Démarrer avec Symfony 7
Blog

TABLE DES MATIERES

Section 1 Créer le projet

1.1 But

- Créer le projet Blog
- Utiliser Composer

1.2 Enoncé

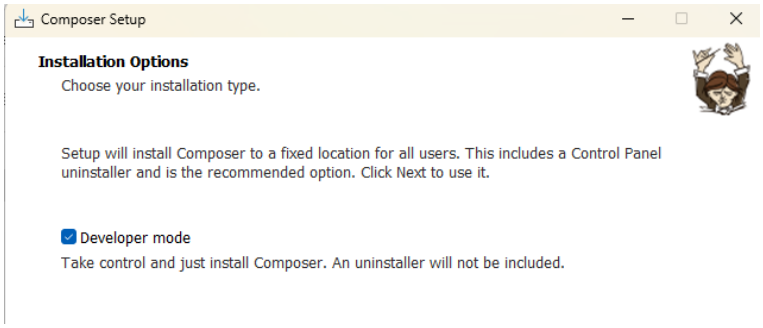
Vérifier les versions de PHP . Dans une fenêtre cmd :

`php -v` La version doit être supérieure ou égale à 8.2

Récupérer Composer sur le site https://getcomposer.org/

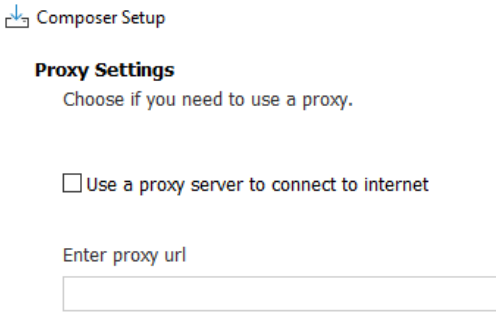
L'installer

Clic sur Developper mode



Vérifier que la versoin PHP est ≥ 8.2

Laisser vide la coche suivante :



composer –V Montre la présence de Composer

Créer depuis un explorateur de fichiers un répertoire pour accueillir le projet blog. Pas nécessaire d'être sous www de Wamp car on n'utilisa pas le serveur Apache de Wamp.

Avec la fenêtre cmd se déplacer dans ce répertoire par cd

On peut réduire le prompt de la fenêtre par :

```
set prompt= $G
```

Si l'on tapait la commande suivante, c'est symfony 6 qui serait installé (en 12/2023)

```
composer create-project symfony/website-skeleton blog
```

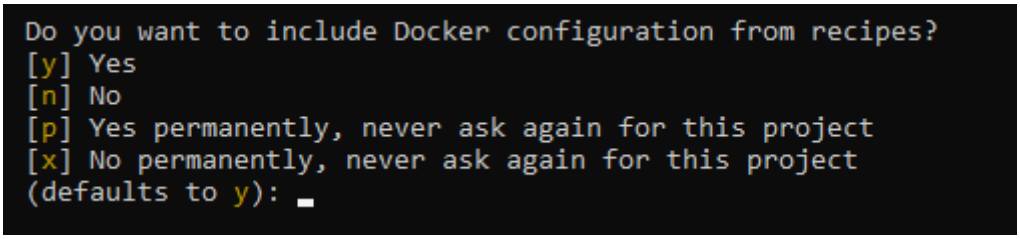
taper plutôt la commande suivante pour utiliser Symfony 7:

```
composer create-project symfony/skeleton:"7.0.*" blog
cd blog
composer require webapp
```

Pour cette commande il est peut être nécessaire d'autoriser l'antivirus à ne pas protéger cet espace blog.

De plus l'option –no-cache est utile si le PC dispose d'une ancienne version devenue incompatible.

Aux questions suivantes répondre np

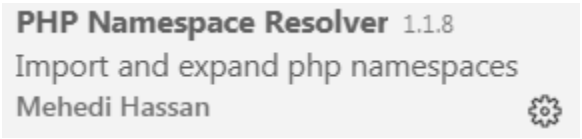


Puis

```
cd blog
```

lancer VS Code en tapant **code .**

Ajouter dans VS code l'extension « PHP Namespace Resolver » de Mehedi Hassan (utilisable par ctrl+Alt+i)



VS code dispose d'une fenêtre de commande que nous allons utiliser (évite de changer de fenêtre).

Cette fenêtre est accessible depuis Terminal => New terminal

```
php -S localhost:8000 -t public -d display_errors=1
```

Une alerte d'un antivirus peut apparaître sur la menace IDP.ALEXA.51 causée par le fichier router.php. Ajouter une exception à l'antivirus pour cette menace pour continuer.

Dans un navigateur Web, utiliser l'URL localhost:8000

On doit obtenir ceci :

Regarder avec VS Code la structure du projet.

Nous allons travailler dans src et templates

Le travail sur le site qui suit est inspiré des articles

https://www.youtube.com/watch?v=UTusmVpwJXo&fbclid=IwAR0PTGWtdHIba0ZQUqMEDmq5blW-vkhupJd85ycxXvU8K8OV_yLxsF3Wo1Q

Il est conseillé d'utiliser un outil de versionning. Nous choisissons ici Git

Télécharger Git et l'installer. https://git-scm.com/download/win

Utiliser TurtoiseGit pour disposer d'une interface graphique à Git. https://tortoisegit.org/download/

Section 2 Créer un Controller

2.1 But

- Créer un Controller
- Utiliser Composer
- Utiliser le contenu de variables

2.2 Enoncé

Dans l'arborescence du projet, le répertoire src/Controller va contenir les différents controllers.

Nous allons créer notre 1er controller :

BlogController

Dans la fenêtre cmde tapper

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

Indiquer le nom du controller avec la convention CamelCase :

```
BlogController
```

Cette commande a créé les fichiers src/Contoller/BlogController.php et templates/blog/index.html.twig

Regarder src/Contoller/BlogController.php

Remarquer l'annotation #[Route()]

(@Route dans des versions précédentes de Symfony)

Arrêter le serveur web depuis VS code et le relancer dans une fenêtre cmde isolée (en dehors de VS code) par

```
php -S localhost:8000 -t public -d display_errors=1
```

Appeler la page /blog dans le navigateur

Nous allons ajouter une page d'accueil.

