mets dans le tableau va être le nom de la variable qui va être générée dans ma vue twig, et la valeur sera la valeur que je pourrai afficher dans ma vue twig.

Vous pouvez envoyer plusieurs données dans une même vue.

Générer un contrôleur

Nous venons de voir comment nous pouvons créer un contrôleur manuellement, mais Symfony nous permet de pouvoir créer un contrôleur avec une commande en terminal :

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

Grâce à cette commande, Symfony va vous générer un nouveau contrôleur déjà prêt avec sa vue dans le dossier twig.

Twig :

Twig est le moteur de template utiliser par Symfony, il va permettre de créer des gabarits de pages (templates) et de pouvoir y intégrer des variables que nous avec notre controller simplement et de manière beaucoup plus lisible qu'en PHP.

Les fichiers Twig sont utilisés par Symfony pour faire le rendu des pages de l'application, grossièrement, on peut dire que Twig ressemble au langage HTML, mais avec plus de fonctionnalités pour le rendu de variables PHP.

Tous les fichiers Twig doivent se trouver dans le dossier templates de votre projet (c'est dans ce dossier que Symfony va chercher les fichiers Twig), ils doivente également avoir l'extentsion `fichier.html.twig`.

TWIG reprend la strucure et les balises du HTML mais utilise des balises spécifiques pour afficher des variables, ou mettre en place des structures logiques à l'intérieur du code. Ces balises sont simple afin de ne pas avoir un code trop lourd et illisble.

Voici les différentes balises Twig :

```
{{ maVariable }}
```

```
{% if foo = 0 %}
  Condition 1
{% else %}
  Sinon 1
{% endif %}
```

Les balises twig avec `{{ }}` 2 acolades permettent d'afficher des variables.

Les balises avec `{% %}` 1 acolades et pourcent permettent d'afficher des structures logiques, à noter que la pluparts du temps, cest balise doivent êtres fermées avec `{% endif %}` (ou endblock, ou endfor... Tout dépend la structure logique que vous avez ouverte).

Ces fonctionnalitées vont nous permettre d'avoir un affichage dynamique et de couvrir un maximum les besoins de n'importe quelles application web.

Voici un exemple de fichier Twig :

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

{% block title %}Page d'accueil du site{% endblock %}

{% block body %}

  {% if maVariable is defined and maVariable is not null %}

    <div class="row">
      <div class="content">
        <p>{{ maVariable }}</p>
      </div>
    </div>

  {% else %}

    <div class="row">
      <div class="content">
        <p>La variable n'existe pas !</p>
      </div>
    </div>

  {% endif %}

{% endblock %}
```

Nous pouvons voir que le TWIG supporte le HTML, il permet d'afficher des variables de manière dynamique et également intégrer des strucures logiques : boucles For, While, condition If etc...

Le fichier base Twig :

Les différents fichiers TWIG de votre projet vont tous étendre d'un fichier de base, souvent nommé `base.html.twig`.

Ce fichier est la base de toutes vos pages, la structure globale de vos pages, c'est dans ce fichier que vous allez créer votre l'architecture de votre code (la partie head avec les balises meta, et la partie body avec votre contenu)

Voici un exemple d'un fichier base.html.twig :

```
!DOCTYPE html>
```

```
<html>
  <head>
    <!-- Les métas qui doivent être sur toutes les pages et qui seront toujours
appelées -->
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-
fit=no">
    <title>

    {% block title %}{% endblock %}  <!-- La balise qui indique où le contenu du block
Title des autres fichiers va être intégré -->

    </title>

    {% block stylesheets %}{% endblock %} <!-- La balise qui indique où le contenu du
block CSS des autres fichiers va être intégré -->

  </head>
  <body>
    <nav>
      <!-- La navbar de votre site qui sera sur toutes les pages -->
    </nav>

    {% block body %}{% endblock %} <!-- La balise qui indique où le contenu body des
autres fichiers va être intégré -->

  {% block javascripts %}{% endblock %} <!-- La balise qui indique où les balises
javascript des autres fichiers va être intégré -->

  </body>
</html>
```

Maintenant voyons comment les autres fichiers vont appeler ce fichier de base et intégrer du contenu dans les balises que nous avons préparées dans le fichier base.html.twig :

```
{% extends 'base.html.twig' %} <!-- Le fichier commence par la balise extends qui va permmetre de définir le fichier de base à utiliser pour ce fichier -->

{% block title %}
  Titre de la page
{% endblock %}

{% block stylesheets %}
  <link rel="stylesheet" href="/css/index.css">
{% endblock %}

{% block body %}
  Le contenu de votre page
{% endblock %}
```

Comme vous pouvez le voir, nous avons utilisé les balises que nous avons placé dans le fichier base.html.twig, mais cette fois, nous avons intégré le contenu spécifique pour ce fichier(page). cette page va donc reprendre tout le fichier base.html.twig en intégrant le contenu des balises du fichiers. Cette méthode permet de pouvoir gagner du temps en ne devant pas redévelopper la structure à chaque fichier et de pouvoir faire de la réutilisation de composent.

Création de la table User

Pour presque toutes les applications web, nous avons besoin de créer une table User afin de pouvoir gérer nos utilisateurs ainsi que les admins qui seront amener à utiliser le back office de notre application.

Pour ça, Symfony dispose d'une commande très simple qui permet de créer une classe User avec les propriétés adéquates pour gérer nos utilisateurs :

```
php bin/console make:user
```

Cette commande va vous poser plusieurs questions pour la gestion de vos utilisateur, en répondant aux questions dans le terminal, Symfony va vous générer le fichier classe User ainsi que faire la configuration du bundle sécurité de Symfony.

Créer ou modifier une Entity

Une Entity dans Symfony c'est simplement une classe de modele, c'est à dire que chaque Entity est une classe PHP qui représente une table dans votre base de données.

Ici encore, Symfony dispose d'une commande en terminal pour créer ainsi que modifier une entity existante :

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

Le nom que vous allez donner après make:entity sera le nom de votre future entity, ou si vous souhaitez modifier une entity déjà existante, vous n'avez qu'à mettre le nom d'une classe existante, Symfony va automatiquement vérifier si la classe existe ou non et va soit créer le fichier, soit modifier un fichier existant.

Toutes les informations vous sont données avec la commande make:entity (nom du champ type de données etc...)

Modifier la table User

Par défaut l'Entity user générée par la commande `php bin/console make:user`, est un peu vide (très peu d'information sur l'utilisateur), si nous voulons rajouter le nom, le prénom, une adresse email, une adresse etc... Il va falloir modifier notre entity User.

Donc vous allez devoir rentrer la commande :

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

Et automatiquement, Symfony va savoir que vous voulez modifier la classe User.

Une fois que vous avez ajouté toutes les propriétés que vous le vouliez, il faut maintenant utiliser l'ORM Doctrine (package qui nous permet de traduire nos classes PHP en requête SQL pour créer ou modifier nos tables).

Quand vous souhaitez faire une migration en base de données (appliquer les changements fait sur Syfmony dans votre base de données), vous devez le faire en 2 étapes :

1. Générer le fichier de migration

Pour se faire, vous devez simplement lancer la commande :

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

Avec cette commande, vous allez générer un fichier de migration qui sera disponible dans le dossier migrations, ce fichier de migration va comporter les requêtes SQL à faire en base de données pour mettre à jour les informations et synchroniser ce que vous avez sur votre application Symfony (vos entity) et les tables de votre base de données.

2. Executer la migration

Une fois que le fichier de migration et généré, vous n'avez plus qu'à dire à doctrine d'exécuter le fichier que vous venez de générer :

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

Avec cette commande, votre fichier de migration va être exécuté, et votre base de données sera à jour.

Sécurisation de votre application

Avant toute chose, il faut penser à sécuriser le BO de votre application, seul les utilisateurs avec le role admin pourront accéder aux pages du BO.

Symfony à déjà préparé le terrain pour vous, il vous suffit de vous rendre dans le dossier config > packages et vous trouvez un fichier **Security.yaml**.

C'est dans ce fichier que vous allez configurer la sécurisation de votre application :

```
security:
    # 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
    providers:
        # used to reload user from session & other features (e.g. switch_user)
        app_user_provider:
            entity:
                class: App\Entity\User
                property: username
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: app_user_provider

    # 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 }

when@test:
    security:
        password_hashers:
        # By default, password hashers are resource intensive and take time. This is
        # important to generate secure password hashes.
        # are not important, waste resources and increase test times. The following
        # reduces the work factor to the lowest possible values.
            Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface:
                algorithm: auto
                cost: 4 # Lowest possible value for bcrypt
                time_cost: 3 # Lowest possible value for argon
                memory_cost: 10 # Lowest possible value for argon
```

Si vous regardez bien le fichier, vous devriez voir une partie `access_control`, c'est ici que nous allons définir les règles ainsi que les accès par certains utilisateur.

Vous pouvez décommenter la ligne `# - { path: ^/admin, roles: ROLE_ADMIN }` ce qui veut dire que toutes url qui débute par /admin ne sont accessible qu'aux utilisateur admin.

Le formulaire de connexion

Maintenant que notre admin est sécurisé, il va falloir créer une page pour se connecter à l'application, sinon personne ne pourra jamais accéder au BO.

Pour ça vous allez devoir créer un nouveau contrôleur que nous appellerons SecurityController, donc vous allez lancer la commande :

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

Cette commande va vous créer un nouveau controller.

Nous allons légèrement le modifier pour qu'il puisse générer une page avec le formulaire de connexion :

```php
<?php

namespace App\Controller\Security;

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 SecurityController extends AbstractController
{
    #[Route('/login', name: 'login')]
    public function index(AuthenticationUtils $authenticationUtils): Response
    {
        // get the login error if there is one
        $error = $authenticationUtils->getLastAuthenticationError();
        // last username entered by the user
        $lastUsername = $authenticationUtils->getLastUsername();

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

Ici vous voyez que nous faisons 2 choses avant de rendre la vue, d'abord nous instancions la classe AuthenticationUtils qui va nous permettre de récupérer les erreurs s'il y en a pour une tentative de connexion ainsi que le dernier utilisateur de connecté sur la session de l'utilisateur.

Ensuite nous envoyons ces informations à la notre vue (dans le dossier templates/Security/login.html.twig).

