

# PHP - Complément Symfony

Support de cours IUT de Calais

Kevin GUERRIER

## Préambule

Lien vers le support de cours PHP Général

<https://bit.ly/2024-iut-php-cm-kguerrier>

Lien vers ce support de complément Symfony

<https://bit.ly/2024-iut-php-symfony-kguerrier>

*Ce support est fortement inspiré :*

- *de la documentation officielle disponible sur*  
<https://symfony.com/doc/current>
- *de la trame de pratique de Romain MUGUET,*  
*intervenant également à l'IUT de Calais sur Symfony*

## Au commencement...

Symfony est l'un des frameworks PHP les plus anciens et les plus robustes. Créé en 2005 par Fabien Potencier, est reconnu pour sa flexibilité, sa modularité et sa capacité à être utilisé pour des projets de toutes tailles.

Symfony est particulièrement adapté aux projets complexes et aux grandes applications d'entreprise, mais peut également être un accélérateur de développement même lors de vos développements plus modestes dès lors où vous aurez suffisamment d'expérience pour le maîtriser.

# Principes et Caractéristiques

Si on devait résumer en quelques mots les principes et caractéristiques de Symfony, alors voici ce que cela pourrait donner :

- Flexibilité et Modularité :
  - Symfony est basé sur un ensemble de composants PHP réutilisables. Chaque fonctionnalité est disponible sous forme de bundle, ce qui rend le framework très modulaire.
- Composants Symfony :
  - Ces composants étant indépendants, on peut les retrouver et les utiliser dans d'autres projets (par exemple, Laravel utilise certains composants de Symfony).
- Système de Routage :
  - Symfony offre un système de routage avancé qui associe les URL aux méthodes des contrôleurs, ce qui permet de gagner un temps de développement considérable.
- Doctrine ORM :
  - Symfony utilise Doctrine pour gérer les bases de données relationnelles. C'est un ORM (Object-Relational Mapping) très puissant, flexible qui permet également de gagner du temps de développement.
- Gestion des Formulaires :
  - Symfony propose un système de formulaires sophistiqué pour générer, valider et traiter des formulaires HTML.
- Boîte à outils en ligne de commande :
  - Symfony Console est un puissant outil CLI qui permet de générer du code, des configurations et de gérer la base de données directement depuis un terminal.

Avantages de Symfony	Inconvénients de Symfony
Grande flexibilité pour des projets complexes et personnalisés.	Courbe d'apprentissage plus abrupte comparée à d'autres frameworks.
Utilisé par des grandes entreprises (BlaBlaCar, Spotify, etc.), ce qui en fait un atout considérable dans votre CV.	Nécessite plus de configuration pour des projets simples.
Mises à jour fréquentes et robustes en termes de sécurité.	Un peu plus lourd en termes de performance pour des applications de taille réduite.

## Plus concrètement...

Il y a plusieurs manières de mettre en place les prérequis pour utiliser le framework Symfony, et vous retrouverez toutes les étapes d'installation sur la page officielle de Symfony <https://symfony.com/download>

Dans le cadre de ce TP, nous allons utiliser cette occasion pour voir l'utilisation de ce framework au travers de conteneur Docker, et plus précisément en suivant les instructions sur la documentation officielle : <https://symfony.com/doc/current/setup/docker.html>

Pour cela, il est nécessaire d'avoir Docker Compose (v2.10+) disponible sur système. (Si ce n'est pas le cas, vous pouvez suivre les instructions d'installation présentes sur le site de docker en fonction de votre système <https://docs.docker.com/compose/install/> )

## Récupération de l'installation minimale de Symfony à partir du repository Symfony-Docker

*Créez un répertoire TD4-Symfony et placez-vous à l'intérieur avec un terminal*

```
$ git clone https://github.com/dunglas/symfony-docker
```

Une fois les sources téléchargées, vous devriez y trouver une arborescence comprenant notamment le fichier de configuration docker (Dockerfile). Dans la partie concernant les paquets à installer via apt, ajouter les paquets make, npm, yarn et bash :

```
18  # persistent / runtime deps
19  # hadolint ignore=DL3008
20  RUN apt-get update && apt-get install -y --no-install-recommends \
21      acl \
22      file \
23      gettext \
24      git \
25      make \
26      npm \
27      yarn \
28      bash \
29      && rm -rf /var/lib/apt/lists/*
30
```

Ensuite, assurez-vous que les services Docker sont bien démarrés (cf. Docker Desktop par exemple), et exécutez les instructions suivantes :

*Construire les images*

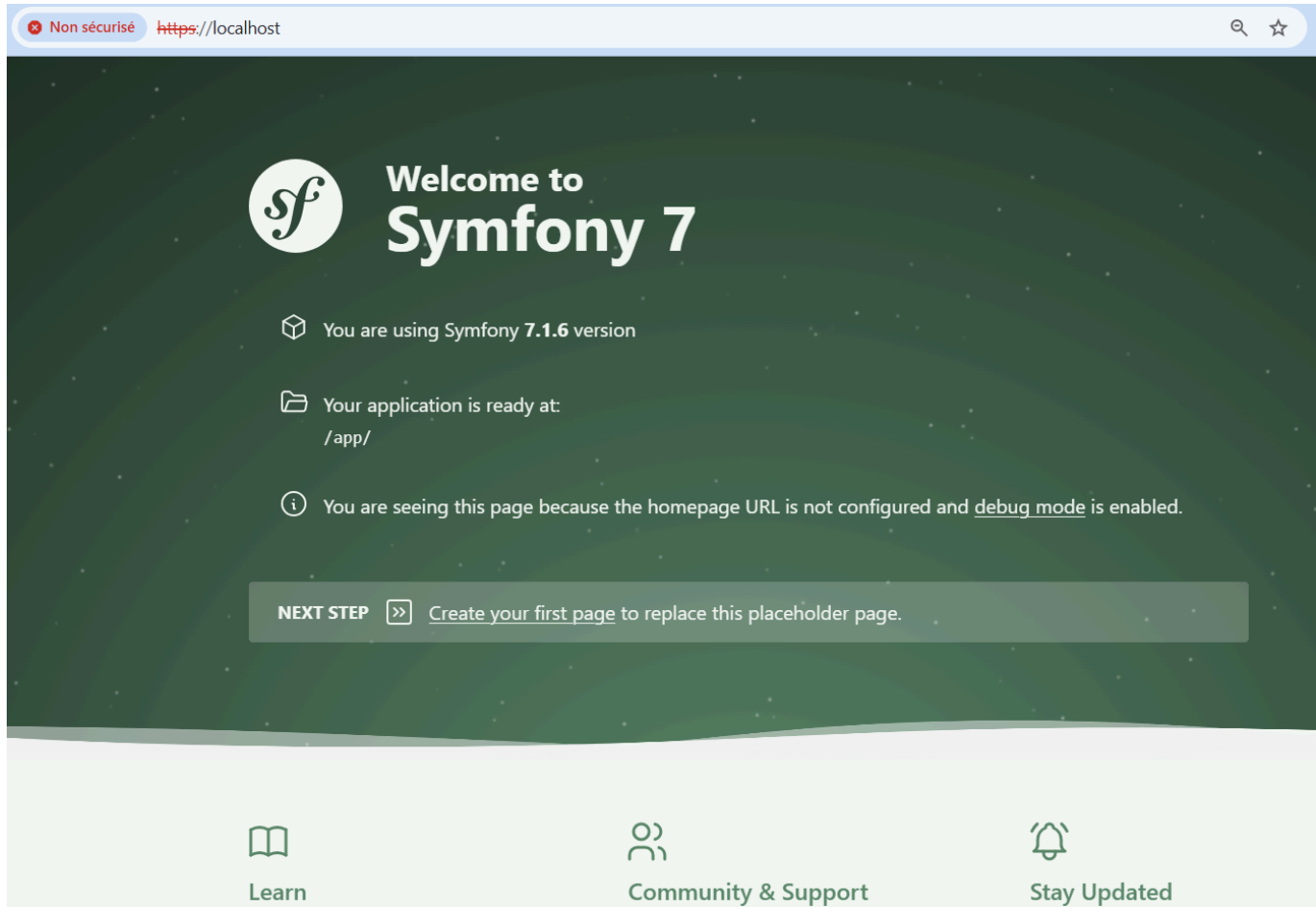
```
$ docker compose build --no-cache
```

### Montez et démarrez le projet

```
$ docker compose up --pull always -d --wait
```

Dès lors, l'application par défaut devrait être disponible depuis votre navigateur dans <https://localhost>

Notez que le certificat TLS est un certificat auto-signé donc il se peut que vous deviez l'acceptez explicitement avant de pouvoir accéder à votre application.