Dans BlogController.php ajouter

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

Dans des versions précédentes le code était :

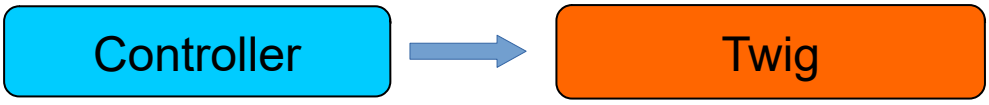
```
/**
* @Route("/", name="home")
*/
public function home()
{
    return $this->render('blog/home.html.twig');
}
```

Créer le fichier blog/home.html.twig

Mettre un titre h1 et un paragraphe (utiliser dans VS code h1 tab et p>lorem40*2 tab)

Visualiser cette page

Nous allons partager des variables entre le Controller et cette vue :



Dans cette vue blog/home.html.twig ajouter :

```
<h2>{{title}}</h2>

{% if age > 18%}
    <p>Tu es Majeur</p>
{% else %}
    <p>Tu es Mineur</p>
{% endif %}
```

Dans BlogController.php remplacer le code de home() par celui-ci :

```
public function home():Response
```

```
{
  return $this->render('blog/home.html.twig',[
    'title'=>"Utilisation d'une variable",
    'age'=>27
  ] );
}
```

La partie basse de la page doit ressembler à ceci :

Regarder le lien https://twig.symfony.com/doc/2.x/templates.html fournit des informations sur les possibilités de Twig 2.x

Essayer quelques syntaxes :

- créer une chaîne localement au fichier twig avec le contenu « <h1>Ceci est un titre </h1>
- l'afficher
- l'afficher à nouveau sans les tags html et en mettant en majuscule chaque nouveau mot
- afficher la date courante

Section 3 Le rendu du site

3.1 But

- Utiliser une bibliothèque de présentation de type Bootstrap

3.2 Enoncé

Il faut d'abord vérifier que VS Code est sensible aux abréviations "emmet" dans des pages html.twig.

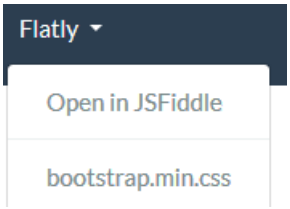
Si ce n'est pas le cas, dans VS Code File/Preferences/Settings/Extensions/Emmet puis Edit in settings.json. Y mettre ce code :

```
{
        "files.associations": {"*.twig": "html"}
}
```

Nous allons définir le rendu général du site.

Utilisons https://bootswatch.com

Choisir le thème Flatly

Dans le haut de la page cliquer sur 

boorstap.min.css avec le bouton du milieu. Copier le lien vers ce css source.

Ouvrir templates/**base.html.twig**

Dans la partie <head> ajouter un <link> vers le lien css précédent. C'est-à-dire les lignes suivantes :

```
    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.1/dist/js/bootstrap.min.js"
integrity="sha384-
Atwg2Pkwv9vp0ygtn1JAojH0nYbwNJLPhwyoVbhoPwBhjQPR5VtM2+xf0Uwh9KtT"
crossorigin="anonymous"></script>
  <link rel="stylesheet"
href="https://bootswatch.com/5/flatly/bootstrap.min.css">
```

On doit obtenir ceci :

```
<html>
    <head>
        <meta charset="UTF-8">
        <title>{% block title %}Welcome!{% endblock %}</title>
        <link rel="icon" href="data:image/svg+xml,<svg xmlns=%22http://www.w3.org/2000/svg%22 viewBo
        <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.1/dist/js/bootstrap.min.js" integrit
        <link rel="stylesheet" href="https://bootswatch.com/5/flatly/bootstrap.min.css">
        {% block stylesheets %}
            {{ ux_controller_link_tags() }}
        {% endblock %}
```

Ajouter la Navbar du thème de la façon suivante :

Sur le site Flatly, copier le code du navbar de votre choix.

Le coller dans le haut, juste sous la balise <body> de **base.html.twig**

Regarder la page /blog

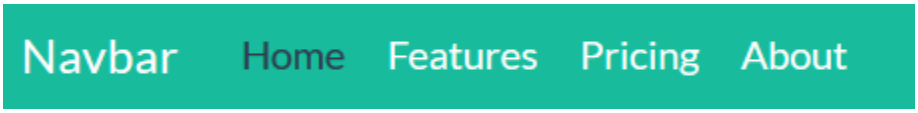
Le navbar doit être visible.

Ce n'est pas encore le cas de la page d'accueil

Dans home.html.twig ajouter en se basant sur le contenu de index.html.twig

```
{% extends 'base.html.twig' %}
{% block body %}
{% endblock %}
```

Essayer la page



Revenir dans base.html.twig

Enlever les items superflus du navbar pour obtenir .

Pour ce faire les lignes suivantes montrent les liens href retenus :

```
<a class="navbar-brand" href="/">Symfony Blog</a>
<a class="nav-link" href="/blog">Articles<span class="sr-
only">(current)</span></a>
<a class="nav-link" href="/blog/new">Créer un article</a>
```

Pour une bonne présentation, dans le fichier base.html.twig, le bloc <body> doit être placé dans un container

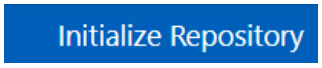
```
<div class="container">
    {% block body %}{% endblock %}
</div>
```

Essayer la page. Elle doit se présenter ainsi :

3.3 Réaliser une archive locale git

On suppose que git a été installé sur le poste de travail.

Sous VS Code appui sur 

Clic sur 

Appui sur le + de Changes 

Saisir un label correspondant au commit (l'archivage)

Section 4 Améliorer la page des articles

4.1 But

- Modifier une page twig
- Ajouter une page de détails

4.2 Enoncé

Nous allons maintenant mettre un contenu plus riche dans la page des articles avec quelques articles

Dans la page index.html.twig :

- Supprimer tout le contenu du bloc <body>
- Ajouter un article et <h2> en utilisant les snippets Emmet wtf suivant

section.articles puis tab

- A l'intérieur de cette section :

(article>h2{Titre de l'article}+div.metadata{rédigé le 10/10/2019}+div.content>img+(p>lorem20)*2+a.btn.btn-primary{Lire la suite})*3 puis tab

- Dans chaque <img src="" placer http://placehold.it/350x150

Essayer la vue. On doit voir 3 fois cette partie

Pour le fun il est possible d'afficher des images aléatoires avec https://picsum.photos/200/300 Modifier si vous voulez

4.3 Page de détail

Nous souhaitons maintenant afficher des pages montrant le détail de chaque article.