La vue

Maintenant que nous avons notre contrôleur, nous devons créer la vue dans le fichier login.html.twig dans le dossier templates/security, à l'intérieur, nous allons simplement mettre un formulaire de connexion simple :

```
{% extends "layout.html.twig" %}

{% block title %}
  Se connecter |
  {{ parent() }}
{% endblock %}

{% block body %}
  <div class="container-fluid mt-4">
    <form action="{{ path('login') }}" method="post" class="p-3 form card w-50 mx-
auto">
      {% if error %}
        <div class="alert alert-danger" role="alert">{{
error.messageKey|trans(error.messageData, 'security') }}</div>
      {% endif %}
      <h1 class="title text-center">Se connecter</h1>
      <label for="username" class="form-label">Username:</label>
      <input type="text" id="username" name="_username" class="form-control" value="{{
last_username }}"/>

      <label for="password" class="form-label mt-2">Mot de passe:</label>
      <input type="password" id="password" name="_password" class="form-control"/>

      <div class="d-grid gap-2 d-md-block mt-4 text-center">
        <button type="submit" class="btn btn-inline btn-primary">Se connecter</button>
      </div>
    </form>
  </div>
{% endblock %}
```

Configuration des routes

Dernière chose à faire pour que notre formulaire de connexion soit opérationnel, de gérer les routes avec le fichier sécurity.yaml.

En effet nous allons ajouter un peu de configuration à ce fichier pour que notre application puisse savoir vers qu'elle route rediriger s'il y a besoin de se connecter.

```yaml
firewalls:
  dev:
    pattern: ^/(_(profiler|wdt)|css|images|js)/
    security: false
  main:
    lazy: true
    provider: app_user_provider
    form_login:
      login_path: login
      check_path: login
    logout:
      path: /logout
      target: /
```

En rajoutant dans le main firewall le form_login ainsi que le logout vous avez gérer les routes pour se connecter ainsi que pour se déconnecter.

Les Fixtures

Pour le moment, nous avons gérer la sécurisation de notre application mais pour le moment, nous n'avons pas d'utilisateurs en base de données. Il faut donc en créer, mais plutôt que de devoir rentrer dans le vif du sujet tout de suite en générant des formulaires, créer des contrôleurs, des vues etc... Nous allons vouloir créer un utilisateur rapidement en PHP, et pour ça, nous allons utiliser des fixtures.

Tout d'abord, nous allons devoir installer le bundle avec la commande :

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

Vous devriez voir un nouveau dossier dans le dossier Src qui se nomme DataFixtures, et à l'intérieur un fichier AppFixtures.php.

C'est avec ce fichier que nous allons pouvoir créer un user rapidement, ouvrez le fichier et modifier le comme suit :

```php
<?php

namespace App\DataFixtures;

use App\Entity\User;
use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Persistence\ObjectManager;

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

```
        $user = new User();
        $user->setPrenom('Pierre')
            ->setNom('Bertrand')
            ->setAge(25)
            ->setUsername('Pierre-brtrd')
            ->setEmail('pierre@example.com')
            ->setPassword('test1234') // Nous devons hasher le password
            ->setRoles(["ROLE_ADMIN"])
            ->setVille('Chambéry');

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

Ici nous avons créer un nouvel objet de type User et définit ces propriétés, mais attention pour le mot de passe, il va falloir le hasher afin que notre application soit bien sécurisé.

Nous allons tout d'abord importer la classe UserPasswordHasherInterface que va nous permettre de hasher notre password, pour se faire, nous allons devoir créer un constructeur :

```
public function __construct(
  private UserPasswordHasherInterface $hasher
) {
}
```

Maintenant, dans notre fonction de création de fixture user, nous allons pouvoir utiliser le password hash avec `$this->hash->hashPassword()` :

```
public function __construct(
  private UserPasswordHasherInterface $hasher
) {
}

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

  $user = new User();
  $user->setPrenom('John')
    ->setNom('Doe')
    ->setAge(25)
    ->setUsername('John-Do')
    ->setEmail('john@example.com')
    ->setRoles(["ROLE_ADMIN"])
    ->setPassword($this->hasher->hashPassword($user, 'test1234'))
```

```
        ->setVille('Chambéry');

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

Maintenant, il faut dire à doctrine de charger en base de données cet fixture :

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

Et voilà, vous avez maintenant un utilisateur en base de données.

Création de la table article

Maintenant que nous avons sécurisé notre application, nous pouvons commencer à construire notre BO, commençons par la table article.

Vous devez créer une table article avec les propriétés suivantes :

- titre -> string
- content -> string
- createdAt -> datetime
- updatedAt -> datetime

Vous pouvez créer la classe avec la commande :

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

Qui va vous générer la classe automatiquement.

Le timestample

Pour les champs createdAt et updatedAt, nous voulons que ces champs soit mis à jour automatiquement, nous allons donc utiliser une extension de doctrine pour permette cela :

```
composer require stof/doctrine-extensions-bundle
```

Autorisé les recettes pour que la configuration se fasse.

Une fois finit, vous devriez voir dans votre dossier config/packages un nouveau fichier `stof_doctrine_extensions.yaml`, vous devez l'ouvrir et le modifier comme suit :

```
stof_doctrine_extensions:
    default_locale: fr_FR
    orm:
        default:
            timestampable: true
            sluggable: true
```

Nous venons d'ajouter 2 extensions à notre ORM, timestampable pour mettre à jour automatiquement les champs createdAt et updatedAt ainsi que sluggable que va nous permettre de générer des slugs rapidement.

Maintenant retour dans notre entity User pour mettre en place le timestampable ainsi que le slug :

```php
<?php

namespace App\Entity;

use App\Repository\ArticleRepository;
use Doctrine\ORM\Mapping as ORM;
use Gedmo\Mapping\Annotation as Gedmo;

#[ORM\Entity(repositoryClass: ArticleRepository::class)]
class Article
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column(type: 'integer')]
    private $id;

    #[ORM\Column(type: 'string', length: 255)]
    private $titre;

    #[ORM\Column(type: 'text')]
    private $content;

    #[ORM\Column(length: 260, unique: true)]
    #[Gedmo\Slug(fields: ['id', 'titre'])]
    private $slug;

    #[ORM\Column(type: 'datetime_immutable')]
    #[Gedmo\Timestampable(on: 'create')]
    private $createdAt;

    #[ORM\Column(type: 'datetime_immutable')]
    #[Gedmo\Timestampable(on: 'update')]
    private $updatedAt;

    public function getId(): ?int
    {
        return $this->id;
    }

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

```php
    public function setTitre(string $titre): self
    {
        $this->titre = $titre;

        return $this;
    }

    public function getContent(): ?string
    {
        return $this->content;
    }

    public function setContent(string $content): self
    {
        $this->content = $content;

        return $this;
    }

    public function getSlug()
    {
        return $this->slug;
    }

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

    public function setCreatedAt(\DateTimeImmutable $createdAt): self
    {
        $this->createdAt = $createdAt;

        return $this;
    }

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

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

        return $this;
    }
}
```

Et voilà, nous sommes prêt à faire notre migration `php bin/console make:migration` et l'exécuter `php bin/console doctrine:migrations:migrate`.

Les formulaires

Maintenant que nous avons créé une table article, nous allons vouloir créer une page de création d'article avec un formulaire.

Pour commencer, nous allons créer un nouveau contrôleur pour l'admin (AdminController) vous pouvez utiliser la commande `php bin/console make:controller AdminController` :

```
<?php

namespace App\Controller\Backend;

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

#[Route('/admin')]
class AdminController extends AbstractController
{
}
```

Nous avons préfixé les routes avec /admin, ce qui veut dire que chaque méthode que nous allons créer, au niveau des routes, elles auront toujours au début `/admin`.

Nous allons simplement créer une méthode qui nous servira pour la page de création des articles :

```
 #[Route('/article/create', name: 'admin.article.create')]
public function createArticle(Request $request)
{
  return $this->render('Backend/Article/create.html.twig');
}
```

Pour le moment nous ne faisons rien d'autre que d'appeler la vue, mais il va falloir que nous générions notre formulaire et que nous l'envoyions à notre vue (n'oubliez pas de créer le fichier twig pour la vue).

Générer un formulaire

Vous pouvez générer automatiquement une classe formulaire qui va permettre d'automatiser la génération d'un formulaire en indiquant l'entity qui sera rattaché.

La commande à faire est :

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

Le terminal va simplement vous demander quelle entity doit être relié au formulaire et vous générer un nouveau fichier dans le dossier src/form qui contient :

```php
<?php

namespace App\Form;

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

class ArticleType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('titre')
            ->add('content');
    }

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

La méthode buildForm va générer les différent champs à intégrer dans le formulaire (ces champs représente toutes les propriétés que nous voulons remplir par le formulaire pour nos articles).

Par défaut, Symfony va essayer de données le type à l'input automatiquement en fonction du type de données qui est relié, mais vous pouvez modifier le type de l'input facilement en ajoutant une option après le nom du champ :

```php
<?php

namespace App\Form;

use App\Entity\Article;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\Extension\Core\Type\TextareaType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class ArticleType extends AbstractType
```

```
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('titre', TextType::class)
            ->add('content', TextareaType::class);
    }

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

N'oubliez pas d'importer les classes (les uses). Ensuite vous pouvez également ajouter un 3ème paramètre à vos inputs, un tableau d'option (vous pouvez retrouver toutes les informations sur la documentation) :

```
<?php

namespace App\Form;

use App\Entity\Article;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\TextareaType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class ArticleType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('titre', TextType::class, [
                'label' => 'Titre:',
                'required' => true
            ])
            ->add('content', TextareaType::class, [
                'label' => 'Contenu:',
                'required' => true
            ]);
    }

    public function configureOptions(OptionsResolver $resolver): void
    {
```

```
        $resolver->setDefaults([
            'data_class' => Article::class,
        ]);
    }
}
```

Et voilà, notre classe ArticleType va permettre de générer automatiquement un formulaire pour créer un article.