Enfin, pour arrêter le conteneur, utilisez la commande suivante :

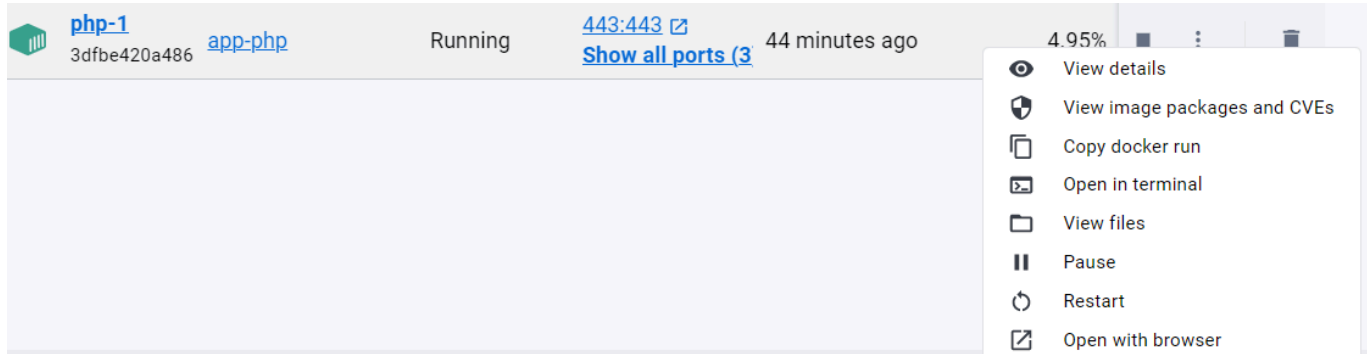
```
$ docker compose down --remove-orphans
```

### Note pour les lenteurs sur certains environnements

Dans le fichier `compose.override.yaml`, il est possible de décommenter la ligne `- /app/vendor` dans `services:php:volumes` afin d'améliorer les performances globales et ainsi ne plus avoir des temps de latences importants lors de l'exécution de Symfony.

# Configuration de Symfony et NPM

Pour la suite de cette configuration, vous devrez exécuter les commandes depuis le terminal du conteneur en service



il vous est également possible d'entrer directement dans un terminal plus complet (sous bash) en utilisant la commande :

```
$ docker exec -it symfony-docker-php-1 bash
```

Cela vous permettra d'avoir un peu plus de flexibilité dans votre terminal pour exécuter les commandes suivantes (par exemple la complétion)

```
# composer require symfony/maker-bundle --dev
# composer require symfony/twig-bundle
# composer require symfony/webpack-encore-bundle
```

Maintenant que notre environnement est prêt, il est temps de créer une première page. Cette étape se déroule en 2 temps :

- Il faut tout d'abord créer un contrôleur. C'est ce composant qui va construire la page. Il va prendre l'ensemble des informations d'une requête et les utiliser pour créer un objet de réponse Symfony. Cet objet peut contenir du contenu HTML, une chaîne JSON ou même un fichier binaire comme une image ou un PDF.
  - ⇒ Consultez les informations sur la manière dont Symfony gère les requêtes et les réponses HTTP : [Symfony and HTTP Fundamentals](#)
- Ensuite il faut créer une route, c'est-à-dire l'URL qui permet d'accéder à la page et pointer vers le contrôleur.

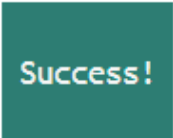
Et c'est à ce moment-là que l'outil de développement maker-bundle que nous avons installé ci-dessus entre en jeu ! Cet outil permet de générer automatiquement des composants en ligne de commande, par exemple :

*Pour créer un controller*

```
# php bin/console make:controller

Choose a name for your controller class (e.g. VictoriousPuppyController):
> LuckyController

created: src/Controller/LuckyController.php
created: templates/lucky/index.html.twig
```



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

Voici le code du Controller créé :

```
<?php
namespace App\Controller;

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

class LuckyController extends AbstractController
{
    #[Route('/lucky', name: 'app_lucky')]
    public function index(): Response
    {
        return $this->render('lucky/index.html.twig', [
            'controller_name' => 'LuckyController',
        ]);
    }
}
```

Vous pouvez y retrouver la route qui est associée à la fonction index dans la partie :


```
#[Route('/lucky', name: 'app_lucky')]
```

Enfin, la mise en page est précisée dans le fichier lucky/index.html.twig dont le code est le suivant :

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

{% block title %}Hello LuckyController!{% endblock %}

{% block body %}
<style>
    .example-wrapper { margin: 1em auto; max-width: 800px; width: 95%; font: 18px/1.5
sans-serif; }
    .example-wrapper code { background: #F5F5F5; padding: 2px 6px; }
</style>

<div class="example-wrapper">
    <h1>Hello {{ controller_name }}! 
```

Etudiez ce code (et celui présent dans base.html.twig) ainsi que la documentation sur les Templates [Creating and Using Templates \(Symfony Docs\)](#) pour vous familiariser avec ce langage.

Des fonctionnalités complémentaires utiles pour manipuler vos affichages à l'intérieur de vos fichiers twig sont disponibles en installant les packages suivants :

```
# composer require twig/string-extra twig/extra-bundle
```

Enfin, le dernier outil que nous allons mettre en place est npm, notamment pour le bundle encore que nous avons installé. Si ce n'est pas encore fait, vous pouvez retrouver les informations relatives à ce composant sur [Encore: Setting up your Project \(Symfony Docs\)](#)

#### Installer les sources JS et CSS et les construire

```
# npm install && npm run build
```

Et tant que nous sommes sur npm, nous allons en profiter pour installer le framework bootstrap

```
# npm install bootstrap @popperjs/core --save
```

Ainsi, nous pouvons importer les fichiers CSS et JS dans le fichier assets/app.js

```
assets > JS app.js
1  /*
2  * Welcome to your app's main JavaScript file!
3  *
4  * We recommend including the built version of this JavaScript file
5  * (and its CSS file) in your base layout (base.html.twig).
6  */
7
8  // Importation des fichiers CSS
9  import 'bootstrap/dist/css/bootstrap.min.css';
10 // Importation des fichiers JavaScript
11 import 'bootstrap/dist/js/bootstrap.min.js'; 60.9k (gzipped: 16.7k)
12
13 // any CSS you import will output into a single css file (app.css in this case)
14 import './styles/app.css';
15
```

et comme cela est précisé, on ajoute dans le base.html.twig l'appel à notre assets via encore