Dans BlogController.php ajouter

```
    public function show()
    {
        return $this->render('blog/show.html.twig');
    }
```

Ajouter l'annotation pour définir la route /blog/12

et le nom 'blog_show'

Dans la page index.html.twig l'appui du bouton du 1er article doit appeler la page show.html.twig

Affecter le href du bouton comme suit

```
<a href="{{ path('blog_show')}}" class="btn btn-primary">Détail de
l'article</a>
```

blog_show est le nom logique du lien que sait interpréter la commande twig path()

Regarder la page depuis le navigateur et constater l'URL associée aux boutons

Localhost:8000/blog/12

Créer la page show.html.twig avec le contenu suivant

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

{% block title %}Un article{% endblock %}

{% block body %}
    <article>
        <h2>Titre de l'article</h2>
        <div class="metadata">rédigé le 10/10/2019</div>
        <div class="content">
            <img src="http://placehold.it/350x150" alt="">
            <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit.
Accusamus at laborum ducimus accusantium illum totam, suscipit enim cumque!
Illum, cum.</p>
            <p>Itaque esse reiciendis culpa, consequuntur ipsa nemo unde ad?
Dolorem tempore quos, commodi consequuntur atque enim eaque molestias beatae
error!</p>
            <hr>
            <p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Et,
magnam aspernatur quae nihil maxime pariatur animi ducimus recusandae dolore
nesciunt assumenda fugit numquam, exercitationem voluptatem explicabo dolores
odit delectus veniam laudantium saepe facere. Assumenda, itaque accusamus amet
```

```
quas culpa totam consequuntur enim ut sunt? Aut ipsum cum ab expedita atque?
</p>
            <p>Saepe perferendis fugit reiciendis alias quae repellat ad
accusantium qui soluta quod ab dolores corporis neque, nulla impedit tempore
harum, molestias enim quis itaque consequuntur modi vero earum officia.
Exercitationem dolorem quasi maiores fugit ipsam consectetur similique ipsum
corporis iusto nam hic, ullam quam aut aliquid. Aspernatur blanditiis
accusamus fugiat!</p>
        </div>
    </article>

{% endblock %}
```

Essayer la vue de détail

Archiver avec git

Section 5 Utiliser une base de données

5.1 But

- Créer une base de données
- Emploi de Doctrine

5.2 Enoncé

Avec Visual Studio Code, dans le fichier .env modifier la ligne

DATABASE_URL=mysql://root:@127.0.0.1:3306/blog

Lancer le serveur MySQL avec Wamp

Dans la fenêtre console de VS code créer la base de données :

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

La base de données blog est créée. On peut le vérifier avec phpMyAdmin

Pour créer la table Article et classe associée :

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

A la Class name demandée, répondre **Article**

Remarquer que la ligne de commande indique que 2 fichiers sont créés :

created: src/Entity/Article.php

created: src/Repository/ArticleRepository.php

Continuons à répondre aux questions sur les champs (les colonnes). Nous allons créer :

title string non nullable

content text non nullable

image string non nullable

createdAt datetime non nullable

Quelques précisions sur createdAt :

- Écrit au format camelCase
- Sera traduit en SQL par un created_at
- La commande make:entity est sensible à “At” et propose par défaut un type datetime

Regarder le code de src/Entity/Article.php et src/Repository/ArticleRepository.php

Pour autant la base de données ne contient pas encore cette table. Exécuter :

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

Ceci va créer un fichier de migration dans src/Migrations. Regarder ce fichier.

Puis produire cette migration par la commande

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

Vérifier avec phpMyAdmin

Pour information :

- Le site

  https://symfony.com/doc/master/bundles/DoctrineMigrationsBundle/index.html
  fournit les possibilités de **`doctrine:migrations`**
- Essayer :
- **`php bin/console doctrine:migrations:status`**
- **`php bin/console doctrine:migrations:up-to-date`**

Faire un commit git

Section 6 Créer un jeu de données

6.1 But

- Créer une Fixture : un jeu de données
- Emploi de Doctrine

6.2 Enoncé

Créer un jeu de données : une fixture

D'abord installer le composant par :

`composer require orm-fixtures --dev`

puis

`php bin/console make:fixtures`

`avec ArticlesFixtures`

Editer le fichier généré src/DataFixtures/ArticlesFixtures.php

Remplacer le contenu actuel par celui-ci :

```php
<?php

namespace App\DataFixtures;

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

class ArticlesFixtures extends Fixture
{
    public function load(ObjectManager $manager):void
    {
        for($i = 1; $i <= 10 ; $i++)
        {
            $article = new Article();
            $article->setTitle("Titre de l'article n°$i")
                ->setContent("<p>Contenu article $i</p>")
                ->setImage("http://placehold.it/350x150")
                ->setCreatedAt(new \Datetime());
            $manager->persist($article);
        }
        $manager->flush();
    }
```

```
}
```

Remarques sur ce contenu :

- Le “ `use App\Entity\Article;`” est nécessaire pour que le `new Article()` fonctionne
- Comprendre pourquoi l’imbrication des setters est possible

  ex `$article->setTitle()->setContent()-> …`

- Le ‘\’ devant \Datetime() sigifie que Datetime() fait partie du namespace global de PHP, pas de use en tête du fichier.
- `$manager->persist($article); enregistre l’objet dans le manager`
- `$manager->flush();` réalise les requête SQL et les envoie au serveur Mysql

Pour faire prendre en compte les fixtures, exécuter la commande :

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

A ce stade la base de données et table Article existent, avec un jeu de test.

Vérifier le contenu depuis phpMyAdmin

title	content	image	created_at
Titre de l'article n°1	Contenu article 1	http://placehold.it/350x150	2019-11-01 15:11:22
Titre de l'article n°2	Contenu article 2	http://placehold.it/350x150	2019-11-01 15:11:22
Titre de l'article n°3	Contenu article 3	http://placehold.it/350x150	2019-11-01 15:11:22
Titre de l'article n°4	Contenu article 4	http://placehold.it/350x150	2019-11-01 15:11:22
Titre de l'article n°5	Contenu article 5	http://placehold.it/350x150	2019-11-01 15:11:22
Titre de l'article n°6	Contenu article 6	http://placehold.it/350x150	2019-11-01 15:11:22
Titre de l'article n°7	Contenu article 7	http://placehold.it/350x150	2019-11-01 15:11:22
Titre de l'article n°8	Contenu article 8	http://placehold.it/350x150	2019-11-01 15:11:22
Titre de l'article n°9	Contenu article 9	http://placehold.it/350x150	2019-11-01 15:11:22
Titre de l'article n°10	Contenu article 10	http://placehold.it/350x150	2019-11-01 15:11:22

Noter le nom du champ created_at, créé par ceatedAt

Il est possible de demander directement à Doctrine les données :

```
php bin/console doctrine:query:sql "select * from article"
```

Section 7 Utiliser les données

7.1 But

- Compléter le code du controller
- Emploi de Doctrine

7.2 Enoncé

Lien Doctrine :

https://symfony.com/doc/current/doctrine.html#migrations-adding-more-fields

https://www.doctrine-project.org/projects/doctrine-orm/en/2.6/tutorials/getting-started.html#entity-repositories

7.2.1 Modifier le Controller

Il faut utiliser cette base dans BlogController.php

Modifier la fonction index() comme suit