Envoyer le formulaire à la vue

Maintenant que nous avons notre classe de génération de notre formulaire, il faut l'appeler et le générer dans le contrôleur pour l'envoyer à la vue, donc retour dans l'AdminController :

```
#[Route('/article/create', name: 'admin.article.create')]
public function createArticle(Request $request)
{
  $article = new Article();

  $form = $this->createForm(ArticleType::class, $article);

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

Nous avons simplement créé un article vide pour le moment, ensuite nous avons créé un formulaire avec notre classe ArticleType, en lui passant l'article vide (qui sera rempli par l'utilisateur dans le formulaire).

Enfin nous envoyons le form à la vue, Attention pour que Twig puisse gérer l'affichage du formulaire en frontend il faut envoyer la variable $form avec la méthode `createView()` qui va permettre de générer le form afin que twig puisse lire le formulaire et l'afficher.

La vue

Maintenant, il faut seulement afficher ce formulaire dans notre vue :

```
{% extends "layout.html.twig" %}

{% block title %}
  Création d'un article |
  {{ parent() }}
{% endblock %}

{% block body %}
  <div class="container-fluid mt-4">
    <div class="card w-50 mx-auto p-3">
      <h1 class="text-center">Création d'un article</h1>
      {{ form_start(form) }}
```

```
        {{ form_end(form) }}
      </div>
    </div>
{% endblock %}
```

Ici nous utilisons la balise `form_start(form)` et `form_end(form)` pour que twig ouvre la balise form avec le formulaire que le contrôleur à envoyé.

Juste en faisant ça, vous devriez voir sur la page le formulaire, mais sans bouton de soumission.

Si nous ne stipulons pas à Twig à l'intérieur des balises `form_start(form)` et `form_end(form)` où et comment nous voulons afficher nos champs, il va les mettre automatiquement les uns à la suite des autres, mais nous nous voulons rajouter à la fin un bouton de soumission, mais il doit être à l'intérieur du form.

Nous allons donc devoir réécrire légèrement le code de notre formulaire :

```
{{ form_start(form) }}
      {{ form_row(form.titre) }}
      {{ form_row(form.content) }}
      <div class="text-center">
        <button type="submit" class="btn btn-primary btn-inline">Créer</button>
      </div>
{{ form_end(form) }}
```

Avec les balise `form_row()` vous pouvez afficher spécifiquement un champ, le form_row va vous afficher le lable ainsi que l'input, il faut stipuler le nom du formulaire ainsi que le champ concerné (à afficher).

Mais imaginons que vous ne vouliez pas du label, seulement de l'input en intégrant un placeholder ? Et bien c'est possible grâce à la balise `form_widget()`

```
{{ form_start(form) }}
      {{ form_widget(form.titre, {'attr': {'class': 'mb-4', 'placeholder': "Titre de
l'article"} } ) }}
      {{ form_widget(form.content, {'attr': {'class': 'mb-4', 'placeholder': "Contenu
de l'article", 'rows': 10} } ) }}
      <div class="text-center">
        <button type="submit" class="btn btn-primary btn-inline">Créer</button>
      </div>
{{ form_end(form) }}
```

Ici nous avons utilisé `form_widget(form.titre)` en ajoutant un tableau d'option `attr` pour ajouter un attribut sur l'input, et ensuite ajouté une classe ains que des placeholder.

Vous savez maintenant comment manipuler vos formulaires dans vos vues twig.

Validation d'un formulaire

Maintenant il nous reste la validation du formulaire pour que notre article soit créé, nous allons écrire la validation directement dans le contrôleur, donc AdminController :

```
 #[Route('/article/create', name: 'admin.article.create')]
public function createArticle(Request $request)
{
  $article = new Article();

  $form = $this->createForm(ArticleType::class, $article);
  $form->handleRequest($request);

  if ($form->isSubmitted() && $form->isValid()) {
    // Formulaire soumis est valide
  }

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

Nous avons simplement utilisé la classe Request qui permet de récupérer par exemple les variable superglobale `$_GET` ou `$_POST`, Symfony vous facilite la vie étant donnée que vous n'avez seulement qu'à récupérer la request, et ensuite de vérifier si le formulaire est soumis et s'il est valide.

Ensuite nous allons devoir importer l'`ArticleReposiroty` qui va nous permettre d'envoyer ce nouvel article en base de données. Pour importer la classe, vous avez 2 solutions :

1. Importer directement la classe en paramètre par défaut la méthode que nous sommes entrain de développer :

```
#[Route('/article/create', name: 'admin.article.create')]
public function createArticle(Request $request, ArticleReposiroty $repoArticle)
{
  $article = new Article();

  $form = $this->createForm(ArticleType::class, $article);
  $form->handleRequest($request);

  if ($form->isSubmitted() && $form->isValid()) {
    $repoArticle->add($article, true)
  }

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

Nous créons un article simplement en appelant la méthode `add()` de notre ArticleRepository (cette méthode est par défaut dans tous les repository's), cette méthode prend 2 paramètres, d'abord l'instance d'article, et ensuite un boolean (true ou false) pour savoir s'il doit exécuter le flush (sauvegarde en bdd).

2. Importer globalement la classe EntityManagerInterface pour pouvoir l'utiliser dans d'autre méthode de notre classe AdminController, pour ce faire nous allons l'instancier dans le constructeur de notre classe AdminController :

```
public function __construct(
  private ArticleRepository $repoArticle
){
}

#[Route('/article/create', name: 'admin.article.create')]
public function createArticle(Request $request)
{
  $article = new Article();

  $form = $this->createForm(ArticleType::class, $article);
  $form->handleRequest($request);

  if ($form->isSubmitted() && $form->isValid()) {
    $this->repoArticle($article, true);
  }

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

Avant de vouloir envoyer l'article en base de données, nous allons vouloir lui rattacher un auteur (un utilisateur).

Pour ça nous allons devoir modifier légèrement notre table Article pour ajouter une relation **OneToMany** avec la table User étant donnée qu'un article ne peut avoir qu'un auteur, et qu'un auteur peut écrire plusieurs articles.

Faites la commande `php bin/console make:entity Article` et ajoutez le champs **user** de type relation à la table user.

Vous devriez avoir se résultat :

```
/* .... Autres propriétés */

#[ORM\ManyToOne(inversedBy: 'articles')]
private ?User $user = null;

/* .... Autres accesseurs */
public function getUser(): ?User
{
```

```
    return $this->user;
}

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

    return $this;
}
```

Maintenant vous pouvez envoyer ces modifications en base de données (`php bin/console make:migration` PUIS `php bin/console doctrine:migrations:migrate`).

Maintenant, vos articles ont besoins d'avoir un utilisateur, donc si nous soumettons le formulaire, nous n'allons pas définir l'auteur automatiquement.

Nous allons devoir ajouter un peu de logique :

Pour écrire un article, il faut que l'utilisateur soit connecté sur notre application, donc nous allons pouvoir récupérer l'utilisateur très facilement.

Dans votre **Admin Controller**, vous devez importer dans le constructeur la class **Security** qui permet entre autre, de récupérer rapidement l'utilisateur qui est connecté.

```
public function __construct(
    private ArticleRepository $repoArticle,
    private Security $security
) {
}
```

Une fois que c'est fait, nous allons devoir ajouter l'utilisateur automatiquement si le formulaire de création d'un article est soumis, donc rendez vous toujours dans votre AdminController dans la méthode `createArticle` :

```
public function __construct(
    private ArticleRepository $repoArticle,
    private Security $security
){
}

#[Route('/article/create', name: 'admin.article.create')]
public function createArticle(Request $request)
{
    $article = new Article();

    $form = $this->createForm(ArticleType::class, $article);
    $form->handleRequest($request);

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

```
    // On récupère l'article et on lui injecte l'utilisateur avant de l'envoyer en base
    $article->setUser($this->security->getUser());
    $this->repoArticle->add($article, true);
  }

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

Et voilà, votre article a bien un utilisateur maintenant !

Dernière chose à faire, ajouter un petit message et rediriger vers une autre page :

```
#[Route('/article/create', name: 'admin.article.create')]
public function createArticle(Request $request)
{
  $article = new Article();

  $form = $this->createForm(ArticleType::class, $article);
  $form->handleRequest($request);

  if ($form->isSubmitted() && $form->isValid()) {
    $article->setUser($this->security->getUser());
    $this->repoArticle->add($article, true);
    $this->addFlash('success', 'Article créé avec succès');

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

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

La route admin

Maintenant que nous avons une page qui crée des articles nous allons pouvoir faire une page qui liste les articles ainsi que les utilisateurs dans l'admin.

Donc dans l'AdminController nous allons rajouter une nouvelle méthode :

```
#[Route('', name: 'admin')]
public function index(): Response
{
  return $this->render('Backend/index.html.twig');
}
```

N'oubliez pas de créer le fichier twig dans templates.

Maintenant nous voulons pouvoir récupérer tous les users ainsi que tous les articles de notre bases de données, pour ça nous allons devoir utiliser les Repository, avec Symfony, ces classes vous permettent de pouvoir rechercher rapidement dans une table de votre base de données.

Nous allons devoir importer 2 classe globalement (dans notre constructeur) le UserRepository ainsi que le ArticleRepository :

```
public function __construct(
  private ArticleRepository $repoArticle,
  private UserRepository $repoUser,
  private EntityManagerInterface $em
){
}
```

Maintenant nous pouvons appeler la classe ArticleRepository et UserRepository n'importe où dans notre AdminController avec les alias `$this->repoArticle` et `$this->repoUser`.

Donc nous pouvons écrire dans notre méthode de liste admin :

```
 #[Route('', name: 'admin')]
public function index(): Response
{
  // Récupérer tout les users
  $users = $this->repoUser->findAll();

  // Récupérer tout les articles
  $articles = $this->repoArticle->findAll();

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

Il ne vous reste plus qu'à afficher les informations sur votre vue.

Modifier un article

Il nous reste tout de même à laisser la possibilité de modifier un article.

Toujours dans l'AdminController, nous allons créer une nouvelle méthode qui va être très proche de celle que nous avons créé pour la création d'un article à quelques détails prêt :

```
 #[Route('/article/edit/{id}', name: 'admin.article.edit', methods: 'GET|POST')]
public function editArticle(Article $article, Request $request)
{
  $form = $this->createForm(ArticleType::class, $article);
  $form->handleRequest($request);
```

```
    if ($form->isSubmitted() && $form->isValid()) {
      $this->repoArticle->add($article, true);
      $this->addFlash('success', 'Article modifié avec succès');

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

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

Dans un premier temps, vous voyez que dans le commentaire pour définir la route, nous avons passer un paramètre dynamique `/article/edit/{id}` les accolades dans une url symbolise un paramètre qui sera envoyé directement dans l'url, en l'occurrence, nous voulons récupérer l'id de l'article à modifier afin de le retrouver en base de données et de générer le formulaire avec les informations qui sont disponible en base de données.

Ensuite plutôt que de créer un nouvel article avant la génération du formulaire, nous avons simplement recherché l'article avec l'id, et nous avons envoyé cet article dans le formulaire.

Vous pouvez maintenant créer ou modifier des articles.

Ajouter une image à un article

Maintenant que nous pouvons créer et modifier des articles, nous allons vouloir rajouter une image à un article, pour ce faire, nous allons utiliser un bundle de Symfony : **VichUploader** qui va nous simplifier l'upload de l'image sur les articles.

Installation et configuration du bundle

La première à faire est d'installer le bundle, pour ce faire vous devez entrez la commande suivante dans un terminal :

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

N'oubliez pas d'autoriser la recette de cette installation (configuration automatique du bundle) ce qui va permettre de créer les fichier de config.

Une fois l'installation finit, vous pourrez retrouver un nouveau fichier dans le dossier **config/packages/vich_uploader.yaml** qui va configurer votre upload d'image.

Vous allez devoir créer un nouveau mapping, ce qui signifie que vous allez dire à votre nouveau bundle où il doit uploader vos fichiers pour les articles ainsi que les options :