```
templates > ./ base.html.twig
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset="UTF-8">
5          <title>
6              {% block title %}Welcome!{% endblock %}
7          </title>
8          <link rel="icon" href="data:image/svg+xml,<svg xmlns=%22http://www.w3.org/2000/svg%22 viewBox=%220
9              {% block stylesheets %}
10                 {# 'app' must match the first argument to addEntry() in webpack.config.js #}
11                 {{ encore_entry_link_tags('app') }}
12
13                 <!-- Renders a link tag (if your module requires any CSS)
14                 | | | | | <link rel="stylesheet" href="/build/app.css"> -->
15                 {% endblock %}
16
17                 {% block javascripts %}                 {{ encore_entry_script_tags('app') }}                 {% endblock %}
18             </head>
19             <body>
20                 {% block body %}{% endblock %}
21             </body>
22     </html>
```

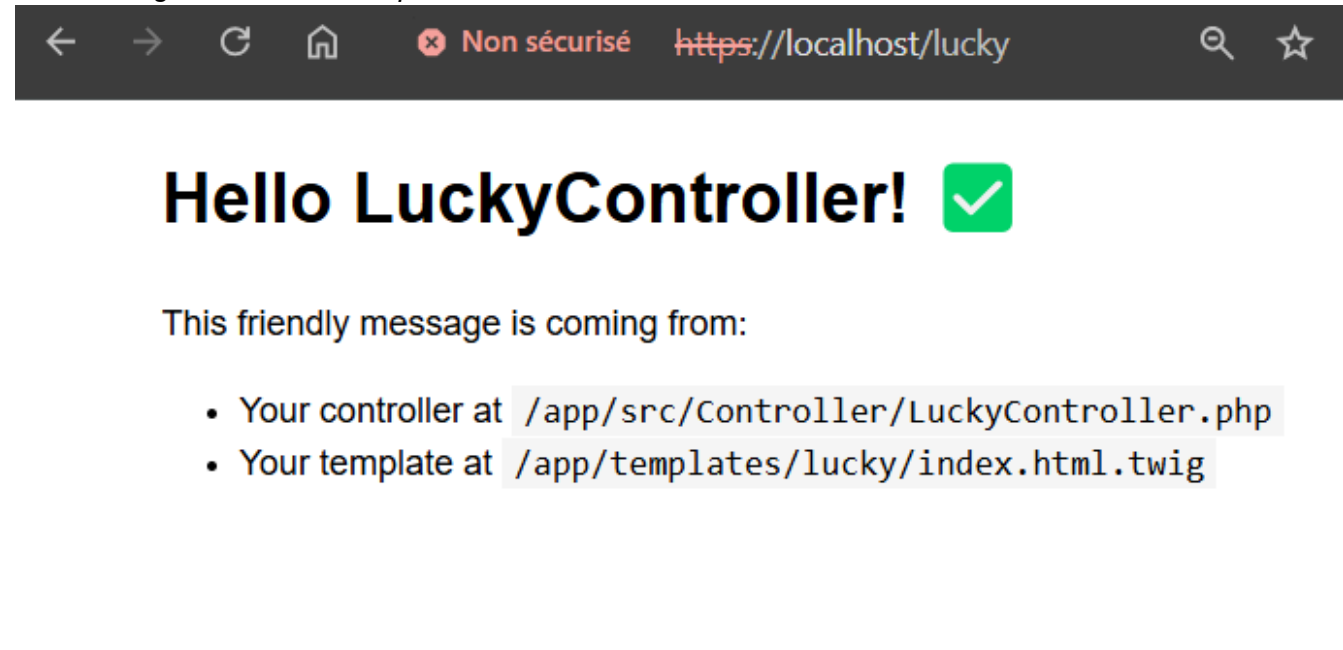


Relancer la construction à jour des JS et CSS

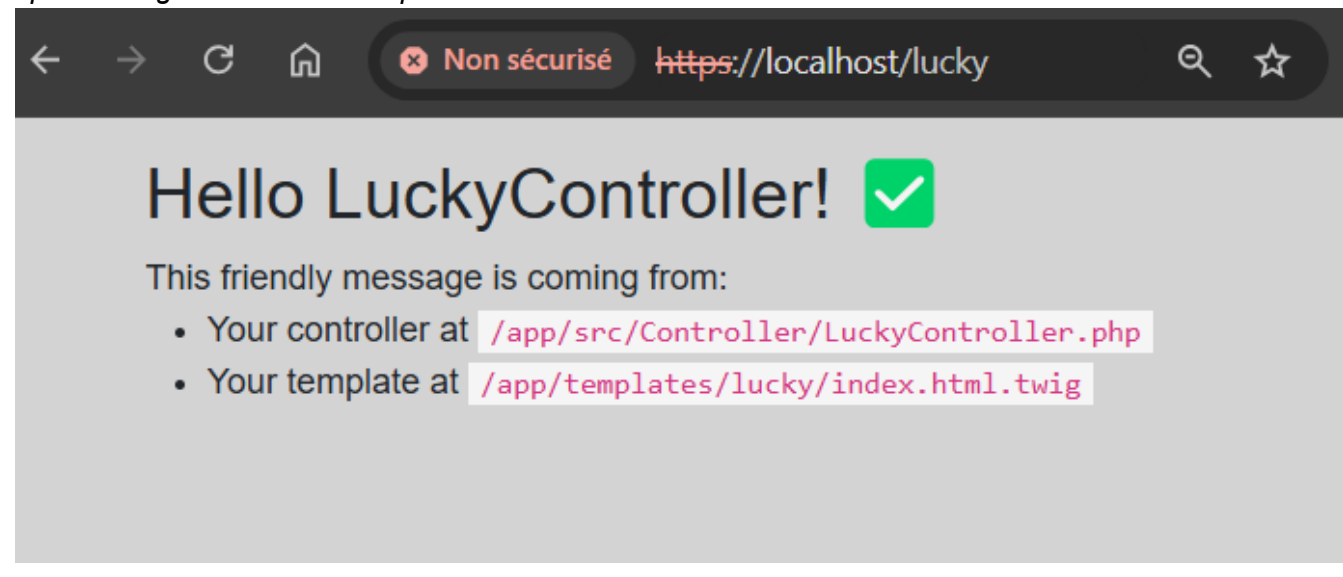
```
# npm run build
```

Et voilà, votre résultat intègre à présent le framework bootstrap :

Avant l'intégration de bootstrap dans l'asset



Après l'intégration de bootstrap dans l'asset



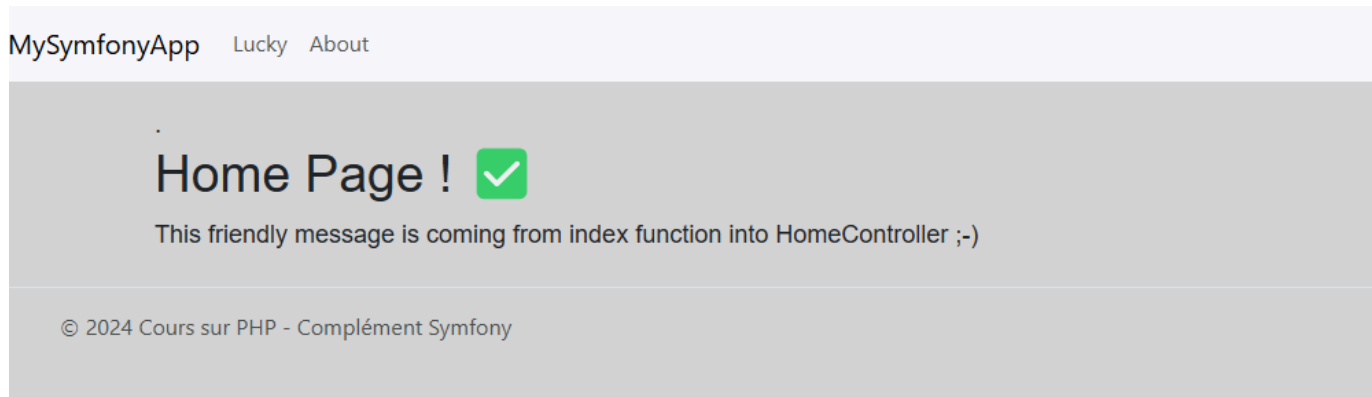
Si vous le souhaitez, vous pouvez utiliser d'autres outils complémentaires pour vous faciliter le codage avec Encore

<https://symfony.com/doc/current/frontend/encore/index.html>

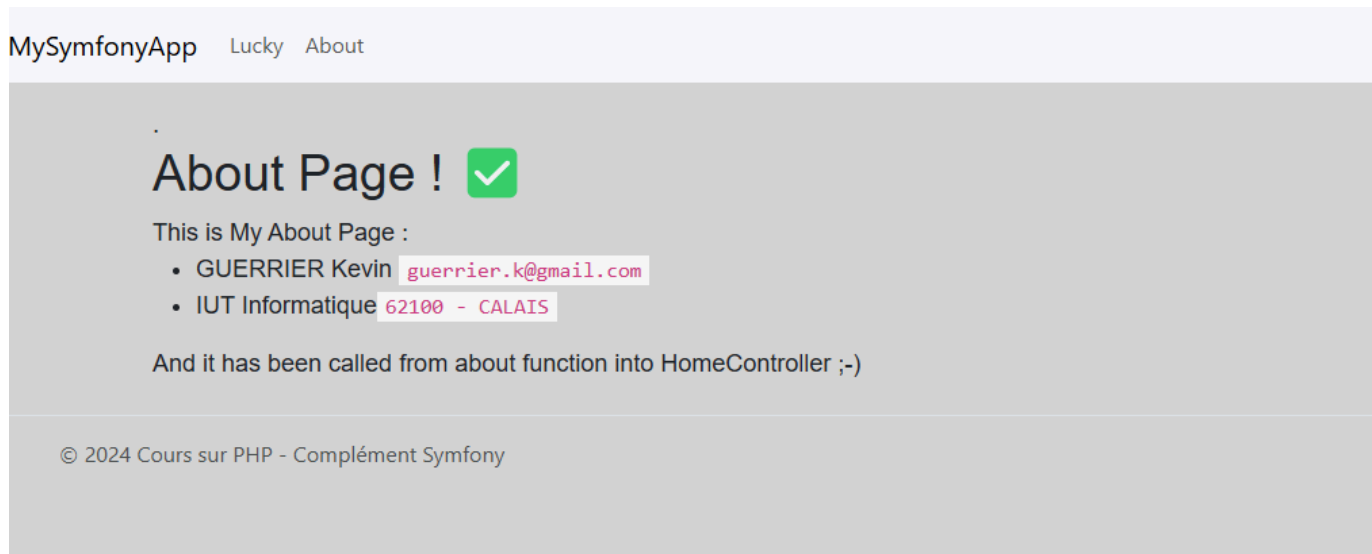
# Exercice 1

Vous réaliserez un controller nommé HomeController qui permettra de gérer :

- La page d'accueil (donc accessible depuis la route / ), accessible par défaut lors de l'ouverture de l'application



- Cette page sera également accessible par la route /home (et c'est d'ailleurs vers ce lien que pointeront la première entrée du menu haut MySymfonyApp)
- Une page About (accessible depuis la route /about ), accessible depuis la 3ème entrée du menu haut About



Et pour réaliser cela, les fonctions de ce controller appelleront toutes deux un fichier twig différent mais qui devront étendre un home.html.twig commun.

Ce home.html.twig devra être découpé en plusieurs parties permettant de faire évoluer facilement chaque zone indépendamment (le menu du haut, le pied de page, ...)

Et voilà, c'est tout pour cette première fois avec Symfony ^\_^ On se retrouvera la prochaine fois pour mettre en place un moteur de blog basique en utilisant un l'ORM Doctrine.

# Les Bases de données et l'ORM Doctrine

Symfony fournit tous les outils dont vous avez besoin pour utiliser des bases de données dans vos applications grâce à Doctrine, le meilleur ensemble de bibliothèques PHP pour travailler avec des bases de données. Ces outils prennent en charge les bases de données relationnelles comme MySQL et PostgreSQL ainsi que les bases de données NoSQL comme MongoDB.

## Installation de Doctrine

Normalement, vous avez déjà intégré précédemment le maker-bundle dans votre projet et il vous est donc uniquement utile d'installer le orm-pack (toujours depuis le shell de votre conteneur docker)