```
#[Route('/blog', name: 'app_blog')]
  public function index(ManagerRegistry $doctrine)
  {
    $repo = $doctrine->getRepository(Article::class);
    $articles = $repo->findAll();
    return $this->render('blog/index.html.twig', [
      'controller_name' => 'BlogController',
      'articles' => $articles
    ]);
  }
```

Ajouter

```
use App\Entity\Article;
```

Utiliser le use adéquat pour résoudre l'erreur sur `ManagerRegistry`

On utilise ici de « l'injection de dépendance » expliqué plus loin.

Dans des versions antérieures de Symfony le code suivant était possible :

```
public function index()
{
    $repo = $this->getDoctrine()->getRepository(Article::class);
    $articles = $repo->findAll();
    return $this->render('blog/index.html.twig', [
        'controller_name' => 'BlogController',
        'articles' => $articles
    ]);
}
```

7.2.2 Modifier la vue

Modifier index.html.twig. Les 3 articles en dur sont remplacés par une séquence avec une boucle for

```
<section class="articles">
    {% for article in articles%}
        <article>
            <h2>{{article.title}}</h2>
            <div class="metadata">rédigé le {{article.createdAt |
date('d/m/Y')}} à {{article.createdAt | date('H:i')}}</div>
            <div class="content">
                <img src="{{article.image}}" alt="">
                {{article.content | raw}}
                <a href="{{ path('blog_show')}}" class="btn btn-primary">Lire
la suite</a>
            </div>
        </article>
    {% endfor %}
 </section>
```

Date() et raw sont des **filtres twig**

https://twig.symfony.com/doc/2.x/filters/index.html

raw permet de considérer des tags html dans les données …

Essayer dans le navigateur.

Section 8 Route paramétrée

8.1 But

- Implémenter une route paramétrée

8.2 Enoncé

Tous les boutons pointent en fixe sur l'article 12.

Ceci vient de la définition de la Route

```
#[Route('/blog/12', name: 'blog_show')]
```

ou ceci en Symfony 5.x

```
@Route("/blog/12", name="blog_show")
```

Nous allons définir une route paramétrée.

Modifier BlogController.php :

```
#[Route('/blog/{id}', name: 'blog_show')]
```

ou ceci en Symfony 5.x

```
    /**
     * @Route("/blog/{id}", name="blog_show")
     */
```

Puis modifier show() comme suit :

```
  public function show(ManagerRegistry $doctrine, $id)
  {
    $repo = $doctrine->getRepository(Article::class);
    $article = $repo->find($id);
    return $this->render('blog/show.html.twig',[
      'article' => $article
    ]);
  }
```

Il faut adapter show.html.twig pour réceptionner les valeurs réelles de l'article

Mettre ce nouveau Body :

<article>

```
    <article>
        <h2>{{article.title}}</h2>
        <div class="metadata">rédigé le {{article.createdAt | date('d/m/Y')}}
à {{article.createdAt | date('H:i')}}</div>
        <div class="content">
            <img src="{{article.image}}" alt="">
            {{article.content | raw}}
        </div>
    </article>
```

Adapter aussi index.html.twig avec du code javascript pour fournir l'id {'id':article.id})}

Il faut ajouter cette information ou second paramètre de path()

Dans le code suivant modifier la partie path()

```
{% block body %}
<section class="articles">
    {% for article in articles%}
        <article>
            <h2>{{article.title}}</h2>
            <div class="metadata">rédigé le {{article.createAt |
date('d/m/Y')}} à {{article.createAt | date('H:i')}}</div>
            <div class="content">
                <img src="{{article.image}}" alt="">
                {{article.content | raw}}
                <a href="{{ path('blog_show', {'id':article.id})}}"
class="btn btn-primary">Lire la suite</a>
            </div>
        </article>
    {% endfor %}
 </section>
{% endblock %}
```

Section 9 Injection de dépendance

9.1 But

- Voir plusieurs formes d'injection de dépendances

9.2 Enoncé

L'injection de dépendance , le service container

https://symfony.com/doc/current/service_container.html

actuellement dans BlogConroller.php l'injection de dépendance est déjà utilisée :

```
public function index(ManagerRegistry $doctrine)
```

L'objet de classe `ManagerRegistry` est alloué par le service container à **un seul exemplaire**

On peut cependant ici faire un code plus direct : faire allouer directement un objet `ArticleRepository`

Dans BlogConroller.php ajouter

```
use App\Repository\ArticleRepository;
```

Modifier

```
public function index(ArticleRepository $repo)
```

et

```
public function show(ArticleRepository $repo, $id)
```

Commenter les lignes

```
$repo = $this->getDoctrine()->getRepository(Article::class);
```

Constater que les pages Articles et de détail fonctionnent toujours.

Section 10 Créer un formulaire

10.1 But

- Créer un formulaire

10.2 Enoncé

https://symfony.com/doc/current/forms.html

Dans BlogConroller.php ajouter avant la méthode show()

```
#[Route('/blog/new', name: 'blog_create')]
public function create()
{
    return $this->render('blog/create.html.twig');
}
```

Il est nécessaire de placer create() avant show() pour ne pas avoir de conflit dans les url blog/new et blog/1

Créer la vue blog/create.html.twig avec le contenu suivant :

```
{% extends 'base.html.twig' %}
{% block body %}
<h1>Création d'un article</h1>
{% endblock %}
```

Essayer la page

Modifier base.html.twig pour utiliser un chemin dynamique avec path() dans le Navbar, à la place de /blog/new

```
<a class="nav-link" href="{{path('blog_create')}}">Créer un article</a>
```

Faire la même chose pour les liens /blog et /

Nous allons créer un formulaire dans create.html.twig avec une méthode “classique”

Dans create.html.twig créer un formulaire avec les commandes emmet

```
form:post
input[name="title" placeholder="titre"]
```

```
textarea[name="content" placeholder="Contenu"]
input[name="image" placeholder="Image"]
input:submit
```

Essayer ce formulaire.

Il faut ensuite alimenter les champs de ce formulaire : dans BlogController.php

```
    #[Route('/blog/new', name: 'blog_create')]
    public function create(Request $request, ManagerRegistry $doctrine
)
    {
        dump($request);
        if ($request->request->count() > 0 )
        {   // retour de formulaire
            $article = new Article();
            $article->setTitle($request->request->get('title'))
            ->setContent($request->request->get('content'))
            ->setImage($request->request->get('image'))
            ->setCreatedAt(new \DateTime());