```
vich_uploader:
    db_driver: orm

    mappings:
```

```
        # Mapping pour l'image des articles
        articles_image:
            uri_prefix: /images/articles
            upload_destination: "%kernel.project_dir%/public/images/articles"
            namer: Vich\UploaderBundle\Naming\SmartUniqueNamer
            inject_on_load: false
            delete_on_update: true
            delete_on_remove: true

    metadata:
        type: attribute
```

Configuration de la table article

Maintenant que vous avez configuré votre nouveau mapping, il va falloir modifier votre Entity Article pour modifier votre table et stocker le nom de l'image rattachée à votre article.

Rendez-vous donc dans le fichier **src/Entity/Article.php**.

Première chose à faire c'est d'indiquer à Symfony que cette table contient des champs uploadable :

```
namespace App\Entity;

use App\Repository\ArticleRepository;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\HttpFoundation\File\File;
use Vich\UploaderBundle\Mapping\Annotation as Vich;

#[ORM\Entity(repositoryClass: ArticleRepository::class)]
#[Vich\Uploadable]
class Article
{
}
```

N'oubliez pas les uses pour importer les classes.

Maintenant il va falloir ajouter les champs utiles pour l'upload d'image :

```
// [...] Autres propriétés de la table

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

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

#[ORM\Column]
private ?int $imageSize = null;
```

Pour fonctionner, VichUploader à besoin de 3 champs indispensables :

- **ImageFile** qui va stoker l'image et permettre l'upload, notez que cette propriété ne sera pas en base de données, elle sert simplement à VichUpload de récupérer le fichier et de faire le traitement de l'upload.
  Nous avons également rajouter des options à cette propriété pour que VichUploader gère tout seul les informations :
  - `mapping` -> qui indique le nom du mapping à utiliser (celui que nous avons configurer dans le fichier yaml du bundle), il faut que le nom du mapping soit identique à celui que vous avez mis dans le fichier vich_uploader.yaml.
  - `fileNameProperty` -> qui va automatiquement remplir le champ **imageName** en base de données pour stocker le nom de l'image rattachée à l'article en base de données.
  - `size` -> qui va automatiquement remplir le champ **ImageSize** en base de données stockant la taille de l'image.

Maintenant il faut ajouter les accesseurs pour ces propriétés :

```
// [...] Autres accesseurs
public function setImageFile(?File $imageFile = null): void
{
  $this->imageFile = $imageFile;

  if (null !== $imageFile) {
    // Il faut biensur que la propriété updatedAt soit crée sur l'Entity.
    $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;
}

public function setImageSize(?int $imageSize): void
{
  $this->imageSize = $imageSize;
}

public function getImageSize(): ?int
```

```
{
  return $this->imageSize;
}
```

Une fois que vous avez fait vos modifications, n'oubliez pas de faire les deux commande pour envoyer ces modifications en base de données :

- `php bin/console make:migration`
- `php bin/console doctrine:migrations:migrate`

Ajout du champ dans le formulaire

Maintenant que nous avons nos champs en base de données et fait la configuration du bundle, il ne nous reste plus qu'à ajouter le champ dans le formulaire.

Ouvrez le fichier **src/Form/ArticleType.php** et ajoutez le champs `imageFile` :

```
<?php

namespace App\Form;

use App\Entity\Article;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\Extension\Core\Type\TextareaType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;
use Vich\UploaderBundle\Form\Type\VichImageType;

class ArticleType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('titre', TextType::class, [
                'label' => 'Titre:',
                'required' => true
            ])
            ->add('content', TextareaType::class, [
                'label' => 'Contenu:',
                'required' => true
            ])
            ->add('imageFile', VichImageType::class, [
              'required' => false,
              'download_uri' => false,
              'image_uri' => true,
              'asset_helper' => true,
              'label' => 'Image',
            ]);
```

```
    }

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

N'oubliez pas d'importer les classes que nous utilisons.

Modification de la vue

Maintenant, il ne nous reste plus qu'à modifier notre vue pour afficher le nouvel input de type file qui va nous permettre d'uploader une image dans la vu qui est utilisée pour la création et l'édition d'un article en ajoutant où vous souhaitez dans votre formulaire le champ :

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

Supprimer un article

Maintenant pour avoir toutes les actions **CRUD** sur nos articles, il faut gérer la suppression d'un article.

Pour ça nous allons devoir créer une nouvelle méthode dans l'AdminController qui va nous permettre de supprimer un article.

Cependant, quand on parle de suppression en base de données, il faut être très prudent, donc nous allons devoir faire des vérifications plus poussées avant de faire quoique ce soit en base.

Pour ce faire Symonfy génère automatiquement des tokens de validation de formulaire, ce qui va nous permettre de vérifier si un formulaire est bien soumis via notre application et que la requête n'a pas été interceptée ni modifier entre temps, c'es le **token Csrf** de Symfony, nous allons devoir en générer un dans la vue sur le bouton de soumission du formulaire de suppression, et vérifier dans le controller que le token est valide avant de supprimer l'élément.

Tout d'abord, faisons notre méthode dans le controller :

```
#[Route('/article/delete/{id}', name: 'admin.article.delete', methods: 'DELETE|POST')]
public function deleteArticle(Article $article, Request $request)
{
  // Méthode de suppression d'un article
}
```

Nous pouvons voir que dans l'url nous allons passer l'id d'un article afin de pouvoir identifier quel article doit être supprimé que nous stockons dans la variable php $article.

Maintenant nous allons devoir vérifier le token Csrf avant de supprimer cet article :

```php
#[Route('/article/delete/{id}', name: 'admin.article.delete', methods: 'DELETE|POST')]
public function deleteArticle(Article $article, Request $request)
{
  if ($this->isCsrfTokenValid('delete' . $article->getId(), $request->get('_token'))) {
    // Le token est valide on suprrime l'article
    $this->repoArticle->remove($article, true);
    $this->addFlash('success', 'Article supprimé avec succès');

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

  $this->addFlash('error', 'Le token n\'est pas valide');

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

Maintenant nous devons modifier la vue qui liste nos article en ajoutant un formulaire avec le token que nous allons générer ainsi que le bouton de suppression pour chaque article :

```twig
<form method="POST" action="{{ path('admin.article.delete', {'id': article.id}) }}"
onsubmit="return confirm('Êtes-vous sûr de vouloir supprimer cet article ?')">
  <input type="hidden" name="_method" value="DELETE">
  <input type="hidden" name="_token" value="{{ csrf_token('delete' ~ article.id) }}">
  <button class="btn btn-danger text-light">Supprimer</button>
</form>
```

Ajout de catégories aux articles

Maintenant, nous allons vouloir ajouter une nouvelle table et une nouvelle fonctionnalité à notre application, l'attribution de catégorie à nos article afin de les trier.

Création de la table

Pour commencer nous allons stocker les catégories dans une nouvelle table, vous devez donc créer une nouvelle entity.

Pour ce faire utiliser le maker bundle avec la commande **`php bin/console make:entity Categorie`**, pour la structure de votre table vous devez avoir les champs suivants :

- `titre` -> string maxi 100 caractères
- `articles` -> relation ManyToMany avec la table article

La relation va être géré automatiquement avec doctrine, donc vous avez simplement à suivre les instructions dans le terminal pour créer ce champ et cette relation.

Une fois que vous avez ajouter les deux champs, faites la migration en base de données -> `php bin/console make:migration` PUIS `php bin/console doctrine:migrations:migrate`.

Génération du CRUD

Nous avons vu précédemment comment mettre en place manuellement notre CRUD sur les articles, mais Symfony nous laisse la possibilité de générer automatiquement le controller avec toutes les méthodes CRUD déjà prête, ainsi que toutes les vues.

Pour générer tout ça, vous avez simplement à lancer la commande `php bin/console make:crud`, ensuite vous devez simplement définir qu'elle entity va être concernée pour la génération du CRUD, et enfin quelle doit être le nom du controller que Symfony va vous générer.

Et automatiquement, vous avez un nouveau controller avec toutes les méthodes CRUD pour votre entity, ainsi que le Form builder pour votre entity et enfin toutes les vues pour les différentes pages du CRUD.

Vous pouvez déplacer les fichiers générés automatiquement dans les différents dossiers pour garder une architecture plus propre, mais attention à changer les namespaces ainsi que les liens vers les vues dans le controller si besoin.

Ajouter des catégories

Maintenant vous pouvez vous rendre sur la page de création de catégories pour en créer plusieurs.

Il ne restera plus qu'à modifier le formulaire des articles pour que l'on puisse récupérer toutes les catégories stockées en base de données et de pouvoir rattacher des catégories à nos articles.

Relier les catégories aux articles

Maintenant que tout est en place, il faut modifier le formulaire des articles pour ajouter des catégories, modifier donc votre fichier `ArticleType` :

```
<?php

namespace App\Form;

use App\Entity\Article;
use App\Entity\Categorie;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\Extension\Core\Type\TextareaType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;
use Vich\UploaderBundle\Form\Type\VichImageType;
use Symfony\Bridge\Doctrine\Form\Type\EntityType;

class ArticleType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('titre', TextType::class, [
                'label' => 'Titre:',
```

```
                'required' => true
            ])
            ->add('categories', EntityType::class, [
                'class' => Categorie::class,
                'label' => 'Categories:',
                'multiple' => true,
                'choice_label' => 'titre',
                'by_reference' => false,
            ])
            ->add('content', TextareaType::class, [
                'label' => 'Contenu:',
                'required' => true
            ])
            ->add('imageFile', VichImageType::class, [
              'required' => false,
              'download_uri' => false,
              'image_uri' => true,
              'asset_helper' => true,
              'label' => 'Image',
            ]);
    }

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

Nous avons rajouté le champs categories qui est de type **`EntityType`**, ce qui veut dire que Symfony va générer soit une liste déroulante soit une liste à choix multiple avec toutes les entrées de la table Catégorie, et va pouvoir attribuer les catégories aux articles rapidement et automatiquement avec le formulaire.

Pour finir, n'oubliez pas de modifier la vue avec les formulaire des articles pour afficher ce nouveau champ où vous souhaitez.

Création des pages frontend article

Maintenant que nous avons géré la partie backend, nous pouvons créer les pages qui vont afficher un seul article.

Pour ce faire, nous allons créer un nouveau contrôleur ArticleController qui sera dans le dossier **src/frontend** afin de ne pas mélanger les controllers pour le frontend et ceux pour le backend.

Pour créer le nouveau contrôleur, faites la commande `php bin/console make:controller ArticleController` ce qui va vous générer le controller, ensuite vous devez le ranger dans le bon dossier (src/frontend) et changer le namespace du fichier généré.

