



Introduction

Les frameworks

Le MVC

Installation de Symfony

Création d'un projet de démonstration

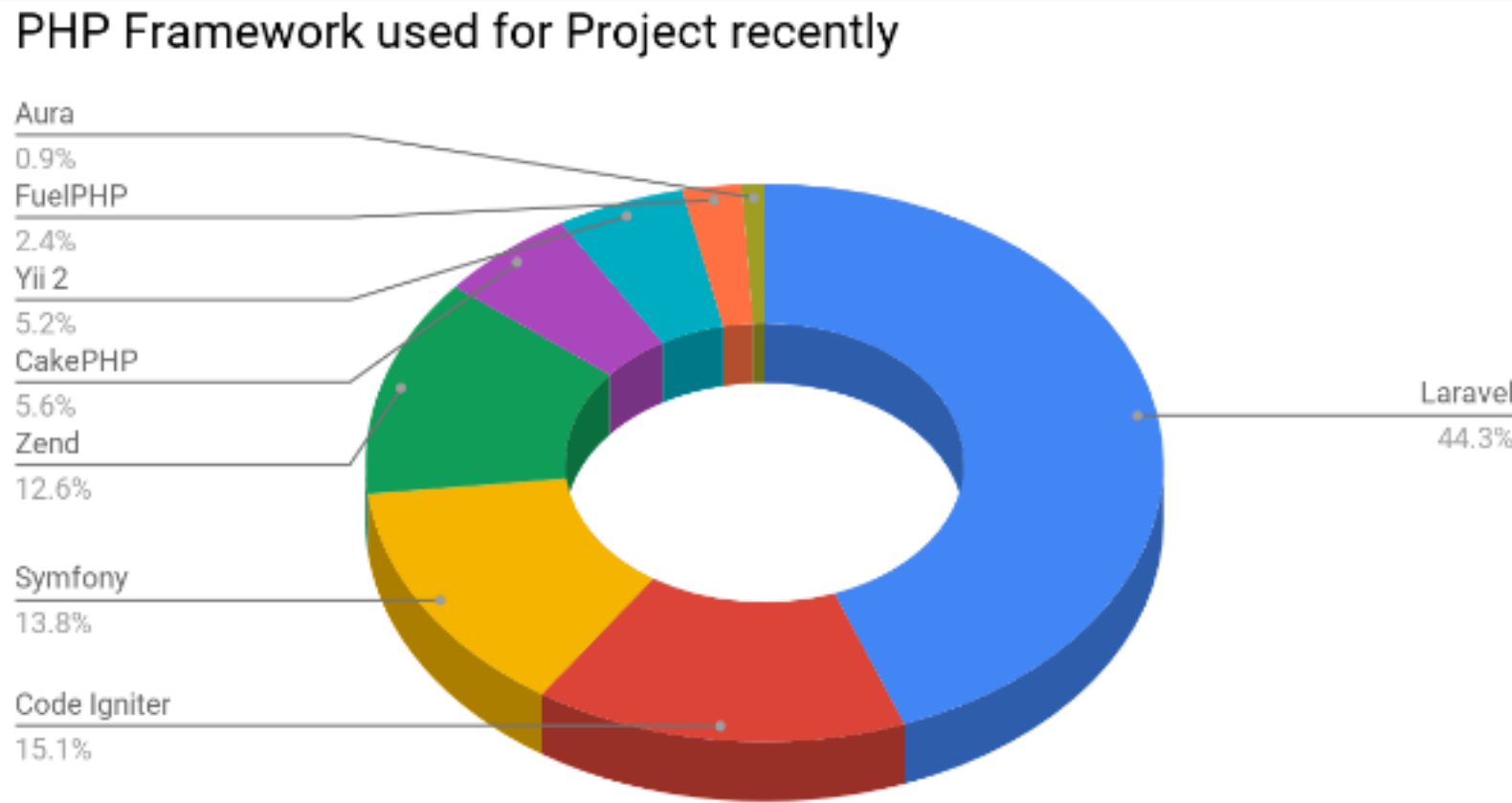
Les environnements de développement et de production

Notions supplémentaires

Principaux frameworks 2023



Principaux frameworks 2023



Qu'est-ce qu'un framework ?

Un Framework est une sorte de cadre applicatif qui permet de réduire le temps de développement des applications, tout en répondant de façon efficace aux problèmes rencontrés le plus souvent par les développeurs.

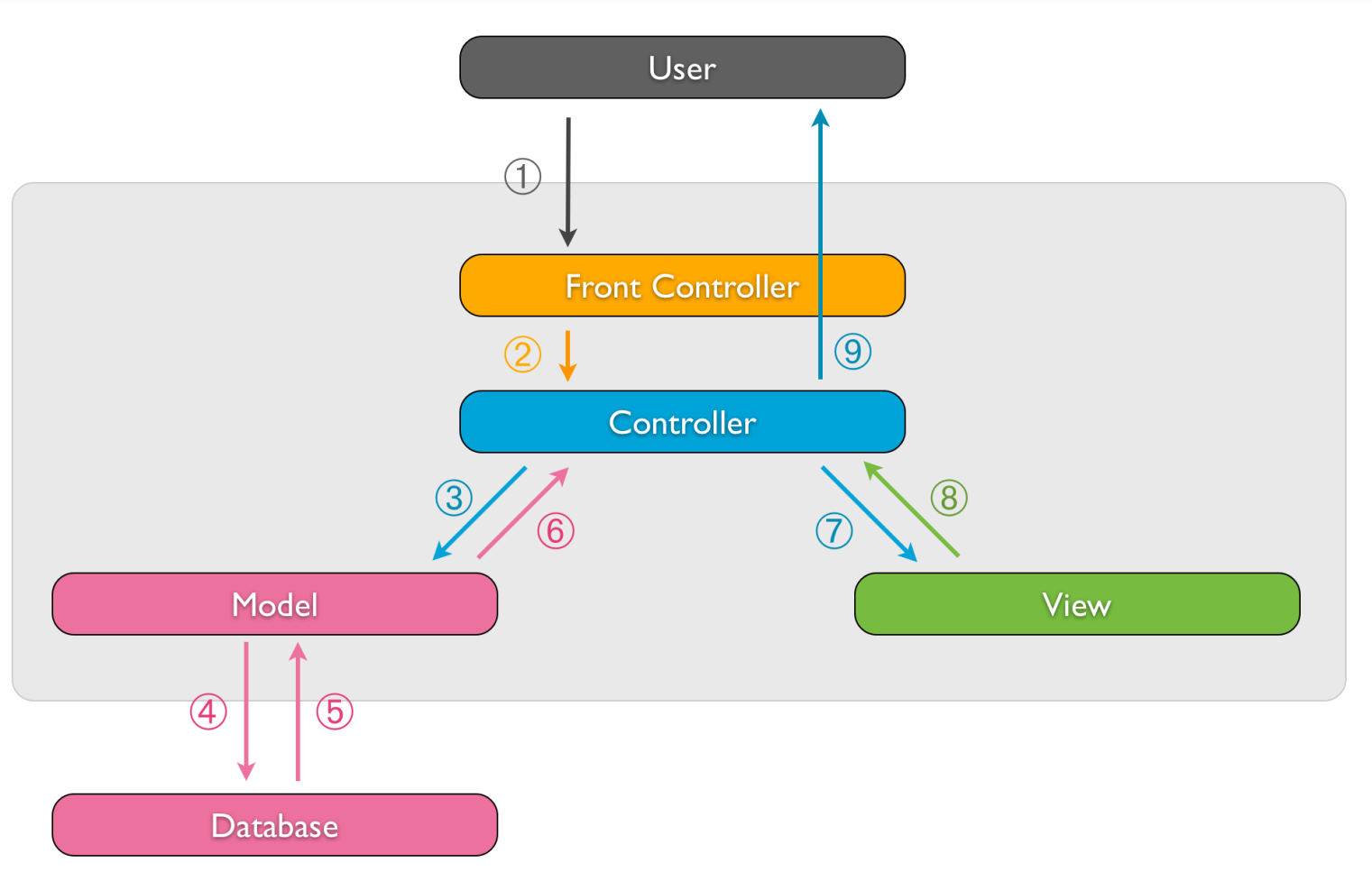
Il inclut généralement de nombreuses fonctionnalités prêtes à l'emploi dont les implémentations sont bien rodées et utilisent des modèles de conception standard et réputés. Le temps ainsi gagné sur les questions génériques pourra être mis à profit pour les parties spécifiques de l'application.

Enfin un framework c'est aussi le fruit du travail de centaines voir de milliers de personnes qui s'appliquent à corriger les problèmes ou les failles de sécurité découvertes par l'ensemble des utilisateurs et à proposer de nouvelles fonctionnalités. De ce fait, les programmes d'un Framework sont en général mieux conçus et mieux codés, mais aussi mieux débogués et donc plus robustes que ce que pourrait produire un unique programmeur. Outre le gain de temps, on obtient un important gain en termes de qualité.

Caractéristiques d'un framework

- Fournir des bibliothèques éprouvées permettant de s'occuper des tâches usuelles (formulaire, validation, sécurité, mail, template,...).
- Assurer une couche d'abstraction avec les données (ORM).
- Incitation aux bonnes pratiques: il vous incite, de par sa propre architecture, à bien organiser votre code. Et un code bien organisé est un code facilement maintenable et évolutif.
- Améliore la façon dont vous travaillez et facilite le travail équipe.
- Fournir une communauté active et qui contribue en retour:
 - code source maintenu par des développeurs attitrés
 - code qui respecte les standards de programmation
 - support à long terme garanti et des mises à jour qui ne cassent pas la compatibilité
 - proposer des bibliothèques tierces (bundle: <http://knpsbundles.com>)

MVC



MVC

MVC signifie « Modèle / Vue / Contrôleur ». C'est un découpage très répandu pour développer des sites Internet, car il sépare les couches selon leur logique propre :

Le Contrôleur (ou Controller) : son rôle est de générer la réponse à la requête HTTP demandée par notre visiteur. Il est la couche qui se charge d'analyser et de traiter la requête de l'utilisateur. Le contrôleur contient la logique de notre site Internet et va se contenter « d'utiliser » les autres composants : les modèles et les vues.

Concrètement, un contrôleur va récupérer, par exemple, les informations sur l'utilisateur courant, vérifier qu'il a le droit de modifier tel article, récupérer cet article et demander la page du formulaire d'édition de l'article.

MVC

- **Le Modèle** (ou Model) : son rôle est de gérer vos données et votre contenu. Reprenons l'exemple de l'article. Lorsque je dis « le contrôleur récupère l'article », il va en faire appel au modèle **Article** et lui dire : « donne-moi l'article portant l'id 5 ». C'est le modèle qui sait comment récupérer cet article, généralement via une requête au serveur, mais ce pourrait être depuis un fichier texte ou ce que vous voulez.
- Au final, il permet au contrôleur de manipuler les articles, mais sans savoir comment les articles sont stockés, gérés, etc. C'est une couche d'abstraction.

MVC

- **La Vue** (ou View) : son rôle est d'afficher les pages. Reprenons encore l'exemple de l'article. Ce n'est pas le contrôleur qui affiche le formulaire, il ne fait qu'appeler la bonne vue. Si nous avons une vue Formulaire, les balises HTML du formulaire d'édition de l'article y seront et au final le contrôleur ne fera qu'afficher cette vue sans savoir vraiment ce qu'il y a dedans.
- En pratique, c'est le designer d'un projet qui travaille sur les vues. Séparer vues et contrôleurs permet aux designers et développeurs PHP de travailler ensemble sans entrer perturber le développement.

Installation de Symfony

Configuration de l'éco système

Ce dont vous allez avoir besoin avant l'installation

Serveur local
Version de PHP
Git
Composer

Installation de Git

- Git est un logiciel de gestion de versions (VCS) permettant notamment de contrôler les différentes versions de votre projet et d'améliorer le travail en équipe.
 - Rendez-vous sur le site officiel pour télécharger la dernière version: <https://git-scm.com/download/win>
 - Sélectionnez « Git Bash » et « Git Gui »
 - Laissez les autres options par défaut.
 - Testez le bon fonctionnement avec la commande: `git --version`

Installation de Composer

- Composer est un logiciel de dépendances pour PHP. Il permet d'installer des librairies supplémentaires dans le cadre de Symfony ou de n'importe quel projet personnel.
- Installez composer:
 - Url : <https://getcomposer.org/download/>
 - Pour Windows : télécharger et exécuter [composer-Setup.exe](#)
 - Lors de l'installation vérifiez bien la version de PHP
 - Tester l'installation en saisissant dans l'invite de commande l'instruction « composer »

L'installation se fait dans l'environnement global et **Composer** peut être utilisé dans tous vos projets. Il sera à nouveau utilisé pour l'installation de bundles, dépendances et autres librairies. Si vous possédez déjà **Composer** exécutez la commande **composer self-update** pour une mise à jour.

Installation de Symfony

Serveur local
Version de PHP
Git
Composer

Installation de Symfony.

1. Avant l'installation, créez un dossier Symfony pour tous vos projets dans votre localhost (www).
2. Rendez-vous sur la [doc officielle](#) pour l'installation de la CLI (Command-line interface) de Symfony. Le plus simple est d'installer [Scoop](#) (Scoop Package Manager est un programme d'installation en ligne de commande pour Windows) avant de commencer. Il suffit de saisir les deux lignes de commandes suivantes dans le terminal:
 1. `Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser`
 2. `Invoke-RestMethod -Uri https://get.scoop.sh | Invoke-Expression`
3. Une fois Scoop installé, saisissez la commande suivante pour installer la CLI de Symfony:
 1. `scoop install symfony-cli`
4. Ouvrez une fenêtre PowerShell (invite de commande) à partir du dossier Symfony.
5. Saisissez la commande: `symfony new --webapp demo` (demo étant le dossier de notre premier projet découverte).

Installation de Symfony.

6. Après confirmation, la structure du nouveau projet a été créée en local. Ouvrez le dossier dans votre IDE pour constater qu'une arborescence et de nombreux fichiers composent maintenant votre futur projet.
7. Pour démarrer, ouvrez le terminal et saisissez: `symfony server:start`. Cette commande va exécuter le serveur interne de Symfony et vous permettre via le lien affiché en bas d'accéder à votre projet. Pour quitter le serveur utilisez la commande CONTROL + C
8. Cliquez sur le lien proposé ou saisissez l'url <http://127.0.0.1:8000/> pour accéder à la page d'accueil par défaut de Symfony.

Installation de Symfony.

Pour obtenir toutes les données techniques concernant Symfony, tapez en mode console :

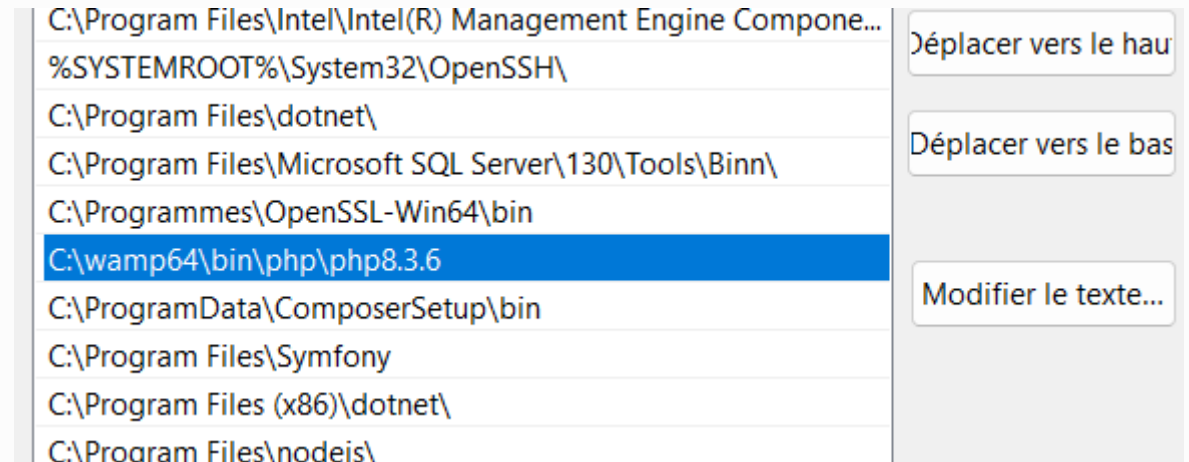
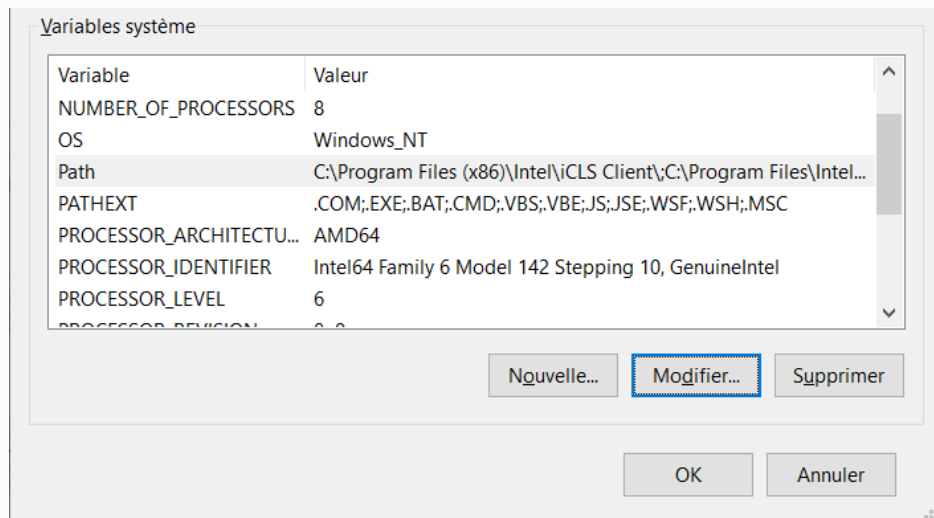
```
php bin/console about
```

Vous obtiendrez les informations:

- Sur la version, la maintenance, et le support de Symfony
- Sur le Kernel de Symfony (noyau)
- Sur PHP (version, debug,...)

Le problème de version de PHP – Path

- En fonction de votre configuration antérieure, il est possible qu'à la fin de l'installation, un message d'erreur apparaisse vous signifiant un problème de version de PHP.
- Il s'agit d'une mauvaise configuration des variables d'environnement. Dans les paramètres Windows, recherchez variables d'environnement et ensuite la commande « **Modifier les variables d'environnement système** ». Sélectionnez « **Path** » et ensuite le bouton « **Modifier** ». Le Path doit correspondre à une version supérieure à PHP 7.1. Modifiez en fonction de votre version installée ou faites **Parcourir** pour rechercher l'exécutable de PHP.



Le problème de version de PHP – Symfony server

Vérifiez dans un terminal et en ligne de commande votre version php: `php -v`

Il est possible que le serveur de Symfony refuse de fonctionner en raison d'une configuration erronée de la version de PHP.

Vous pouvez forcer la modification des paramètres du serveur dans le terminal avec la commande:

```
echo 8.3 > .php-version
```

Un autre problème peut également se poser lors du démarrage du serveur. Il est possible que la commande `symfony server:start` ne soit pas reconnue. C'est également lié au Path mais celui de Symfony.

C:\Program Files\Symfony ou C:\Programmes\Symfony (il faut aussi laisser les deux)

N'oubliez de redémarrer Windows

Le problème de version stable

Si vous souhaitez installer un nouveau projet à partir de la commande symfony il peut arriver que l'opération soit interrompue et que vous obteniez le message suit:

```
unable to run C:\ProgramData\ComposerSetup\bin\composer.phar create-project symfony/website-skeleton  
C:\wamp64\www\Symfony2021\practice --no-interaction
```

Dans ce cas, vous devez passer par Composer pour l'installation:

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

Le problème de la version PHP de la CLI

- Vérifier dans les paramètres PhpStorm (CTRL + ALT + S) rubrique php:
 - PHP language level
 - CLI interpréter
- Vous pouvez vérifier avec la commande: `symfony check:requirements` si votre environnement est opérationnel avant de créer votre premier projet avec Symfony.

Installation d'une version LTS

- Pour installer une version antérieure qui est maintenue par l'équipe de SensioLabs (long-term support) utilisez `composer` et saisissez la commande suivante:
- `symfony new my_project_name --version=lts`
- Ceci vous permet de travailler sur la dernière version stable et maintenue de Symfony.

Configuration d'une base de données

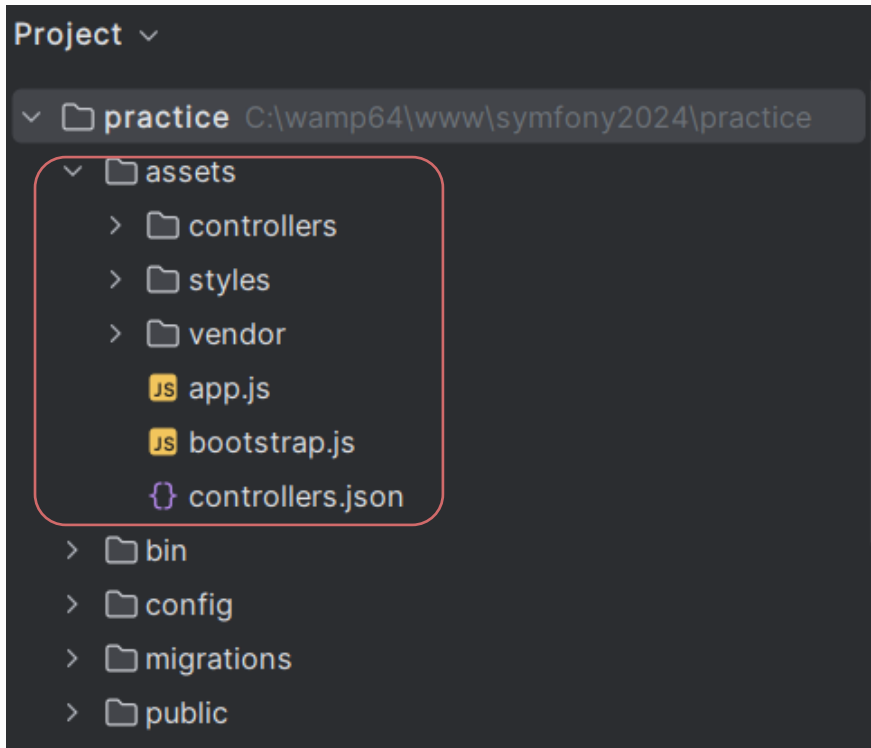
A partir de la **version 6.1** de Symfony, vous n'êtes plus obligé de configurer une base de données pour tout nouveau projet. En principe c'est évident dès lors que vous utilisez un framework. Cependant, pour notre découverte de Symfony une base données n'est pas nécessaire.

Si vous souhaitez en configurer une malgré tout, vous pouvez configurer et installer une base de données vide uniquement pour ce projet ou alors configurer une base de données existantes.

Ouvrez le fichier **.env** de la racine du projet décommentez la ligne vers le driver mysql et modifiez là.