            $manager = $doctrine->getManager();
            $manager->persist($article);
            $manager->flush();

            return $this->redirectToRoute('blog_show', ['id' => $article-
>getId()]);
        }
        return $this->render('blog/create.html.twig');
    }
```

Puis ajouter les use :

```
use Symfony\Component\HttpFoundation\Request;
use Doctrine\Persistence\ObjectManager;
```

Essayer ce formulaire.

Saisir au moins un article.

Est-ce que la faille XSS existe ?

Fonctionne mais fastidieux et manque la vérification du contenu de chaque champ;

Faire un commit git

Section 11 Utiliser le générateur de formulaire

11.1 But

- Créer un formulaire automatiquement par Symfony

11.2 Enoncé

Nous allons utiliser le générateur de formulaire de Symfony

https://symfony.com/doc/current/forms.html

dans BlogController.php remplacer le contenu actuel de create() par ceci ::

```
public function create(Request $request, ObjectManager $manager)
{
    $article = new Article();

    $form = $this->createFormBuilder($article)
                ->add('title')
                ->add('content')
                ->add('image')
                ->getForm();
    dump ($form);
    dump($form->createView());
    return $this->render('blog/create.html.twig', [
        'formArticle' => $form->createView()
    ]);
}
```

Dans create.html.twig

Commenter le précédent formulaire et remplacer par

```
{{ form(formArticle)}}
```

Afficher la page.

Noter que le champ content est un text area, automatiquement détecté par Symfony du fait de son type.

Il est possible de forcer le choix. Ex

```
add('content', TextType::class)
```

ajouter dans ce cas la référence

```
use Symfony\Component\Form\Extension\Core\Type\TextType;
```

Se référer à https://symfony.com/doc/current/reference/forms/types.html

ajouter la saisie de la date createdAt et constater que le formulaire propose un calendrier.

Ce formulaire n’a pas de submit. Le rajouter.

Le traitement du formulaire n’existe pas. Ajouter ce traitement en s’inspirant du traitement du formulaire précédent.

Section 12Utiliser un thème pour le formulaire

12.1 But

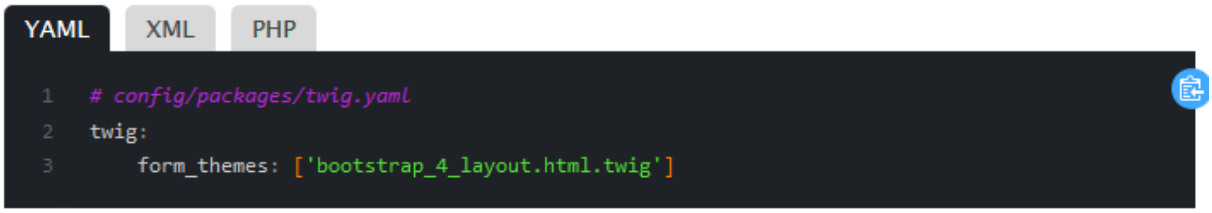
- Utiliser un thème Bootstrap

12.2 Enoncé

Nous allons employer un thème Bootstrap

https://symfony.com/doc/current/form/bootstrap4.html

Aller sur ce site et copier la ligne form_themes



Dans le fichier config/packages/twig.yaml ajouter cette ligne form_themes.

On doit obtenir :

```
twig:
    default_path: '%kernel.project_dir%/templates'
    debug: '%kernel.debug%'
    strict_variables: '%kernel.debug%'
    form_themes: ['bootstrap_4_layout.html.twig']
```

Dans create.html.twig copier en tête, après extends, la ligne en provenance du même site

```
{% form_theme formArticle 'bootstrap_4_layout.html.twig' %}
```

Afficher la vue

Pour appliquer des attributs à ce formulaire, utiliser form-row() .

Dans create.html.twig remplacer le contenu par :

```
{% extends 'base.html.twig' %}
{% form_theme formArticle 'bootstrap_4_layout.html.twig' %}
{% block body %}
    <h1>Création d'un article</h1>
    {{form_start(formArticle)}}
```

```
    {{form_row(formArticle.title, {'attr':{'placeholder' : "Titre de
l'article"},
        'label':'Titre'})}}
    {{form_row(formArticle.content, {'attr':{'placeholder' : "Contenu de
l'article"},
        'label':'Contenu'})}}
    {{form_row(formArticle.image, {'attr':{'placeholder' : "URL de
l'image"}})}}

    <button type="submit">Ajouter</button>

    {{form_end(formArticle)}}


{% endblock %}
```

Essayer la page

faire un commit git

Section 13 Utiliser et traiter les informations du formulaire

13.1 But

- Utiliser les données du formulaire pour créer un article en base de données

13.2 Enoncé

En fin de chapitre 11 le traitement a dû être fait. Nous voyons ici un autre code.

Il faut maintenant exploiter les données du formulaire pour persister l'objet en BD :

Dans BlogController.php, remplacer le contenu de create()

```
    public function create(Request $request, ManagerRegistry $doctrine)
    {
        $manager = $doctrine->getManager();
        $article = new Article();

        $form = $this->createFormBuilder($article)
                    ->add('title')
                    ->add('content')
                    ->add('image')
                    ->getForm();
        $form->handleRequest($request);  // fait le bind entre les éléments du
formulaire et objet Article
        dump($article);
        if ($form->isSubmitted() && $form->isValid())
        {
            $article->setCreatedAt(new \DateTime());
            $manager->persist($article);
            $manager->flush();

            return $this->redirectToRoute('blog_show', ['id' => $article-
>getId()]);
        }

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

Regarder le résultat et ajouter un article

Section 14Utiliser le même formulaire pour les modifications

14.1 But

- Utiliser le formulaire pour modifier un article existant

14.2 Enoncé

On souhaite que le même formulaire serve pour créer et mettre à jour des articles.