Le fichier doit ressembler à cela :

```php
<?php

namespace App\Controller\Frontend;

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

#[Route('/article')]
class ArticleController extends AbstractController
{
}
```

Maintenant nous allons créer une méthode qui va afficher une page avec 1 seul article, pour ce faire nous devons envoyer dans l'url le **slug** en paramètre pour que notre controller puisse retrouver l'article en base de données et envoyer toutes les informations à la vue :

```php
#[Route('/details/{slug}', name: 'article.show')]
public function index(?Article $article, ArticleRepository $repo): Response
{
  // Si aucun article trouvé, on redirige vers la page d'accueil avec un message
d'erreur
  if (!$article instanceof Article) {
    $this->addFlash('error', 'Article non trouvé');

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

  // On envoie les informations à la vue
  return $this->renderForm('frontend/article/show.html.twig', [
    'article' => $article,
  ]);
}
```

Rien de plus simple, maintenant il faut créer le fichier pour la vue et afficher les informations.

Une fois votre vue mise en place, vous pouvez maintenant modifier la page d'accueil pour qu'elle puisse afficher la liste de tous les articles et intégrer des liens vers les pages des articles, pour ça modifiez simplement le MainController :

```php
<?php

namespace App\Controller\Frontend;

use App\Repository\ArticleRepository;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
```

```php
use Symfony\Component\Routing\Annotation\Route;

class MainController extends AbstractController
{
    #[Route('/', name: 'home')]
    public function index(ArticleRepository $repository): Response
    {
        $articles = $repository->findAll();

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

On utilise l'`ArticleRepository` pour chercher tous les articles dans la table et on les envoie à la vue, il ne vous reste plus qu'à modifier la vue pour avoir les articles sur la page d'accueil.

Gérer les utilisateurs

Maintenant, il nous reste une partie importante sur notre application, la gestion des utilisateurs, que ce soit pour enregistrer un nouvel utilisateur, ou pour que l'administrateur puisse modifier les droits de chaque utilisateur.

Inscription des utilisateurs

Dans un premier temps, nous allons créer la page d'inscription utilisateur.

Pour ça nous allons créer un nouveau formulaire pour les inscriptions des utilisateurs, vous pouvez générer ce formulaire avec la commande `php bin/console make:form` et vous pouvez appeler ce nouveau formulaire **RegistrationFormType**

Une fois que vous avez générez le fichier vous allez devoir le modifier légèrement :

```php
<?php

namespace App\Form;

use App\Entity\User;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
use Symfony\Component\Form\Extension\Core\Type\EmailType;
use Symfony\Component\Form\Extension\Core\Type\IntegerType;
use Symfony\Component\Form\Extension\Core\Type\PasswordType;
use Symfony\Component\Form\Extension\Core\Type\RepeatedType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;
use Symfony\Component\Validator\Constraints\Length;
```

```
use Symfony\Component\Validator\Constraints\NotBlank;
use Vich\UploaderBundle\Form\Type\VichImageType;

class RegistrationFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('email', EmailType::class, [
                'label' => 'Votre Email:',
                'required' => true,
            ])
            ->add('password', RepeatedType::class, [
                'type' => PasswordType::class,
                'first_options' => [
                    'attr' => ['autocomplete' => 'new-password'],
                    'constraints' => [
                        new NotBlank([
                            'message' => 'Entrez un mot de passe',
                        ]),
                        new Length([
                            'min' => 6,
                            'minMessage' => 'Votre mot de passe doit faire plus de {{
limit }} caractères',
                            // max length allowed by Symfony for security reasons
                            'max' => 4096,
                        ]),
                    ],
                    'label' => 'Mot de passe',
                ],
                'second_options' => [
                    'attr' => ['autocomplete' => 'new-password'],
                    'label' => 'Répétez le mot de passe',
                ],
                'invalid_message' => 'Les mot de passe ne correspondent pas.',
                // Instead of being set onto the object directly,
                // this is read and encoded in the controller
                'mapped' => false,
            ])
            ->add('prenom', TextType::class, [
                'label' => 'Prénom:',
                'required' => true,
            ])
            ->add('nom', TextType::class, [
                'label' => 'Nom:',
                'required' => true,
            ])
            ->add('address', TextType::class, [
                'label' => 'Adresse:',
```

```
                'required' => true,
            ])
            ->add('zipCode', IntegerType::class, [
                'label' => 'Code postal:',
                'required' => true,
            ])
            ->add('ville', TextType::class, [
                'label' => 'Ville:',
                'required' => true,
            ])
            ->add('imageFile', VichImageType::class, [
                'label' => 'Image: ',
                'required' => false,
                'download_uri' => false,
                'image_uri' => true,
                'by_reference' => false,
            ]);
    }

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

Maintenant que vous avez un beau formulaire, vous allez pouvoir créer la page qui peut enregistrer de nouveau utilisateurs, donc passer par un controller, vous pouvez passer par le SecurityController si vous le souhaitez et ajouter la méthode suivante :

```
#[Route('/register', name: 'register')]
public function register(
  Request $request,
  UserPasswordHasherInterface $passwordEncoder,
  UserRepository $repo
) {
  $user = new User();

  $form = $this->createForm(RegistrationFormType::class, $user);
  $form->handleRequest($request);

  if ($form->isSubmitted() && $form->isValid()) {
    // Hash the password
    $user->setPassword(
      $passwordEncoder->hashPassword(
        $user,
        $form->get('password')->getData()
      )
```

```
        );

        $repo->add($user, true);

        $this->addFlash('success', 'Vous êtes bien inscrit à notre application');

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

    return $this->renderForm('Security/register.html.twig', [
        'form' => $form,
    ]);
}
```

Et il ne vous reste plus qu'à créer votre vue.

Gestion des rôles

Maintenant que vous avez gérer l'inscription, il va falloir laisser la possibilité aux utilisateurs admin de pouvoir modifier le rôles des utilisateurs.

Pour ça vous allez créer un nouveau controller dans le dossier **src/backend** que vous nommerez UserController et dans lequel vous allez vouloir créer les méthodes CRUD (sans le create) :

```
<?php

namespace App\Controller\Backend;

use App\Entity\User;
use App\Form\UserType;
use App\Repository\UserRepository;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

#[Route('/admin/user')]
class UserController extends AbstractController
{
    #[Route('/', name: 'app_user_index', methods: ['GET'])]
    public function index(UserRepository $userRepository): Response
    {
        return $this->render('Backend/user/index.html.twig', [
            'users' => $userRepository->findAll(),
        ]);
    }

    #[Route('/{id}/edit', name: 'app_user_edit', methods: ['GET', 'POST'])]
```

```
    public function edit(Request $request, User $user, UserRepository $userRepository):
Response
    {
        $form = $this->createForm(UserType::class, $user);
        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {
            $userRepository->add($user, true);

            return $this->redirectToRoute('app_user_index', [],
Response::HTTP_SEE_OTHER);
        }

        return $this->renderForm('Backend/user/edit.html.twig', [
            'user' => $user,
            'form' => $form,
        ]);
    }

    #[Route('/{id}', name: 'app_user_delete', methods: ['POST'])]
    public function delete(Request $request, User $user, UserRepository
$userRepository): Response
    {
        if ($this->isCsrfTokenValid('delete' . $user->getId(), $request->request-
>get('_token'))) {
            $userRepository->remove($user, true);
        }

        return $this->redirectToRoute('app_user_index', [], Response::HTTP_SEE_OTHER);
    }
}
```

Je vous laisse bien entendu faire vos vue.

Si vous suivez bien, vous aurez remarqué que pour la partie admin, nous utilisons un autre FormType pour les utilisateurs, c'est normal car nous ne voulons pas que l'admin puisse modifier les informations personnel des utilisateurs, seulement les rôles.

Donc nous allons créer un nouveau FormType qui va intégrer plusieurs conditions :

- Si l'utilisateur connecté en tant qu'Admin
- Si l'utilisateur connecté est le même que l'utilisateur qui va être modifié (si oui, il aura accès à plus de champs pour modifier son profil)

Nous faisons cela pour vous montrer que nous pouvons générer des formulaires en intégrant des conditions, et que certains champs vont être affiché de manière dynamique en fonction de condition sans besoin de devoir créer plusieurs FormType pour le même objet.

Donc générez votre fichier UserType.php et modifiez-le comme ceci :

```
<?php
```

```php
namespace App\Form;

use App\Entity\User;
use Symfony\Component\Form\FormEvent;
use Symfony\Component\Form\FormEvents;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Security\Core\Security;
use Symfony\Component\Form\FormBuilderInterface;
use Vich\UploaderBundle\Form\Type\VichImageType;
use Symfony\Component\OptionsResolver\OptionsResolver;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\EmailType;
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
use Symfony\Component\Form\Extension\Core\Type\IntegerType;

class UserType extends AbstractType
{
    public function __construct(
      private Security $security
    ){
    }

    public function buildForm(FormBuilderInterface $builder, array $options)
    {
      // On ajoute une condition dans notre FormType
      $builder->addEventListener(FormEvents::PRE_SET_DATA, function (FormEvent $event)
{
        $user = $event->getData();
        $form = $event->getForm();

        // On vérifie si l'utilisateur qui est connecté est celui qui va être modifié
        if ($user == $this->security->getUser()) {
          // On ajoute les champs pour la modification de son propre profil
          $form
            ->add('prenom', TextType::class, [
              'label' => 'Prénom:',
              'required' => true,
            ])
            ->add('nom', TextType::class, [
              'label' => 'Nom:',
              'required' => true,
            ])
            ->add('address', TextType::class, [
              'label' => 'Adresse:',
              'required' => true,
            ])
            ->add('zipCode', IntegerType::class, [
              'label' => 'Code postal:',
```

```
                'required' => true,
            ])
            ->add('ville', TextType::class, [
                'label' => 'Ville:',
                'required' => true,
            ])
            ->add('imageFile', VichImageType::class, [
                'label' => 'Image: ',
                'required' => false,
                'download_uri' => false,
                'image_uri' => true,
                'by_reference' => false,
            ]);
        }

        // On vérifie si l'utillisateur connecté est un admin
        if ($this->security->isGranted('ROLE_ADMIN')) {
          // On ajoute le champ rôle pour gérer les rôles
          $form->add('roles', ChoiceType::class, [
            'choices' => [
              'Utilisateur' => 'ROLE_USER',
              'Éditeur' => 'ROLE_EDITOR',
              'Administrateur' => 'ROLE_ADMIN',
            ],
            'label' => 'Roles:',
            'required' => false,
            'expanded' => true,
            'multiple' => true,
          ]);
        }
      });
    }

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

Vous noterez que dans ce formulaire, nous n'intégrons pas la modification du mot de passe, nous allons mettre en place une structure de réinitialisation de mot de passe plus complexe et plus sécurisé (que nous ferons plus tard).

Maintenant la vue de votre formulaire d'édition d'utilisateur doit également être modifié :

```
{{ form_start(form) }}
{% if form.prenom is defined and form.nom is defined %}
```

```
  <div class="row">
    <div class="col-md-6">
      {{ form_row(form.prenom) }}
    </div>
    <div class="col-md-6">
      {{ form_row(form.nom) }}
    </div>
  </div>
{% endif %}