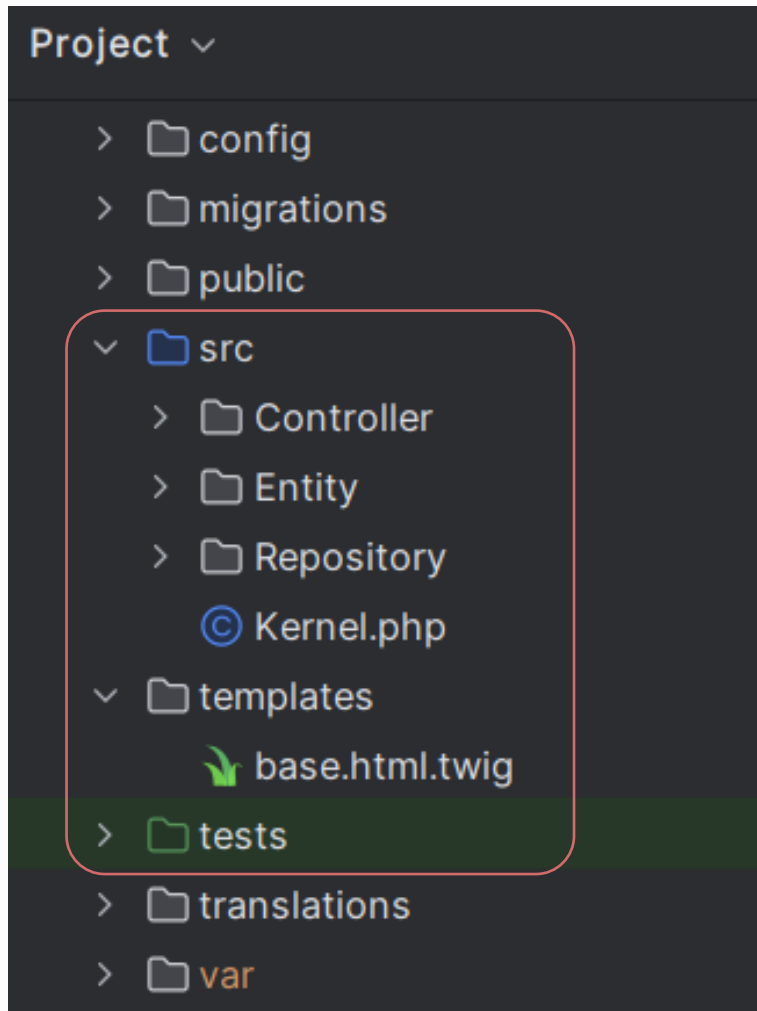
```
DATABASE_URL="mysql://root:@127.0.0.1:3306/weblearning?serverVersion=8.0.28.32&charset=utf8mb4"
```


l'architecture d'un projet vide



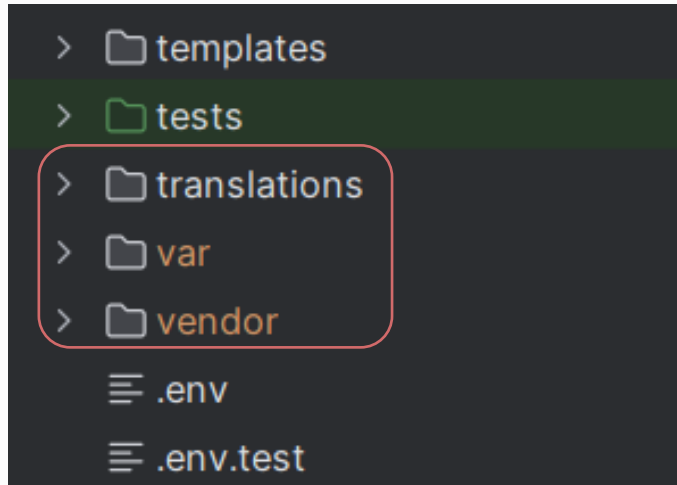
- **Assets**: Ce dossier fait partie d'un nouveau composant de Symfony 7 (AssetMapper). Contient les fichiers nécessaires au Front: Css, images, graphisme, votre JavaScript et toutes les librairies externes en JS.
- **bin**: ce répertoire contient l'exécutable dont nous allons nous servir en mode console pendant le développement. La commande pour y accéder est la suivante: `php bin/console`. Vous avez également la commande pour les tests unitaires.
- **config**: contient tous les fichiers de configuration au format yaml (routing, security, configuration de développement ou de production,...).
- **migrations**: Reprend les fichiers de migrations permettant la création et l'update de la DB
- **public**: C'est là que seront dirigés chacun de vos internautes une fois votre site web mis en ligne (via le fichier "index.php"). Il est le seul dossier accessible sur le net, tout les autres étant protégés par votre hébergeur.

l'architecture d'un projet vide



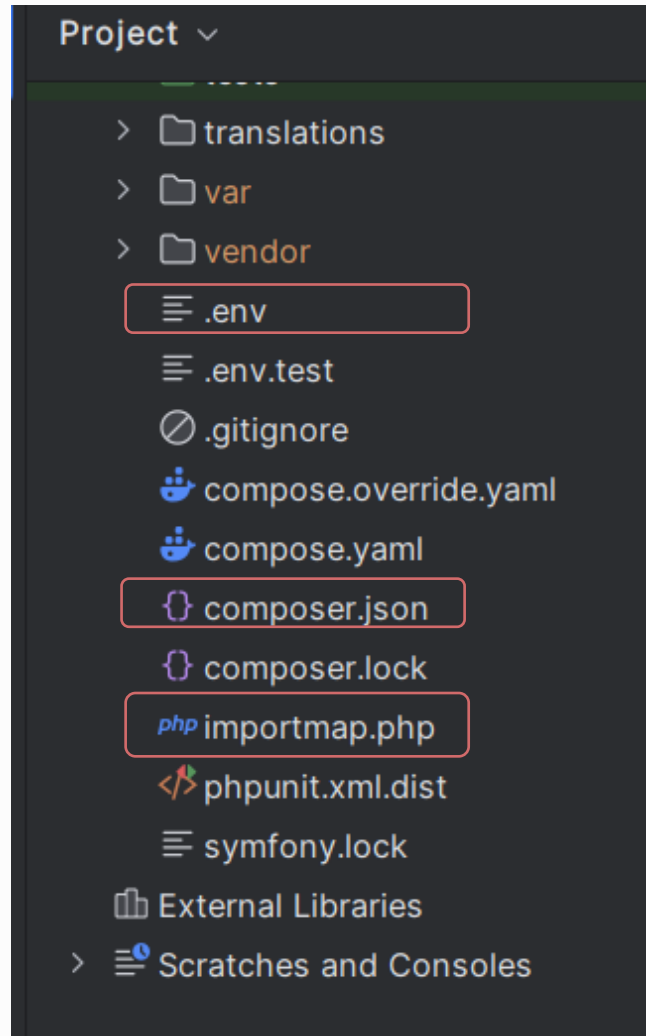
- **src**: ce dossier comporte toute l'architecture php de votre site web avec les Controllers, les Entités, les Repositories, les forms et autres fichiers(que nous créerons par la suite). Il correspond au namespace **App** défini dans l'autoloader de Composer.
- **templates**: Ici, se trouvent toutes les vues, les templates liés au moteur de template Twig (tout votre HTML). actuellement il contient un seul fichier: base.html.twig. Il nous servira de modèle pour la création de toutes nos vues.
- **tests**: dossier réservé aux tests fonctionnels et unitaires.

l'architecture d'un projet vide



- **translations**: contiendra les fichiers de traduction en Json ou xml.
- **var**: Il contient tout ce que Symfony va écrire durant son process : les logs, le cache, informations de session et d'autres fichiers nécessaires à son bon fonctionnement.
- **vendor**: le «core framework». L'ensemble des librairies et dépendances dont vous avez besoin (celles déjà fournies et celles que vous allez installer).

l'architecture d'un projet vide



- Le fichier `.env`: il contient la configuration de l'environnement d'exécution de notre code (notamment la configuration à la base de données ou la configuration de l'environnement). Le tout sous la forme de variables globales.
- Le fichier `composer.json`: il contient la liste des dépendances de votre projet. Utile pour le transfert, le partage et les mises à jour.
- Le fichier `importmap.php`: il contient un tableau multidimensionnel des dépendances (js et css) nécessaires pour la gestion des assets.

Premiers pas avec

Symfony

Créer ses premières pages Web

Création d'un contrôleur

Création d'une vue

Création et paramétrage d'une route

Le contrôleur

Principe

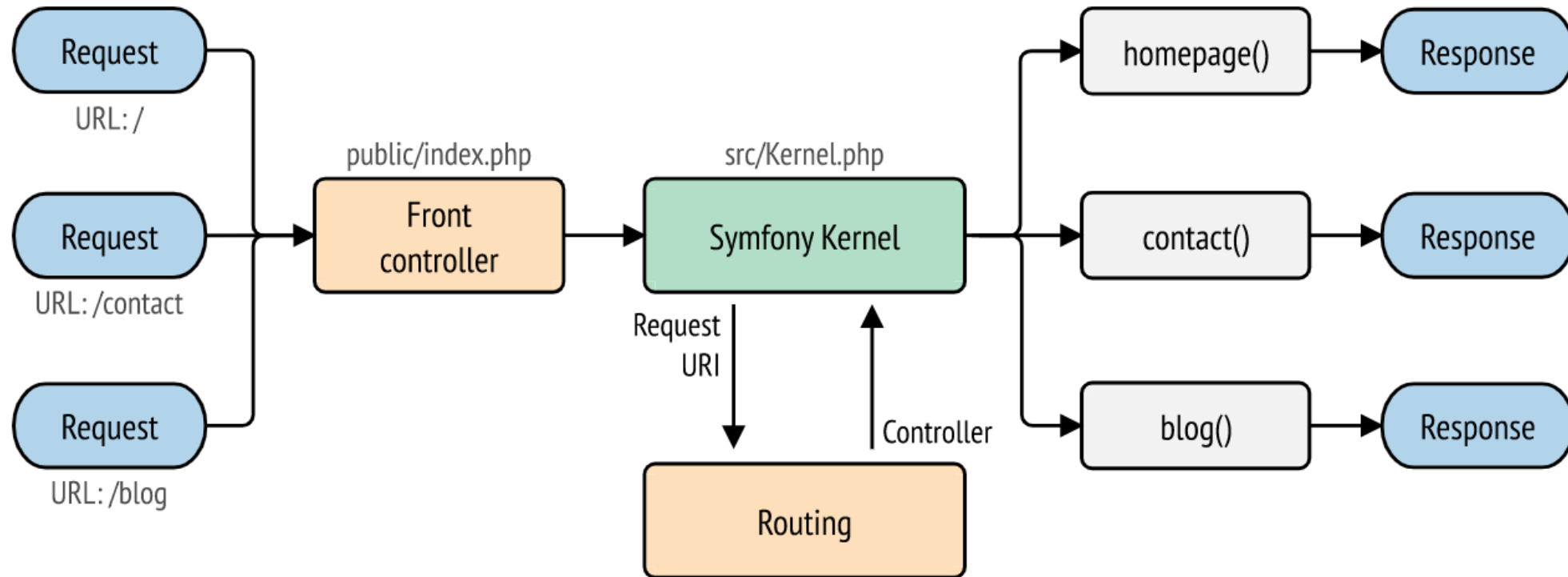
Syntaxe

Namespace

La route

Action du contrôleur

Une requête Symfony



Objectif de cette partie

Créer ses premières pages et découvrir les composants essentiels du framework:

1. Le contrôleur, les méthodes et les routes.
2. Les vues (template).

Ces notions sont parmi les plus importantes de Symfony et seront utilisées régulièrement dans tout projet Web. Lors de l'utilisation, vous devrez respecter certains principes généraux fixés par le framework: le nommage, la structure des dossiers et le positionnement des fichiers. Mais aussi les principes et la syntaxe de l'architecture MVC et de la programmation orientée objet.

Principes de base

Pour la création d'une nouvelle page (vue), qu'elle soit une page HTML, un point final JSON ou un contenu XML, l'opération est simple et composée de deux étapes :

1. Dans le contrôleur, **créer une route** : une route correspond à l'**URL** (ex : /about) de votre page et pointe sur une méthode (action).
2. **Créer une méthode** : une méthode va vous permettre de construire votre page. Vous prenez les requêtes d'information entrantes et les utilisez pour créer un objet Symfony, lequel va prendre en charge le contenu HTML, une chaîne JSON ou autre.

Tout comme sur le Web, chaque interaction est initiée par une requête HTTP. Votre travail est simple : comprendre une requête et retourner une réponse.

Le contrôleur et sa syntaxe

```
<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

class HomeController extends AbstractController
{

}
```

Créez un nouveau fichier PHP (HomeController.php) dans le dossier src/Controller. La classe doit toujours porter le même nom que le fichier (UpperCamelCase).

La ligne use... s'ajoute automatiquement lors de la création de la classe (PhpStorm)

Créez la classe HomeController et faites-la hériter de AbstractController (nous développerons)

Analyse – Les Namespaces et les Uses

En PHP, les espaces de noms sont conçus pour résoudre deux problèmes que rencontrent les auteurs de librairies et ceux d'applications lors de la réutilisation d'éléments tels que des classes ou des bibliothèques de fonctions :

1. Collisions de noms entre le code que vous créez, les classes, fonctions ou constantes internes de PHP, ou celles de bibliothèques tierces.
2. La capacité de faire des alias ou de raccourcir des noms extrêmement longs pour aider à l'écriture du code (héritage, instanciation...) et améliorer la lisibilité du code.

Analyse – Les Namespaces

Les espaces de noms sont déclarés avec le mot-clé namespace. Un fichier contenant un espace de noms doit le déclarer au début du fichier, avant tout autre code (sauf les commentaires !).

```
<?php  
namespace App\Controller;
```

Analyse – Le mot clé use

Permet de définir le chemin d'accès de la classe qui va être étendue ou utilisée.

```
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
```

```
class HomeController extends AbstractController  
{  
  
}
```

Câbler une route sur une action (routing)

- La route va, à partir d'une URL, déterminer quelle méthode sera appelée et avec quels paramètres (optionnels). Cela permet de configurer son application pour avoir des URL propres et légères.
- Les routes sous la forme d'attributs: #[Route()]

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

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

N'oubliez pas d'indiquer le use suivant en dessous du précédent sinon vous obtiendrez un message d'erreur lors de l'accès à la page. Il spécifie la classe "Route" que vous allez utiliser.

```
use Symfony\Component\Routing\Attribute\Route;
```

Création de l'action du contrôleur

- La méthode `index()` intercepte la requête (l'url et ses paramètres) et retourne une réponse.
- La méthode `render()` de `l'AbstractController` permet de renvoyer une vue. C'est-à-dire un template contenant du HTML et des instructions Twig pour le côté dynamique. Elle est invoquée sur `$this` représentant l'objet en cours.
- Créez un fichier « `index.html.twig` » dans le dossier `templates/home` et écrivez un simple contenu HTML.
- Ajoutez la méthode `index()` dans le contrôleur:

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

Contrôleur et sa méthode (action)

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

Le code fonctionne mais n'est pas aux normes php. Effectivement php recommande une déclaration de type pour ses méthodes ou ses fonctions. Nous allons déclarer le type de la méthode mais celui-ci fait partie du framework. Alors, un nouveau **use** s'impose pour ce type.

Déclaration du type de la méthode

- Ajoutez le use suivant:

```
use Symfony\Component\HttpFoundation\Response;
```

- Typez la méthode avec le type : Response

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

Contrôleur finalisé

```
<?php
namespace App\Controller;

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

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

Récapitulatif sur le processus

- Chaque requête envoyée par le client est analysée par le routeur de Symfony via la page index.php.
- Le système de routage établit la correspondance entre l'URL entrante et la route spécifique, puis retourne les informations relatives à la route, dont le contrôleur qui devra être exécuté.
- La méthode correspondante à la route est exécutée : c'est là que votre code crée et retourne l'objet approprié (la réponse). Dans notre cas, il retourne une vue.

La vue

Twig
Template

Twig – Moteur de rendu

- Permet la création du template (vue) en séparant le code PHP du code HTML. Via son pseudo-langage (Twing), il offre la possibilité de réaliser des fonctionnalités pour du code dynamique. Celui-ci est plus lisible et plus adapté pour l'affichage des pages Web que le PHP.
- Grace à son système de cache le rendu est optimisé et le traitement n'est pas plus long que du PHP.



Le template de base

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>{% block title %}Welcome!{% endblock %}</title>
    <link rel="icon" href="">
    {% block stylesheets %}
    {% endblock %}

    {% block javascripts %}
      {% block importmap %}{{ importmap('app') }}{% endblock %}
    {% endblock %}
  </head>
  <body>
    {% block body %}{% endblock %}
  </body>
</html>
```

- templates/base.html.twig
- Ce fichier est une trame de départ pour la conception des vues. Vous pouvez par la suite le modifier. Toutes vos vues devront en hériter.
- Des blocs ont déjà été prévus pour vos contenus, vos styles et votre JavaScript. Vous pourrez par la suite y ajouter vos propres blocs et d'autres feuilles de style.

Utiliser Twig dans notre première vue

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

{% block title %}Home - Symfony{% endblock %}

{% block body %}
<h1>Symfony - Accueil</h1>
<h2>Introduction aux vues</h2>
{% endblock %}
```

- Étendre le modèle de base de Twig ce qui vous obligera à respecter les règles du parent ! Pour ce faire on utilise la commande `{% extends 'base.html.twig' %}`
- Le contenu doit être inséré dans des « blocks » prévus à cet effet et définis dans le parent. Ici, les blocs `title` et `body`.
- La barre d'outils de débogage de Symfony est réapparue en mode (Web Debug Toolbar).

Debug Toolbar et Profiler

Debug Toolbar

Profiler

Fonction dump()

La Web Debug Toolbar

HTTP status: permet de déterminer le résultat d'une requête et indiquer au client une erreur.

200 : succès de la requête;

301 et 302 : redirection;

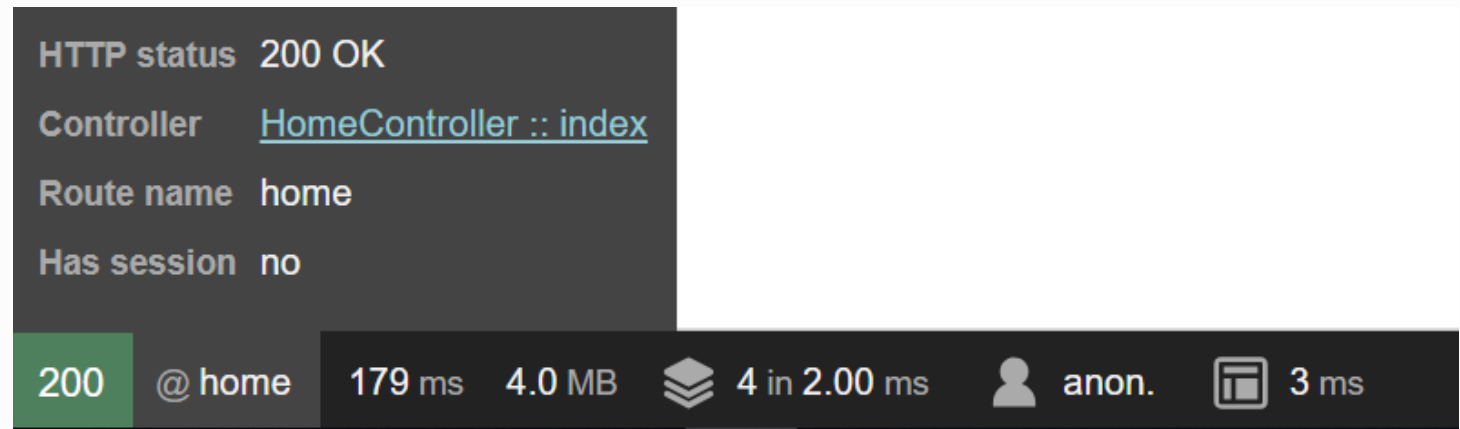
401 : utilisateur non authentifié;

403 : accès refusé;

404: page non trouvée;

500 et 503 : erreur serveur.

Informations sur le Controller (nom), la route et la session. D'autres indications techniques sont également fournies dans la barre d'outils.



Autres informations

- Temps de rendu de la page et temps d'initialisation
- Mémoire (RAM) utilisée (indicateur de pique mémoire)
- Authentification (anonyme, token)
- Informations Twig (Temps de rendu du template, nombre de templates,...)
- Sur la droite, volet proposant les informations sur Symfony et PHP

Le Profiler

The screenshot displays the Symfony Profiler interface. At the top, the Symfony logo and 'Symfony Profiler' text are on the left, and a search bar with 'search on symfony.com' and a 'Search' button is on the right. Below this, a green bar shows the URL 'http://localhost/symfony/theory/public/' and details: 'Method: GET', 'HTTP Status: 200', 'IP: ::1', 'Profiled on: Sat, 15 Sep 2018 08:08:47 +0000', and 'Token: 8f09a7'.

On the left side, there is a sidebar with icons and labels for various profiler sections: 'Request / Response' (selected), 'Performance', 'Validator', 'Forms', 'Exception', 'Logs', 'Events', 'Routing', and 'Cache'. Above these icons are buttons for 'Last 10', 'Latest', and a search icon.

The main content area shows the title 'HomeController :: index' with a link icon. Below it are tabs for 'Request', 'Response' (selected), 'Cookies', 'Session', and 'Flashes'. The 'Response Headers' section is active, displaying a table with the following data:

Header	Value
cache-control	"no-cache, private"
content-type	"text/html; charset=UTF-8"
date	"Sat, 15 Sep 2018 08:08:47 GMT"
x-debug-token	"8f09a7"

Pour accéder au Profiler un simple clic sur une icône de la Debug Toolbar suffit

La fonction dump

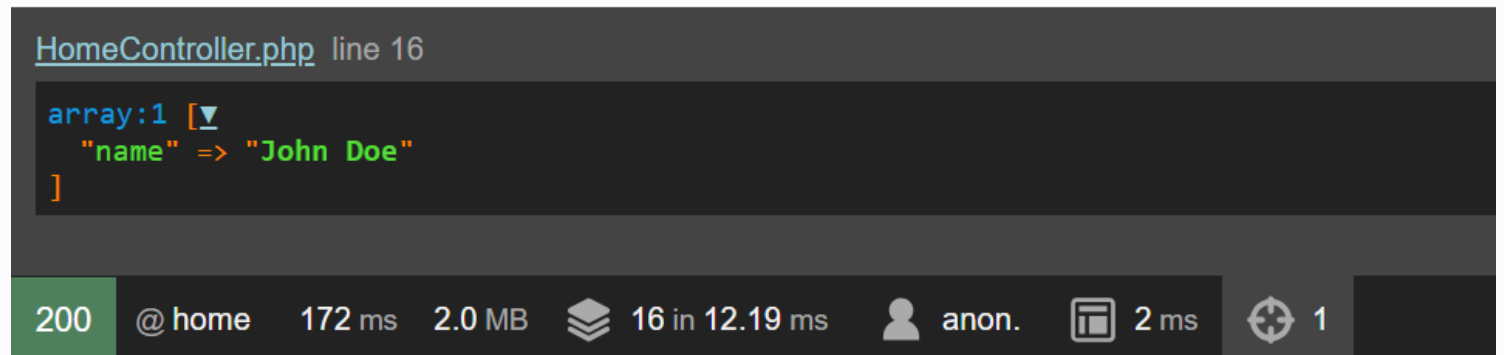
- Nous utilisons régulièrement la fonction `var_dump()` qui affiche les informations structurées d'une variable, y compris son type et sa valeur. Nous la faisons suivre d'une instruction d'arrêt pour afficher le contenu sans exécuter le reste du script.

```
var_dump($var);exit;
```

- Dans symfony, il est possible d'employer une fonction `dump()` qui affichera le contenu de la variable dans la "Debug Toolbar". L'intérêt de cet utilitaire est de ne pas casser le template avec de l'affichage de débogage.

```
dump($var)
```

```
dump($request)
```



The screenshot shows a Symfony Debug Toolbar. The top bar indicates the file `HomeController.php` at line 16. Below it, a dump of an array is shown: `array:1 [▼`, `"name" => "John Doe"`, and `]`. The bottom bar displays various performance metrics: status 200, location @ home, time 172 ms, memory 2.0 MB, 16 in 12.19 ms, user anon., and other icons.

Notions de base supplémentaires

Les routes

Les contrôleurs

Les vues

Les services

Les frameworks CSS

L'utilitaire de lignes de commandes

Configuration d'une route

```
#[Route()]
```

Syntaxe

```
use Symfony\Component\Routing\Attribute\Route;
```

Le Use

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

Accès à la racine (index)

```
#[Route('/products', name: 'app_products')]
```

Ajout d'un niveau supplémentaire

```
#[Route('/products/{cat}', name: 'app_products')]
```

Ajout d'un paramètre: n'importe quoi mais obligatoire

```
#[Route('/products/{cat?}', name: 'app_products')]
```

Le paramètre n'est pas obligatoire

```
#[Route('/products/{cat}/{order?}', name: 'app_products')]
```

Deux paramètres dernier pas obligatoire

Récupération du paramètre

Paramètre obligatoire

```
#[Route('/products/{cat}', name: 'app_products')]
public function products(string $cat): Response
{
    return $this->render('product/products.html.twig',
        [
            'cat' => $cat,
        ]
    );
}
```

Paramètre non obligatoire

```
#[Route('/products/{cat?}', name: 'app_products')]
public function products(?string $cat): Response
{
    return $this->render('product/products.html.twig',
        [
            'cat' => $cat,
        ]
    );
}
```

Récupération du paramètre dans la vue

Le paramètre est envoyé dans la vue sous la forme d'un tableau associatif passé en second argument de la méthode render().

```
return $this->render('index.html.twig', ['cat' => $cat]);
```

Pour afficher la valeur du paramètre dans la vue on encadre le nom du paramètre par des doubles accolades {{ paramName }}. Il s'agit simplement d'interpoler une variable récupérée du code PHP (contrôleur).