Create() devient

```
#[Route('/blog/new', name: 'blog_create')]
#[Route('/blog/{id}/edit', name: 'blog_edit')]
```

Avec les route Symfony 5 :

```
    /**
     * @Route("/blog/new", name="blog_create")
     * @Route("/blog/{id}/edit", name="blog_edit")
     */
    public function createModify(Article $article = null, Request $request,
ManagerRegistry $doctrine)
    {
        if (!$article)
        {
            $article = new Article();
        }

        $form = $this->createFormBuilder($article)
                    ->add('title')
                    ->add('content')
                    ->add('image')
                    ->getForm();
        $form->handleRequest($request); // fait le bind entre les éléments du
formulaire et objet Article
        dump($article);
        if ($form->isSubmitted() && $form->isValid())
        {
            if (!$article->getId())
            {
                $article->setCreatedAt(new \DateTime());
            }
            $manager = $doctrine->getManager();
            $manager->persist($article);
            $manager->flush();

            return $this->redirectToRoute('blog_show', ['id' => $article-
>getId()]);
        }
```

```
        return $this->render('blog/create.html.twig', [
            'formArticle' => $form->createView(),
            'editMode' => $article->getId() !== null
        ]);
    }
```

Dans create.html.twig le code du bouton submit devient :

```
<button type="submit">
    {% if editMode %}
        Modifier l'article
    {% else %}
        Ajouter
    {% endif %}
</button>
```

Essayer la page par exemple avec 12/edit

Section 15 Création automatique de formulaire et validation

15.1 But

- Création automatique d'un formulaire
- Ajouter des contraintes aux éléments de formulaire pour validation

15.2 Enoncé

Dans une fen cmd :

`php bin/console make:form`

Avec ArticleType et Article

Le fichier src/form/ArticleType.php est créé. L'éditer et regarder. Enlever la date de création.

Modifier BlogController.php pour utiliser ce formulaire :

Commenter $form = actuel

Ajouter

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

Ajouter aussi

```
use App\Form\ArticleType;
```

Cette méthode permet de disposer d'une classe qui alimente le formulaire et qui peut être employée à plusieurs endroits si nécessaire.

Nous allons ajouter des validations aux champs du formulaire

https://symfony.com/doc/current/validation.html

Ceci va être fait en ajoutant des annotations dans Article.php

Ajouter

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

Pour title ajouter l'annotation

En Symfony 7

```
#[Assert\Length(min:10, max :255)]
```

En Symfony ≤ 5.1

```
@Assert\Length(min=10, max =255)
```

Pour content

```
@Assert\Length(min=10)
```

Pour image

```
@Assert\Url()
```

Tester le formulaire et éventuellement inspecter les contraintes html ajoutées.

Section 16 Gestion d'autres Entités

16.1 But

- Ajout d'autres tables en relation avec la table Article
- Gérer ces deux tables par Symfony

16.2 Enoncé

Fonctionnellement un Article est associé à une Catégorie et à un commentaire.
Un Article possède une seule catégorie.
Une catégorie peut être associée à plusieurs Articles

Création de la table Catégorie

`php bin/console make:entity`

`Category`

`title` string 255 non null

`description` text null possible

`articles`

Il faut ajouter une relation . Taper ?

Choisir

de type **`relation`** relié à la classe **`Article`** en **`OneToMany`**

Ajouter dans la table **`Article`** le champ **`category`** proposé par défaut

Pas d'article sans catégorie, répondre no

No sur delete orphaned

Regarder le contenu de src/Entity/Article.php et Category.php

Il faut ensuite créer une migration :

`php bin/console make:migration`

Sous PHPMyAdmin, supprimer les données de Article (car `category_id INT NOT NULL`

`php bin/console doctrine:migrations:migrate`

Vérifier que la table Category est créée.

Idem pour commentaire :

php bin/console make:entity

Comment

author string 255 non null

content text non null

createdAt datetime non null

Il faut ajouter une relation pour associer un article et un commentaire.

Choisir

article

relation

Article

ManyToOne

no (pas de commentaire sans article)

yes (disposer de $article->getComments())

comments (choix par défaut)

yes (delete orphaned)

Regarder le code de Comment

Il faut ensuite créer une migration :

php bin/console make:migration

php bin/console doctrine:migrations:migrate

Regarder la base de données

Il faut maintenant faire un jeu de données : faire une fixture

Nous allons utiliser un générateur de contenu plus diversifié : Faker

https://github.com/fzaninotto/Faker

composer require fzaninotto/faker --dev

Modifier ArticlesFixtures.php comme ceci :

```php
<?php

namespace App\DataFixtures;

use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Common\Persistence\ObjectManager;
use App\Entity\Article;
use App\Entity\Category;
use App\Entity\Comment;
use Faker;

class ArticlesFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        $faker = Faker\Factory::create('fr_FR');
        // créer 3 catégories avec faker
        for ($i = 1; $i <= 3 ; $i++)
        {
            $category = new Category();
            $category->setTitle($faker->sentence())
                    ->setDescription($faker->paragraph());
            $manager->persist($category);
            for($j = 1; $j <= mt_rand(4,6) ; $j++)
            {
                $article = new Article();
                $content = '<p>';
                $content .= join($faker->paragraphs(5), '</p><p>'); // $faker-
>paragraphs rend une liste
                $content .= '</p>';
                $article->setTitle($faker->sentence())
                    ->setContent($content)
                    ->setImage($faker->imageUrl())
                    ->setCreatedAt($faker->dateTimeBetween('-6 months'))
                    ->setCategory($category);
                $manager->persist($article);

                // affecter des commentaires à l'article
```

```
                for ($k = 1 ; $k <= mt_rand(4,8) ; $k++)
                {
                    $comment = new Comment();
                    $content = '<p>' . join($faker->paragraphs(5),
'</p><p>') . '</p>';
                    // chercher une date réaliste pour le commentaire
                    $now = new \DateTime();
                    $interval = $now->diff($article->getCreatedAt());
                    $days = $interval->days;
                    $minimum = '-' . $days .' days';

                    $comment->setAuthor($faker->name)
                            ->setContent($content)
                            ->setCreatedAt($faker->dateTimeBetween($minimum))
                            ->setArticle($article);
                    $manager->persist($comment);
                }
            }
        }
        $manager->flush();
    }
}
```

Puis

`php bin/console doctrine:fixtures:load`

Regarder le résultat en BD

Cependant faker n'existe plus pour Symfony 7.

Adapter ArticlesFixtures.php précédent our alimenter la BD.