{% if form.imageFile is defined %}
  <div class="row">
    {{ form_row(form.imageFile)}}
  </div>
{% endif %}

{% if form.roles is defined %}
  <div class="col-md-12">
    {{ form_row(form.roles) }}
  </div>
{% endif %}

{% if form.address is defined and form.zipCode is defined and form.ville is defined %}
  <div class="row">
    <div class="col-md-8">
      {{ form_row(form.address) }}
    </div>
    <div class="col-md-4">
      {{ form_row(form.zipCode) }}
    </div>
  </div>

  <div class="col-md-6">
    {{ form_row(form.ville) }}
  </div>
{% endif %}

<div class="text-center">
  <button class="btn btn-primary">{{ button_label|default('Save') }}</button>
</div>
{{ form_row(form._token) }}
{{ form_end(form, {render_rest: false}) }}
```

Avant de vouloir afficher les champs, on vérifie qu'ils ont été généré et qu'ils existent.

Maintenant, quand vous allez modifier un user, votre formulaire va prendre en compte toute nos conditions pour afficher les bons champs en fonction de la situation.

La partie frontend des utilisateurs

Nous avons géré pour le moment seulement le cas où les admin souhaite modifier un utilisateur, mais nous n'avons pas géré le cas où un utilisateur qui est connecté souhaite afficher ces informations ainsi que modifier ces informations personnelles.

Dans un premier temps, vous allez devoir modifier votre bundle security pour ajouter une règle sur les urls qui commencent par '/compte', pour que ces pages soient disponible uniquement aux utilisateurs connecté.

Ouvrez-le fichier **config/packages/security.yaml** et faites la modification suivante :

```
access_control:
    - { path: ^/admin, roles: ROLE_ADMIN }
    - { path: ^/compte, roles: ROLE_USER }
```

De ce fait, vous avez sécurisé les urls qui commencent par '/compte' pour qu'elles soient disponible seulement aux utilisateurs connecté.

Maintenant il va falloir créer les pages de compte et d'édition pour les utilisateurs qui sont connecté, donc un nouveau controller.

Générez un nouveau controller dans le dossier frontend du nom de UserController :

```
<?php

namespace App\Controller\Frontend;

use App\Entity\User;
use App\Form\UserType;
use App\Repository\UserRepository;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Security\Core\Security;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

#[Route('/compte')]
class UserController extends AbstractController
{
    public function __construct(
        private Security $security,
        private UserRepository $repo
    ) {
    }

    #[Route('', name: 'compte')]
    public function show(): Response
    {
        $user = $this->security->getUser();
```

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

    #[Route('/edit-account', name: 'front_user_edit', methods: ['GET', 'POST'])]
    public function edit(
        Request $request,
        UserRepository $userRepository
    ): Response {

        $user = $this->security->getUser();

        $form = $this->createForm(UserType::class, $user);
        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {
            $userRepository->add($user, true);

            return $this->redirectToRoute('compte', [], Response::HTTP_SEE_OTHER);
        }

        return $this->renderForm('Backend/user/edit.html.twig', [
            'user' => $user,
            'form' => $form,
            'title_heading' => 'Editez votre profil'
        ]);
    }
}
```

Vous devez biensur créer vos vue maintenant.

Le reset password

Dernière chose à mettre en place pour les utilisateurs, c'est la fonctionnalité de changer de mot de passe, en effet pour le moment nous ne laissons pas la possibilité à nos utilisateurs de modifier leur mot de passe (par mesure de sécurité).

Nous voulons ajouter cette feature de sorte à ce qu'un utilisateur qui veut changer son mot de passe doit faire une demande sur l'application en rentrant son adresse email, et qu'automatiquement notre application lui envoie un email avec un lien sécurisé pour réinitialiser son mot de passe.

Pour ça nous allons utiliser un bundle de Symfony pour simplifier les choses.

Installation du bundle

Pour installer le bundle vous devez rentrer la commande :

```
composer require symfonycasts/reset-password-bundle
```

Une fois que vous avez installé le bundle entrez la commande :

```
php bin/console make:reset-password
```

Ce qui va vous générer tout les fichiers dont vous avez besoin pour effectuer un reset de password (même les vues !).

Ensuite vous allez devoir faire la migration en base de données des changements que le bundle à fait (création d'un table qui stocke les demande de réinitialisation de mot de passe avec les token), donc -> `php bin/console make:migration` PUIS `php bin/console doctrine:migrations:migrate`.

Configuration du bundle

Avec l'installation, vous avez un nouveau fichier **config/packages/reset_password.yaml**, c'est le fichier de configuration du bundle de reset password. Modifié le comme ceci :

```
symfonycasts_reset_password:
    request_password_repository: App\Repository\ResetPasswordRequestRepository
    lifetime: 5300
    enable_garbage_collection: true
```

Ensuite, afin de pouvoir tester l'envoi d'email, vous pouvez utiliser mailldev ou mailltrap qui va nous permettre de pouvoir tester l'envoie d'email de notre application sans avoir besoin d'avoir un serveur SMTP sous la main.

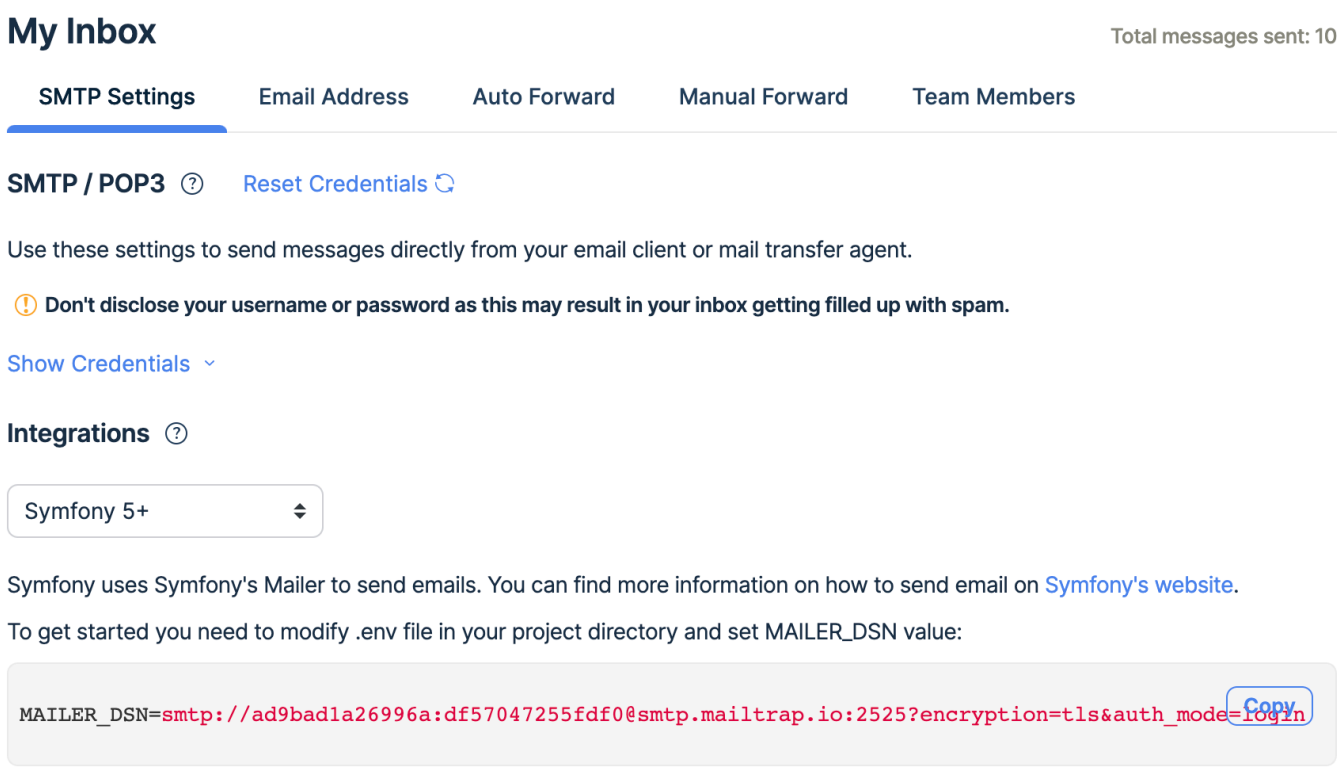
Première étapes, vous devez modifier le fichier **config/packages/messenger.yaml** :

```
framework:
    messenger:
        failure_transport: failed

        transports:
            # https://symfony.com/doc/current/messenger.html#transport-configuration
            async:
                dsn: "%env(MESSENGER_TRANSPORT_DSN)%"
                options:
                    use_notify: true
                    check_delayed_interval: 60000
                retry_strategy:
                    max_retries: 3
                    multiplier: 2
            failed: "doctrine://default?queue_name=failed"
            # sync: 'sync://'

        routing:
            # Commentez la ligne ci dessous
            #Symfony\Component\Mailer\Messenger\SendEmailMessage: async
            Symfony\Component\Notifier\Message\ChatMessage: async
            Symfony\Component\Notifier\Message\SmsMessage: async
```

```
            # Route your messages to the transports
            # 'App\Message\YourMessage': async
```

Ensuite vous allez devoir vous créer un compte sur le site de mailTrap (service gratuit).

Une fois que vous l'avez fait, vous devriez tomber sur la page d'intégration de mailtrap dans vos applications, vous devriez voir un menu déroulant avec marqué `curl`, ouvrez ce menu et sélectionnez Symfony 5+ :



Copiez ensuite la ligne qui commence par **MAILER_DSN** et allez dans le fichier .env de votre projet Symfony, cherchez la ligne qui commence par **MAILER_DSN** dans le fichier .env et remplacez-là par celle que vous avez copié sur Mailtrap.

Dernière étape lancez la commande :

```
composer dump-env dev
```

Terminez le process avec la commande :

```
php bin/console cache:clear
```

Tester le reset de password

Maintenant il ne nous reste plus qu'à tester que tout fonctionne correctement, pour ça, rendez-vous sur l'url reset-password et rentrez une adresse email d'un utilisateur existant dans votre base de données.

En soumettant le formulaire, cela va envoyer un email avec le lien pour modifier le mot de passe de l'utilisateur qui a fait la demande.

Vous pourrez retrouver ce lien en allant sur votre compte mailtrap en ligne dans l'onglet Inboxes > My inbox.

Cliquez sur le lien et modifiez le mot de passe, ensuite essayer de vous connecter avec l'utilisateur dont vous avez modifié le mot de passe pour vérifier qu'il a bien été changé.

Gestion des commentaires

Maintenant que nous avons une application fonctionnelle avec des utilisateurs, la possibilité de créer des articles, les modifier, les supprimer, nous allons pouvoir ajoute un **module de commentaires** sur les articles.