```
<h3>Login: {{ cat }}</h3>
```


Les contrôleurs

Le contrôleur contient toute la logique de notre site Internet. Cependant, cela ne veut pas dire qu'il contient beaucoup de code. En fait, il ne fait qu'utiliser **des services**, les modèles et appeler la vue. Finalement, c'est un chef d'orchestre qui se contente de faire la liaison entre tous les composants de l'application.

Son rôle principal est de retourner une réponse. Pour cela, il utilise la classe **Response** qui renvoie directement une réponse sous la forme d'un simple texte, d'un contenu HTML ou Json. Mais aussi la **méthode Render()** de la classe **AbstractController** qui permet de générer du contenu Twig. Tous les deux sont des représentations objet des concepts HTTP.

La plupart du temps, la méthode **render()** contiendra également en paramètre des données (strings, variables, arrays, json,...).

Twig

Les templates vont nous permettre de séparer le code PHP du code HTML. Seulement, pour faire du HTML de présentation, on a toujours besoin d'un peu de code dynamique : faire une boucle pour afficher tous les articles, créer des conditions pour afficher un menu différent pour les utilisateurs authentifiés ou non, etc. Pour faciliter ce code dynamique dans les templates, le moteur de templates Twig offre son pseudo-langage à lui. Ce n'est pas du PHP, mais c'est plus adapté et plus lisible (avantages pour les développeurs Front End).

Twig: Principes de base

Les réalisations les plus fréquentes sont l'affichage de variables et l'exécution d'une instruction. Afficher le nom de l'utilisateur, l'identifiant de l'article... Exécuter une instruction comme une boucle ou un test.

`{{ ... }}` affiche quelque chose: des paramètres, des valeurs de variables...

`{% ... %}` fait quelque chose: des instructions ou fonctions Twig

`{# ... #}` syntaxe des commentaires

Les filtres Twig

- Un filtre agit comme une fonction. Ils sont déjà définis mais vous pouvez créer les vôtres si vous le souhaitez. Pour appliquer un filtre à une variable, il faut séparer le nom de la variable et celui du filtre par un pipe (|). Je vous propose ici de voir quelques filtres utiles.
- Syntaxe: `{{ variable|filter }}` pour illustrer les filtres, je remplace la variable par une chaîne.
- `{{ 'jane doe'|upper }}` // JANE DOE VS lower
- `{{ 'jane doe'|title }}` // Jane Doe
- `{{ 'jane doe'|capitalize }}` // Jane doe
- `{{ ' jane doe '|trim }}` // jane doe
- `{{ 'jane doe.|trim('.') }}` // jane doe

Les filtres (suite)

- `{{ 'HelloWorld'|humanize }}` // Hello world
- `{{ date()|date }}` // July 11, 2020 18:01
- `{{ date()|date('d/m/Y') }}` // 07/11/2020
- `{{ 5.5|round }}` // 6
- `{{ '5.473'|round(1) }}` // 5,5

Couper une chaîne de caractères

- `{{ long text|u.truncate(100, '...', false) }}`
- Erreur: The "u" filter is part of the StringExtension, which is not installed/enabled; try running "**composer require twig/string-extra**" in "post/posts.html.twig".

Le tag for

Pour la création d'une simple boucle la syntaxe est la suivante:

```
{% for i in 0..10 %}
```

```
  {{ i }}<br>
```

```
{% endfor %}
```

```
{% for i in 'A'..'Z' %}
```

```
  {{ i }} -
```

```
{% endfor %}
```

Le tag for (tableau indexé)

Controller

```
/**
 * @Route("/for", name="for")
 */
public function for()
{
    $courses = ['HTML', 'PHP', 'CSS', 'JavaScript'];
    return $this->render(
        'for.html.twig',
        [
            'courses' => $courses
        ]
    );
}
```

Twig

```
{% block body %}
{% for course in courses %}
    <p>{{ course }}</p>
{% endfor %}
{% endblock %}
```

Le tag for (tableau associatif)

```
/**
 * @Route("/forassoc", name="forassoc")
 */
public function forassoc(): Array
{
    $courses = ['HTML' => 100, 'PHP' => 120,
    'CSS' => 60, 'JavaScript' => 80];
    return $this->render(
        'forassoc.html.twig',
        [
            'courses' => $courses
        ]
    );
}
```

```
{% block body %}
{% for course, duration in courses %}
<p>{{ course }} ({{ duration }}
périodes)</p>
{% endfor %}
{% endblock %}
```


Le tag For & variable de boucle

- Lors d'une itération sur des données, vous pouvez utiliser la variable de boucle: `loop`

```
<ul>
  {% for function, name in team %}
    <li>
      ({{ loop.index }}) {{ name }} -
      {{ function }}
    </li>
  {% endfor %}
</ul>
```

- Il existe plusieurs variantes:
 - `loop.length` (nombre d'items)
 - `loop.first` (true si premier)
 - `loop.last` (true si dernier)

Le tag if

```
{% if age >= 40 %}  
<p>Senior Developer</p>  
{% elseif age >= 30 %}  
<p>Experimented Developer</p>  
{% else %}  
<p>Junior Developer</p>  
{% endif %}
```

```
{% if temperature > 18 and temperature < 27 %}  
    <p>It's a nice day for a walk in the park.</p>  
{% endif %}
```

```
{% if not user.subscribed %}  
    <p>You are not subscribed to our mailing list.</p>  
{% endif %}
```

Introduction aux services

- Un service est simplement un objet PHP qui remplit une fonction et peut être utilisé n'importe où dans votre code.
- Cette fonction peut être simple : envoyer des e-mails, vérifier qu'un texte n'est pas un spam, etc. Mais elle peut aussi être bien plus complexe : gérer une base de données (le service Doctrine !), etc.
- Un service est donc un objet PHP qui a pour vocation d'être accessible depuis n'importe où dans votre code. Pour chaque fonctionnalité dont vous aurez besoin dans toute votre application, vous pourrez créer un ou plusieurs services (et donc une ou plusieurs classes et leur configuration). Un service est avant tout une simple classe.

Introduction aux services

- L'avantage de réfléchir sur les services est que cela force à bien séparer chaque fonctionnalité de l'application. Comme chaque service ne remplit qu'une seule et unique fonction, ils sont facilement réutilisables. Et vous pouvez surtout facilement les développer, les tester et les configurer puisqu'ils sont assez indépendants. Cette façon de programmer est connue sous le nom d'architecture orientée services, et n'est pas spécifique à Symfony ni au PHP.
- Pour éviter une surcharge du code métier dans le contrôleur, on doit le déporter et en créer un service.

Introduction aux services

```
namespace App\Service;  
  
class Utils  
{  
    public function clean($string)  
    {  
        return ucfirst(trim($string));  
    }  
}
```

Objectif: créez une fonctionnalité permettant le formatage d'une chaîne de caractères.

1. Créez un dossier "Service" dans le répertoire "src"
2. Créez une classe portant le nom du service "Utils"
3. Ajoutez une méthode toute simple permettant de retirer les espaces et de mettre une première majuscule au param login.

Utilisation du service dans le contrôleur

```
public function index(Utills $clean, $login)
{
    $login = $clean->clean($login);
    return $this->render('home/index.html.twig', ['login' => $login]);
}
```

- Il suffit de passer un paramètre dans la méthode et de le typer avec le nom de la classe.
- Ensuite, on invoque la méthode clean() sur l'objet \$clean et on stocke le résultat dans \$login.
- Symfony s'occupe d'instancier l'objet à votre place. Ce principe se nomme l'injection de dépendances.

PhpDoc Blocks

- Vous l'aurez constaté, PhpStorm vous avertit en soulignant les paramètres de la fonction que vous oubliez quelque chose (Argument PHPDoc Missing).
- Vous devez rajouter en dessous de la route les tags @param et @return dans le DockBlock.

```
/**
 * @route("/{login}", name="home")
 * @param Utils $clean
 * @param string $login
 * @return \Symfony\Component\HttpFoundation\Response
 */
```

Installation d'un framework CSS (1)

A partir de Symfony 7 un nouveau composant "AssetMapper" a été introduit. Nous en avons parlé brièvement pour les images et les CSS. Ce composant a préinstallé un dossier **Assets** dans la racine du projet dans lequel nous allons y mettre tous les actifs: css, js, images, librairies,...

Cette nouveauté permet de versionner les fichiers pour une meilleure gestion du cache du navigateur mais aussi une gestion plus moderne de ces contenus un peu comme le ferai Webpack (Bundler) avec NodeJs.

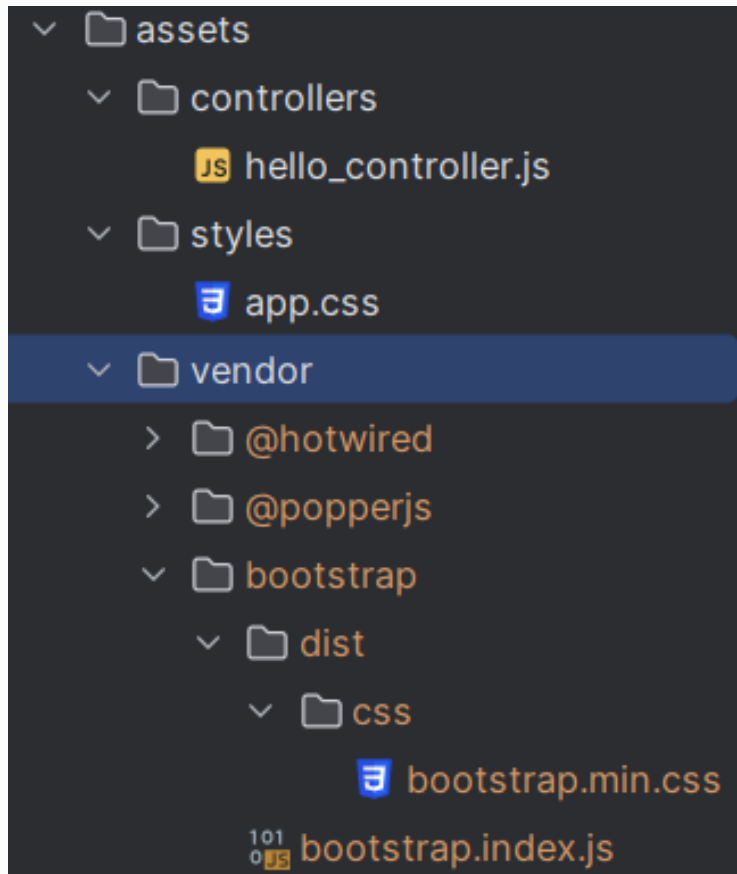
Installation de Bootstrap

Terminal

```
php bin/console importmap:require bootstrap
```


Installation d'un framework CSS (2)

Cela ajoute les fichiers dans le dossier Assets et le bootstrap package dans le fichier importmap.php.



```
],  
'bootstrap' => [  
  'version' => '5.3.3',  
],  
'@popperjs/core' => [  
  'version' => '2.11.8',  
],  
'bootstrap/dist/css/bootstrap.min.css' => [  
  'version' => '5.3.3',  
  'type' => 'css',  
],
```

Installation d'un framework CSS (3)

- Vous devez maintenant modifier le fichier app.js du dossier Assets pour y inclure vos dépendances.

```
import './bootstrap.js';  
  
// Bootstrap css  
import 'bootstrap/dist/css/bootstrap.min.css';  
// Custom css  
import './styles/app.css';  
// Bootstrap JS  
import 'bootstrap';
```

Installation d'un framework CSS (4)

Dans le code source (CTRL + u) de la page des modifications ont été réalisées:

- Liens vers le fichier min.css versionné (cache)

```
<link rel="stylesheet" href="/assets/vendor/bootstrap/dist/css/bootstrap.min-1712f0378f8675ca7cd423d6262fccccf.css">
```

- Des imports pour Bootstrap et PopperJs

```
"imports": {
```

```
  "bootstrap/dist/css/bootstrap.min.css": "data:application/javascript,",
```

```
  "@popperjs/core": "/assets/vendor/@popperjs/core/core.index-ceb5b6c0f9e1d3f6c78ef733facfdcd.js",
```

```
  "bootstrap": "/assets/vendor/bootstrap/bootstrap.index-c0423c99f6075e6b1cef7579b5c00d32.js",
```

```
}
```

- Les preloadings des modules JS nécessaires pour Bootstrap.

Création de liens dans les vues

- Utilisation du helper path() de Twig

```
<a class="nav-link" href="{{ path('about') }}">About Us</a>
```

- Path('name'): indiquez la valeur du name="about" c'est-à-dire le nom de la route créée en annotation pour l'action du contrôleur.
- En principe, le texte du lien devrait correspondre au path de la route.

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

Insertion d'une image dans la vue

- Pour insérer une image dans une vue, vous allez utiliser la fonction `asset()`. Elle prend en paramètre le chemin logique vers votre ressource.

```

```

- Vous utilisez ici le chemin **logique** vers l'image. Il faut juste un dossier **images** dans **assets** ou dans **public**.

Compilation et gestion des assets

1. Pour la production du projet (mise en ligne) vous devrez compiler les assets avec la commande:

```
php bin/console asset-map:compile
```

Symfony va copier l'ensemble des fichiers versionnés dans le dossier public du projet.

2. Par défaut, le dossier assets/**vendor** n'est pas comité avec le reste des fichiers (.gitignore). Pour le réinstaller après le clonage, utilisez la commande:

```
php bin/console importmap:install
```

3. Pour exécuter un audit sur la sécurité des dépendances, utilisez la commande:

```
php bin/console importmap:audit
```

Création d'une application

Etape 1 – le CRUD

- Objectifs et configuration du projet
- Créer et modifier une entité avec Doctrine
- Persister des objets avec les fixtures
- Ajouter des données depuis un formulaire
- Lister les enregistrements
- Afficher un seul enregistrement
- Supprimer un enregistrement
- Mettre à jour un enregistrement

Objectifs

- Nous allons créer une application en plusieurs étapes. Il s'agit de la gestion d'articles (posts) publiés par des membres et concernant le domaine du web.
- Dans un premier temps, nous souhaitons seulement gérer une homepage et les articles. Ceux-ci seront affichés sur la page **posts**
- Nous créerons ensuite une partie admin pour le **CRUD**

Projet de départ

1. Installez symfony dans un nouveau projet: `webposts`

`symfony new --webapp webposts`

2. Configurer le fichier `.env` et la variable d'environnement `DATABASE_URL`. Par défaut le driver de base de données est celui de postgresql, commentez-le (#) et décommentez celui de mysql en adaptant les propriétés.

`DATABASE_URL="mysql://root:@127.0.0.1:3306/webposts?serverVersion=8.3.0&charset=utf8mb4"`

3. Créez la base de données vide "webpost" avec la ligne de commande suivante:

`php bin/console doctrine:database:create`

4. Installez Bootstrap avec le composant `AssetMapper` dans le dossier Assets (`php bin/console importmap:require bootstrap`).
5. Installez également une librairie d'icônes (icofont.com) aussi dans le dossier Assets. Copier les fichiers dans le dossier `styles/icofont` (dossier fonts et fichier icofont.css).
6. Supprimer le contenu du fichier `app.css`
7. Modifier le fichier `app.js` comme ci-dessous.

app.js

```
// Stimulus
import './bootstrap.js';

// Icofont.css
import './styles/icofont/icofont.min.css';
// Bootstrap css
import 'bootstrap/dist/css/bootstrap.min.css';
// Custom css
import './styles/app.css';
// Bootstrap JS
import 'bootstrap';
```

Création du contrôleur et de la vue Home

- Symfony est capable de créer automatiquement la structure d'un contrôleur et de la vue. Ce premier contrôleur aura pour objectif la gestion des articles (CRUD). Il devra porter le même nom que le modèle et aussi que la table de la DB.

`php bin/console make:controller`

- Choose a name for your controller class: `HomeController`
- Symfony vient de créer le contrôleur dans le dossier src et la vue (index.html.twig) dans le dossier templates/home.
- Du code HTML et Twig ont été ajoutés. Supprimer tout ce code sauf l'instruction d'héritage et le bloc de titre. Le bloc body peut rester mais doit être vide.

Contrôleur: HomeController.php

```
namespace App\Controller;

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

class HomeController extends AbstractController
{
    #[Route('/', name: 'app_home')]
    public function index(): Response
    {
        return $this->render('home/index.html.twig', [

        ]);
    }
}
```

Création de la navbar

Pour ne pas inclure la navbar directement dans base.html.twig nous allons utiliser la technique des **partials** et **des includes** en Twig.

- Créez un nouveau dossier **partials** dans templates et nommez-le navbar.html.twig
- Créez un `{% block navbar %}`
- Insérez ou créez le code Bootstrap d'une navbar avec un dropdown.

WebPosts Home Articles Catégories ▼

- Insérer la commande twig '**include**' dans base.html.twig

```
<body>
  {% include 'partials/navbar.html.twig' %}
  {% block body %}{% endblock %}
</body>
```

Création du footer

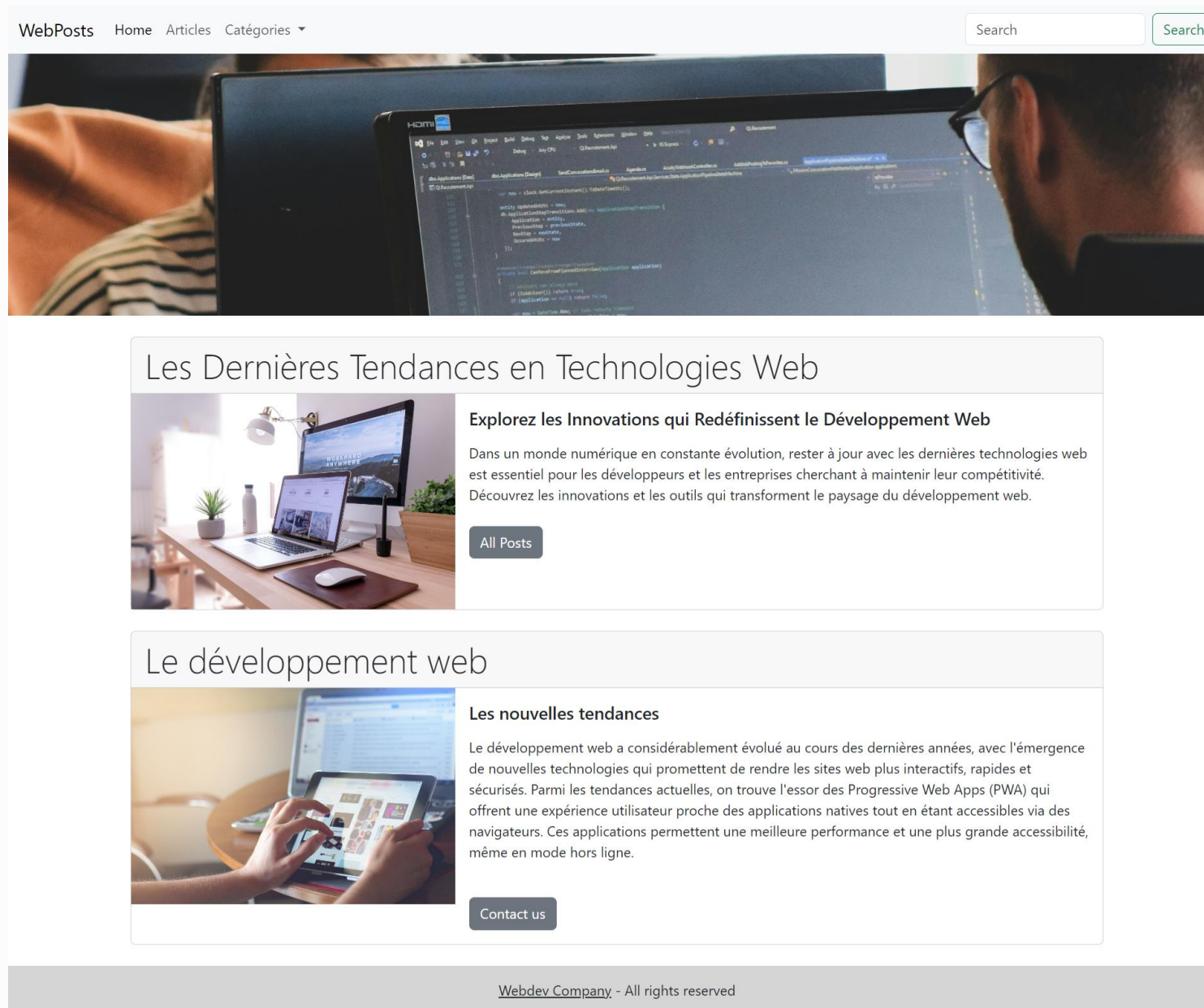
- Utilisez le même principe que pour la navbar. Créez un nouveau fichier dans `partials/footer.html.twig`
- Ajoutez l'include dans `base.html.twig` après le block body.

```
{% block footer %}
    <footer class="col-md-12 mt-4 p-3" style="background-color: #d5d5d5; text-align: center">
        <a href="" class="text-reset btn-link" target="_blank">Webdev Company</a> - All rights reserved
    </footer>
{% endblock %}
```

Contenu de la HomePage

- Créez un contenu (pas en html) à l'aide de ChatGpt destiné à un site proposant des articles sur les technologies Web (titre, sous-titre, blockout et deux paragraphes).
- Ajoutez une bannière.
- Mettez le tout en HTML en utilisant les classes Bootstrap. L'objectif n'est pas d'obtenir le template parfait mais une présentation correcte de vos contenus statiques ou dynamiques. J'ai utilisé ici la classe card de Bootstrap avec une image sur la droite. Des modifications sont à effectuer dans les classes.

Exemple basic de HomePage



Création du PostController et de sa vue.

- Utilisez la même commande que précédemment:

Php/bin/console make:controller PostController

- Le message confirmant les actions apparaît:

created: src/Controller/PostController.php

created: templates/post/index.html.twig

- J'ai juste ajouté un (s) dans la route car le controller doit afficher tous les articles. Ici rien de neuf.

- Modifiez le nom de la vue, toutes celles-ci ne doivent pas porter le nom par default (index.html.twig).

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

Création de la vue

- Effacez ce qui n'est pas nécessaire.
- Préparez une page qui permettra d'afficher dynamiquement (DB) les articles. Comme rien n'est en base de données pour l'instant je n'aurais besoin que de quelques balises html.
- N'oubliez pas d'ajouter les liens de la navbar et des boutons dans la vue homePage, comme ceci:

`href="{{ path('app_posts') }}">Articles`