```
# composer require symfony/orm-pack
```

## Configuration de la base de données

Les informations de connexion à la base de données sont stockées dans une variable d'environnement appelée DATABASE\_URL. Pour le développement, vous pouvez donc la trouver et la personnaliser à l'intérieur .env:

```
# .env (or override DATABASE_URL in .env.local to avoid committing your changes)

###> doctrine/doctrine-bundle ###
#
# DATABASE_URL="sqlite:///kernel.project_dir%/var/data.db"
#
DATABASE_URL="mysql://app:!ChangeMe!@127.0.0.1:3306/app?serverVersion=8.0.32&charset=utf8mb4"
#
DATABASE_URL="mysql://app:!ChangeMe!@127.0.0.1:3306/app?serverVersion=10.11.2-MariaDB&
charset=utf8mb4"
DATABASE_URL="postgresql://app:!ChangeMe!@127.0.0.1:5432/app?serverVersion=16&charset=
utf8"
###< doctrine/doctrine-bundle ###
```

Maintenant que vos paramètres de connexion sont configurés, après down / rebuild / up de votre conteneur, Doctrine devrait donc être disponible. Vous pouvez le vérifier notamment en exécutant la commande suivante pour voir la liste complète des commandes Doctrine utilisables :

```
# php bin/console list doctrine
```

Nous allons à présent créer une entité qui sera destinée à être manipulée via Doctrine.

## Créer une classe d'entité en interactif

Supposons que vous construisez une application dans laquelle des articles doivent être affichés. Sans même penser à Doctrine ou aux bases de données, vous savez déjà que vous avez besoin d'un objet Article pour représenter ces produits.

Vous pouvez utiliser la commande `make:entity` pour créer cette classe et tous les champs dont vous avez besoin. : titre (string), texte (text), publie (boolean), date (datetime\_immutable)

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

Cette commande est interactive c'est à dire qu'elle vous posera quelques questions - répondez-y pour créer chaque élément.

```
root@fdb8aa0c5e0b:/app# php bin/console make:entity

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

created: src/Entity/Article.php
created: src/Repository/ArticleRepository.php

Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-running this command.

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

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

Vous pouvez voir le résultat dans le fichier `src/Entity/Article.php` et il devrait ressembler à ceci :

```
<?php
namespace App\Entity;

use App\Repository\ArticleRepository;
use Doctrine\DBAL\Types\Types;
use Doctrine\ORM\Mapping as ORM;

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

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

    #[ORM\Column(type: Types::TEXT)]
    private ?string $texte = null;

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

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

```

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

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

public function setTitre(string $titre): static
{
    $this->titre = $titre;

    return $this;
}

public function getTexte(): ?string
{
    return $this->texte;
}

public function setTexte(string $texte): static
{
    $this->texte = $texte;

    return $this;
}

public function isPublie(): ?bool
{
    return $this->publie;
}

public function setPublie(bool $publie): static
{
    $this->publie = $publie;

    return $this;
}

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

public function setDate(\DateTimeImmutable $date): static
{
    $this->date = $date;

    return $this;
}
}

```

Vous pouvez ensuite lancer la migration de cette Entité vers une Table de votre base de données en utilisant la commande

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

```
root@fdb8aa0c5e0b:/app# php bin/console make:migration
created: migrations/Version20241203205159.php

Success!

Review the new migration then run it with php bin/console doctrine:migrations:migrate
See https://symfony.com/doc/current/bundles/DoctrineMigrationsBundle/index.html
```

Ce qui génère automatiquement un fichier sql correspondant à cette entité, et vous pouvez l'exécuter automatiquement en lançant la commande

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

```
root@fdb8aa0c5e0b:/app# php bin/console doctrine:migrations:migrate
WARNING! You are about to execute a migration in database "app" that could result in schema changes and data loss. Are you sure you wish to continue? (yes/no) [yes]:
>

[notice] Migrating up to DoctrineMigrations\Version20241203205159
[notice] finished in 16.7ms, used 12M memory, 1 migrations executed, 2 sql queries

[OK] Successfully migrated to version: DoctrineMigrations\Version20241203205159
```

Cette commande exécute tous les fichiers de migration qui n'ont pas encore été exécutés sur votre base de données.

#### Note complémentaire :

Si vous devez ajouter de nouveaux champs à votre Entité, alors vous pouvez relancer les commandes

```
$ php bin/console make:entity
$ php bin/console make:migration
$ php bin/console doctrine:migrations:migrate
```

Chaque fois que vous apportez une modification à votre schéma, exécutez ces deux commandes pour générer la migration, puis exécutez-la. Assurez-vous de valider les fichiers de migration et de les exécuter lors du déploiement.

Continuons à présent avec la création d'un contrôleur pour manipuler notre nouvelle entité.

## Créer un contrôleur pour cette Entité

Il est temps d'enregistrer un objet Article dans la base de données ! Et cette étape passe donc par la création d'un nouveau contrôleur :

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

```
root@fdb8aa0c5e0b:/app# php bin/console make:controller ArticleController
created: src/Controller/ArticleController.php
created: templates/article/index.html.twig
```

Success!

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

Vous pouvez aller consulter le code généré à l'intérieur de ce nouveau contrôleur. Celui-ci devrait ressembler à ceci :

```
<?php
namespace App\Controller;

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

class ArticleController extends AbstractController
{
    #[Route('/article', name: 'app_article')]
    public function index(): Response
    {
        return $this->render('article/index.html.twig', [
            'controller_name' => 'ArticleController',
        ]);
    }
}
```

ainsi que dans le twig concerné :

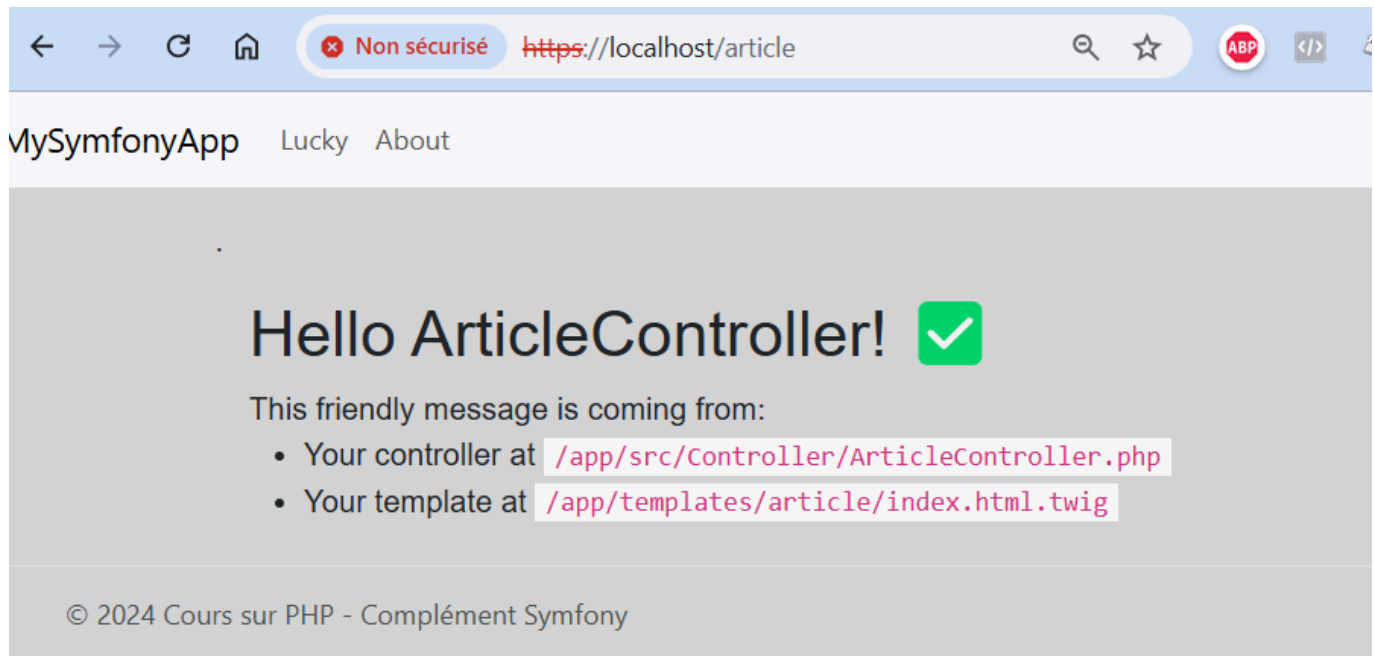
```
{% extends 'base.html.twig' %}
{% block title %}Hello ArticleController!{% endblock %}
{% block body %}
<style>
    .example-wrapper { margin: 1em auto; max-width: 800px; width: 95%; font: 18px/1.5
sans-serif; }
    .example-wrapper code { background: #F5F5F5; padding: 2px 6px; }
</style>
```