Nous voulons que chaque utilisateur identifié (connecté) puisse pouvoir laisser un commentaire pour chaque article.

La table commentaire

Première chose à faire, nous devons créer la table commentaire, pour ça faites la commande `php bin/console make:entity Comment`

Ensuite vous allez devoir ajouter les champs suivants :

- Titre -> string
- Content -> string
- createdAt -> date time
- updatedAt -> date time
- Note -> integer
- Active -> boolean
- Rgpd -> Boolean
- User -> Relation ManyToOne avec la table user
- Article -> Relation ManyToOne avec la table article

Une fois que votre classe est faite, il faut envoyer les modifications en base de données avec `php bin/console make:migration` puis `php bin/console doctrine:migrations:migrate`.

Le formulaire de commentaire

Maintenant, il faut créer un formulaire pour pouvoir laisser la possibilité de poster un commentaire, donc pour créer le formulaire, faites la commande `php bin/console make:form` et sélectionnez l'entity que nous venons de créer (Comment).

Maintenant il ne reste plus qu'à modifier notre fichier CommentType pour lui donner les champs souhaitez :

```
<?php

namespace App\Form;

use App\Entity\Comments;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\Extension\Core\Type\CheckboxType;
```

```php
use Symfony\Component\Form\Extension\Core\Type\RangeType;
use Symfony\Component\Form\Extension\Core\Type\TextareaType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;
use Symfony\Component\Validator\Constraints\Length;
use Symfony\Component\Validator\Constraints\NotBlank;

class CommentsType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('titre', TextType::class, [
                'label' => 'Titre:',
                'attr' => [
                    'placeholder' => 'Titre du commentaire',
                ],
                'required' => true,
                'constraints' => [
                    new Length([
                        'min' => 10,
                        'minMessage' => 'Le titre de votre commentaire doit être
supérieur à {{ limit }} caractères',
                        'max' => 150,
                        'maxMessage' => 'Le titre de votre commentaire ne doit pas
dépasser {{ limit }} caractères',
                    ]),
                ],
            ])
            ->add('content', TextareaType::class, [
                'label' => 'Contenu:',
                'attr' => [
                    'placeholder' => 'Contenu de votre commentaire',
                ],
                'required' => true,
            ])
            ->add('note', RangeType::class, [
                'label' => 'Note:',
                'attr' => [
                    'min' => 0,
                    'max' => 5,
                    'value' => 3,
                ],
                'help' => 'Sélectionnez une note pour l\'article',
                'required' => true,
            ])
            ->add('rgpd', CheckboxType::class, [
                'label' => 'Rgpd',
```

```
                'help' => 'En cochant cette case vous acceptez notre politique de
confidentialité',
                'constraints' => [
                    new NotBlank([
                        'message' => 'Veuillez cocher la case RGPD pour poster un
commentaire',
                    ]),
                ],
            ]);
    }

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

Envoyer le formulaire à la vue

Maintenant que nous avons tout préparé, nous allons envoyer ce nouveau formulaire à une vue pour que nos utilisateurs puissent laisser des commentaires.

Pour ça, nous allons utiliser la page de details d'un article pour afficher le formulaire de commentaires, donc rendez vous dans le fichier **src/Controller/Frontend/ArticleController.php** pour envoyer le formulaire de soumission de commentaire.

Tout d'abord nous allons devoir importer le repository pour retrouver les commentaires s'il y en a :

```
/* Constructeur de notre classe */
public function __construct(
  private ArticleRepository $repo,
  /* On ajoute le repository des commentaires pour retrouver les commentaires s'il y en
a */
  private CommentsRepository $repoComment,
) {
}
```

Ajouter une méthode de recherche custom

Avant d'aller plus loins dans l'envoi du formulaire à la vue, il va falloir qu'on ajoute une méthode de recherche custom pour retrouver les commentaires, en effet, nous allons vouloir **récupérer tous les commentaires pour 1 seul article** (étant donnée que nous allons afficher les commentaires sur la page de details d'un article).

De plus nous allons vouloir créer 2 méthodes de recherche, une pour le frontend qui affichera **seulement les commentaires actifs**, et une autre pour le backend qui affichera tout les commentaires pour les gérer.

Pour ajouter une méthode de rechercher dans une table custom, nous avons simplement à rajouter une nouvelle méthode dans le repository de la table ciblée, dans notre cas, le **CommentRepository**.

Donc rendez-vous dans le fichier pour ajouter nos méthodes de recherche, d'abord la recherche de commentaires actif pour un seul article :

```
public function findActiveByArticle(int $articleId)
{
  return $this->createQueryBuilder('C');
}
```

Pour le moment, notre nouvelle méthode envoie seulement une queryBuilder (un builder de requête), vous noterez que nous **donnons un alias à notre queryBuilder** pour lui ajouter plus facilement les options.

Autre point à noter ici, notre **méthode prend l'id de l'article en paramètre**, ce qui veut dire que quand on voudra executer cette requête, nous allons **obligatoirement devoir lui passer l'id de l'article**.

Mais maintenant, nous allons devoir lui stipuler les options de recherches il doit ajouter avant de lui demander de récupérer toutes les entrées qui correspondent à notre recherche :

```
public function findActiveByArticle(int $articleId)
{
  return $this->createQueryBuilder('c')
    /* On filtre sur l'article (on utilise la relation entre les deux tables */
    ->andWhere('c.article = :id')
    /* On attribut la valeur du marker de la ligne ci-dessus */
    ->setParameter('id', $articleId)
    /* On ne sélectionne que les commentaires actifs */
    ->andWhere('c.active = :active')
    /* On attribut la valeur du marker de la ligne ci-dessus */
    ->setParameter('active', true)
    /* On ordonnance les résultats en récupérant les derniers commentaires en premier
*/
    ->orderBy('c.createdAt', 'DESC')
    /* On récupère la requête SQL complète */
    ->getQuery()
    /* On récupère les résultats de la requête que notre méthode va renvoyer */
    ->getResult();
}
```

Maintenant nous allons faire la deuxième méthode de recherche qui va récupérer tout les commentaires (même inactif) pour un article :

```
public function findByArticle(int $articleId, string $slug)
{
  return $this->createQueryBuilder('c')
    ->join('c.article', 'a')
    ->andWhere('a.id = :articleId')
    ->setParameter('articleId', $articleId)
    ->andWhere('a.slug = :slug')
    ->setParameter('slug', $slug)
    ->orderBy('c.createdAt', 'DESC')
    ->getQuery()
    ->getResult();
}
```

C'est exactement la même chose que précédemment, mais cette fois, on ne filtre pas sur le champ actif pour récupérer tous les commentaires.

Utiliser nos recherches custom

Maintenant que nous avons nos méthode de recherche, nous allons pouvoir les utiliser, d'abord sur la partie frontend afin de récupérer seulement les articles actifs, donc dans le ArticleController (frontend), nous allons ajouter une variables comments qui va stocker les commentaires de l'article, les envoyer à la vue, et ensuite nous allons générer notre formulaire de soumission de commentaire :

```
#[Route('/article/details/{slug}', name: 'article.show')]
public function show(?Article $article, Security $security, Request $request): Response
{
  if (!$article instanceof Article) {
    $this->addFlash('error', 'Article non trouvé');

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

  /* On cherche les commentaires de notre article */
  $comments = $this->repoComment->findActiveByArticle($article->getId());

  /* On instancie le commentaire vide */
  $comment = new Comments();

  /* On génère le formulaire */
  $form = $this->createForm(CommentsType::class, $comment);
  $form->handleRequest($request);

  return $this->renderForm('Frontend/Article/show.html.twig', [
    'article' => $article,
    /* On envoie le formulaire ainsi que les commentaires à la vue */
    'form' => $form,
    'comments' => $comments,
  ]);
```

```
}
```

Maintenant il va falloir gérer la validation du formulaire pour créer un commentaire :

```php
if ($form->isSubmitted() && $form->isValid()) {
  /* On ajoute l'utilisateur connecté au commentaire */
  $comment->setUser($security->getUser())
    /* On ajoute l'article qui va être ratacché au commentaire */
    ->setArticle($article)
    /* On définit par défaut actif le commentaire */
    ->setActive(true);

  /* On ajoute le commentaire en base de données */
  $this->repoComment->add($comment, true);

  /* On redirige sur la même page pour mettre à jour les commentaires */
  $this->addFlash('success', 'Votre commentaire a été posté avec succès');

  return $this->redirectToRoute('article.show', [
    'slug' => $article->getSlug(),
  ], 301);
}
```

Envoyer à la vue

Très bien, maintenant il va falloir gérer la vue pour afficher les commentaires s'il y en a, et afficher le formulaire mais seulement si l'utilisateur est connecté, dans le cas contraire, on lui demande de se connecter.

Dans ce contexte, l'idéal est de séparer ces fichiers twig en créant un petit fichier qui va afficher les commentaires, et de l'appeler sur la page de details via un include, ce qui va éviter d'avoir un seul fichier trop lourd et incompréhensible.

Donc dans le dossier template/Frontend/Article, créez un nouveau dossier Commentaires dans lequel vous allez mettre tout les fichiers partials qui concerne les commentaires.

Commençons par l'affichage du formulaire :

```twig
{# Commentaires/_formComment.html.twig #}

{{ form_start(form) }}
{{ form_row(form.note) }}
{{ form_widget(form.titre, {'attr' : {'class': 'mt-4'} }) }}
{{ form_errors(form.titre)}}
{{ form_widget(form.content, {'attr' : {'class': 'mt-4 mb-4', 'rows': 5} }) }}
{{ form_row(form.rgpd)}}
<button type="submit" class="btn btn-primary">{{ button_label|default('Save')}}
</button>
{{ form_end(form) }}
```

Maintenant, sur le fichier article show.html.twig, nous allons devoir vérifier si l'utilisateur est connecté ou non, si oui on lui affiche le formulaire, sinon on lui affiche un message pour qu'il se connecte :

```
{# Article/show.html.twig #}

<div class="comments card p-3 mt-4">
  <h2>Ajouter un commentaire</h2>
  {% if app.user %}
    {% include "Frontend/Article/Commentaires/_formComment.html.twig" with {
button_label: 'Envoyer'} %}
  {% else %}
    <div class="alert alert-info mt-2" role="alert">
      <p>
      <b>Attention!</b>
      <br/>
      Vous devez
      <a href="{{ path('login') }}">être connecté</a>
      pour laisser un commentaire.
      </p>
    </div>
  {% endif %}
</div>
```

Et voilà, votre formulaire ne va s'afficher que si un utilisateur est connecté !

Dernière chose à faire, afficher les commentaires s'il y en a, pour a rien de plus simple, vous savez que votre variable `comments` stocke les commentaires actif, donc vous allez pouvoir l'utiliser pour afficher les commentaires sur la vue :