```
{% block body %}
  <main class="container">
    <section class="posts">
      <div class="row">
        <div class="col-md-12">
          <h2>Liste des articles</h2>
        </div>
      </div>
    </section>
  </main>
{% endblock %}
```

L'ORM Doctrine

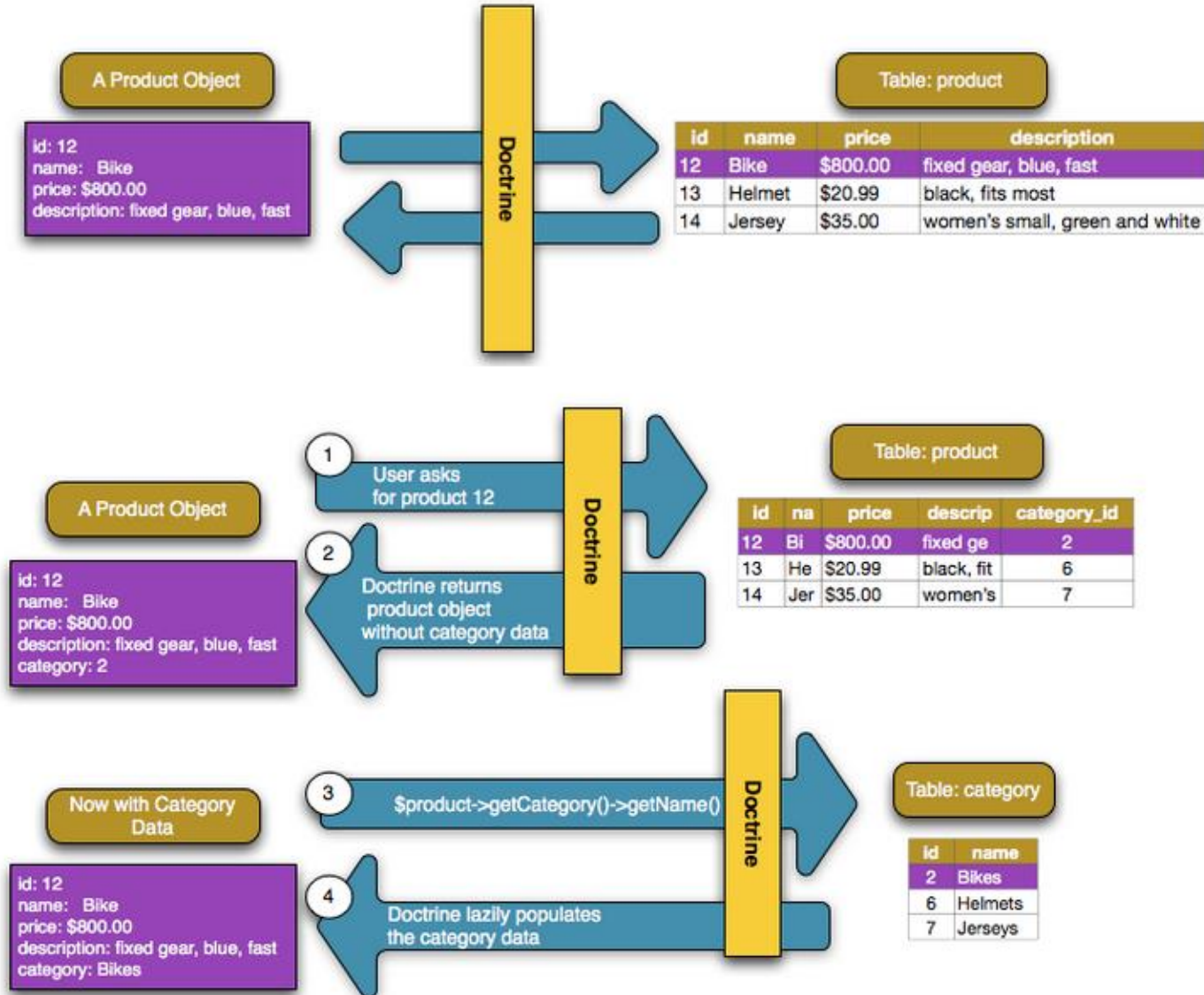
Object Relation Mapper

Doctrine est un **ORM** (couche d'abstraction à la base de données) pour PHP. Il s'agit d'un logiciel libre utilisé par défaut dans Symfony.

Son objectif est de se charger du traitement de vos données (CRUD: Create, Read, Update, Delete) sans manipuler la base de données et sans la création de requêtes SQL.

Les données que vous allez manipuler via la base de données sont des **objets** (objet Category ou objet Post). Pour ce faire, vous devez créer (automatiquement) les entités (entity) qui se chargeront de faire l'interface entre la DB et le traitement en PHP sous la forme d'objets.

Conversion enregistrement => objet



Composants de Doctrine

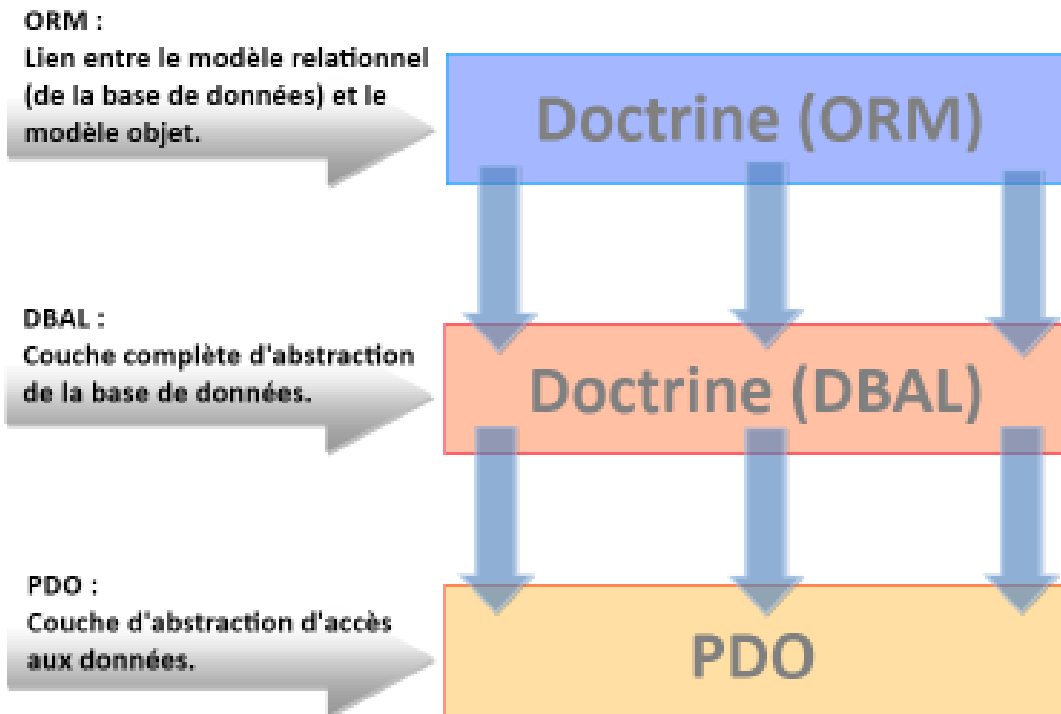
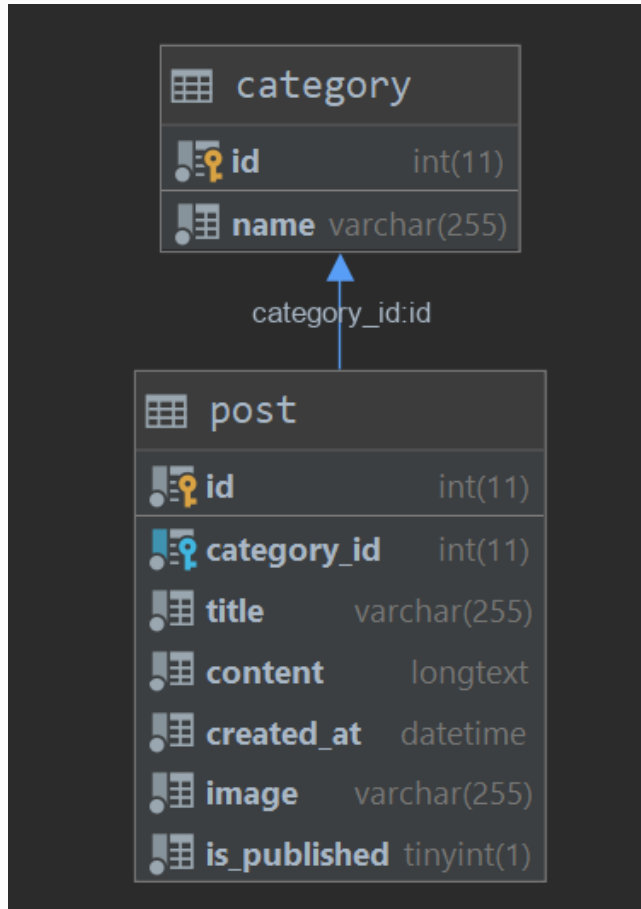


Schéma de départ



Dans un premier temps, nous allons créer une DB contenant deux tables. Ce qui représente deux entités (Post et Category) dans Symfony.

Nous partons sur une relation classique en Symfony: ManyToOne avec comme entité principale les articles (Post).

L'entité Category

Dans le modèle MVC, une entité correspond à un modèle. La base de données (webposts) a déjà été créée précédemment mais elle ne contient aucune table (entités), c'est que nous allons faire maintenant. Pour ça, nous utiliserons une commande intégrée `make:entity` appartenant au bundle `maker` de Symfony

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




- Class name of the entity to create or update: `Category`
- Add the ability to broadcast entity updates using Symfony UX Turbo? (yes/no) [no]: `no`
- L'entité et son repository viennent d'être créés. Vous devez maintenant saisir les champs (attributs) de l'entité. Vous devrez fournir au fur et mesure les propriétés (type, length,...). La clé primaire sera ajoutée automatiquement par Doctrine (id).
- Pour le choix du type de champ, vous pouvez saisir un point d'interrogation pour obtenir la liste complète.

L'entité Category

Une seule propriété est nécessaire pour les catégories.

name

1. New property name (press <return> to stop adding fields): name
2. Field type [string]:
3. Field length [255]:
4. Can this field be null in the database (nullable) (yes/no) [no]:
5. Add another property? Enter the property name (or press <return> to stop adding fields):
6. Terminez l'ajout des champs avec « **Enter** »

	category	
	id	int(11)
	name	varchar(255)


```

namespace App\Entity;
use App\Repository\CategoryRepository;
use Doctrine\ORM\Mapping as ORM;

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

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

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

    public function getName(): ?string
    {
        return $this->name;
    }

    public function setName(string $name): static
    {
        $this->name = $name;
        return $this;
    }
}

```

L'entité Category (classe PHP)

Représentation objet des catégories

La classe a automatiquement été ajoutée dans le dossier **Entity** et contient entre autre:
 La description des différentes propriétés (champs) sous la forme d'**attributs** de type #[ORM\
 Construction des **getters et des setters** nécessaires
 Vous pouvez aussi rajouter ou corriger manuellement une nouvelle propriété et de nouvelles méthodes.

Le repository CategoryRepository (classe PHP)

Méthodes contenant les requêtes pour les catégories

Pour l'instant la classe ne contient qu'un constructeur, deux méthodes commentées que nous pourrons utiliser par la suite. Elle nous permet en premier d'utiliser une série de méthodes magiques de type query sur nos données (find, findAll...). Par la suite nous pourrons y intégrer nos propres requêtes en DQL ou SQL.








L'entité Post

Propriété title

- New property name (press <return> to stop adding fields): title
- Field type [string]:
- Field length [255]:
- Can this field be null in the database (nullable) (yes/no) [no]:

Propriété content

- New field name (press <return> to stop adding fields): content
- Field type [string]: text
- Can this field be null in the database (nullable) (yes/no) [no]:

post	
 id	int(11)
 category_id	int(11)
 title	varchar(255)
 content	longtext
 created_at	datetime
 image	varchar(255)
 is_published	tinyint(1)

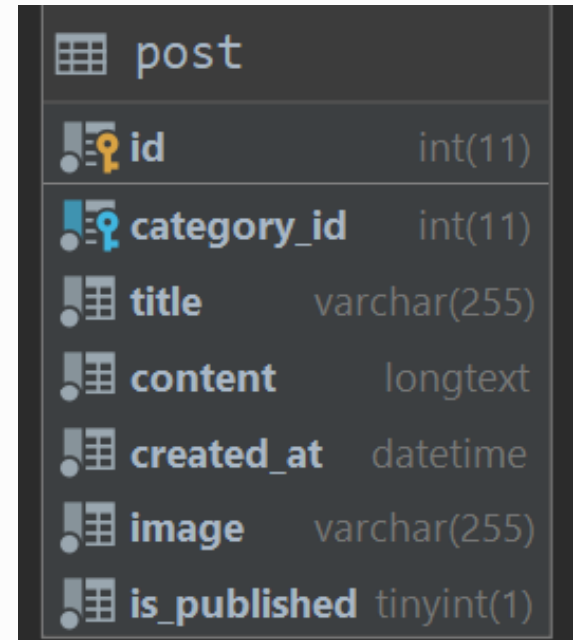
L'entité Post

Propriété createdAt

- New field name (press <return> to stop adding fields): created
- Field type (enter ? to see all types) [datetime_immutable]:
- Can this field be null in the database (nullable) (yes/no) [no]:

Propriété image








- New field name (press <return> to stop adding fields): image
- Field type [string]:
- Field length [255]: 255
- Can this field be null in the database (nullable) (yes/no) [no]: no



post	
id	int(11)
category_id	int(11)
title	varchar(255)
content	longtext
created_at	datetime
image	varchar(255)
is_published	tinyint(1)

L'entité Post

- **Propriété isPublished**
 - Field type (enter ? to see all types) [boolean]:
 - Can this field be null in the database (nullable) (yes/no) [no]:

post	
 id	int(11)
 category_id	int(11)
 title	varchar(255)
 content	longtext
 created_at	datetime
 image	varchar(255)
 is_published	tinyint(1)

Ajouter une relation entre les posts et les catégories

Les deux entités ont été créées mais il n'existe aucune relation entre les deux. Pourtant, un article doit avoir une catégorie. Nous aurions pu le faire directement lors de la création de l'entité **Post** mais nous allons le faire après cela nous permettra de voir comment on modifie une entité après coup.

symfony console make:entity Post

C'est la même commande que pour la création mais comme l'entité existe Symfony nous propose d'ajouter une ou plusieurs propriétés.

Ajouter une relation entre les posts et les catégories

1. New property name (press <return> to stop adding fields): `category`
2. Field type (enter ? to see all types) [string]: `relation`
3. What class should this entity be related to?: `Category`
4. Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]: `ManyToOne`
5. Is the Post.category property allowed to be null (nullable)? (yes/no) [yes]: `no`
6. Do you want to add a new property to Category so that you can access/update Post objects from it - e.g. `$category->getPosts()`? (yes/no) [yes]: `yes`
7. A new property will also be added to the Category class so that you can access the related Post objects from it.
8. New field name inside Category [`posts`]:
9. Do you want to automatically delete orphaned App\Entity\Post objects (orphanRemoval)? (yes/no) [no]: `no`
10. press <`return`> to stop adding fields

L'entité Post

Dans la classe `Post` la propriété `category` a été ajoutée. Elle correspond à la clé étrangère qui permettra d'enregistrer l'id d'une catégorie. Vous remarquerez que l'annotation est différente puisqu'il s'agit d'une propriété relationnelle. Il s'agit ici d'une double annotation précisant la relation `ManyToOne` avec l'entité `Category` et l'obligation d'avoir une valeur correspondante (l'id de la catégorie).

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

Un getter et un setter ont été aussi ajoutés pour accéder à la catégorie à partir de l'objet `Product`.

```
public function getCategory(): ?category  
{  
    return $this->category;  
}  
  
public function setCategory(?category $category): static  
{  
    $this->category = $category;  
  
    return $this;  
}
```

L'entité Category

La propriété `posts` a été ajoutée avec la relation `OneToMany` vers l'entité Post (targetEntity). Cela permettra d'accéder aux articles à partir d'une catégorie. Le type de la propriété Collection est un type qui ressemble et agit presque exactement comme un tableau, mais avec une flexibilité supplémentaire. Ce type est automatiquement affecté dans le contrôleur.

```
#[ORM\OneToMany(mappedBy: 'category', targetEntity: Post::class)]  
private Collection $posts;
```

```
public function __construct()  
{  
    $this->posts = new ArrayCollection();  
}
```

Le getter a été créé ainsi que deux méthodes supplémentaires (addPost et removePost). Nous pourrions ainsi accéder à tous les posts d'une catégorie:

```
$category->getPosts();
```


La migration vers la base de données

- Une fois les entités ajoutées, vous allez maintenant les migrer vers la base de données. Il s'agit d'une double procédure:
 - Création du fichier de migration contenant le code sql permettant la création des tables:
 - `php bin/console make:migration`
 - Migration vers la base de données en exécutant le code SQL généré par Symfony dans le fichier de migration:
 - `php bin/console doctrine:migrations:migrate`

Fichier de migration

Celui-ci se trouve dans le dossier migrations et porte un numéro de version commençant par l'année, le mois et le jour.

Il contient deux méthodes:

1. une méthode **up** permettant la création des deux tables (create table) et l'ajout de la clé étrangère (alter table)
2. Une méthode **down** permettant l'annulation de la méthode up (drop table)

Vous ne pouvez rien modifier car cela ne correspondrait plus aux informations des entités.

Modifier une entité

Pour ajouter une propriété vous avez deux possibilités:

- Repasser par la commande `make:entity` vous permettra de rajouter la propriété manquante de manière rapide.
- Ajoutez manuellement la propriété et l'annotation correspondante dans le fichier des entités. Ne vous occupez pas du getter et du setter, ils seront ajoutés automatiquement lors de la régénération de l'entité. Cette méthode vous permet également de modifier ou de corriger les noms et les propriétés ainsi que d'en supprimer une.

Modifier une entité

- En mode console saisissez (utilisez cette étape uniquement si vous avez modifié l'entité à la main). Si vous avez supprimé une propriété (champ) cela n'est pas nécessaire:
 - `php bin/console make:entity --regenerate`
 - Enter a class or namespace to regenerate [App\Entity]: <<enter>>
 - -> updated: src/Entity/Post.php
- Créez un nouveau fichier de migration:
 - `php bin/console make:migration`
- Migrez la modification (nouveau champ) en base de données.
 - `php bin/console doctrine:migrations:migrate`
 - Confirmez la migration.

Autres commandes doctrine

- `doctrine:mapping:import :`

Permet de créer les entités à partir d'une base de données existante. C'est une technique appelée: "reverse engineering".

- `doctrine:database:create`

Permet de créer et configurer la base de données.

- `doctrine:database:drop --force`

Permet d'effacer la base de données configurée dans le projet

- `doctrine:migrations:execute --down 20190925110032`

ou

- `doctrine:migrations:migrate prev`

ou

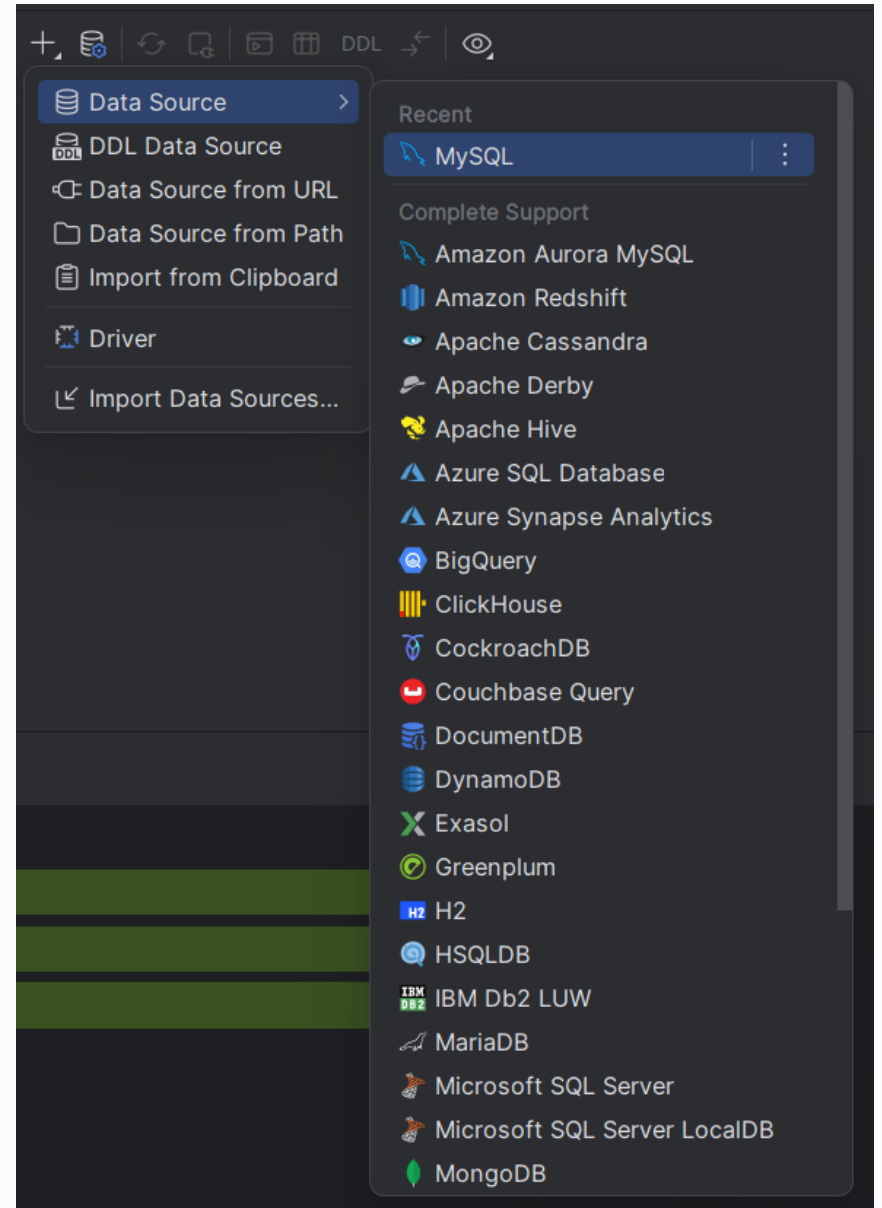
- `doctrine:schema:drop --force`

Permet d'annuler la migration précédente, une migration précise ou n'importe laquelle

- ...

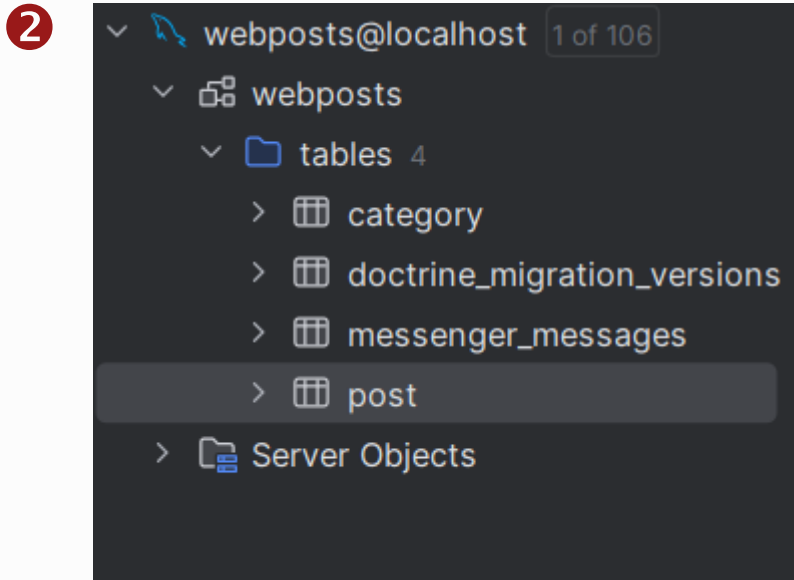
Configurez votre DB dans Symfony

- Dans les icones affichées à droite de votre éditeur, cliquez sur DB.
- Cliquez sur le bouton + (en haut), Data Source et le type de votre DB MySQL.
- Dans la nouvelle fenêtre remplissez certaines données manquantes. Voir la copie ci-dessous

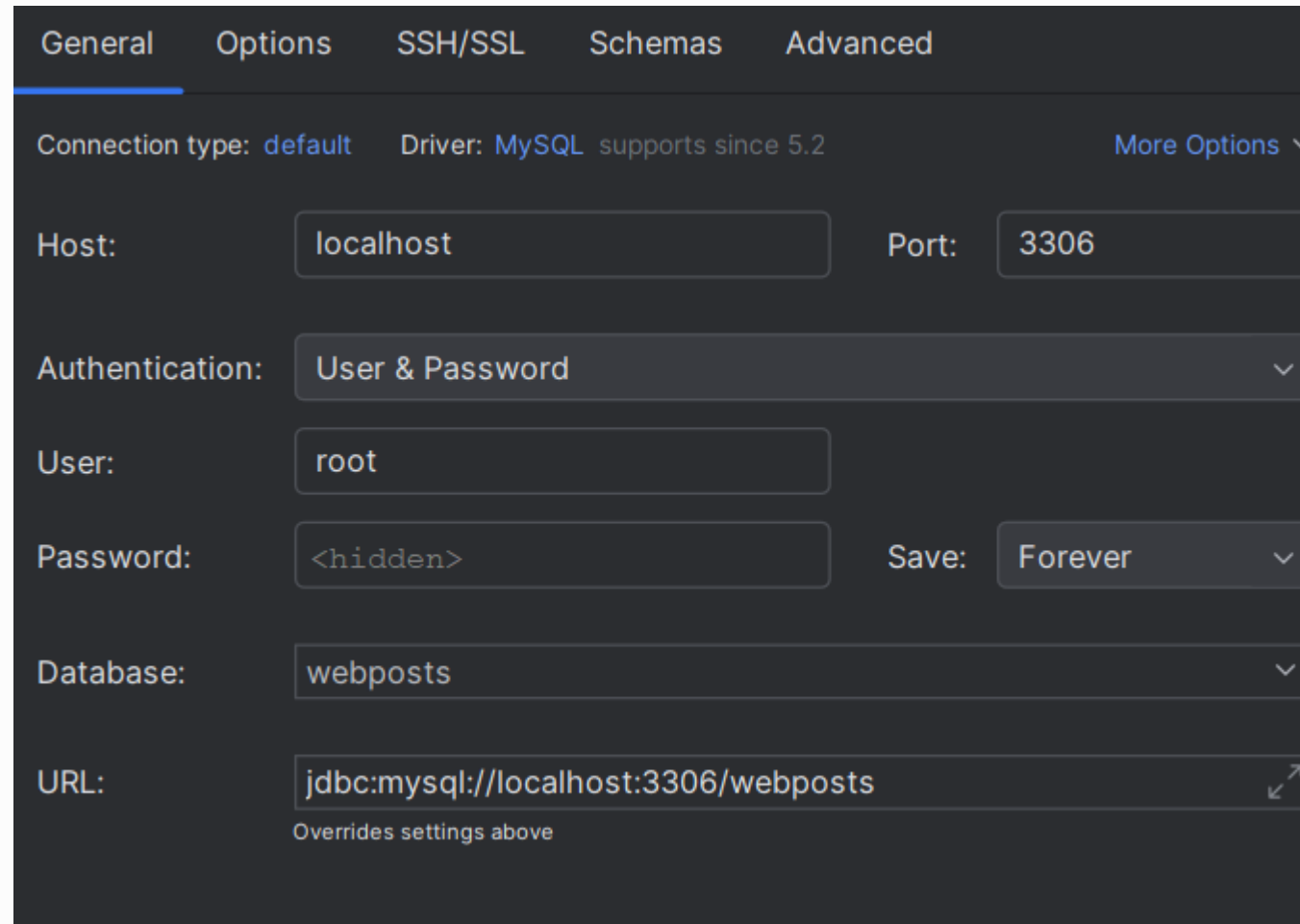


Configurez votre DB dans Symfony

- Maintenant dans le volet droit de votre éditeur vous pouvez visualiser la base de données webposts sur le localhost. Développez et vous afficherez l'ensemble des tables de la DB.



1



Les relations entre les entités

Notions de base

Introduction

- L'objectif de cette partie est de comprendre comment relier les entités entre elles comme il est possible de le faire avec Mysql et SQL.
- Il existe plusieurs manières d'établir des relations entre les entités en fonction du schéma relationnel que vous souhaitez représenter:

OneToOne

OneToMany

ManyToMany

- Avant de voir en détail les relations, il faut comprendre comment elles fonctionnent.

Introduction – la notion d'entité propriétaire

- Dans une relation entre deux entités, il y a toujours une entité dite propriétaire. **L'entité propriétaire** est celle qui contient **la référence à l'autre entité**. On parle ici de clé étrangère ou de «foreign key» que l'on représente par une propriété correspondante à la clé primaire de l'autre entité.
- Prenons l'exemple d'un article pouvant avoir plusieurs commentaires. En SQL, pour créer une relation entre ces deux tables, vous allez mettre une colonne `post_id` dans la table `comment`. La table `comment` est donc propriétaire de la relation, car c'est elle qui contient la colonne de liaison `post_id`.

Introduction – la notion d'unidirectionnalité et de bidirectionnalité

- Une relation entre deux entités avec Doctrine peut être à sens unique ou à double sens. La relation que nous avons créée entre Post et Category est **bidirectionnelle**. C'est-à-dire que l'on peut obtenir la catégorie à partir d'un article mais également tous les articles correspondants aux catégories.
- Cette bidirectionnalité est indiquée au niveau de la classe Category avec la propriété **mappedBy** et la relation inversée: OneToMany. Nous avons choisi cette option lors de la création de l'entité Post.

```
/**
 * @ORM\OneToMany(targetEntity=Post::class, mappedBy="category")
 */
private $posts;
```

Et dans l'entité Post, la bidirectionnalité est indiquée dans l'annotation de l'attribut category avec la propriété **inversedBy**.