```

<div class="example-wrapper">
    <h1>Hello {{ controller_name }}! 

```

N'oubliez pas de l'ajuster pour étendre votre fichier home.html.twig ;-)



Enfin, pour générer un premier article et le stocker dans votre table, vous pouvez ajouter une nouvelle route faisant appel au code suivant :

```

#[Route('/article/generate', name: 'generate_article')]
public function generateArticle(EntityManagerInterface $entityManager): Response
{
    $article = new Article();
    $str_now = date('Y-m-d H:i:s', time());
    $article->setTitre('Titre aleatoire #' . $str_now);
    $content = file_get_contents('http://loripsum.net/api');
    $article->setTexte($content);
    $article->setPublie(true);
    $article->setDate(\DateTimeImmutable::createFromFormat('Y-m-d H:i:s',
    $str_now));
}

```



```

    // tell Doctrine you want to (eventually) save the Product (no queries yet)
    $entityManager->persist($article);

    // actually executes the queries (i.e. the INSERT query)
    $entityManager->flush();

    return new Response('Saved new article with id '.$article->getId());
}

```

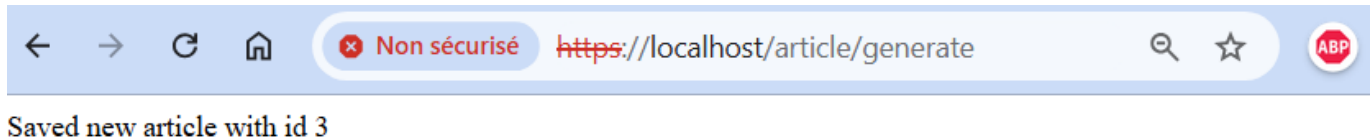
Sans oublier d'ajouter en début de fichier les dépendances utilisées :

```

use App\Entity\Article;
use Doctrine\ORM\EntityManagerInterface;

```

Ainsi, l'appel à votre nouvelle route devrait vous donner générer ce résultat



Et vous devriez pouvoir consulter votre article directement depuis la base de données en utilisant la commande :

```
$ php bin/console dbal:run-sql 'SELECT * FROM article'
```

## Exercice 2

Dans un premier temps, ajustez le menu haut pour obtenir un menu déroulant nommé Article, qui permettra d'accéder aux différentes actions relatives aux articles que vous réaliserez au fur et à mesure des exercices.

Vous réaliserez les fonctions rattachées aux routes suivantes :

- Création d'une route pour lire la liste des articles (sur /article/list)
- Création d'une route pour lire un article précis (sur /article/show/ suivi de l'id de l'article)

Vous réaliserez les templates twig correspondantes pour maintenir un affichage cohérent avec l'ensemble des pages. Pensez à vous inspirer des documentations officielles pour vous aider dans ces exercices

Dans la suite de ce TP, vous pourrez utiliser l'outil suivant pour vous aider à debugger :

```
$ composer require symfony/web-profiler-bundle
```

A présent, il nous reste à créer des formulaires pour transmettre nos données

# Mais avant cela, un outil complémentaire : ux-icons

Le package `symfony/ux-icons` propose des moyens simples et intuitifs pour afficher des icônes SVG dans votre application Symfony. Il fournit une fonction Twig pour inclure toutes les icônes locales ou distantes de vos modèles.

UX Icons vous donne un accès direct à plus de 200 000 icônes vectorielles provenant d'ensembles d'icônes populaires tels que FontAwesome, Bootstrap Icons, Tabler Icons, Google Material Design Icons, etc.

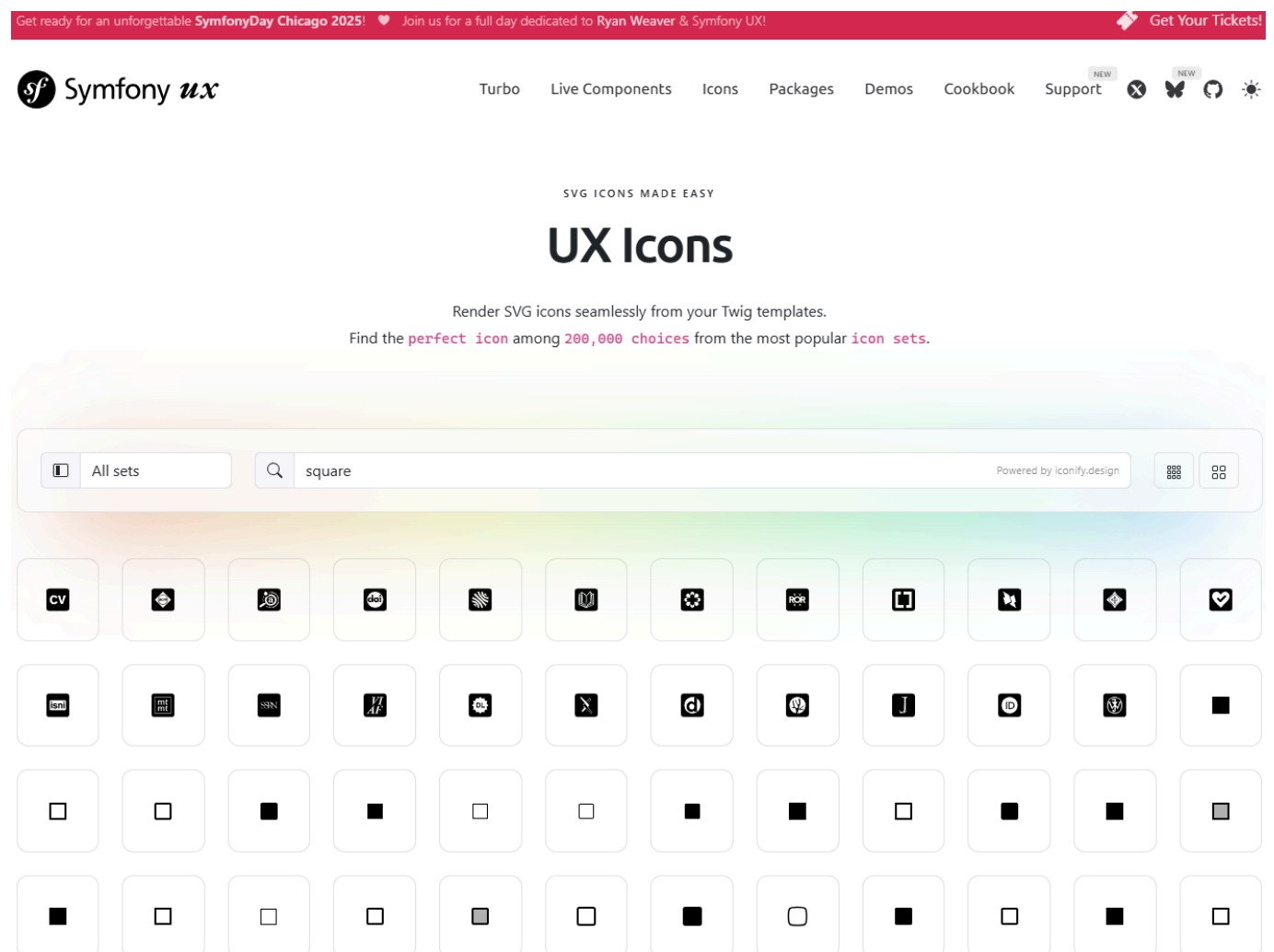
TOUTES LES ICONES NE SONT PAS SYSTÉMATIQUEMENT CHARGÉES !

C'est vous qui décidez quelles icônes vous choisissez d'inclure dans votre application.

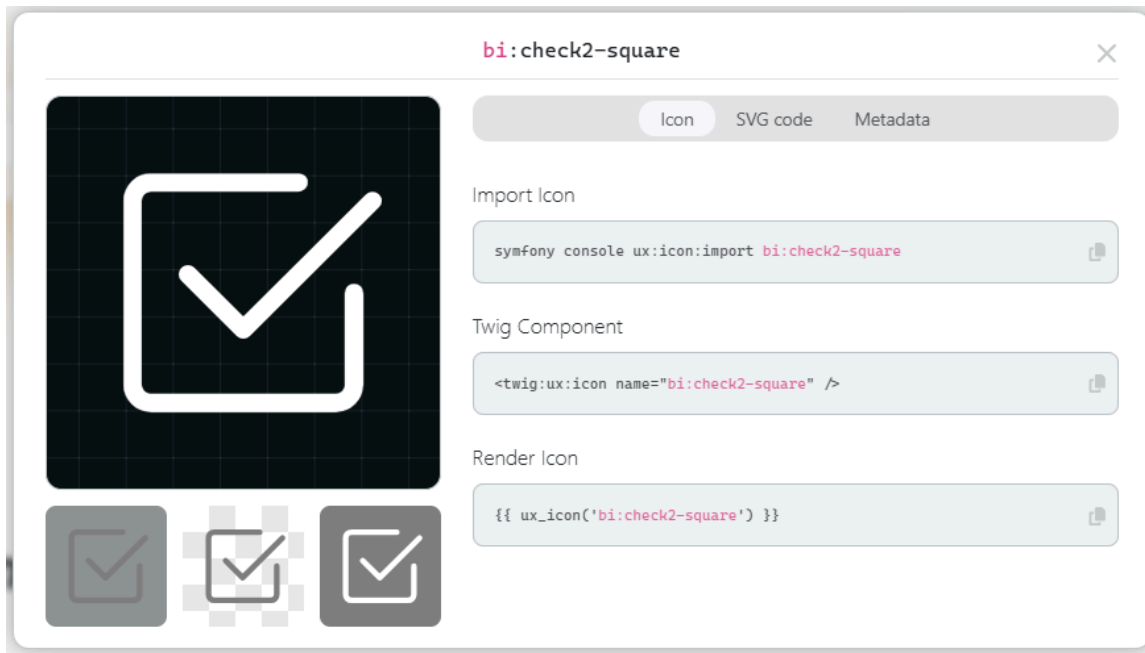
Comme pour le reste, Symfony vous propose un outil de génération des formulaires. Vous pouvez l'installer avec la commande :