```
{# Article/show.html.twig #}

{% if comments|length > 0 %}
  <div class="show-comments mt-4">
    <h2>Commentaires: </h2>
    <div class="comments-list">
      {% for comment in comments %}
      {% include "Frontend/Article/Commentaires/_comment.html.twig" %}
      {% endfor %}
    </div>
  </div>
{% endif %}
```

Vous l'aurez compris, il faut maintenant créer le fichier partial `_comment.html.twig` pour afficher chaque commentaire :

```
{# Commentaires/_comment.html.twig #}

<div class="comment-item">
```

```
  <div class="comment-header">{{ comment.titre }}</div>
  <div class="comment-item-content">
    <em class="card-text">{{ comment.user.fullName }}</em>
    <p class="card-text">{{ comment.content }}</p>
    <div class="ratings">
      {% for number in range(1,5) %}
        {% if number <= comment.note %}
          <i class="fas fa-star rating-color"></i>
        {% else %}
          <i class="far fa-star rating-color"></i>
        {% endif %}
      {% endfor %}
    </div>
    <p class="card-text">
      <small class="text-muted">{{ comment.createdAt|ago }}</small>
    </p>
  </div>
</div>
```

Attention, ici nous avons utilisé un filtre twig `|ago` qui n'existe pas par défaut. Ce filtre va nous permettre de calculer depuis combien de temps le commentaire est posté et d'afficher un message. Mais il faut rajouter un bundle pour pouvoir l'utiliser, c'est le bundle knpTimeBundle, et pour l'installer, il vous suffit de rentrer la commande :

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

Une fois que vous avez installé ce bundle, votre vue va fonctionner et vous verrez depuis combien de temps votre commentaire a été posté.

Voilà pour la partie frontend des commentaires !

Gestion backend des commentaires

Maintenant que nos utilisateurs peuvent ajouter des commentaires, nous allons devoir ajouter la gestion en admin des commentaires pour pouvoir les rendre inactif ou supprimer les commentaires si besoin.

Tout d'abord, nous allons vouloir créer les nouvelles pages qui vont nous permettre de récupérer tous les commentaires d'un article.

Allez dans le fichier **src/backend/ArticleController.php** et ajoutez cette nouvelle méthode :

```
public function __construct(
  private ArticleRepository $repoArticle,
  /* On rajoute le comment repository pour pouvoir chercher dans cette table */
  private CommentsRepository $repoComments
) {
}

/* Autres méthode de la classe... */
```

```php
#[Route('/article/{id}/comments', name: 'admin.article.comments')]
public function adminComments(?Article $article)
{
  /* On vérifie si l'id en url appartient bien à un article, sinon on redirige */
  if (!$article instanceof Article) {
    $this->addFlash('error', 'Article non trouvé');

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

  /* On récupère les commentaires de l'article */
  $comments = $this->repoComments->findByArticle($article->getId());

  /* Si pas de commentaires, on redirige */
  if (!$comments) {
    $this->addFlash('error', 'Pas de commentaires trouvés');

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

  /* Si tout va bien on affiche la vue */
  return $this->render('Backend/Article/comments.html.twig', [
    'comments' => $comments,
  ]);
}
```

Ici nous avons tout d'abord **importé de manière globale le repository des commentaires**, ensuite nous avons créer une nouvelle méthode qui va nous permettre d'afficher **tous les commentaires d'un seul article**.

Mais pour le moment, nous n'avons pas encore écrite la fonction `findByArticle()` qui va nous permettre de retrouver avec l'id d'un article tous les commentaires en questions.

Pour la créer, vous allez devoir modifier votre `CommentRepository` et ajouter la méthode suivante :

```php
/* On passe en paramètre l'id de l'article concerné */
public function findByArticle(int $articleId)
{
  return $this->createQueryBuilder('c')
    /* On filtre sur le champ article_id qui doit être égale à l'id envoyé en paramètre
*/
    ->andWhere('c.article = :articleId')
    ->setParameter('articleId', $articleId)
    ->orderBy('c.createdAt', 'DESC')
    ->getQuery()
    ->getResult();
}
```

Maintenant, notre fonction va bien nous renvoyer tous les commentaires de notre article, nous pouvons passer à la phase vue.

Pour ça nous allons devoir créer une nouvelle vue **Backend/Article/comments.html.twig** :

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

{% block title %}
  Admin des commentaires |
  {{ parent() }}
{% endblock %}

{% block stylesheets %}
  {{ parent() }}
  {{ encore_entry_link_tags('admin') }}
{% endblock %}

{% block javascripts %}
  {{ parent() }}
  {{ encore_entry_script_tags('admin') }}
{% endblock %}

{% block body %}
  <section class="container mt-4 show-comments">
    <h1 class="text-center">{{ 'comment.admin.index.heading1'|trans(domain = 'content')
}}</h1>
    <div class="comments-list mt-4">
      {% for comment in comments %}
        <div class="comment-item">
          <div class="comment-header">{{ comment.titre }}</div>
          <div class="comment-item-content">
            <em class="card-text">{{ comment.user.fullName }}</em>
            <p class="card-text">{{ comment.content }}</p>
            <div class="ratings">
              {% for number in range(1,5) %}
                {% if number <= comment.note %}
                  <i class="fas fa-star rating-color"></i>
                {% else %}
                  <i class="far fa-star rating-color"></i>
                {% endif %}
              {% endfor %}
            </div>
            <div class="form-check form-switch">
              <input type="checkbox" class="form-check-input" value="{{ comment.id }}"
role="switch" data-switch-active-comment {{ comment.active ? 'checked' }}>
              <label class="form-check-label">{{ 'main.btn.enable'|trans(domain =
'content') }}</label>
            </div>
            <div class="row">
```

```
              <form method="POST" action="{{ path('admin.comment.delete', {'id':
comment.id}) }}" onsubmit="return confirm('Êtes-vous sûr de vouloir supprimer ce
commentaire ?')">
                <input type="hidden" name="_method" value="DELETE">
                <input type="hidden" name="_token" value="{{ csrf_token('delete' ~
comment.id) }}">
                <button class="btn btn-danger text-light">{{
'main.btn.delete'|trans(domain = 'content') }}</button>
              </form>
            </div>
            <p class="card-text">
              <small class="text-muted">Posté il y a
                {{ comment.createdAt|ago }}</small>
            </p>
          </div>
        </div>
      {% endfor %}
    </div>
  </section>
{% endblock %}
```

Dernière chose à faire, sur la page de liste des commentaires, il va falloir ajouter un bouton seulement s'il y a des commentaires rattaché à nos article pour afficher la page de liste des commentaires.

Donc nous allons modifier la vue qui gère la liste des articles :

```
{% if article.comments|length > 0 %}
  <div class="row">
    <div class="col-md-4 mt-2">
      <a href="{{ path('admin.article.comments', {'id': article.id}) }}" class="btn
btn-info text-light">Commentaires</a>
    </div>
  </div>
{% endif %}
```

Et maintenant sur votre page de liste des articles en admin, vous allez pouvoir voir le bouton commentaire.

Suppression des commentaires

Maintenant que nous pouvons afficher les commentaires et que nous avons fait notre vue de liste, nous devons gérer la suppression des commentaires.

Pour ça allez dans le fichier **src/Controller/Backend/ArticleController.php** et ajoutez la méthode suivante :

```
#[Route('/comment/delete/{id}', name: 'admin.comment.delete', methods: 'DELETE|POST')]
public function deleteComment(Comments $comment, Request $request)
{
  if ($this->isCsrfTokenValid('delete' . $comment->getId(), $request->get('_token'))) {
```

```
            $this->repoComments->remove($comment, true);
            $this->addFlash('success', 'Commentaire supprimé avec succès');

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

        $this->addFlash('error', 'Le token n\'est pas valide');

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

Et voilà, maintenant sur la page de liste des commentaires vous allez pouvoir supprimer un commentaire avec le bouton de suppression.

Gérer la visibilité des commentaires en Ajax

Nous avons préparé un bouton switch sur la vue des commentaires qui devait servir à mettre un commentaires actif ou inactif, pour le moment notre bouton ne sert à rien, mais maintenant, nous allons ajouter un peu de javascript est une nouvelle méthode pour pouvoir tout gérer en Ajax.

Le Js avec l'ajax

Dans un premier temps nous allons préparer notre javascript, nous voulons que quand nous cliquons sur le switch d'un des commentaires, il envoie une requête en Ajax pour changer la visibilité du commentaire.

Pour l'ajax nous allons utiliser la library **axios** qui est une library JS qui va nous permettre d'envoyer des requêtes en Ajax très simplement, pour l'installer rentrez la commande :

```
yarn add axios
```

Une fois que vous avez installez axios, vous pouvez vous créer un nouveau fichier js dans le dossier **assets/js/** et ajoutez ces lignes de code :

```
/* On importe la library Axios */
import axios from 'axios';

/* On récupère tous les inputs switch */
let switchs = document.querySelectorAll('[data-switch-active-comment]');

/* Si switchs n'est pas vide */
if (switchs) {
    /* On fait une boucle foreach pour parcourir chaque élément du tableau */
    switchs.forEach((element) => {
        /* On ajoute une écoute d'événement au changement */
        element.addEventListener('change', () => {
            /* On récupère l'id du commentaire dans la valeur de l'input */
            let commentId = element.value;
            /* On envoie la requête en ajax */
            axios.get(`/admin/comments/switch/${commentId}`);
        });
```

```
    });
}
```

Dernière chose à faire pour que cela fonctionne, importer ce nouveau fichier js à notre admin.js :

```
// Import scss
import './styles/admin.scss';

// Import js
import './js/visibilityComments';
```

Et ensuite finir avec la commande `yarn watch`.

Maintenant, nous envoyons une requête Ajax avec l'id du commentaire à chaque changement sur l'input switch, mais le problème c'est qu'aucune route n'existe pour le moment, c'est pour ça que nous allons devoir rajouter une **nouvelle méthode** dans notre **ArticleController** de la partie Admin :

```
#[Route('/comments/switch/{id}', name: 'admin.comments.switch', methods: 'GET')]
public function switchVisibilityComment(?Comments $comment)
{
  if (!$comment instanceof Comments) {
    return new Response('Commentaires non trouvé', 404);
  }

  if ($comment) {
    /* Si le commentaire existe on change le champs actif à l'inverse de ce qu'il était
avant */
    $comment->isActive() ? $comment->setActive(false) : $comment->setActive(true);
    $this->repoComments->add($comment, true);

    return new Response('Visibility changed', 201);
  }
}
```

Et voilà, votre visibilité d'article en Ajax est fonctionnelle !

Vous avez maintenant toues les bases de Symfony, il ne reste plus qu'à pratiquer et de toujours plus se documenter sur la documentation officielle de Symfony !