```
/**
 * @ORM\ManyToOne(targetEntity=Category::class, inversedBy="posts")
 * @ORM\JoinColumn(nullable=false)
 */
private $category;
```

Persister des objets à l'aide des fixtures

Avec le services doctrine-fixtures

Avec le bundle Faker

Doctrine-fixtures-bundle

- Il s'agit d'un composant permettant de créer des jeux de données afin de tester votre application. Il n'est pas installé par défaut et doit l'être via la commande:

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

- Un dossier `DataFixtures` a été créé dans votre application (src). C'est dans ce répertoire que vous ajouterez les classes permettant de générer les données fictives. Pour ajouter de nouvelles classes de fixtures en CLI, utilisez la commande `php bin/console make:fixture`
- Renommer le fichier `AppFixtures` en `CategoryFixtures`.
- Une fois les Fixtures réalisées, il faut les envoyer en base de données avec la commande: `php bin/console doctrine:fixtures:load`

CategoryFixtures

```
class CategoryFixtures extends Fixture
{
    ① private array $categories = ['PHP 8', 'IA', 'Symfony', 'Laravel', 'Security', 'Angular',
    JavaScript', 'Bootstrap'];

    ② public function load(ObjectManager $manager)
    {
        ③ foreach($this->categories as $category) {
            ④ $cat = new Category();
            ⑤ $cat->setName($category);
            ⑥ $manager->persist($cat);
        }
        ⑦ $manager->flush();
    }
}
```

Résumé du processus:

Comme propriété de la classe, on utilise un tableau contenant les catégories.

Dans la boucle, on instancie l'entité Category et on se sert du setter() pour initialiser la première donnée du tableau.

On persiste l'objet à l'aide du Manager de Doctrine (opération utilisée uniquement lors d'une insertion).

On enregistre les objets en DB avec la méthode flush().

PostFixtures avec la librairie Faker

- Il est possible d'écrire toutes vos fixtures sur le même fichier ou de créer un fichier séparé par entité. Nous allons adapter cette dernière stratégie.
- Pour les articles, nous allons utiliser une librairie supplémentaire qui nous permettra de générer facilement des contenus, des titres, des dates...
- Il s'agit de la librairie **Faker** qui permet de créer des jeux de fausses données pour de nombreuses catégories. Elle permettra de persister en base de données des milliers d'objets pour remplir temporairement des tables de produits, des personnes, d'associations, d'adresses...

PostFixtures

1. Création d'un nouveau script de fixtures:

```
php bin/console make:fixtures PostFixtures
```

2. Installation de la librairie Faker

- `composer require fakerphp/faker`

Ou

- `composer require fakerphp/faker --with-all-dependencies -W`

3. Création du script de fixtures

trois notions supplémentaires sont à considérer:

- 3.1 L'utilisation de Faker pour les données.
- 3.2 La récupération des catégories de l'entité Category (pour la clé étrangère)
- 3.3 L'utilisation d'une interface (DependentFixtureInterface) pour déterminer l'ordre de chargement des fichiers de fixtures. En premier, les catégories et ensuite les articles pour respecter l'intégrité référentielle.

PostFixtures

Etapas dans la méthode `load()` de la classe:

1. Instancier le générateur Faker (classe statique).
2. Récupérer les catégories sous la forme d'un tableau d'objets.
3. Créer la boucle pour générer les fausses données.
4. Utiliser les méthodes de Faker pour la création des données (voir la documentation de faker).
5. Générer aléatoirement une clé de tableau sur les catégories (`category_id`).

```

use App\Entity\Category;
use App\Entity\Post;
use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Persistence\ObjectManager;
use Faker\Factory;

class PostFixtures extends Fixture
{
    public function load(ObjectManager $manager): void
    {
        $faker = Factory::create();
        $categories = $manager->getRepository(Category::class)->findAll();
        for($i = 1; $i <= 30; $i++){
            $post = new Post();
            $post->setTitle($faker->words($faker->numberBetween(3,5), true))
                ->setContent($faker->paragraphs(3, true))
                ->setCreatedAt('\DateTimeImmutable::createFromMutable($faker->dateTimeBetween('-30 days', 'now'))))
                ->setImage($i.'.png')
                ->setPublished($faker->boolean(90))
                ->setCategory($categories[$faker->numberBetween(0,count($categories) -1)]);
            $manager->persist($post);
        }
        $manager->flush();
    }
}

```

Comme le type de la propriété createdAt (entité) est un objet de type DateTimeImmutable, vous devez utiliser la méthode statique createFromMutable de la classe DateTimeImmutable de php. Faker renvoie un objet de type DateTime.

Le backSlash devant la classe indique qu'elle fait partie du domaine PHP et pas de Symfony. Nous pouvons utiliser un use à la place du symbole.

PostsFixtures

- Il nous reste maintenant à utiliser l'interface `DependentFixtureInterface` pour définir l'ordre de chargement des fixtures. Il faut impérativement que Symfony envoie en premier les catégories.
- Pour rappel, lors de l'utilisation d'une interface vous devez implémenter ses méthodes. Ici il y en a une seule: `getDependencies()`
- Cette méthode retourne un tableau contenant de manière ordonnée les fixtures à charger préalablement

```
use Doctrine\Common\DataFixtures\DependentFixtureInterface;
```

```
class PostFixtures extends Fixture implements DependentFixtureInterface
{
    public function load(ObjectManager $manager): void
    {
        // code de la méthode
    }
    public function getDependencies(): array
    {
        return [
            CategoryFixtures::class
        ];
    }
}
```

Ajoutez l'implémentation de l'interface.

Modifiez la méthode `getDependencies()` pour qu'elle charge en premier les catégories. Elle retourne un tableau avec l'ordre des fixtures prioritaires. Pour l'instant une seule.

Par défaut Symfony charge les fixtures dans l'ordre alphabétique mais dans la plupart du temps cela ne respecte pas l'intégrité référentielle.

Commande:

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

Entity Post

1. Ajoutez une nouvelle propriété dans l'entité **Post**

- Slug
- String
- Length: 255

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

2. Commande: `php bin/console make:migration`

3. Commande: `php bin/console d:m:m`

PostFixtures

1. Ajoutez dans la classe le constructeur permettant d'injecter la SluggerInterface.

```
public function __construct(private readonly SluggerInterface  
$slugger) {  
}
```

2. Utilisez la méthode `slug()` sur l'objet slugger.

```
->setSlug($this->slugger->slug($post->getTitle()))
```

3. Commande: `php bin/console d:f:l`

Réalisez la même procédure pour **les catégories**

PostController

Modifiez la méthode `post()` permettant d'afficher un article par son slug et non plus par son id. Le `ParamConverter` ne peut plus être utilisé ici, il faut passer par le `Répository`.

```
#[Route('/post/{slug}', name: 'app_post')]  
public function post($slug, PostRepository $repository): Response  
{  
    $post = $repository->findOneBy(['slug' => $slug]);  
    return $this->render('post/post.html.twig', [  
        'post' => $post,  
    ]);  
}
```

La vue posts.html.twig

Changez le lien sur le bouton "Lire la suite" et passez le slug à la place de l'id

```
<li class="list-group-item"><a href="{{ path('app_post',  
{slug:post.slug}) }}" class="btn btn-outline-dark">Lire la  
suite</a></li>
```

Lister les données

Introduction

Le contrôleur

La vue

Introduction

- L'une des principales fonctions de la couche Modèle dans une application MVC, c'est la récupération des données. Récupérer des données n'est pas toujours évident, surtout lorsqu'on veut récupérer seulement certaines données, les classer selon des critères, etc. Tout cela se fait grâce aux repositories.
- Un repository centralise tout ce qui touche à la récupération de vos entités. Concrètement, cela veut dire que vous ne devez pas faire la moindre requête SQL ailleurs que dans un repository, c'est la règle. On va donc y construire des méthodes pour récupérer une entité par son id, pour récupérer une liste d'entités suivant un critère spécifique, etc. Bref, à chaque fois que vous devez récupérer des entités dans votre base de données, vous utiliserez le repository de l'entité correspondante.
- Le repository contient déjà quelques méthodes (findAll, findBy, find...) qu'il hérite de l'EntityRepository mais vous pouvez construire vos propres requêtes DQL à l'aide de la classe QueryBuilder. Dans un premier temps nous utiliserons les méthodes de sélection existantes.

Introduction

- Cela permet de bien organiser son code. Bien sûr, cela n'empêche pas qu'un repository utilise plusieurs entités, dans le cas d'une jointure par exemple.
- Vos repositories héritent de la classe Doctrine\ORM\EntityRepository grâce à l'héritage de la classe ServiceEntityRepository, qui propose déjà quelques méthodes très utiles pour récupérer des entités. Nous n'écrirons rien pour le moment dans la classe Repository.

Le PostController

```
use App\Repository\PostRepository;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Attribute\Route;

class PostController extends AbstractController
{
    /**
     * @param PostRepository $repository
     * @return Response
     */
    #[Route('/posts', name: 'app_posts')]
    public function posts(PostRepository $repository): Response
    {
        $posts = $repository->findAll();
        return $this->render('post/posts.html.twig', [
            'posts' => $posts,
        ]);
    }
}
```

- Le principe est simple, dans la méthode posts() nous allons utiliser l'**injection de dépendances**. On ajoute la classe PostRepository qui permet de réaliser des requêtes dans la méthode. Ensuite une variable (objet) qui permet d'utiliser les méthodes de la classe sans instantiation car elle se fait via l'injection de dépendance.
- Après, on invoque la méthode **findAll()** sur l'objet **\$repository** et on enregistre toutes les données dans **\$posts**.
- Enfin, on retourne une vue avec les données sous la forme d'un tableau associatif.
- Dans la vue, c'est la clé **posts** qui sera utilisée pour l'affichage des données

Lister les enregistrements dans la vue

Dans la vue, vous devez ajouter une boucle twig pour afficher l'ensemble des enregistrements envoyés par le contrôleur.

L'affichage des données est simple, la variable post suivie d'un point et de l'attribut de l'entité entre accolades: {{ post.title }}

```
<main class="container">
  <section id="posts">
    <div class="row">
      <div class="col-md-10 offset-md-1">
        <h2>Liste des articles</h2>
        {% for post in posts %}
          <p>{{ post.title }} ( {{ post.createdAt|date('d/m/Y')}})</p>
        {% endfor %}
      </div>
    </div>
  </section>
</main>
```

Pour éviter une erreur de type la date,
doit être manipulée avec le filtre
Twig: date(format).

Refactoring du contrôleur.

- La méthode magique findAll() récupère la totalité des articles. Hors nous souhaitons seulement afficher ceux dont le champ (attribut) isPublished est à true. De plus, les données ne sont pas triées.
- Pour y remédier, nous allons utiliser une autre des méthodes magiques: findBy() qui prend en paramètre un tableau associatif permettant de définir des critères.

```
$posts = $repository->findBy(  
    ['isPublished' => true],  
    ['createdAt'    => 'ASC']  
);
```


Syntaxe

```
$repository->findBy(  
    array $criteria,  
    array $orderBy = null,  
    $limit = null,  
    $offset = null  
);
```

La vue finalisée

Tout nos articles

Voluptatem et deserunt.



Symfony

Sunt magni et sapiente. Consequatur explicabo commodi occaecati fugiat. Et aut quibusdam officiis et....

17/10/2024

Kaylie Gottlieb

Lire la suite

Voluptatibus eius laboriosam.



Symfony


Iusto rerum rem vel reiciendis porro et. Et mollitia et earum id placeat. Quidem consequuntur expedita...

15/10/2024

Alysson Muller

Lire la suite

Rerum libero dolorum.



Laravel


Qui libero ipsa recusandae neque. Blanditiis molestias eos perferendis quo. Non dicta exercitationem...

15/10/2024

Aliah Fritsch

Lire la suite

Pariatur autem et.



Symfony

Deserunt quis minus laudantium et qui enim. Amet ratione repellat saepe laborum accusantium consequuntur....

15/10/2024

Nathaniel Bogan

Lire la suite

Voluptas distinctio iure.



Angular

Facilis voluptatem asperiores omnis accusamus repudiandae. Velit quod nihil et quod delectus eos. Officia...

15/10/2024

Geraldine Rowe

Lire la suite

Nihil corporis possimus.



PHP 8

Est nesciunt architecto facere aliquid. Possimus explicabo ex quidem voluptas. Fugit autem vero repellendus...

15/10/2024

Dorris Kutch

Lire la suite

Refactoring de la vue

```
<table class="table table-hover">
  <thead>
    <tr>
      <th>Image</th>
      <th>Titre</th>
      <th>Catégorie</th>
      <th>Publication</th>
    </tr>
  </thead>
  <tbody>
    {% for post in posts %}
      <tr>
        <td></td>
        <td>{{ post.title|capitalize }}</td>
        <td>{{ post.category.name }}</td>
        <td>{{ post.createAt|date('d/m/Y') }}</td>
      </tr>
    {% endfor %}
  </tbody>
</table>
```

- Utilisation de plusieurs contrôleurs dans une même vue. Le cas du menu déroulant dans la navbar.

Afficher un seul article

Contrôleur

Créer la vue avec un seul article

Créer le lien dans la vue avec tous les articles

Le contrôleur – PostController

Comme il s'agit toujours d'articles nous travaillons sur le même contrôleur

1. Ajoutez une nouvelle méthode: post (au singulier)
2. Créez la route avec un paramètre (id)
3. Utilisez la méthode find(\$id) et non findAll
4. Renvoyez vers une nouvelle vue: post.html.twig

```
#[Route('/post/{id}', name: 'app_post')]
public function post(PostRepository $repository, int $id): Response
{
    $post = $repository->find($id);
    // dd($post);
    return $this->render('post/post.html.twig', [
        'post' => $post
    ]);
}
```


Le ParamConverter

- Il s'agit de transformer automatiquement un paramètre de route, comme {id} par exemple, en un objet, une entité \$post
- Le **ParamConverter** va nous convertir nos paramètres directement en entités Doctrine. L'idée est la suivante : dans la méthode, au lieu de récupérer le paramètre de route {id} sous forme de variable \$id, on va récupérer directement une instantiation de entité Post sous la forme d'une variable \$post, qui correspond à l'article portant l'id \$id.
- Et un bonus en prime : on veut également que, s'il n'existe pas d'article portant l'id \$id dans la base de données, alors une exception 404 soit levée.

Refactoring de la méthode

```
#[Route('/post/{id}', name: 'app_post')]
public function post(Post $post): Response
{
    return $this->render('post/post.html.twig', [
        'post' => $post,
    ]);
}
```

On passe l'entité Post en argument de la méthode et Doctrine nous retourne l'objet correspondant à l'id. On ne va plus chercher l'annonce manuellement en passant par le Repository et la méthode `find()` c'est Symfony qui le fait automatiquement.

Le use "use App\Entity\Post" a été ajouté dans la liste.

Lecture d'un article à partir de la liste (posts.html.twig)

- Utilisation du Helper `path()` pour générer un lien vers la vue avec en paramètre id de l'article.
- Le lien correspond au name de la route de la méthode post.
- La commande dans le path permet de passer le paramètre de l'article en court.

```
<td>
  <a href="{{ path('app_post', { id:post.id }) }}"
    class="link-table">{{ post.title | capitalize }}</a>
</td>
```

La vue detail.html.twig



Voluptatem et deserunt.

Symfony

Sunt magni et sapiente. Consequatur explicabo commodi occaecati fugiat. Et aut quibusdam officiis et eos. Praesentium non adipisci ut quia illo.

Laborum optio est iure. Incidunt blanditiis tempora eos reiciendis ut consectetur aut quisquam. Ipsum dolores et deleniti sit earum aut incidunt. Totam necessitatibus qui ad provident magnam qui.

Eius ab impedit ipsa debitis reiciendis sit. Sint excepturi odio omnis quia odit. Praesentium odit labore repellendus laboriosam et iusto facilis sequi. Alias porro eaque dolorem dolores necessitatibus quia ad ad.

Créé par Kaylie Gottlieb

Ajouté le 17/10/2024

[Retour aux articles](#)

Méthodes de type query

- **findAll()**: retourne toutes les entités de la base de données sous la forme d'un tableau.
- **find(\$id)**: retourne l'entité correspondant à l'id
- **findBy()**: permet de passer des paramètres afin d'organiser vos données. Il s'agit tout simplement comme pour une requête SQL d'ajouter des tris, des conditions, des limites.
- **findOneBy()**: fonctionne sur le même principe que la méthode `findBy()`, sauf qu'elle ne retourne qu'une seule entité. Les arguments `orderBy`, `limit` et `offset` n'existent donc pas.
- **findByX(\$valeur)**: permet de remplacer « X » par le nom d'une propriété de votre entité: `findByCategory("php")`. Cette méthode fonctionne comme si vous utilisiez `findBy()` avec un seul critère, celui du nom de la méthode.
- **findOneByX(\$valeur)**: identique à la précédente mais pour une seule entité.

Administration

Ajouter des données

Supprimer des données

Modifier des données

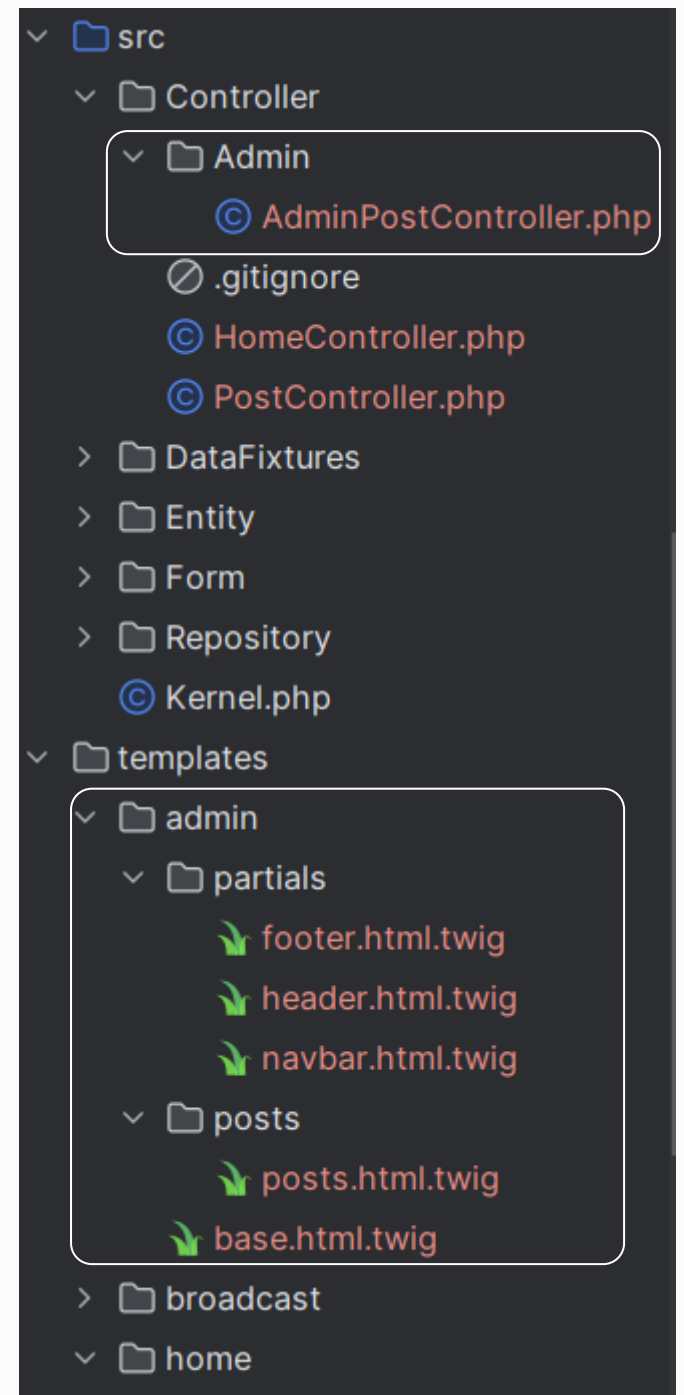
Ajouter des données

Introduction

La classe PostType (formulaire)

Principe

- Pour l'organisation des fichiers, nous allons créer un dossier **Admin** dans le dossier **Controller** et aussi un dossier **admin** dans les templates. La gestion et la présentation de l'admin est différente de la partie publique. Nous allons avoir besoin d'une série de fichiers pour son affichage. Créez, déplacez et renommez dossiers et fichiers comme la figure à droite.
- Modifiez le Controller pour qu'il retourne sur la nouvelle vue sans paramètres pour l'instant.
- Modifiez base.html.twig et ajoutez de nouveaux contenus pour le header, la navbar et éventuellement le footer. Voir diapos suivantes.



Base.html.twig (admin)

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8" lang="fr">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>{% block title %}Administration{% endblock %}</title>

  {% block stylesheets %}
  {% endblock %}

  {% block javascripts %}
    {% block importmap %}{{ importmap('app') }}{% endblock %}
  {% endblock %}
</head>
<body>

  {% include 'admin/partials/navbar.html.twig' %}
  {% block body %}{% endblock %}
  {% include 'admin/partials/footer.html.twig' %}
</body>
</html>
```

Navbar.html.twig (admin)

```
<header>
  <nav class="navbar navbar-expand-lg bg-body-tertiary">
    <div class="container-fluid">
      <a class="navbar-brand" href="#">Administration</a>
      <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarSupportedContent" aria-
controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
      </button>
      <div class="collapse navbar-collapse" id="navbarSupportedContent">
        <ul class="navbar-nav me-auto mb-2 mb-lg-0">
          <li class="nav-item">
            <a class="nav-link active" aria-current="page" href="{{ path('app_admin_posts') }}">Articles</a>
          </li>
          <li class="nav-item">
            <a class="nav-link" href="">Catégories</a>
          </li>
          <li class="nav-item">
            <a class="nav-link" href="">Utilisateurs</a>
          </li>
        </ul>
      </div>
    </div>
  </nav>
</header>
```

Posts.html.twig (admin)

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

{% block title %}Administration{% endblock %}

{% block body %}
    <section class="row">
        <div class="col-md-9 offset-1">
            <h2 class="display-5 my-4">Gestion des articles</h2>
            <a href="" class="btn btn-outline-dark">Nouvel article</a>
        </div>
    </section>
{% endblock %}
```

Création du formulaire

Comme pour les contrôleurs, Symfony propose de créer une classe de type «PostType» dans un dossier «Form». Vous obtiendrez ainsi une classe prête à l'emploi qui vous permettra d'utiliser le composant FormBuilder.

Dans le terminal:

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

- Indiquez le nom de la classe: PostType
- Indiquez le nom de l'entité associée au formulaire: Post
- Dans le dossier Controller, un sous-dossier «Form» et un fichier de classe «PostType.php» viennent d'être créés.
- Ouvrez le fichier pour commencer à utiliser le FormBuilder.
- Il contient déjà: le namespace, les uses nécessaires et la structure de la classe.

La classe de départ – PostType

```
public function buildForm(FormBuilderInterface $builder, array $options): void
{
    $builder
        ->add('title')
        ->add('content')
        ->add('createdAt', null, [
            'widget' => 'single_text',
        ])
        ->add('image')
        ->add('isPublished')
        ->add('category', EntityType::class, [
            'class' => category::class,
            'choice_label' => 'id',
        ])
    ;
}
```

Symfony a ajouté pour chaque propriété de l'entité Post une méthode add(). Cette méthode provient de la classe **FormBuilderInterface** passée en injection de dépendance. Il ne s'agit pas ici de persister les données mais seulement de construire le formulaire (buildForm). La persistance des objets en DB sera confiée au contrôleur.

Création d'une vue pour afficher le formulaire

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

{% block title %}Nouvel article{% endblock %}

{% block body %}
    <section class="row">
        <div class="col-md-6 offset-3">
            <h2 class="display-5 my-4">Ajout d'une formation</h2>
            {{ form(form) }}
        </div>
    </section>
{% endblock %}
```

Pour afficher le formulaire dans la vue nous utilisons un helper twig:

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

Le nom `form` entre parenthèses correspond au paramètre que vous passerez dans le contrôleur avec la méthode `render()`. Le terme `form` est une convention d'écriture.

La méthode newPost du AdminPostController

```
#[Route('/admin/newpost', name: 'app_admin_newpost')]
public function newPost(): Response
{
    $form = $this->createForm(PostType::class);
    return $this->render('admin/posts/newpost.html.twig', [
        'form' => $form,
    ]);
}
```

La méthode `createForm()` de l'AbstractController permet de créer le formulaire. Il prend en paramètre le nom de la classe du formType:

`PostType::class`.

L'objet `$form` est passé en paramètre de la méthode `render()`

Modifier la vue posts.html.twig

- Cette vue permet pour l'instant via un bouton d'ajouter un nouvel article. Le lien du bouton doit être ajouté à l'aide du helper path.

```
<a href="{{ path('app_admin_newpost') }}" class="btn btn-outline-dark">Nouvel article</a>
```

- Dans le path, vous devez indiquer le name de la route du controller et pas le lien vers la vue avec le formulaire. En MVC tout passe par le controller qui génère le formulaire.

La vue et le formulaire

Administration Articles Catégories Utilisateurs

Ajout d'un article

Title

Content

Create at jj-mm-aaaa --:--

Image

Is published ☐

Category 57

A ce moment rien n'est fonctionnel:

- Apparence
- Pas de bouton Submit
- Le nom des champs (propriétés de l'entité)
- La catégorie par ID
- Le Controller affiche le formulaire mais n'est pas encore fonctionnel pour l'enregistrement des données

Si vous inspecter le code, vous verrez le code du formulaire sans l'attribut `action=""`

Plus bas, un champ caché avec un token permet de vérifier que les données du formulaire proviennent bien de celui-ci.