```
$ composer require symfony/ux-icons symfony/http-client
```

Ensuite, lorsque vous voulez ajouter un nouvel icône, vous pouvez utiliser le site <https://ux.symfony.com/icons> pour faire votre recherche.



Et en cliquant sur l'icône souhaitée, vous voyez la commande symfony console pour importer l'icône dans votre projet.



Dans l'exemple ci-dessus, vous pouvez donc importer l'icône dans votre projet avec la commande suivante :

```
$ php bin/console ux:icon:import bi:check2-square
```

```
root@330a9ef60b2f:/app# php bin/console ux:icon:import bi:check2-square

Icon set: Bootstrap Icons (License: MIT)
✓ Imported bi:check2-square

[OK] Imported 1/1 icons.
```

et vous pourrez alors utiliser votre icône dans vos fichiers twig en utilisant la syntaxe présentée dans la partie "Render Icon", en l'occurrence : `{{ ux_icon('bi:square') }}`

```
<td>
    {% if (article.publie) %}
        {{ ux_icon('bi:check2-square') }}
    {% else %}
        {{ ux_icon('bi:square') }}
    {% endif %}
</td>
```

Vous trouverez toutes les manières de personnaliser votre affichage dans la documentation dédiée : <https://symfony.com/bundles/ux-icons/current/index.html>

# Créer un formulaire pour cette Entité

Comme pour le reste, Symfony vous propose un outil de génération des formulaires. Vous pouvez l'installer avec la commande :

```
$ composer require symfony/form
```

et en exécutant :

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

```
root@fdb8aa0c5e0b:/app# php bin/console make:form

The name of the form class (e.g. BraveKangarooType):
> Article

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

created: src/Form/ArticleType.php

Success!

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

Comme vous pouvez le voir, cette instruction génère le fichier `src/Form/ArticleType.php`. Allons donc voir ce qu'il contient :

```
<?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('texte')
            ->add('publie')
            ->add('date', null, [
                'widget' => 'single_text',
            ])
        ;
    }
}
```

```

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

```

On peut notamment y voir la fonction `buildForm` qui permet, à partir d'une interface `$builder` de créer les différents champs (la documentation devrait vous aider à comprendre la manière de l'utiliser pleinement ^\_^)

A nous d'ajouter une nouvelle route dans notre `ArticleController` qui utilisera ce formulaire et l'enverra à une vue.

```

#[Route('/article/new', name: 'article_new')]
public function new(): Response
{
    // creates a article object and initializes some data for this example
    $article = new Article();
    $article->setTitre('Which Title ?');
    $article->setTexte('And which content ?');
    $now = time();
    $str_now = date('Y-m-d H:i:s', $now);
    $article->setDate(\DateTimeImmutable::createFromFormat('Y-m-d H:i:s',
$str_now));

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

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

```

De cette manière, dans notre template `article/new.html.twig`, nous avons accès à 2 variables :

- `form` qui contient le formulaire généré (à vous de voir comment le manipuler au mieux)
- `article` qui contient des valeurs par défaut à utiliser si besoin

et ainsi, vous voyez comment passer d'autres éléments si vous en avez besoin.

## Exercice 3

Vous réaliserez les fonctions rattachées aux routes suivantes :

- Création d'une route pour ajouter un article (sur /article/new)
- Création d'une route pour mettre à jour des articles (sur /article/edit/ suivi de l'id)
- Création d'une route pour supprimer des articles (sur /article/delete/ suivi de l'id)

Vous réaliserez les templates twig correspondantes pour maintenir un affichage cohérent avec l'ensemble des pages. Pensez à vous inspirer des documentations officielles pour vous aider dans ces exercices

Rappel des documentations utiles :

- <https://symfony.com/doc/current/forms.html>
- <https://getbootstrap.com/docs/5.3/components/card/>
- <https://twig.symfony.com/doc/3.x/tags/for.html>

## Utilisation des messages à destination des utilisateurs

Dans votre contrôleur, il est possible de faire appel à la fonction `addFlash` pour transmettre un message à destination de votre utilisateur.

Par exemple, si on veut ajouter un message indiquant à notre utilisateur que l'article a bien été chargé, alors on peut ajuster la fonction `show` de la manière suivante :

```
#[Route('/article/show/{id}', name: 'article_show')]
public function show(EntityManagerInterface $entityManager, string $id): Response
{
    $article = $entityManager->getRepository(Article::class)->find((int)$id);

    if (!$article) {
        throw $this->createNotFoundException(
            'No article found for id ' . $id
        );
    }
    $this->addFlash('success', 'Article loaded !');

    // return new Response('Check out this great titre : '.$article->getTitre());

    // or render a template
    // in the template, print things with {{ article.titre }}
    return $this->render('article/show.html.twig', ['article' => $article]);
}
```

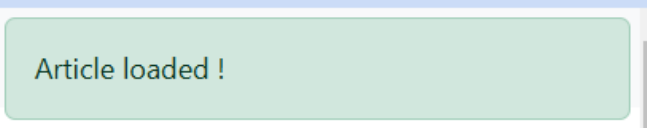
Cela chargera en session le message dans un tableau de “success”. Il ne nous reste donc plus qu’à le traiter. Pour cela, dans notre fichier twig précisant la structure globale de notre page, nous allons ajouter une div spécifique

```
22     <body>
23         <div class="menutop">
24             {% include 'menutop.inc.html.twig' %}
25         </div>
26         <div class="menuleft">
27             {% include 'menuleft.inc.html.twig' %}
28         </div>
29         <div class="maincontent">.
30             {% block maincontent %}{% endblock %}
31         </div>
32         <div class="footer">
33             {% include 'footer.inc.html.twig' %}
34         </div>
35         <div class="messages">
36             {% for message in app.flashes('success') %}
37                 <div class="alert alert-success">
38                     {{ message }}
39                 </div>
40             {% endfor %}
41         </div>
42     </body>
```

et préciser dans notre CSS quelles sont ces propriétés, par exemple :

```
.messages {
    position: fixed;
    top: 5px;
    right: 5px;
    min-width: 350px;
}
```

pour obtenir une boîte de cet aspect



Article loaded !

qui sera positionnée en haut à droite

## Exercice 4

Vous réaliserez à présent une route pour supprimer des articles (sur `/article/delete/` suivi de l'id)

Cette route doit être appelée à partir d'un lien "Supprimer" présent en bas d'une fiche article. Une Confirmation utilisateur devra être faite pour limiter les erreurs de manipulation. Une fois l'article supprimé, l'utilisateur devra être redirigé vers la liste des articles.