Modifier le body de show.html.twig avec ceci

```
{% block body %}
    <article>
        <h2>{{article.title}}</h2>
        <div class="metadata">rédigé le {{article.createdAt | date('d/m/Y')}}
à {{article.createdAt | date('H:i')}}
           dans la catégorie {{article.category.title}}
        </div>

        <div class="content">
            <img src="{{article.image}}" alt="">
            {{article.content | raw}}
        </div>
    </article>
    <section id="commentaires">
        {% for comment in article.comments%}
            <div class="comment">
                <div class="row">
                    <div class="col-3">
                        {{comment.author}} (<small> {{ comment.createdAt |
date('d/m/Y à H:i') }}</small>)
                    </div>
                    <div class="col">
                        {{comment.content |raw}}
                    </div>
                </div>
            </div>
        {% endfor %}
    </section>
{% endblock %}
```

Regarder le résultat

Section 17 Adapter le formulaire

17.1 But

- Prise en compte des nouvelles entités dans le formulaire

17.2 Enoncé

On souhaite faire connaître au formulaire la liste des catégories.

https://symfony.com/doc/current/reference/forms/types/entity.html sert de support

Modifier le contenu du formulaire ArticleType.php

Ajouter

```
use App\Entity\Category;
use Symfony\Bridge\Doctrine\Form\Type\EntityType;
```

buildform devient :

```
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('title')
        ->add('category', EntityType::class, [
            'class' => Category::class,
            'choice_label' => 'title'
        ])
        ->add('content')
        ->add('image')
    ;
}
```

Modifier create.html.twig. Ajouter le ligne

```
{{form_row(formArticle.category)}}
```

Section 18 Authentification

18.1 But

- Permettre le login à l'application
- Utilisation du module Security
- S'appuie sur le site https://www.youtube.com/watch?v=_GjHWa9hQic

18.2 Enoncé

Créer une table d'utilisateurs :

`php bin/console make:entity`

User

email string 255 not null

username string 255 not null

password string 255 not null

Puis

`php bin/console make:migration`

`php bin/console doctrine:migrations:migrate`

Créer les données du formulaire

`php bin/console make:form RegistrationType`

User

Dans RegistrationType.php ajouter sous add('password') :

```
    ->add('confirm_password')
```

Ajouter dans User.php

```
public $confirm_password;
```

Nous allons ajouter un Controller

php bin/console make:controller SecurityController

Voici le code

```
<?php

namespace App\Controller;

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

use App\Entity\User;
use App\Form\RegistrationType;

class SecurityController extends AbstractController
{
    /**
     * @Route("/inscription", name="security_registration")
     */
    public function registration()
    {
        $user = new user();
        $form = $this->createForm(RegistrationType::class, $user);

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

Il faut créer la vue templates/security/registration.html.twig

```
{% extends 'base.html.twig' %}
{% form_theme form 'bootstrap_4_layout.html.twig' %}

{% block body %}
    <h1>Inscription sur ce site</h1>
    {{form_start(form)}}
    {{form_row(form.username, {'attr':{'placeholder' : "Votre nom ..."},
        'label':'Nom'})}}
    {{form_row(form.email, {'attr':{'placeholder' : "Votre mail"},
        'label':'e-Mail'})}}
        {{form_row(form.password, {'attr':{'placeholder' : "Mot de passe"},
        'label':'Mot de passe'})}}
```

```
        {{form_row(form.confirm_password, {'attr':{'placeholder' :
"Confirmation ot de passe"},
        'label':'Mot de passe'})}}

    <button type="submit" class="btn btn-success">Inscription</button>

    {{form_end(form)}}
{% endblock %}
```

Essayer la page par le lien /inscription

Les champs pswd sont visibles. Pour remédier à cela éditer RegistrationType.php

Ajouter

```
use Symfony\Component\Form\Extension\Core\Type\PasswordType;
            ->add('password', PasswordType::class)
            ->add('confirm_password', PasswordType::class)
```

Essayer la page par le lien /inscription

Il faut maintenant enregistrer les données en BD.

Modifier SecurityController.php

```
class SecurityController extends AbstractController
{
    /**
     * @Route("/inscription", name="security_registration")
     */
    public function registration(Request $request, ManagerRegistry $doctrine
)
    {
        $manager = $doctrine->getManager();
        $user = new user();
        $form = $this->createForm(RegistrationType::class, $user);
        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid())
        {
            $manager->persist($user);
            $manager->flush();
        }

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

}

Ne pas oublier les use manquants (Ctrl Alt I)

Essayer la vue et enregistrer un user

Vérifier le mot de passe :

Dans User.php

Ajouter

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

Modifier password et confirm_password

Consulter https://symfony.com/doc/current/reference/constraints.html

En Symfony 7

```
#[ORM\Column(length: 255)]
  #[Assert\Length(min:8, minMessage: "Au minimum 8 caractères")]
  #[Assert\EqualTo(propertyPath:"confirm_password",message:"Mal confirmé")]
  private ?string $password = null;

  #[Assert\EqualTo(propertyPath:"password",message:"Mal confirmé")]
  public $confirm_password;
```

En Symfony ≤ 5.1

```
    /**
     * @ORM\Column(type="string", length=255)
     * @Assert\Length(min="8", minMessage="Au minimum 8 caractères")
     * @Assert\EqualTo(propertyPath="confirm_password", message="Mal
confirmé")
     */
    private $password;

    /**
     * @Assert\EqualTo(propertyPath="password", message="Mal confirmé")
     */
    public $confirm_password;
```

Essayer le formulaire et créer un user (essayer en vous trompant d'abord)

Section 19 Sécuriser les données, Symfony ≤ 5.1

19.1 But

- Cripter le mot de passe
- Utilisation du module Security

19.2 Enoncé

Dans SecurityController.php

```
class SecurityController extends AbstractController
{
    /**
     * @Route("/inscription", name="security_registration")
     */
    public function registration(Request $request, ObjectManager $manager,
    UserPasswordEncoderInterface $encoder)
    {
        $user = new user();
        $form = $this->createForm(RegistrationType::class, $user);
        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid())
        {
            $hash = $encoder->encodePassword($user,$user->getPassword());
            $user->setPassword($hash);

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

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

https://symfony.com/doc/4.0/security/password_encoding.html

Modifier RegistrationType.php

```
use Symfony\Component\Form\Extension\Core\Type\PasswordType;
```

et

```
            ->add('password', PasswordType::class)
```

```
            ->add('confirm_password', PasswordType::class)
```

Modifier également User.php pour que User implémente une interface

https://symfony.com/doc/3.3/security/entity_provider.html

```
use Symfony\Component\Security\Core\User\UserInterface;

class User implements Userinterface

    public function eraseCredentials()
    {}
    public function getSalt()
    {}
    public function getRoles()
    {
        return ['ROLE_USER'];
    }
```

Modifier config\packages\security.yaml

```
security:
    encoders:
        App\Entity\User:
            algorithm: bcrypt
```

Essayer et créer un nouvel utilisateur

Ensuite depuis phpMyAdmin, supprimer les users

Section 20 Sécuriser les données en Symfony 7.x

20.1 But

- Cripter le mot de passe
- Utilisation du module Security
- utilisr https://symfony.com/doc/current/security.html#the-user

20.2 Enoncé

Dans SecurityController.php