Rendre le formulaire fonctionnel

Donnez du style avec Bootstrap

- Nous avons déjà installé Bootstrap mais comme nous utilisons le helper form de Twig, nous n'avons pas accès aux champs de formulaire pour ajouter des classes.
- Pour styler les formulaires avec Bootstrap, nous allons modifier le fichier de configurations de Twig twig.yaml dans config/packages.
- On ajoute une nouvelle clé:

```
twig:  
  file_name_pattern: '*.twig'  
  form_themes: ['bootstrap_5_layout.html.twig']
```

- Ceci provient de la documentation de Symfony et d'autres thèmes sont disponibles:
https://symfony.com/doc/current/form/form_themes.html

Rendre le formulaire fonctionnel

Ajoutez un bouton Submit

- Vous avez deux possibilités:

Soit ajouter le bouton dans la vue du formulaire mais vous ne pourrez plus utiliser la fonction `{{ form(form) }}`

```
{{ form_start(form) }}  
  {{ form_widget(form) }}  
  <input type="submit" value="Ajouter" class="btn btn-outline-dark">  
{{ form_end(form) }}
```

- Soit modifier le `PostType` pour rajouter le bouton et utiliser la fonction `{{ form(form) }}`.

```
->add('save', SubmitType::class, [  
    'label' => 'Ajouter'  
)
```

Rendre le formulaire fonctionnel

Modifier les noms des champs, définir les entityType (types des champs) et autres options.

Symfony a seulement créé une méthode de base que nous allons compléter pour la rendre opérationnelle.

La méthode add() peut prendre plusieurs paramètres:

1. La propriété de l'entité concernée (title, content,...)
2. Le type de la propriété sous la forme d'une classe (permet de définir le type de champs HTML affiché dans le formulaire)
3. Un tableau associatif contenant des options (label, data, format,...)

```
$builder  
    ->add('title', TextType::class, [  
        'label' => 'Titre de l\'article'  
    ])
```

4. Tous les types sont disponibles dans la doc officielle:

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

Rendre le formulaire fonctionnel

Affichez le nom de la catégorie et pas l'ID.

Vous devez retourner sur le `PostType` pour y faire une seule modification. Modifiez la valeur de la clé `'choice_label'` pour y mettre le nom (name) de la catégorie et pas l'Id. Remarquez aussi le type qui est différent, il s'agit d'un type `'EntityType'` dont le nom de l'entité figure dans la clé `'class'`

```
->add('category', EntityType::class, [  
    'class' => category::class,  
    'choice_label' => 'name',  
])
```

```

public function buildForm(FormBuilderInterface $builder, array $options): void
{
    $builder
        ->add('title', TextType::class, [
            'label' => 'Title de l\'article',
        ])
        ->add('content', TextareaType::class, [
            'label' => 'Contenu de l\'article',
        ])
        ->add('createAt', DateType::class, [
            'label' => 'Date de création',
            'widget' => 'single_text',
            'input' => 'datetime_immutable',
        ])
        ->add('image', TextType::class, [
            'label' => 'Image',
            'data' => 'default.png',
        ])
        ->add('category', EntityType::class, [
            'label' => 'Catégorie',
            'class' => Category::class,
            'choice_label' => 'name',
            'placeholder' => 'Sélectionnez...',
        ])
        ->add('save', SubmitType::class, [
            'label' => 'Ajoutez'
        ])
    ;
}

```

La classe PostType et la méthode buildForm()

La vue du formulaire

Ajout d'un article

Title de l'article

Contenu de l'article

Date de création

Image

Catégorie

Ajoutez

Le contrôleur AdminPost et la méthode newPost()

Cette méthode va jouer un double rôle:

1. Permettre tout simplement d'afficher le formulaire
2. Persister les données une fois le formulaire soumis

Etape 01

1. Pour afficher le formulaire, on utilise la méthode `createFormBuilder()` héritée de la classe `AbstractController` qui permet d'initialiser la construction du formulaire. On lui passe deux paramètres: la classe `PostType` et une instance de la classe `Post`.
2. Ensuite, dans le `render()` de la vue on utilise la méthode `createView()` sur l'objet `$form`. Cet objet représente un ensemble d'utilitaires que l'on ne peut pas utiliser directement pour l'envoyer à la vue.

Depuis Symfony 5.3 une nouvelle méthode (`renderForm()` de la classe `AbstractController`) concernant l'envoi du formulaire à la vue a été ajoutée. Elle remplace la méthode `render()`.

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

Le contrôleur Post et la méthode addPost()

- Maintenant la méthode permet de lier et d'afficher le formulaire dans la vue mais pas encore de persister les données en DB.

```
public function addPost ()
{
    $post = new Post;
    $form = $this->createForm(PostType::class, $post);
    return $this->render('post/add.html.twig', [
        'form' => $form
    ]);
}
```

Le contrôleur Post et la méthode addPost()

Etape 02

- Avant de persister, on doit récupérer dans la requête les données soumises par le formulaire. Pour cela, on doit indiquer en paramètre de la méthode que l'on a besoin d'une instance de la classe Request.

```
public function addPost (Request $request)
```

- Après nous devons gérer cette requête avec la méthode handleRequest()

```
$form->handleRequest($request);
```

- Ensuite, il faut vérifier si le formulaire a été soumis et s'il est valide (permettra de définir par la suite des critères de validation) et dans la condition on pourra persister les données.

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

Le contrôleur Post et la méthode addPost()

- Maintenant dans la condition, nous allons persister les données. Pour cela nous avons besoin du Manager de Doctrine qui nous permettra d'indiquer à doctrine que nous souhaitons enregistrer les données et exécuter la requête d'insertion.

```
$manager = $this->getDoctrine()->getManager();
```

```
$manager->persist($post);
```

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

- Comme le Manager de Doctrine est une dépendance, nous pouvons aussi l'injecter directement en paramètre de la méthode et nous passer alors de la première ligne.

```
public function addPost (Request $request, EntityManagerInterface $manager)
```

Le contrôleur AdminPostController et la méthode newPost()

```
#[Route('/admin/newpost', name: 'app_admin_newpost')]
public function newPost(EntityManagerInterface $manager, Request $request): Response
{
    $post = new Post();
    $form = $this->createForm(PostType::class, $post);
    $form->handleRequest($request);
    if($form->isSubmitted() && $form->isValid()) {
        $post->setPublished(true);
        $post->setCreatedAt(new \DateTimeImmutable());
        $manager->persist($post);
        $manager->flush();
        return $this->redirectToRoute('app_admin_posts');
    }
    return $this->render('admin/posts/newpost.html.twig', [
        'form' => $form,
    ]);
}
```

Code de la vue

```
{% block body %}
    <main class="container">
        <section class="row">
            <div class="col-md-8 offset-md-2">
                <h2>Ajouter un article</h2>
                {{ form(form) }}
            </div>
        </section>
    </main>
{% endblock %}
```

Pour afficher le formulaire en Twig vous avez différentes possibilités, voici la plus simple et la plus rapide. On utilise le helper `form()` qui prend en paramètre le même nom que celui transmis via le `render()` de la méthode

Afficher tous les articles sur la vue de l'admin

Lors de l'accès à l'admin (via url et non sécurisé) il est important d'afficher tous les articles (même ceux dont `isPublished = false`) sous la forme d'un tableau. Nous ajouterons des colonnes pour les opérations de suppression, d'édition et d'afficher/masquer.

Ouvrez le Controller `AdminPostController` et ajoutez une méthode permettant d'afficher tous les articles.

```
#[Route('/admin/posts', name: 'app_admin_posts')]
public function posts(PostRepository $repository): Response
{
    $posts = $repository->findBy(
        [],
        ['createdAt' => 'DESC']
    );
    return $this->render('admin/posts/posts.html.twig', [
        'posts' => $posts,
    ]);
}
```


La vue

Elle est pour l'instant identique à la partie publique.

Gestion des articles

Nouvel article

Liste des articles

Image	Titre	Catégorie	Publication
	Quisquam velit cupiditate numquam	Angular	22/05/2024
	Nihil hic eius quia	Angular	21/05/2024
	Culpa quidem pariatur	Symfony	20/05/2024
	Maiores est accusantium	Laravel	20/05/2024
	Modi provident sed assumenda laudantium	Laravel	19/05/2024
	In repellat est eveniet provident	IA	18/05/2024
	Vitae tempore et officia quaerat	Symfony	18/05/2024

Ajoutez un champ de type upload dans le formulaire (Vich)

Il est possible d'uploader un fichier directement dans Symfony, mais le plus simple et le plus complet est d'utiliser le Bundle

VichUploaderBundle. Toutes les étapes que je vais décrire sont issues de la documentation officielle de Vich:

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

Étapes:

- **Installation**: composer require vich/uploader-bundle
- **Configuration** du fichier config/packages/**vich_uploader.yaml**.
Modifiez le mapping existant pour l'adapter à votre configuration (paths). Le mapping a été nommé **posts** pour des raisons de lisibilité. Vous devrez en créer un différent pour chaque **upload file** dans votre projet (image, pdf,...).
La dernière clé: **namer** permet à Vich d'ajouter un suffixe (clé) pour conserver un nom unique dans le dossier des images.
Lors de la suppression d'un article, Vich effacera l'image du dossier posts.

```
vich_uploader:
  db_driver: orm

mappings:
  posts:
    uri_prefix: /images/posts
    upload_destination: '%kernel.project_dir%/assets/images/posts'
    namer: Vich\UploaderBundle\Naming\SmartUniqueNamer
```

Ajoutez un champ de type upload dans le formulaire (Vich)

Vous devez maintenant modifier l'entité `Post` pour l'adapter au bundle Vich. Nous devons créer un lien entre le système de fichiers et l'entité que l'on souhaite rendre téléchargeable.

Étapes:

- Ajoutez deux nouveaux use dans l'entité et l'attribut `#[Vich\Uploadable]`:
- Ensuite, vous devez créer les deux propriétés nécessaires au fonctionnement du bundle (imageFile et updatedAt). Vich impose l'ajout de la propriété `updatedAt` pour effectuer l'upload.
- Ajoutez le getter et le setter pour la propriété `imageFile` (voir la doc) et pour la propriété `updatedAt` (utilisez PhpStorm)

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

#[ORM\Entity(repositoryClass: PostRepository::class)]
#[Vich\Uploadable]
```

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

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

```
public function setImageFile(?File $imageFile = null): void
{
    $this->imageFile = $imageFile;

    if (null !== $imageFile) {
        $this->updatedAt = new \DateTimeImmutable();
    }
}

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

Ajoutez un champ de type upload dans le formulaire (Vich)

Modifiez le `PostType` (formulaire) pour ajouter le champ d'uploadFile spécifique à Vich.

Modifiez la classe `PostFixtures` pour l'adapter à l'ajout de la propriété `updatedAt`.

Retirez du `PostController` le setter de l'image par défaut.

- Créez la migration: `make:migration`
- Réalisez la modification en DB: `d:m:m`
- Loades les fixtures en DB: `d:f:l`

```
use Vich\UploaderBundle\Form\Type\VichImageType;
```

```
->add('imageFile', VichImageType::class, [  
    'required' => false,  
    'allow_delete' => false,  
    'download_uri' => false,  
    'asset_helper' => true,  
)
```

```
->setUpdatedAt(new \DateTimeImmutable());  
$post->setImage($i.'.jpg');
```

Supprimer un article

Principe

La méthode delete()

Le lien de suppression dans la vue index.html.twig

La méthode delPost pour effacer un enregistrement

Pour effacer un objet, nous avons besoin de:

- Récupérer le composant Entity Manager pour utiliser ses méthodes
- Récupérer l'enregistrement (objet) à effacer via le repository et la méthode find()
- Utiliser l'Entity Manager pour supprimer les données et exécuter la requête.

Méthode delPost()

L'injection de dépendance (EntityManagerInterface) et ParamConverter pour transformer l'id en un objet correspondant simplifie le code.

```
#[Route('admin/delpost/{id}', name: 'app_admin_delpost')]
public function delPost(Post $post, EntityManagerInterface $manager): Response
{
    $manager->remove($post);
    $manager->flush();
    return $this->redirectToRoute('app_admin_posts');
}
```

Le lien dans la vue

```
<td>
  <a href="{{ path('delepost', {id:post.id}) }}">
    <i class="icofont-ui-delete"></i>
  </a>
</td>
```

A l'aide du helper path en Twig, on crée le lien vers la méthode delepost() en passant l'id de l'entité à supprimer. Delepost correspond au name de la route.















Ajouter une icone (icofont) correspondant à la suppression.

La vue avec l'icone de suppression

Gestion des articles

Nouvel article

Liste des articles

Image	Titre	Catégorie	Publication	
	Quisquam velit cupiditate numquam	Angular	22/05/2024	
	Nihil hic eius quia	Angular	21/05/2024	
	Culpa quidem pariatur	Symfony	20/05/2024	
	Maiores est accusantium	Laravel	20/05/2024	
	Modi provident sed assumenda laudantium	Laravel	19/05/2024	
	In repellat est eveniet provident	IA	18/05/2024	
	Vitae tempore et officia quaerat	Symfony	18/05/2024	

Mettre à jour un enregistrement

Principe

Contrôleur

Vue (formulaire)

Vue générale (index)

Le contrôleur et la méthode editPost()

- Il est temps de terminer cette partie consacrée au CRUD avec la mise à jour d'un enregistrement.
- Le procédé est presque identique à l'insertion mais nous allons ici utiliser le `ParamConverter` pour convertir l'id de l'enregistrement en une entité correspondante.

Le contrôleur et la méthode editPost()

```
#[Route('admin/editpost/{id}', name: 'app_admin_editpost')]
public function editPost(Post $post, EntityManagerInterface $manager, Request $request): Response
{
    $form = $this->createForm(PostType::class, $post);
    $form->handleRequest($request);
    if($form->isSubmitted() && $form->isValid()) {
        $manager->flush();
        return $this->redirectToRoute('app_admin_posts');
    }
    return $this->render('admin/posts/editpost.html.twig', [
        'posts' => $post,
        'form' => $form,
    ]);
}
```

La vue avec le formulaire d'édition

1. Créez une nouvelle vue: edit.html.twig
2. Ajoutez y le formulaire avec la méthode d'affichage Twig

```
{% block body %}
    <main class="container">
        <section class="row">
            <div class="col-md-8 offset-md-2">
                <h2>Mettre à jour un article un article</h2>
                {{ form(form) }}
            </div>
        </section>
    </main>
{% endblock %}
```

3. Ajouter le lien de l'édition avec une icone dans la vue générale (helper path)

Problème de date dans le formulaire d'update

- Lors de la mise à jour d'un enregistrement la date du jour est automatique proposée dans le formulaire. C'est normal, nous avons demandé dans le champ `createdAt` de la classe `PostType` de prendre comme valeur la date du jour. Mais pour un update on ne peut pas modifier la date de création de l'article. Pour ce faire, il suffit de retirer la méthode `add()` pour le type `createAt`. Comme il est retiré du formulaire, le champ ne sera jamais mis à jour. Mais en faisant cela, nous retirons également la date de création dans le formulaire d'insertion comme nous utilisons le même `PostType` pour l'insert et l'update.
- Pour insérer la date du jour pour la création de l'article, nous allons devoir procéder via la méthode `newPost` du controller.

Modification de la méthode newPost()

- Ajoutez le setter avec la date du jour avant de persister les données

```
if($form->isSubmitted() && $form->isValid()) {  
    $post->setPublished(true);  
    $post->setCreatedAt(new \DateTimeImmutable());  
    $manager->persist($post);  
    $manager->flush();  
    return $this->redirectToRoute('app_admin_posts');  
}
```

La vue posts.html.twig (Admin)

Gestion des articles

Nouvel article

Liste des articles

Image	Titre	Catégorie	Publication		
	Quisquam velit cupiditate numquam	Angular	22/05/2024		
	Nihil hic eius quia	Angular	21/05/2024		
	Culpa quidem pariatur	Symfony	20/05/2024		
	Maiores est accusantium	Laravel	20/05/2024		
	Modi provident sed assumenda laudantium	Laravel	19/05/2024		
	In repellat est eveniet provident	IA	18/05/2024		
	Vitae tempore et officia quaerat	Symfony	18/05/2024		
	Ea impedit eius enim saepe	Symfony	17/05/2024		
	Sed facere repudiandae	Symfony	17/05/2024		
	Repudiandae iste repudiandae	Security	16/05/2024		
	Quibusdam aut aut corrupti	Security	14/05/2024		
	Aperiam sunt quibusdam	Angular	13/05/2024		

Afficher ou masquer un article

Contrôleur

Vue Admin

Résultat

Afficher ou masquer un article

- Nous souhaitons afficher ou masquer un article sans passer par le formulaire d'édition. Pour ce faire nous allons ajouter une méthode viewPost dans le controller et une colonne supplémentaire dans le tableau de l'admin.
- Le controller:

```
#[Route('/admin/viewpost/{id}', name: 'app_admin_viewpost')]
public function viewPost(Post $post, EntityManagerInterface $manager): Response
{
    $post->setPublished(!$post->isPublished());
    $manager->flush();
    return $this->redirectToRoute('app_admin_posts');
}
```

Afficher ou masquer un article

- La vue Admin:

```
<td><a href="{{ path('app_admin_viewpost', {id:post.id}) }}">
  {% if post.isPublished %}
    <i class="icofont-eye"></i>
  {% else %}
    <i class="icofont-eye" style="color: red;"></i>
  </a>
  {% endif %}
</td>
```
































- On modifie la couleur de l'icone (rouge par ex.) si l'article n'est pas publié.

Afficher ou masquer un article

Gestion des articles

Nouvel article

Liste des articles

Image	Titre	Catégorie	Publication			
	Test	Security	05/06/2024			
	Quisquam velit cupiditate numquam	Angular	22/05/2024			
	Nihil hic eius quia	Angular	21/05/2024			
	Culpa quidem pariatur	Symfony	20/05/2024			
	Maiores est accusantium	Laravel	20/05/2024			
	Modi provident sed assumenda laudantium	Laravel	19/05/2024			
	In repellat est eveniet provident	IA	18/05/2024			

Les messages flash

Principe

Contrôleur

Vue

Principes

- Vous pouvez stocker des messages de type warning, success, danger... appelés messages "flash", dans la session de l'utilisateur. Les messages flash sont conçus pour être utilisés une seule fois, ils disparaissent automatiquement de la session dès que vous les avez affichés.
- Cette fonctionnalité rend les messages "flash" particulièrement intéressants pour stocker tout types de notifications destinés aux utilisateurs.

Le contrôleur

- Nous allons créer le message de type success lors de l'ajout d'un article en DB. Dans le contrôleur, ajoutez le code suivant entre la méthode `flush()` et la méthode `redirectToRoute()`.

```
$manager->flush();  
$this->addFlash(  
    'success',  
    'L\'article a bien été ajouté');  
return $this->redirectToRoute('app_admin_posts');
```

Il s'agit d'une méthode `addFlash()` de l'`AbstractController` invoquée sur l'objet en cours et prenant deux paramètres à envoyer en session : **le type et le message**. Le paramètre type peut être n'importe quoi mais utilisez un terme conventionnel. Il vous permettra d'afficher avec Bootstrap le message dans la vue. Si vous ajoutez un article, vous pourrez visualiser le message dans le Profiler, catégorie Flashes.

Le partial et La vue

- Pour récupérer le message dans la vue, on utilise deux boucles for et la variable globale de Twig qui est `app.variable`. Elle donne accès à certaines information sur votre application (app.user, app.session (session de l'utilisateur lors de son identification), app.flashes...) Les deux boucles imbriquées permettront d'afficher plusieurs messages flash si il y a plus d'une erreur dans la session. Elle prend en paramètre le message(s) et le type(s) que vous avez mentionner dans la méthode le ou les messages addFlash(). Dans notre cas success qui sera concaténer avec une classe native de Bootstrap.
- Pour plus de clarté et éviter des redondances de codes, il convient de créer un partial dans le dossier admin des vues: `flash.html.twig`. Vous allez inclure ce partial dans la vue contenant l'administration: `post.html.twig`

```
{% for label, messages in app.flashes %}
  {% for message in messages %}
    <div class="alert alert-{{ label }}">
      {{ message }}
    </div>
  {% endfor %}
{% endfor %}
```

```
<div class="col-md-9 offset-1">
  {% include 'admin/partials/flash.html.twig' %}
  <h2 class="display-5 my-4">Gestion des articles</h2>
  <a href="{{ path('app_admin_newpost') }}" class="btn
  btn-outline-dark">Nouvel article</a>
</div>
```

Les utilisateurs et le composant Security

Fonctionnement

L'entité User

Passwords

Fixtures

Inscription

Authentification

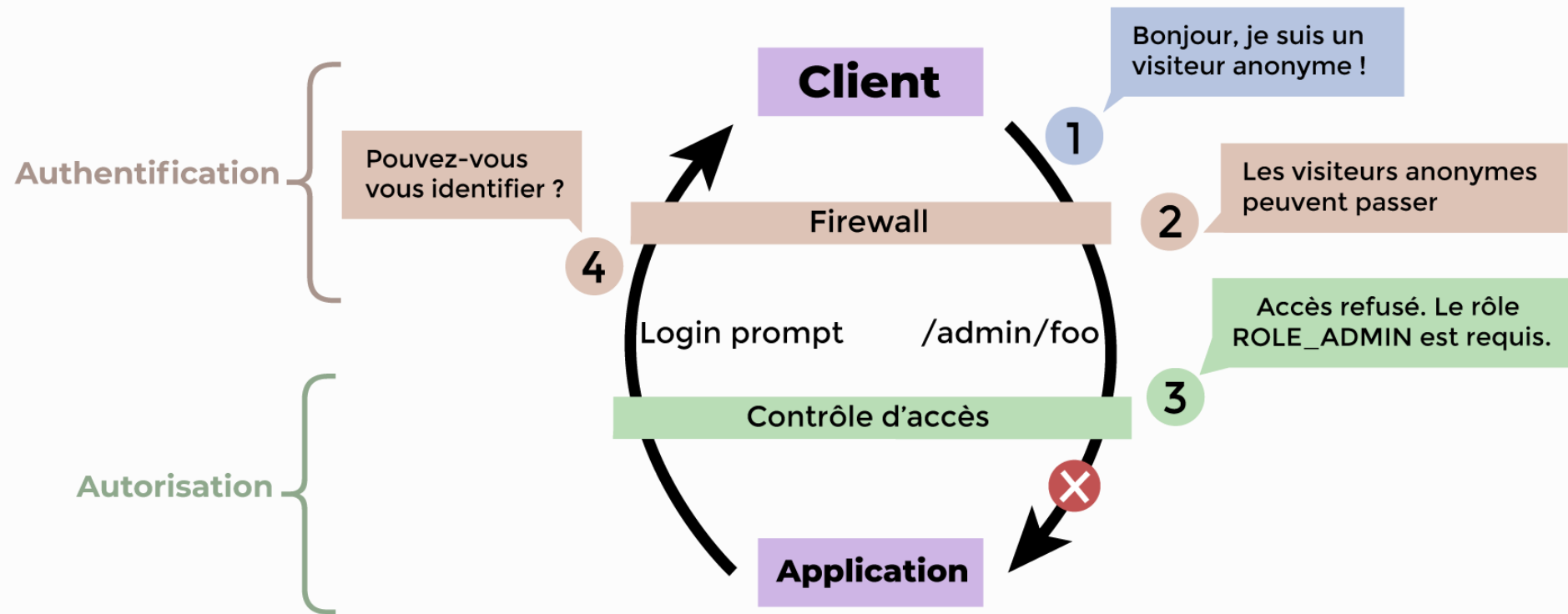
Fonctionnement

- Le contrôle de la sécurité sous Symfony fonctionne sous le double principe de l'authentification et de l'autorisation.
- **L'authentification**, est le procédé qui permet de déterminer qui est votre visiteur. Il y a deux cas possibles :
 - le visiteur est anonyme car il ne s'est pas identifié,
 - le visiteur est membre de votre site car il s'est identifié.
- Sous Symfony, c'est le **firewall** qui prend en charge l'authentification. Régler les paramètres du firewall va vous permettre de sécuriser le site. En effet, vous pouvez restreindre l'accès à certaines parties du site uniquement aux visiteurs qui sont membres. Autrement dit, il faudra que le visiteur soit authentifié pour que le firewall l'autorise à passer.

Fonctionnement

- **L'autorisation** intervient après l'authentification. Comme son nom l'indique, c'est la procédure qui va accorder les droits d'accès à un contenu. Sous Symfony, c'est **l'access control** qui prend en charge l'autorisation.
- Prenons l'exemple de différentes catégories de membres. Tous les visiteurs authentifiés ont le droit de poster des messages sur le forum mais uniquement les membres administrateurs ont des droits de modération et peuvent les supprimer. C'est **l'access control** qui permet de faire cela.

Le composant Security



Introduction

- Cette partie concerne la gestion des utilisateurs et la sécurité. Nous allons mettre en place un système de gestion des articles en relation avec les utilisateurs, l'inscription et l'identification de ceux-ci.
- Avant de mettre en place le composant Security de Symfony, nous allons réaliser une copie du projet et recréer la base de données.

Projet: `webpostuser`

DB: `webpostuser`(modifier le .env)

- Commande: `php bin/console doctrine:database:create`
- Supprimer le ou les anciens fichiers de migration

Création de l'entité User

Vous avez la possibilité de créer et de gérer les utilisateurs de deux manières:

1. Manuellement en créant l'entité User comme une entité classique et en ajouter les fonctionnalités propres aux users (rôle, password, unicité...). Vous devrez aussi modifier le fichier `security.yaml` pour configurer l'encoder (password) et le Firewall.
2. Utiliser une commande de type `make` vous permettant de gérer automatiquement dans Symfony les utilisateurs. Vous devrez configurer certains points comme l'Access Control mais cette solution est plus rapide. Saisissez `make:user` dans le terminal.