Vous ajusterez ensuite vos route précédemment créées en implémentant des messages Flash pour communiquer des informations avec vos utilisateurs et améliorer vos interfaces.

## Exercice 5

Maintenant que vous avez vu la réalisation d'un CRUD pour gérer des articles, vous allez implémenter par vous même la mise en place d'un système de commentaires sur les articles.

Quelques précisions :

- Lorsque nous sommes sur un article, il est possible de consulter les commentaires déjà présents.
- Il est également possible d'ajouter un nouveau commentaire.
- Pour l'instant, la modification et la suppression ne sont pas attendues car elles seront à conditionner par une authentification.



# La Sécurité et les accès utilisateur

Symfony fournit de nombreux outils pour sécuriser votre application. Certains outils de sécurité liés à HTTP, comme les cookies de session sécurisés et la protection CSRF, sont fournis par défaut.

Le SecurityBundle, que nous allons aborder ici, fournit toutes les fonctionnalités d'authentification et d'autorisation nécessaires pour sécuriser votre application.

## Installation

Comme pour les précédentes fonctionnalités, cette installation se fait depuis le shell de votre conteneur docker :

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

## Configuration de base

Le fichier contenant la configuration est présent dans `config/packages/security.yaml` et contient tout un ensemble de sections. Les 3 principaux que nous aborderons sont :

- providers, qui concerne l'utilisateur
- firewalls, qui concerne le pare-feu et l'authentification des utilisateurs
- access\_control, qui concerne le contrôle d'accès utilisateur, autrement dit les autorisations.

## L'utilisateur

Les permissions dans Symfony sont toujours liées à un objet User. Quand vous avez besoin de sécuriser votre application (en partie ou dans la globalité), vous devez créer une classe User. Il s'agit d'une classe qui implémente `UserInterface`. Comme nous avons déjà vu Doctrine, nous allons l'utiliser pour la générer, mais ce n'est pas un impératif : vous pouvez également utiliser une classe dédiée à la sécurité.

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

La création de ce type d'objet spécifique fonctionne également de manière interactive et vous comprenez d'autant plus les avantages d'un outil comme Doctrine ;-)

```
root@330a9ef60b2f:/app# php bin/console make:user

The name of the security user class (e.g. User) [User]:
> User

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

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

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

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

Success!

#### Next Steps:

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

Ce qui vous génère le fichier `src/Entity/User.php`.

Enfin, n'oubliez pas de recréer les tables correspondantes et de relancer une migration Doctrine

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

Si vous regardez à nouveau votre fichier `config/packages/security.yaml`, vous verrez que la section `providers` a été modifiée lors de la commande `make:user`

```
providers:
    users_in_memory: { memory: null }
```

```
providers:
    # used to reload user from session
    # & other features (e.g. switch_user)
    app_user_provider:
        entity:
            class: App\Entity\User
            property: email
```

C'est ainsi que Symfony sait qu'il doit charger une entité `User` à partir de la base de donnée, et plus précisément en se basant sur la propriété `email`.

Maintenant que nous avons la structure de persistance pour nos utilisateurs, il nous faut implémenter le formulaire d'enregistrement. Et pour cela, Symfony nous propose également un maker : le `registration-form`.

Pour pouvoir l'utiliser, vous devez installer :

```
# composer require symfonycasts/verify-email-bundle validator
```

et ensuite lancer

```
# php bin/console make:registration-form
```

Dans le cadre de ce cours, nous allons :

- Ajouter l'attribut de validation `#[UniqueEntity]`
- désactiver l'envoi d'un email de vérification
- désactiver l'authentification automatique après enregistrement
- rediriger l'utilisateur vers notre route `app_index` après enregistrement

La fonctionnalité de génération des tests unitaires étant expérimentale, nous ne l'utiliserons pas ici.

Enfin, nous allons à présent générer le formulaire de connexion

```
# php bin/console make:security:form-login
```

Dans le cadre de ce cours, nous allons :

- laisser le SecurityController comme nom de la classe
- générer l'URL /logout pour la déconnexion
- ne pas utiliser la génération des tests unitaires.

## Exercice 5

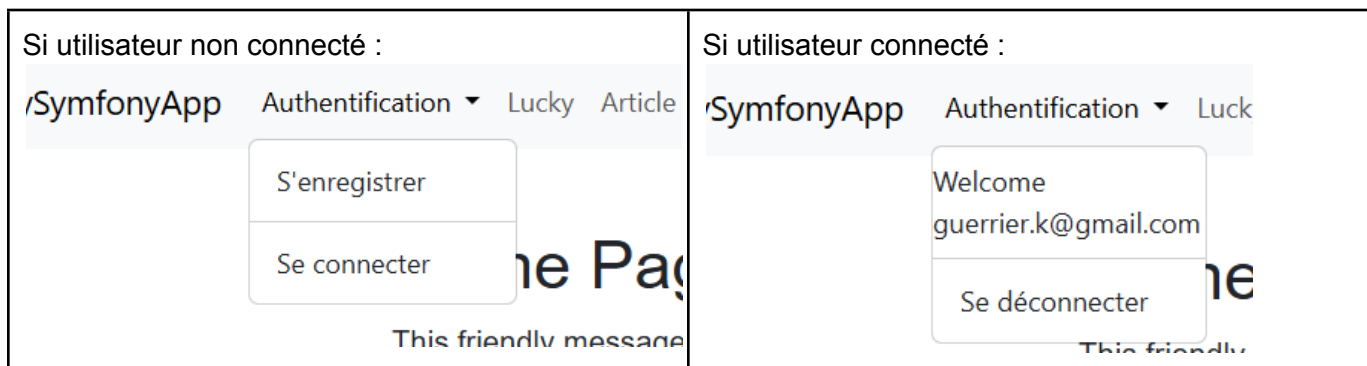
Ajuster le contenu de votre projet pour permettre la prise en charge des utilisateurs, notamment :

- Dans le menu haut, ajouter un menu déroulant pour l'authentification contenant :
  - un lien pour s'enregistrer
  - un lien pour se connecter

*Piste :*

*Nous avons vu que les URLs de navigation sont gérées directement à partir des contrôleurs. La gestion utilisateur ne fait pas exception. Vous devriez donc aller regarder ce qu'il y a dans les contrôleurs qui ont été générés automatiquement*

- Ajuster à présent ce menu pour avoir un comportement différent si l'utilisateur est connecté :



*Piste :*

*Comme toujours, la documentation officielle est votre amie ! Consultez la partie concernant la sécurité disponible à cette adresse : <https://symfony.com/doc/current/security.html> . Vous devriez y trouver ce dont vous avez besoin ;-)*

- Ajuster les fichiers twig pour intégrer les formulaires à votre application et améliorer leurs affichages

*Piste :*

*Vous pouvez personnaliser l'affichage de vos formulaires en utilisant tout un ensemble de fonctionnalités : [https://symfony.com/doc/current/form/form\\_customization.html](https://symfony.com/doc/current/form/form_customization.html) . Et si vous ne l'aviez pas encore fait pour vos formulaires d'articles / de commentaires, profitez-en pour les ajuster !*

# Les contrôles d'accès

Maintenant que les utilisateurs peuvent se connecter à l'application à l'aide du formulaire de connexion, il est nécessaire de préciser qui peut accéder à quoi. Et pour cela, vous allez utiliser l'objet User et notamment son rôle. C'est par ce biais que vous pourrez préciser si un utilisateur peut accéder à une ressource (une URL, un objet modèle, un appel de méthode, ...).

Le processus d'autorisation comporte deux volets différents :

- L'utilisateur reçoit un rôle spécifique lors de la connexion (par exemple ROLE\_ADMIN).
- Vous ajoutez du code pour qu'une ressource (par exemple une URL, un contrôleur) nécessite un « attribut » spécifique (par exemple un rôle comme ROLE\_ADMIN) pour être accessible.

## Les rôles...

Lorsqu'un utilisateur se connecte, Symfony appelle la méthode `getRoles()` de votre objet User pour déterminer les rôles de cet utilisateur. Dans la classe User générée précédemment, les rôles sont un tableau stocké dans la base de données et chaque utilisateur se voit toujours attribuer au moins un rôle `ROLE_USER`:

<pre>/**  * @var list&lt;string&gt; The user roles  */ #[ORM\Column] private array \$roles = [];</pre>	<pre>/**  * @see UserInterface  *  * @return list&lt;string&gt;  */ public function getRoles(): array {     \$roles = \$this-&gt;roles;     // guarantee every user at least has     ROLE_USER     \$roles[] = 'ROLE_USER';      return array_unique(\$roles); }</pre>
--	--

C'est une valeur par défaut intéressante, mais vous pouvez créer tous les rôles que vous souhaitez pour le bon fonctionnement de votre situation.. La seule règle est que chaque rôle doit commencer par le préfixe `ROLE_` (par exemple `ROLE_ARTICLE_ADMIN`).

Vous utiliserez ensuite ces rôles pour accorder l'accès à des sections spécifiques de votre site.

## ... et leur organisation hiérarchique

Au lieu d'attribuer plusieurs rôles à chaque utilisateur, vous pouvez définir des règles d'héritage de rôles en créant une hiérarchie de rôles. Ainsi, dans le fichier `config/packages/security.yaml`, vous pouvez préciser :