```
class SecurityController extends AbstractController
{
  #[Route("/inscription", name:"security_registration")]

  public function registration(Request $request, ManagerRegistry $doctrine,
UserPasswordHasherInterface $passwordHasher)
  {
    $manager = $doctrine->getManager();
    $user = new user();
    $form = $this->createForm(RegistrationType::class, $user);
    $form->handleRequest($request);
    //dd($request);

    if ($form->isSubmitted() && $form->isValid())
    {
      $hash = $passwordHasher->hashPassword($user,$user->getPassword());
      $user->setPassword($hash);
      $manager->persist($user);
      $manager->flush();
    }

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

https://symfony.com/doc/4.0/security/password_encoding.html

Modifier RegistrationType.php

```
use Symfony\Component\Form\Extension\Core\Type\PasswordType;
```

et

```
            ->add('password', PasswordType::class)
            ->add('confirm_password', PasswordType::class)
```

Modifier également User.php pour que User implémente une interface

https://symfony.com/doc/3.3/security/entity_provider.html

```
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface;

#[ORM\Entity(repositoryClass: UserRepository::class)]
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
  public function getRoles(): array
  {
    return ['ROLE_USER'];
  }
  public function eraseCredentials(): void
  {
    // If you store any temporary, sensitive data on the user, clear it here
  }
  public function getUserIdentifier(): string
  {
    return (string) $this->email;
  }
```

Essayer et créer un nouvel utilisateur

Ensuite depuis phpMyAdmin, vérifier que le mot de passe est crypté, ensuite supprimer les users

Section 21 Utilisateur unique

21.1 But

- Contrôler l'@ mail pour qu'il soit unique
- Utilisation du module Security

21.2 Enoncé

Faire en sorte que l'email soit unique

Dans User.php

Ajouter

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

Les annotations de User sont modifiées

En Symfony 7.x

```
#[ORM\Entity(repositoryClass: UserRepository::class)]
#[UniqueEntity('email')]
class User implements UserInterface, PasswordAuthenticatedUserInterface
```

En Symfony ≤ 5.1

```
/**
 * @ORM\Entity(repositoryClass="App\Repository\UserRepository")
 * @UniqueEntity(
 * fields={"email"},
 * message= "Cet email est déjà utilisé"
 * )
 */
class User implements Userinterface
```

L'annotation de $mail est modifiée

En Symfony 7.x

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

En Symfony ≤ 5.1

```
    /**
     * @ORM\Column(type="string", length=255)
     * @Assert\Email()
     */
    private $email;
```

Essayer ces modifications avec des emails identiques.

Section 22 Fenêtre de login

22.1 But

- Ajouter une fenêtre de login

22.2 Enoncé

Dans SecurityController.php

Ajouter

```
/**
 * @Route("/connexion", name="security_login")
 */
public function login(){
    return $this->render('security/login.html.twig');
}
```

Modifier registration() pour que la prise en compte d'un nouvel utilisateur soit suivi de la fenêtre de login

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

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

Créer templates/security/ login.html.twig

Avec ce contenu

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

{% block body %}
    <h1>Connexion</h1>

    <form action="{{ path('security_login')}}" method="post">
        <div class="form-group">
            <input type="text" name="_username" class="form-control"
placeholder="Adresse Mail" required>
        </div>
        <div class="form-group">
                <input type="password" name="_password" class="form-control"
placeholder="Mot de passe" required>
        </div>
        <div class="form-group">
```

```
            <button type="submit" class="btn btn-success">Connexion</button>
        </div>
    </form>
{% endblock %}
```

Modifier security.yaml

En Symfony 7 , LAISSER LA FIN DU FICHIER à partir de access_control

```
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:
    users_in_memory: { memory: null }
    app_user_provider:
      entity:
        class: App\Entity\User
        property: email

  firewalls:
    dev:
      pattern: ^/(_(profiler|wdt)|css|images|js)/
      security: false
    main:
      lazy: true
      provider: app_user_provider #users_in_memory
      form_login:
        login_path: security_login
        check_path: security_login
```

En Symfony ≤ 5.1

```
security:
    encoders:
        App\Entity\User:
            algorithm: bcrypt
    # https://symfony.com/doc/current/security.html#where-do-users-come-from-
user-providers
    providers:
        in_memory: { memory: null }
```

```
        in_database:
            entity:
                class: App\Entity\User
                property: email
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            anonymous: true

            provider: in_database

            form_login:
                login_path: security_login
                check_path: security_login
```

Essayer la page par /connexion

Section 23logout

23.1 But

- Traitement de la déconnexion

23.2 Enoncé

Modifier security.yaml. Ajouter sous form_login

```
logout:
    path: security_logout
    target: /blog
```

Ajouter dans SecurityController.php

```
#[Route("/deconnexion", name:"security_logout")]
public function logout(){
}
```

Améliorer la navbar :

Dans base.html.twig compléter la navbar par

```
                {% if not app.user %}
                    <li class="nav-item">
                        <a href="{{path('security_login')}}" class="nav-
link">Connexion</a>
                    </li>
                {% else %}
                    <li class="nav-item">
                        <a href="{{path('security_logout')}}" class="nav-
link">Déconnexion</a>
                    </li>
                {% endif %}
```

Essayer

Section 24Formulaire de commentaire si connecté

24.1 But

- Créer un formulaire
- Tester la connexion

24.2 Enoncé

Créer le formulaire des commentaires :

`php bin/console make:form`

`CommentType`

`Comment`

Dans Form/CommentType.php supprimer les lignes

```
->add('createdAt')
->add('article')
```

Dans BlogController.php modifier show()

```
    public function show(Request $request, ManagerRegistry $doctrine
,ArticleRepository $repo, $id)
    {
        $article = $repo->find($id);
        $comment = new Comment();
        $form = $this->createForm(CommentType::class,$comment);
        $form->handleRequest($request);
        if ($form->isSubmitted() && $form->isValid())
        {
            $comment->setCreatedAt(new \Datetime())
                    ->setArticle($article);

            $manager = $doctrine->getManager();
            $manager->persist($comment);
            $manager->flush();

            return $this->redirectToRoute('blog_show', ['id' => $article-
>getId()]);
        }
        return $this->render('blog/show.html.twig',[
            'article' => $article,
            'commentForm' => $form->createView()
        ]);
    }
```

Modifier show.html.twig

Ajouter après la boucle for :

```
{% if app.user %}
    {{ form_start(commentForm)}}
    {{ form_row(commentForm.author, {'attr' : {'placeholder' : "Votre
nom"}})}}
    {{ form_row(commentForm.content, {'attr' : {'placeholder' : "Votre
commentaire"}})}}
    <button type="submit" class="btn btn-success">Enregistrer</button>
    {{ form_end(commentForm)}}
{% else %}
    <h2>Il faut être connecté pour rédiger un commentaire</h2>
    <a href="{{path('security_login')}}" class="btn btn-primary">Se
connecter</a>
{% endif %}
```