Nous utiliserons la deuxième méthode mais je vais expliquer toutes les modifications effectuées automatiquement par Symfony.

Utilisation de la commande make:user

```
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]:

```
> yes
```

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

```
> email
```

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

```
> yes
```

created: src/Entity/User.php

created: src/Repository/UserRepository.php

updated: src/Entity/User.php

updated: config/packages/security.yaml

L'entité User

On constate que Symfony a créé une entité reprenant quatre propriétés avec leurs getters et leurs setters: `$id`, `$email`, `$roles` et `$password`

La propriété `$role` est un peu particulière car il s'agit d'un type array. C'est la manière dont Symfony va enregistrer les rôles en base données:

```
['ROLE_USER']
```

```
['ROLE_ADMIN']
```

Son accesseur (`getRoles`) retourne un tableau avec les rôles et dans ce cas, ça sera toujours le rôle user. Cela permet d'avoir au moins le rôle de base et nous pourrons changer ça par après.

Une méthode `eraseCredentials()` provient de l'interface `UserInterface` implémentée dans la classe `User` et une autre, `getUserIdentifier()` provenant de la même interface. Selon l'architecture objet, quand une classe implémente une Interface toutes ses méthodes doivent obligatoirement être ajoutées.

Ajoutez des propriétés à l'entité User

En passant par la CLI, nous n'avons pas pu choisir nos propres propriétés. Il est possible d'en ajouter au besoin en repassant par la commande `make:entity User`

Propriétés à ajouter:

- firstName (string -120)
- lastName (string – 120)
- imageName (string – 255)
- createdAt (datetime_immutable)
- updatedAt (datetime_immutable)
- isDisabled (boolean)

N'oubliez pas la migration:

- Supprimer les anciens fichiers de migrations
- `php bin/console make:migration`
- `php bin/console doctrine:migrations:migrate`

Modifier l'entité Post

Pour mettre en relation les articles et les utilisateurs, nous devons ajouter un attribut dans l'entité Post. Post étant l'entité propriétaire, c'est ici que nous allons définir la relation. `php bin/console make:entity Post`

- New property name: `user`
- Field type: `ManyToOne`
- What class should this entity be related to?: `User`
- Is the Post.user property allowed to be null ? : `no`
- Do you want to add a new property to User ... ? : `yes`
- New field name inside User: `posts`
- Do you want to activate orphanRemoval on your relationship?: `no`

Modifications effectuées dans l'entité post

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

La propriété `$user` avec le type de relation et le nullable

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

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

    return $this;
}
```

Le getter et le setter de la propriété `$user`

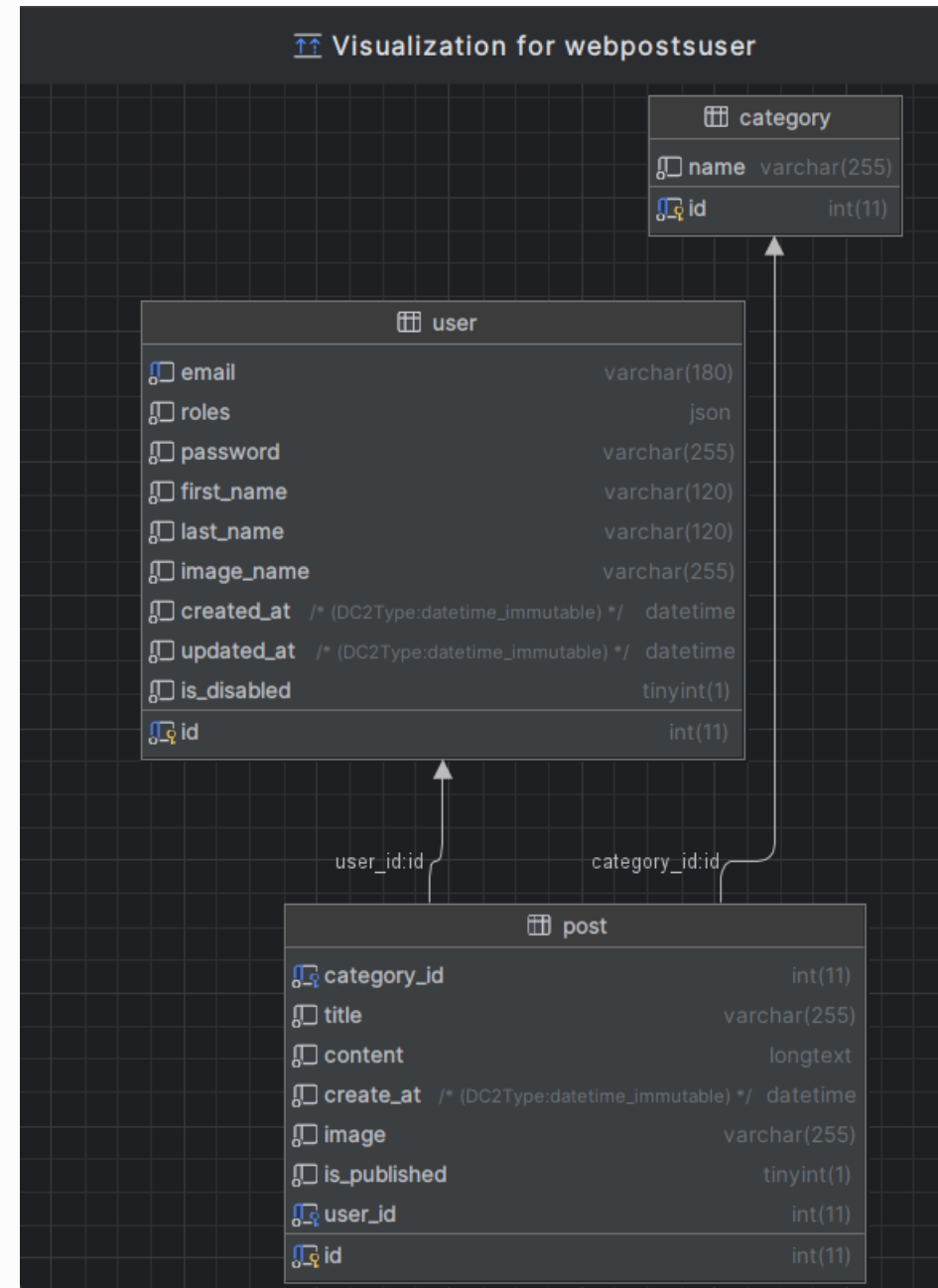
La migration

Supprimez les fichiers de migration du projet

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

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

La nouvelle base de données relationnelle figure bien dans Mysql.



Le fichier security.yaml avant la création des users

```
security:
  password_hashers:
    Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
  providers:
    users_in_memory: { memory: null }
  firewalls:
    dev:
      pattern: ^/(_(profiler|wdt)|css|images|js)/
      security: false
    main:
      lazy: true
      provider: users_in_memory
  access_control:
    # - { path: ^/admin, roles: ROLE_ADMIN }
    # - { path: ^/profile, roles: ROLE_USER }
```

Le fichier security.yaml après la création des users

```
security:
  password_hashers:
    Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
  providers:
    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
  access_control:
    # - { path: ^/admin, roles: ROLE_ADMIN }
    # - { path: ^/profile, roles: ROLE_USER }
```

Security.yaml – le password_hasher

Pour des raisons de sécurité, nous aurons besoin d'encoder les mots de passe dans la base de données. Nous allons vérifier le composant Security. Celui-ci prend en charge tout ce qui concerne la sécurité de votre application.

La clé `password_hashers` permettant de crypter les mots de passe figure déjà dans le fichier. Elle utilise l'interface `PasswordAuthenticated...` qui sera également utilisée dans un controller, pour prendre en charge le hachage du password ou autre champ.

```
security:  
  password_hashers:  
    Symfony\Component\Security\Core\User>PasswordAuthenticatedUserInterface: 'auto'
```

On y retrouve le nom de l'interface pour la gestion des passwords et le type d'algorithme utilisé. Symfony utilise une valeur automatique (auto) ce qui permettra de maintenir et de supporter dans le temps d'autres algorithmes et d'assurer aussi la portabilité.

Security.yaml – le provider et l'authentification

- Nous en avons déjà parlé, l'authentification est gérée par le Firewall de Symfony. Nous devons le déclarer dans le fichier `security.yaml`
- Une fois de plus, il a été configuré à notre place. Symfony a créé un provider du nom de `app_user_provider`. On constate que l'identification des utilisateurs sera vérifiée à partir de la propriété `email` de l'entité `User`.
- Son objectif est de fournir au Firewall les données de l'utilisateur avec comme propriété unique l'email. Nous aurions pu choisir par exemple `username` si nous en avions créé un dans l'entité `User`.

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

Security.yaml – le provider et l'authentification

- Les firewalls dev et **main**. Le premier dev ne concerne que la partie développement et la webdebug toolbar du navigateur. Le main firewalls récupère les données utilisateur du provider situé plus haut: `app_user_provider`

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

- L'accès control plus bas est identique et permettra de définir les permissions et les rôles des utilisateurs.

Fixtures User

- Créez une nouvelle fixture: `php bin/console make:fixture UserFixtures`
- Pour encoder les mots de passe `cryptés` nous aurons besoin de l'interface `UserPasswordHasherInterface` qui implémente trois méthodes: `hashPassword()`, `isPasswordValid()` et `needsRehash()`. C'est la première qui nous intéresse ici.
- Pour utiliser l'interface et sa méthode, nous ne pouvons pas l'injecter en dépendance dans notre méthode `load()` mais via un constructeur.

```
private object $hasher;  
/**  
 * UserFixtures constructor.  
 * @param UserPasswordHasherInterface $hasher  
 */  
public function __construct(UserPasswordHasherInterface  
$hasher)  
{  
    $this->hasher = $hasher;  
}
```

- Maintenant, nous pouvons utiliser la méthode `hashPassword()` dans notre fixture.

Fixtures User finalisé

Propriétés de la classe

```
private object $hasher;  
private array $genders = ['male', 'female'];
```

Constructeur pour le cryptage

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

Méthode pour le chargement des fixtures

```
public function load(ObjectManager $manager): void  
{  
    $faker = Factory::create();  
    for ($i = 1; $i <= 50; $i++) {  
        $user = new User();  
        $gender = $faker->randomElement($this->gender);  
        $user->setFirstName($faker->firstName($gender))  
            ->setLastName($faker->lastName)  
            ->setEmail($user->getFirstName().'.'.$user->getLastName().'@'.$faker->domainName());  
        $gender = ($gender == 'male' ? 'm' : 'f');  
        $user->setImageName('0' . ($i + 10) . $gender.'.jpg')  
            ->setPassword($this->hasher->hashPassword($user, 'password'))  
            ->setDisabled($faker->boolean(10))  
            ->setCreatedAt(new \DateTimeImmutable())  
            ->setUpdatedAt(new \DateTimeImmutable())  
            ->setRoles(['ROLE_USER']);  
        $manager->persist($user);  
    }  
    $manager->flush();  
}
```

Fixtures User finalisé

- Ajoutez deux administrateurs de votre choix dont vous retiendrez facilement les emails.
- Dans notre projet, seuls les administrateurs peuvent poster des articles.

```
// Admin John Doe
$user = new User();
$user ->setFirstName('John')
      ->setLastName('Doe')
      ->setEmail('john.doe@gmail.com')
      ->setImageName('062m.jpg')
      ->setPassword($this->hasher->hashPassword($user, 'password'))
      ->setCreatedAt(new \DateTimeImmutable())
      ->setUpdatedAt(new \DateTimeImmutable())
      ->setDisabled(false)
      ->setRoles(['ROLE_ADMIN']);
$manager->persist($user);
$manager->flush();

// Admin Pat Mar
$user = new User();
$user ->setFirstName('Pat')
      ->setLastName('Mar')
      ->setEmail('pat.mar@gmail.com')
      ->setImageName('074m.jpg')
      ->setPassword($this->hasher->hashPassword($user, 'password'))
      ->setCreatedAt(new \DateTimeImmutable())
      ->setUpdatedAt(new \DateTimeImmutable())
      ->setDisabled(false)
      ->setRoles(['ROLE_ADMIN']);
$manager->persist($user);
$manager->flush();
```

Quelques explications

- Les images fournies dans le dossier avatars représentent des hommes et des femmes. Le suffixe m ou f permet de le définir. Attention le premier fichier commence au numéro 10. Il faudra en tenir compte dans la boucle.
- La propriété \$gender de type array me permettra de choisir aléatoirement le genre male ou female selon Faker. Nous utiliserons \$faker->randomElement(\$gender).
- Pour le prénom \$faker->firstName(\$gender) ce qui permet de générer un prénom féminin ou masculin au hasard.
- Ensuite via un opérateur ternaire on vérifie le contenu de \$gender pour stocker m ou f pour la concaténation du nom de l'image.
- Pour l'insertion du nom de l'image, on commence par mettre un zéro, ajouter 9 à la variable pour commencer à 10 concaténer le genre (m ou f) avec l'extension du fichier.

Modifications de PostFixtures

```
$users = $manager->getRepository(User::class)->findAll();  
for ($i = 1; $i < 35; $i++) {  
    $post = new Post();  
    $post ->setTitle($faker->words(3, true).'.')  
    //  
    ->setUser($users[array_rand($users)]);  
    $post->setImage($i.'.jpg');  
    $manager->persist($post);  
}  
$manager->flush();
```

```
public function getDependencies(): array  
{  
    return [  
        CategoryFixtures::class,  
        UserFixtures::class,  
    ];  
}
```

Lier les articles aux administrateurs

- Modifier la vue index.html.twig pour ajouter une colonne avec le nom de l'utilisateur.

```
<td>{{ post.user.firstName }} {{ post.user.lastName }}</td>
```

L'identification – Login

- Créez un contrôleur pour le formulaire de connexion :

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

- Choose a name for the controller class (e.g. SecurityController) [SecurityController]: (yes)
- Do you want to generate a '/logout' URL? (yes/no) [yes]: (yes)
- Do you want to generate PHPUnit tests? [Experimental] (yes/no) [no]: (no)
- Symfony retourne dans la console les messages suivants:
 - created: src/Controller/SecurityController.php
 - created: templates/security/login.html.twig
 - updated: config/packages/security.yaml

la vue login.html.twig

Cette vue a été créée automatiquement et contient le formulaire de login (identification).

- Nous observons une gestion d'erreur en cas d'erreurs dans le formulaire.
- Ensuite un affichage si l'on est déjà identifié.
- Après les champs nécessaires pour l'identification email et password. Les attributs name="_username" et name="_password" ne doivent pas être modifiés car ils seront gérés par la sécurité de Symfony.
- Enfin un token d'authentification a été ajouté dans le formulaire (champ caché) permettant de s'assurer de la provenance des données.

! Ici le champ unique d'identification est l'email. Mais vous pourriez utiliser un username comme pour certains sites

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

{% block title %}Log in!{% endblock %}

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

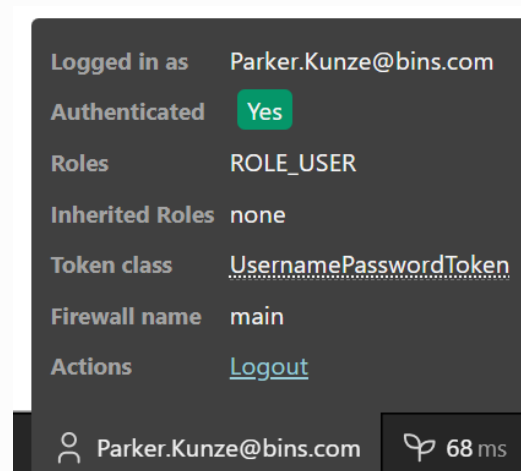
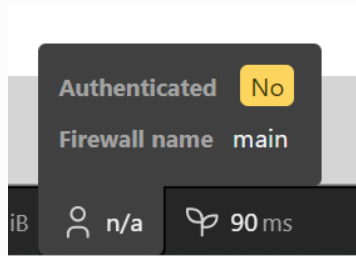
        {% if app.user %}
            <div class="mb-3">
                You are logged in as {{ app.user.userIdentifier }}, <a href="{{ path('app_logout') }}">Logout</a>
            </div>
        {% endif %}

        <h1 class="h3 mb-3 font-weight-normal">Please sign in</h1>
        <label for="username">Email</label>
        <input type="email" value="{{ last_username }}" name="_username" id="username" class="form-control"
autocomplete="email" required autofocus>
        <label for="password">Password</label>
        <input type="password" name="_password" id="password" class="form-control" autocomplete="current-
password" required>

        <input type="hidden" name="_csrf_token"
            value="{{ csrf_token('authenticate') }}">
        <button class="btn btn-lg btn-primary" type="submit">
            Sign in
        </button>
    </form>
{% endblock %}
```


Testez l'authentification

- Jusqu'à maintenant le profiler de Symfony nous indiquait que nous n'étions pas identifiés.
- Maintenant, si vous complétez le formulaire de login avec les données l'Email de la DB et le password "password" (voir les fixtures). Le profiler indiquera l'email de l'utilisateur que vous avez choisi.



- Pour accéder au formulaire de login, tapez login dans l'url et pour vous déconnecter saisissez logout.

Configurer le firewall

- Le fichier `security.yaml` a été modifié automatiquement lors de la création du form security login. Une fois activé, le système de sécurité redirige les visiteurs non authentifiés vers le `login_path` lorsqu'ils tentent d'accéder à un contenu sécurisé.

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

provider: le nom du provider que nous avons choisi ou que Symfony a défini par défaut.

form_login:

login_path: le nom de la route de la méthode login()

check_path: le même nom

Enable_csrf: true (vérification du token du formulaire)

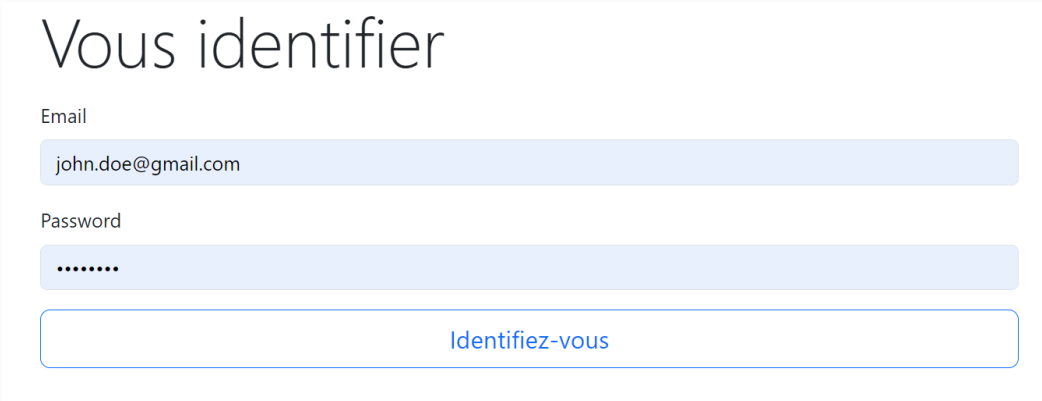
Logout:

path: le nom de la route pour se déconnecter

Les names des routes font parties du SecurityController.

Mise en page du form login

```
<div class="row">
  <div class="col-md-4 offset-md-4">
    <h1 class="display-4 my-3">Vous identifier</h1>
    <div class="form-group my-3">
      <label for="username" class="form-label">Email</label>
      <input type="email" value="{{ last_username }}" name="_username"
id="username" class="form-control" autocomplete="email" required autofocus>
    </div>
    <div class="form-group my-3">
      <label for="password" class="form-label">Password</label>
      <input type="password" name="_password" id="password" class="form-
control" autocomplete="current-password" required>
    </div>
    <div class="form-group my-3">
      <input type="hidden" name="_csrf_token"
value="{{ csrf_token('authenticate') }}">
      <button class="form-control btn btn-lg btn-outline-primary" type="submit">
        Sign in
      </button>
    </div>
  </div>
</div>
```

A visual rendering of the login form. It features a title "Vous identifier" at the top. Below it are two input fields: "Email" with the value "john.doe@gmail.com" and "Password" with masked characters ".....". At the bottom is a button labeled "Identifiez-vous".

Vous identifier

Email

john.doe@gmail.com

Password

.....

Identifiez-vous

Le security Controller

```
#[Route(path: '/login', name: 'app_login')]
public function login(AuthenticationUtils $authenticationUtils): Response
{
    // get the login error if there is one
    $error = $authenticationUtils->getLastAuthenticationError();

    // last username entered by the user
    $lastUsername = $authenticationUtils->getLastUsername();

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

```
#[Route(path: '/logout', name: 'app_logout')]
public function logout(): void
{
    throw new \LogicException('This method can be blank - it will be intercepted by the logout key on your firewall.');
```

Le partial navbar.html.twig

- Dans la navbar du site nous devons afficher les informations de l'utilisateur. Nous ajoutons un bouton se connecter ou se déconnecter en fonction de l'identification ou pas à la place de la fonction search..
- Pour faire le test de connexion, on va utiliser en Twig une variable d'environnement (`app.user`).
- Si l'utilisateur n'est pas connecté, alors on affiche le bouton s'identifier et le bouton s'enregistrer.
- Si l'utilisateur est connecté alors on affiche un "dropdown" avec les futures options de gestion de compte ainsi que le lien vers la déconnexion et vers l'administration s'il dispose d'un rôle admin.
- Vous devrez modifier les css pour un affichage correct.

```
{% if not app.user %}
    <a href="{{ path('app_login') }}" type="button" class="btn btn-dark btn-sm m-2"><i class="icofont-user"></i> S'identifier</a>
    <a href="" class="btn btn-dark btn-sm m-2"><i class="icofont-ui-add"></i> S'enregistrer</a>
{% else %}
    <div class="dropdown">
        <button class="nav-link dropdown-toggle" href="#" type="button" data-bs-toggle="dropdown" aria-expanded="false">
            &nbsp;&nbsp;&nbsp;{{ app.user.firstName }} {{ app.user.lastName }}
        </button>
        <ul class="dropdown-menu">
            <li><a class="dropdown-item" href="#">Mon profile</a></li>
            <li><a class="dropdown-item" href="#">Modifier le profile</a></li>
            <li><hr class="dropdown-divider"></li>
            <li class="dropdown-item"><a href="{{ path('app_logout') }}">Logout</a></li>
            <li><hr class="dropdown-divider"></li>
            <li><a class="dropdown-item" href="#">Administration</a></li>
        </ul>
    </div>
{% endif %}
```

Création du formulaire pour les inscriptions

php bin/console make:registration-form

- Do you want to add a @UniqueEntity validation annotation on your User class to make sure duplicate accounts aren't created? (yes/no) [yes]: yes
- Do you want to send an email to verify the user's email address after registration? (yes/no) [yes]: no
- Do you want to automatically authenticate the user after registration? (yes/no) [yes]: yes
- Do you want to generate PHPUnit tests? [Experimental] (yes/no) [no]: no

Un fichier modifié et deux autres créés:

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

created: src/Form/**RegistrationFormType.php**

created: src/Controller/**RegistrationController.php**

Création du formulaire pour les inscriptions

- Pour le **formType** (RegistrationFormType), une partie du code a été généré par Symfony mais il manque un certain nombre de propriétés. Vous allez devoir ajouter les méthodes **add()** en fonction de l'entité User.
- Pour **la vue** (register.html.twig), nous allons la simplifier et utiliser les helpers twig que nous avons déjà utilisés.
- Pour le **Controller** (RegistrationController), il faudra ajouter les propriétés absentes (firstName, lastName,...).
- Pour l'instant nous ne gérerons pas encore l'upload d'image (default.png) dans le formulaire ni les rôles. Nous mettrons ['ROLE_USER'] bien que ce ne soit pas nécessaire car c'est défini dans l'entité.