```
security:
    [...]

    role_hierarchy:
        # Les utilisateurs avec le ROLE_ADMIN auront également le ROLE_USER
        ROLE_ARTICLE_ADMIN:    ROLE_USER
        ROLE_COMM_ADMIN:       ROLE_USER
        # Les utilisateurs avec ROLE_ADMIN, auront automatiquement
        # ROLE_ADMIN, ROLE_ARTICLE_ADMIN, ROLE_COMM_ADMIN, et ROLE_USER en héritage
        ROLE_ADMIN:            [ROLE_COMM_ADMIN, ROLE_ARTICLE_ADMIN]
        ROLE_SUPER_ADMIN:      ROLE_ADMIN
```

Une fois ces rôles définis, il vous est possible de les utiliser de plusieurs manières :

La première se fait directement dans le fichier `config/packages/security.yaml`, dans la partie `access_control` :

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

Vous noterez que seule la première expression régulière qui match avec la requête sera prise en compte ! Soyez donc attentif à l'ordre dans lequel vous définissez vos règles.

La seconde se fait à partir d'un contrôleur. Par exemple, si nous voulons créer un contrôleur qui gèrera un panel d'administration :

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

Et nous allons y placer quelques méthodes pour vous permettre de tester :

```
<?php
namespace App\Controller;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Attribute\Route;

use Symfony\Component\Security\Http\Attribute\IsGranted;
```

→ Nous allons ajouter la dernière ligne pour avoir accès à IsGranted

```
#[IsGranted('ROLE_USER', statusCode: 403, message: 'You must be logged in.')]
class AdminController extends AbstractController
{
    #[Route('/admin', name: 'app_admin')]
    public function index(): Response
    {
        return $this->render('admin/index.html.twig', [
            'controller_name' => 'AdminController',
        ]);
    }
}
```

→ Avec la première ligne, nous précisons que l'ensemble du contrôleur nécessite le ROLE\_USER

```
#[Route('/admin/test1', name: 'app_admin_test1')]
public function index_test(): Response
{
    $this->denyAccessUnlessGranted('ROLE_ADMIN');
    // or add an optional message - seen by developers
    $this->denyAccessUnlessGranted('ROLE_ADMIN', null, 'User tried to access a
page without having ROLE_ADMIN');

    return $this->render('admin/index.html.twig', [
        'controller_name' => 'AdminController',
    ]);
}
```

→ ici nous utilisons l'appel à denyAccessUnlessGranted à l'intérieur de la fonction

```
#[Route('/admin/test2', name: 'app_admin_test2')]
#[IsGranted('ROLE_SUPER_ADMIN', statusCode: 403, message: 'You are not allowed to
access the Super admin dashboard.')]
public function index_test2(): Response
{
    return $this->render('admin/index.html.twig', [
        'controller_name' => 'AdminController',
    ]);
}
}
```

→ Et pour cette dernière partie, nous utilisons le `IsGranted` directement sur l'appel à la fonction au même titre que la définition de la Route

Et pour terminer, vous pouvez également tester si l'utilisateur en cours dispose d'un rôle spécifique directement à partir d'un template Twig en utilisant la syntaxe suivante :

```
{% if is_granted('ROLE_ADMIN') %}
    <a href="...">Delete</a>
{% endif %}
```

Ce qui peut être utile pour conditionner l'affichage d'un bouton de suppression par exemple

## Exercice 6

- Créez un utilisateur avec une adresse email spécifique et modifiez la fonction `getRoles` dans `Entity/User.php` pour attribuer le `ROLE_SUPER_ADMIN` lorsque l'utilisateur se connecte avec cette adresse.
- Créer un autre utilisateur pour vérifier qu'il ne récupère pas ce rôle là également ;-)
- Implémentez la partie permettant de gérer les utilisateurs à partir de `/admin/users/` et qui vous permet de :
  - lister les utilisateurs
  - supprimer un utilisateur
  - manager les rôles d'un utilisateur.

Ces interfaces seront bien sûr uniquement accessibles avec le `ROLE_SUPER_ADMIN`.

- Implémentez également les niveaux de sécurité pour permettre :
  - l'ajout d'articles aux `ROLE_USER`
  - la suppression d'articles aux `ROLE_ARTICLE_ADMIN`
  - la modération de commentaires aux `ROLE_COMM_ADMIN` (publication / dépublication / suppression)
- Implémentez la partie `/profile` qui permet à un utilisateur de gérer son mot de passe et son adresse email et de supprimer son compte (avec une sécurité pour le `ROLE_SUPER_ADMIN` qui ne peut pas se supprimer)

## La partie Firewall

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

## La protection CSRF

Piste : <https://symfony.com/doc/current/security.html#csrf-protection-in-login-forms>

## Personnalisez vos messages d'erreur

Piste : [https://symfony.com/doc/current/controller/error\\_pages.html](https://symfony.com/doc/current/controller/error_pages.html)

en utilisant notamment :

```
$ composer require symfony/twig-pack
```

## Déployer votre application (passage de dev en prod)

Piste : <https://symfony.com/doc/current/deployment.html>

## Traitement de l'image d'un article

Piste : [https://symfony.com/doc/current/controller/upload\\_file.html](https://symfony.com/doc/current/controller/upload_file.html)

- composer require symfony/mime
- Ajouter au formulaire le traitement de l'article la gestion de l'image
- Mettre à jour l'entité pour inclure le nom du fichier
- Ajout de l'image dans la liste des articles
- Détail de l'article sur page dédiée

## Ajout de la Catégorie d'article

Piste : <https://symfony.com/doc/current/doctrine/associations.html#saving-related-entities>

- Création de l'entité Catégorie
  - Créer une entité catégorie qui a une relation avec l'article (une catégorie est attachée à X articles) La relation se fait dans l'entité article
  - Créer le CRUD des catégories
- Modifier le CRUD article pour associer une catégorie à un article

## Exposition des entités en API

- composer require api
- En cas d'erreur il faut downgrade phpstan/phpdoc-parser:^1.0



# Synthèse des éléments du TP à Rendre

Suite à l'ensemble de ces exercices, vous devriez déjà avoir un grand nombre de ces fonctionnalités implémentées. Assurez-vous que toutes soient fonctionnelles et implémentez celles qui manquent.

- Application accessible publiquement par défaut
- Toutes les pages doivent implémenter la même structure (bandeau de menu, contenu, pied de page) et avoir une CSS avec un minimum de mise en page (pas besoin de faire des effets de folie mais une attention particulière sur l'ergonomie, les formulaires, les titres, les champs, les boutons, ... )
- Dans le menu haut, il y a un accès à :
  - La page d'accueil qui liste
    - Les 3 derniers articles créés (en format aperçu)
  - Le menu déroulant d'authentification contenant :  
si l'utilisateur n'est pas connecté :
    - S'inscrire
    - Se connectersi l'utilisateur est connecté :
    - Bienvenue %user%
    - Mon Profil
    - Se déconnecter
  - Le lien vers le contrôleur Lucky
  - Le menu déroulant de gestion des Articles contenant :
    - La liste des articles
    - Le formulaire de création d'un nouvel article
    - Le lien de génération d'un article aléatoire
  - Le lien vers la page About contenant vos informations (Nom / Prénom / ...)
  - Le menu déroulant d'administration contenant :
    - La gestion des utilisateurs
    - La gestion des articles
    - La gestion des commentaires

Pour chaque entité, le CRUD est implémenté (même si certaines fonctionnalités nécessitent un ROLE spécifique), les messages d'erreurs et de confirmations sont affichés à l'utilisateur dans une zone dédiée, et les pages d'erreur sont personnalisées (404 / 403 / 500 / ...).

## En Bonus :

- Une zone de recherche dans la barre de menu permettant de rechercher un contenu dans les articles (dans les titres titres et/ou textes),
- L'ajout d'une catégorisation des articles,

La date limite pour le rendu de ce TP est fixée au  
02/02/2025 - 23:00  
(Boîte de dépôt sur Moodle)