La vue «registration.html.twig»

S'enregistrer

Votre prénom

Votre nom de famille

Votre E-mail

Votre mot de passe

☐ Acceptez nos conditions générales

RegistrationFormType

```
$builder
->add('firstName', TextType::class, [
    'label' => 'Votre prénom',
    'attr' => [
        'placeholder' => 'Votre prénom'
    ]
])
->add('lastName', TextType::class, [
    'label' => 'Votre nom',
    'attr' => [
        'placeholder' => 'Votre nom'
    ]
])
->add('email', EmailType::class, [
    'label' => 'Votre email',
    'attr' => [
        'placeholder' => 'Votre email'
    ]
])
```

- La propriété roles est gérée via le controller (tous les nouveaux inscrits ont le rôle USER)
- Les propriétés created et updated sont gérées via le Contrôleur
- La propriété ImageName reçoit une valeur par défaut dans le Contrôleur

```
->add('plainPassword', PasswordType::class, [
    'label' => 'Votre mot de passe',
    'mapped' => false,
    'attr' => [
        'autocomplete' => 'new-password',
        'placeholder' => 'Mot de passe'
    ],
    'constraints' => [
        new NotBlank([
            'message' => 'Please enter a password',
        ]),
        new Length([
            'min' => 6,
            'minMessage' => 'Your password should be at least {{ limit }}
characters',
            // max length allowed by Symfony for security reasons
            'max' => 4096,
        ]),
    ],
])
->add('agreeTerms', CheckboxType::class, [
    'mapped' => false,
    'constraints' => [
        new IsTrue([
            'message' => 'You should agree to our terms.',
        ]),
    ],
]);
}
```

Le contrôleur RegistrationController

- Les uses (classes utilisées)

```
use App\Entity\User;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\Extension\Core\Type\CheckboxType;
use Symfony\Component\Form\Extension\Core\Type\EmailType;
use Symfony\Component\Form\Extension\Core\Type>PasswordType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;
use Symfony\Component\Validator\Constraints\IsTrue;
use Symfony\Component\Validator\Constraints\Length;
use Symfony\Component\Validator\Constraints\NotBlank;
```

Registration Controller

```
#[Route('/register', name: 'app_register')]
public function register(Request $request,
    UserPasswordHasherInterface $userPasswordHasher, Security
    $security, EntityManagerInterface $entityManager): Response
{
    $user = new User();
    $form = $this->createForm(RegistrationFormType::class, $user);
    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()) {
        $user->setDisabled(false);
        $user->setCreatedAt(new \DateTimeImmutable());
        $user->setUpdatedAt(new \DateTimeImmutable());
```

```
        $user->setImageName('default.png'); // Avant d'aller plus loin
        $user->setPassword(
            $userPasswordHasher->hashPassword(
                $user,
                $form->get('plainPassword')->getData()
            )
        );

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

        // do anything else you need here, like send an email

        return $security->login($user, 'form_login', 'main');
    }

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

La vue registration.html.twig

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

{% block title %}Register{% endblock %}

{% block body %}
    <main class="section">
        <section class="row">
            <div class="col-md-6 offset-md-3">
                <h2 class="mb-4">S'enregistrer</h2>
                {{ form_errors(registrationForm) }}
                {{ form_start(registrationForm) }}
                {{ form_widget(registrationForm) }}
                <button type="submit" class="btn btn-outline-dark">Enregistrer</button>
                {{ form_end(registrationForm) }}
            </div>
        </section>
    </main>
{% endblock %}
```

Restreindre l'accès à l'administration

- Pour l'instant, nos utilisateurs ont tous le `ROLE_USER`, Ce qui est insuffisant pour la consultation du site web. Pour avoir accès à l'administration, le rôle minimum est le rôle `ROLE_ADMIN`. Nous pouvons en ajouter d'autres en fonction de nos besoins.
- Pour interdire l'accès aux utilisateurs ne disposant que du `ROLE_USER`, nous allons modifier l'accès control du fichier `security.yaml`
- Pour ce faire, vous devez décommenter la clé `path` pour le rôle admin.

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

- Maintenant il faudra obligatoirement le `ROLE_ADMIN` pour accéder à `AdminPostController`.
- Seuls les administrateurs peuvent poster, éditer et supprimer un article.
- Le `ROLE_USER` servira notamment à poster des commentaires sur les articles et gérer son profil par exemple.

Upload d'image dans le formulaire d'enregistrement

- Lors de l'upload d'une image, souvent en utilisant l'entité `User`, le message d'erreur ci-dessous s'affiche:

Message d'erreur lors de l'upload: Serialization of 'Symfony\Component\HttpFoundation\File\File' is not allowed

- Cette erreur survient généralement lorsqu'on tente de sérialiser un objet qui contient une instance de `UploadedFile`, une classe utilisée par Symfony pour gérer les fichiers téléchargés via des formulaires. En effet, `UploadedFile` ne peut pas être directement sérialisé en raison de sa nature complexe et des ressources temporaires qu'il représente. Le composant `Security` utilisé lors de la création de l'entité `User` a besoin de sérialiser les trois propriétés qu'il a créé automatiquement (id, email et password). Comme il ne peut plus le faire à cause de l'upload, nous devons lui indiquer dans l'entité de sérialiser et de désérialiser les trois propriétés.
- Ajoutez les deux méthodes de sérialisation à la fin de votre classe `User`.

```
public function __serialize(): array
{
    return [
        $this->id,
        $this->email,
        $this->password,
    ];
}

public function __unserialize(array $data): void
{
    [
        $this->id,
        $this->email,
        $this->password,
    ] = $data;
}
```

Poster un article avec la référence de l'auteur

Modifiez le contrôleur `PostController` pour ajouter le setter correspondant à l'utilisateur:

```
if($form->isSubmitted() && $form->isValid()) {  
    $post->setUser($this->getUser());  
    $post->setCreatedAt(new \DateTime());  
    $manager->persist($post);  
    $manager->flush();  
    return $this->redirectToRoute('posts');  
}
```

Sécuriser le contrôleur Post

- Pour finaliser la sécurité, vous devez encore empêcher l'accès aux méthodes (delPost, editPost et addPost) du contrôleur Post. En effet, il suffirait de saisir delpost/3 ou editpost/4 pour supprimer ou éditer des articles. Et ce sans être connecté !
- Il est possible de restreindre les accès aux fichiers via l'access_control de security.yaml ou de restreindre des méthodes directement dans le contrôleur concerné.
- Pour cela, il suffit d'ajouter une autorisation sous la forme d'une annotation. Vous devez aussi ajouter le use correspondant

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

```
/**  
 * @Route("/edit-{id}", name="edit")  
 * @IsGranted("ROLE_USER")  
 */
```


La pagination

KnplPaginatorBundle

La pagination

Pour éviter d'afficher la totalité des enregistrements sur la même vue, vous devez paginer vos données à l'aide d'un système d'affichage numéroté. Nous le réaliserons dans la vue affichant tous les articles. Il est possible de le développer avec Symfony mais il est préférable d'utiliser une librairie tierce. La plus connue est KnpPaginatorBundle:

<https://github.com/KnpLabs/KnpPaginatorBundle>

Principe:

- Installation: `composer require knplabs/knp-paginator-bundle`
- Ajoutez un fichier `knp_paginator.yaml` dans le dossier `config/packages`. Son contenu figure dans la doc et il contient les différents paramètres concernant la pagination. Nous les modifierons par la suite.
- Modifiez le contrôleur `PostController` pour y insérer le code permettant d'utiliser l'interface `PaginatorInterface`. Elle possède une seule méthode: `paginate()`.

```
#[Route('/posts', name: 'app_posts')]
public function posts(PostRepository $repository, Request $request, PaginatorInterface
$paginator): Response
{
    $posts = $repository->findBy(
        ['isPublished' => true],
        ['createdAt' => 'DESC'],
    );
    $pagination = $paginator->paginate($posts, $request->query->getInt('page', 1),
    10
    );
    return $this->render('post/posts.html.twig', [
        'posts' => $pagination,
    ]);
}
```

```
public function paginate(mixed $target, int $page = 1, ?int $limit = null, array $options = [])
```

La pagination

- Pour afficher le système de pagination dans la vue (post.html.twig) de la partie publique ajoutez la fonction `knp_pagination_render()`. Le paramètre `posts` correspond à celui passé par le controller. La pagination apparaît par défaut. Pour l'adapter à la sauce Bootstrap, modifiez votre configuration de template dans le fichier `knp_paginator.yml`.
- Ajoutez du CSS pour adapter le système de pagination (espaces, centrer...)
- Adaptez également la méthode `postsCat` selon la même technique (affichage des articles par catégorie).

```
<div>
  {{ knp_pagination_render(posts) }}
</div>
```

```
template:
  pagination:
    '@KnpPaginator/Pagination/bootstrap_v5_pagination.html.twig'  #
    sliding pagination controls template
    rel_links:
    '@KnpPaginator/Pagination/bootstrap_v5_bi_sortable_link.html.twig'
    # <link rel=...> tags template
```

```
<div class="pt-5 d-flex justify-content-center">
  {{ knp_pagination_render(posts) }}
</div>
```

La pagination

Par défaut le texte des boutons est en anglais. Pour les traduire, nous allons utiliser le module de traduction intégré dans Symfony.

1. Définir la variable locale dans le fichier `translation.yaml` (config/packages):

```
framework:  
  default_locale: fr
```

2. Dans le dossier translation, créez le fichier `KnpPaginatorBundle.fr.yaml`

3. Ajoutez les traductions suivantes:

```
label_next: "Suivant"  
label_previous: "Précédent"
```

4. Videz le cache

```
framework:  
  default_locale: fr
```

```
label_next: Suivant  
label_previous: Précédent
```

La validation des données

Principe

- Il est impératif de vérifier les données provenant d'un formulaire avant de les migrer en DB ou de les traiter. La validation en HTML et côté client en JavaScript n'est pas suffisante. Elle doit toujours être (aussi) réalisée en PHP côté serveur.
- Le but de la validation est de vous dire si les données d'un objet sont valides. Pour que cela fonctionne, vous allez configurer une liste de règles (appelées contraintes) que l'objet doit suivre pour être valide. Les règles de validation sont principalement ajoutées dans les **entités** mais on peut aussi les retrouver dans les **formTypes** ou dans certains **controllers**.
- Symfony propose le composant **Validator** permettant de mettre en place vos contraintes sur les propriétés de vos entités. Par exemple vérifier si l'utilisateur a bien rempli un champ dans le formulaire et que celui-ci contient au moins un certain nombre de caractères... Si la contrainte n'est pas respectée, un message d'erreur et éventuellement des explications seront affichés lors de la validation du formulaire.

L'entité Post

- Pour définir les règles de validation dans l'entité, nous allons utiliser le système d'attribut `#[]` pour les propriétés à valider. La première chose à savoir est le namespace des contraintes à utiliser. Souvenez-vous, pour le mapping Doctrine c'était `#[ORM\...]`, ici nous allons utiliser `#[Assert\...]`, dont le namespace complet est le suivant :
`use Symfony\Component\Validator\Constraints as Assert;`
- La classe `Constraints` sera utilisée avec l'alias `Assert` (Convention). Il ne reste plus maintenant qu'à écrire les règles de validation. Une fois de plus, la documentation officielle fournit les informations sur l'ensemble des classes disponibles permettant de valider en fonction du type de donnée (string, number, date...).
<https://symfony.com/doc/current/validation.html#validator-constraint-targets>
- Exemple:

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

L'entité Post

Voici quelques exemples de contraintes pour l'entité **Post**

```
#[Assert\NotBlank]
#[Assert\Length(
    min: 5,
    max: 255,
    minMessage: 'Le titre doit contenir au moins {{ limit }} caractères',
    maxMessage: 'Le titre doit contenir au maximum {{ limit }} caractères'
)]
#[ORM\Column(length: 255)]
private ?string $title = null;
```

```
#[Assert\NotBlank]
#[Assert\Length(
    min: 10,
    minMessage: 'Le contenu doit contenir au moins {{ limit }} caractères',
)]
#[ORM\Column(type: Types::TEXT)]
private ?string $content = null;
```


Autres exemples

- Avec une expression régulière:

```
#[Assert\Regex(  
  pattern: '/^[a-z]+$/',  
  message: 'Votre prénom contient des caractères non autorisés',  
)]  
#[ORM\Column(length: 120)]  
private ?string $firstName = null;
```

- Avec une valeur plus grande que:

```
#[Assert\GreaterThan( value: 18, )]
```

- Comparaison de date:

```
#[Assert\GreaterThan('today')]  
protected \DateTimeInterface $deliveryDate;
```

- La force d'un password:

```
#[Assert\PasswordStrength([  
  'minScore' => PasswordStrength::STRENGTH_VERY_STRONG, // Very strong password required ]  
)]
```

Rendre un champ unique

Cette technique consiste à interdire la saisie d'une donnée déjà existante dans la base de données. Par exemple, dans le cas d'un utilisateur, on ne peut pas avoir deux fois le même Email. Ou encore dans le cas d'un article, deux fois le même titre...

Ici on n'utilise pas les contraintes du composant Validator mais celle de Doctrine. Dans le cas d'une Unique Entity, vous devez la déclarer au niveau de l'entité et non pas de la propriété.

```
#[ORM\Entity(repositoryClass: PostRepository::class)]  
#[Vich\Uploadable]  
#[UniqueEntity(  
    fields: 'title',  
    message: 'Ce titre existe déjà !'  
)]
```

Le use de la classe devrait s'ajouter automatiquement dans votre éditeur:

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

Voici un autre cas d'utilisation (champs multiples):

```
#[UniqueEntity(  
    fields: ['email', 'phoneNumber'],  
    ignoreNull: 'phoneNumber']
```

Création d'un formulaire de contact et envoi d'Email.

Introduction

Tester via un Controller

Création de la classe Contact (entité non mappée)

Création de la classe ContactType (formulaire)

Création ou modification du contrôleur PageController

Création du service ContactService

Création de la vue avec le formulaire

Etapes à réaliser

- Installation d'une application pour récupérer les mails: MailPit
- Création d'une classe Contact
- Création du formType: ContactType
- Création d'un Contrôleur: ContactController

Nous contacter

Firstname

Lastname

Email

Subject

Message

Nous contacter

Introduction

- Pour permettre à un utilisateur d'envoyer un Email, Symfony dispose de deux composants **Mailer** et **Mime** qui constituent un système efficace pour créer et envoyer des e-mails, avec prise en charge des messages en plusieurs parties, intégration de Twig, intégration CSS, pièces jointes et bien plus encore. Consultez la documentation à cette adresse: <https://symfony.com/doc/current/mailer.html>.
- Depuis Symfony 7, le développement d'un service d'envoi de mails a été entièrement revu et amélioré. Faites attention dans vos recherches web car les informations trouvées peuvent être obsolètes.
- Lors du développement, pour tester l'envoi d'un Email, nous devons disposer d'un client qui se chargera de la réception et de la visualisation de celui-ci. Il en existe plusieurs et les plus utilisés sont: maildev (Docker), mailhog et mailpit. C'est ce dernier que je vais utiliser pour sa simplicité d'installation et son usage multi-plateformes.

Introduction

Installation de Mailpit via la documentation officielle: <https://mailpit.axllent.org/docs/install>

Le plus simple est de télécharger un exécutable à la rubrique "Download static binary (Windows, Linux and Mac"

Etapes:

- Décompressez l'archive et copier l'exécutable dans le dossier bin de votre projet.
- Exécutez dans votre terminal, la commande bin/mailpit
- Un message composé de trois parties apparaît:
 - level=info msg="[smtpd] starting on [::]:1025 (no encryption)"
 - level=info msg="[http] starting on [::]:8025"
 - level=info msg="[http] accessible via http://localhost:8025/"
- Cliquez sur le lien pour visualiser l'interface client dans votre navigateur ou saisissez <http://localhost:8025>.



Introduction

Faux problème de sécurité

Il est possible que votre firewall et votre antivirus se mettent en alerte lors de l'exécution de Mailpit (Avast).

- Pour le firewall, acceptez l'exécution de Mailpit.
- Pour l'antivirus, créez une exception pour l'exécutable mailpit.exe
- Relancez la commande `bin/mailpit`

Modifiez le fichier `.env` pour la configuration du Mailer DSN

```
###> symfony/mailer ###  
MAILER_DSN=smtp://localhost:1025  
###< symfony/mailer ###
```

Modifiez le fichier de configuration `messenger.yaml` pour retirer l'utilisation de la file d'attente dans l'envoi de mail en local. Commentez le routing pour l'envoi d'Email.

```
routing:  
  # Symfony\Component\Mailer\Messenger\SendEmailMessage: async  
  Symfony\Component\Notifier\Message\ChatMessage: async  
  Symfony\Component\Notifier\Message\SmsMessage: async
```


Tester l'envoi d'un Email via un Controller

Avant la création d'un formulaire de type Email, nous allons vérifier le fonctionnement des composants via un Controller. Nous l'utiliserons et l'adapterons également pour finaliser le service.

Étapes:

- `Php bin/console make:controller ContactController`
- Comme je l'ai déjà expliqué en introduction, nous utiliserons en dépendance de la méthode la `MailerInterface` qui nous permettra d'utiliser sa méthode `send()` et une instance de la classe `Email` pour la gestion des parties du mail (from, to, subject...). Les deux uses doivent être évidemment ajoutés.
- En ce qui concerne la vue, pour l'instant laissez la totalement vide.
- Testez l'envoi du mail dans maitip en saisissant l'url `contact`.

```
#[Route('/contact', name: 'app_contact')]
public function contact(MailerInterface $mailer): Response
{
    $email = (new Email())
        ->from('johndoe@gmail.com.com')
        ->to('info@webarticle.com')
        ->subject('Question')
        ->text('something...');
    $mailer->send($email);
    return $this->render('contact/contact.html.twig');
}
```

Création de la classe «Contact»

- Nous allons définir l'ensemble des propriétés nécessaires pour la gestion d'un formulaire de contact dans notre application (contact simplifié). L'entité ne sera pas reliée à la base de données et ne contiendra pas l'annotation @ORM permettant de réaliser le mapping entre l'entité et la base de données. Ce qui implique que l'entité sera créée manuellement. Cette classe que nous appellerons src/Class/**Contact.php** est nécessaire pour la représentation sous la forme d'objets des contenus du formulaire.
- Nous ajouterons aussi une série de contraintes permettant la validation des informations avant l'envoi du formulaire.
- Pour ne pas surcharger les données, je n'ai pas spécifié les messages en cas d'erreur de saisie. Ceux-ci seront affichés par défaut (anglais). Cette pratique a déjà été abordée dans les diapos précédentes.
- **Étapes pour la création du ContactType:**
 - > php bin/console make:form
 - The name of the form class:
 - > ContactType
 - The name of Entity or fully qualified model class name that the new form will be bound to (empty for none):
 - > \App\Class>Contact
- La dia suivante représente l'entité Contact sauf les getters et setters générés automatiquement par PhpStorm.

Classe Contact

```
namespace App\Class;

use Symfony\Component\Validator\Constraints as Assert;

class Contact
{
    #[Assert\NotBlank]
    private string $firstName;
    #[Assert\NotBlank]
    private string $lastName;
    #[Assert\NotBlank]
    #[Assert\Email(
        message: 'Votre Email {{ value }} n\'est pas valide.',
    )]
    private string $email;
    #[Assert\NotBlank]
    private string $subject;
    #[Assert\NotBlank]
    private string $message;

    // GETTERS & SETTERS

}
```

Création de la classe ContactType

Le `ContactType` va nous permettre de gérer le formulaire. Comme nous en avons déjà réalisé plusieurs, celui-ci va être simplifié. N'oubliez pas de vérifier les usages en conformité avec les types des champs de formulaire.

La méthode `configureOptions()` permet de définir la classe `Contact` comme étant le modèle pour la création du formulaire: `fully qualified model class`.

```
public function buildForm(FormBuilderInterface $builder, array $options): void
{
    $builder
        ->add('firstName', TextType::class, [
            'label' => 'Votre prénom',
        ])
        ->add('lastName', TextType::class, [
            'label' => 'Votre nom',
        ])
        ->add('email', EmailType::class, [
            'label' => 'Votre email',
        ])
        ->add('subject', TextType::class, [
            'label' => 'Votre sujet',
        ])
        ->add('message', TextareaType::class, [
            'label' => 'Votre message',
        ])
    ;
}
```

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

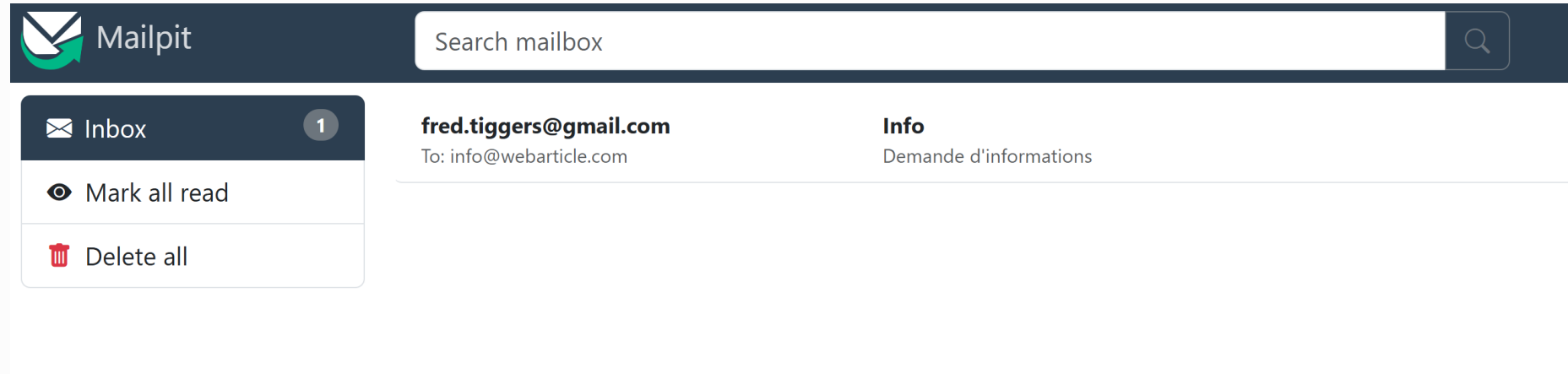
ContactController et la méthode contact() finalisée

```
#[Route('/contact', name: 'app_contact')]
public function contact(MailerInterface $mailer, Request $request): Response
{
    $contact = new Contact();
    $form = $this->createForm(ContactType::class, $contact);
    $form->handleRequest($request);
    if($form->isSubmitted() && $form->isValid()) {
        $email = (new Email())
            ->from($contact->getEmail())
            ->to('info@webarticle.com')
            ->subject($contact->getSubject())
            ->text($contact->getMessage());
        $mailer->send($email);
        return $this->redirectToRoute('app_home');
    }
    return $this->render('contact/contact.html.twig', [
        'form' => $form,
    ]);
}
```

La vue contact.html.twig

```
{% block body %}
    <section class="row">
        <div class="col-md-6 offset-3">
            <h2 class="display-4 my-4">Nous contacter</h2>
            {{ form_start(form) }}
            {{ form_widget(form) }}
            <input type="submit" value="Envoyer">
            {{ form_end(form) }}
        </div>
    </section>
{% endblock %}
```

Le résultat dans MailTip



Les tests avec PHPUnit

Introduction

Mise en place des tests dans Symfony

Création d'un premier test unitaire

Test unitaires sur une méthode

Test fonctionnel sur une vue

Tester une entité

Pourquoi tester ?

- Lorsque vous développez une fonctionnalité dans votre projet, la première chose que faites c'est de tester manuellement son bon fonctionnement dans l'interface. Par exemple, j'ai créé un formulaire d'inscription (register) et je vais le tester en remplissant les différents champs pour ensuite soumettre le formulaire. je réalise ainsi un test manuel qui peut ne pas couvrir l'ensemble des erreurs éventuelles et qui ne sera exécuté qu'à un seul instant dans le développement.
- Si par la suite d'autres fonctionnalités sont ajoutées dans le projet (maintenance, développements futurs...) il est possible que ce formulaire ne soit plus opérationnel (régression) et je ne m'en pas compte immédiatement. Si c'est un autre développeur qui réalise le travail, il est probable qu'il n'y pensera peut-être pas.
- Réalisez des tests automatiques (programmés) vous permettra de vous assurer du bon fonctionnement de vos composants et si du développement est ajouté, le développeur lancera le processus de tests que vous avez programmé pour vérifier s'il n'a rien cassé (régression) dans l'application.

Quand tester ?

- Les tests programmés peuvent être réalisés à différents moments du développement:
- Test First (assure une bonne logique dans le développement).
- Begin dev and test (principe le plus utilisé).
- End test (utilisé surtout lors de l'utilisation de librairies tierces).

Quels sont les différents types de tests ?

Il existe plusieurs types de tests pour une application ou un site web. Nous aborderons uniquement les principaux qui concerne le développement web.

- Les tests unitaires
- Les tests fonctionnels
- Les tests end to end (E2E)

Configuration

- Avec ou sans Symfony, les tests sont réalisés à l'aide de la librairie "**PHPUnit**". Celle-ci est intégrée automatiquement si vous avez optez pour l'installation **–full** de Symfony. L'exécutable se trouve dans le dossier bin de votre projet. Un dossier **tests** figure dans l'arborescence ainsi que deux fichiers: **.env.test** et **phpunit.xml.dist**
- Pour vérifier le bon fonctionnement, lancez la commande: **php bin/phpunit**
- Le message "No tests executed!" apparaît indiquant qu'aucun test n'a été exécuté.

Étapes:

Avant de commencer, consultez la documentation de Symfony concernant les tests:

<https://symfony.com/doc/current/testing.html#unit-tests> et celle de PHPUnit:

<https://docs.phpunit.de/en/9.6/index.html>

- Lancez la commande `php bin/console make:test` pour créer votre premier test.

```
Which test type would you like?:
[TestCase      ] basic PHPUnit tests
[KernelTestCase] basic tests that have access to Symfony services
[WebTestCase   ] to run browser-like scenarios, but that don't execute JavaScript code
[ApiTestCase   ] to run API-oriented scenarios
[PantherTestCase] to run e2e scenarios, using a real-browser or HTTP client and a real web server
> 
```

- Choisissez l'option `TestCase` qui correspond aux tests unitaires. Vous devez l'écrire dans le terminal. Nous aborderons les autres options par la suite.

Etapas:

- Vous obtenez les propositions suivantes: Ecrivez le nom de la classe de test et suffixez avec le mot Test (obligatoire).

```
Choose a class name for your test, like:
* UtilTest (to create tests/UtilTest.php)
* Service\UtilTest (to create tests/Service/UtilTest.php)
* \App\Tests\Service\UtilTest (to create tests/Service/UtilTest.php)

The name of the test class (e.g. BlogPostTest):
> SimpleTest
```

- La classe SimpleTest a été ajoutée dans le dossier tests du projet.

```
<?php
namespace App\Tests;
use PHPUnit\Framework\TestCase;

class SimpleTest extends TestCase
{
    public function testSomething(): void
    {
        $this->assertTrue(true);
    }
}
```

La classe SimpleTest hérite de TestCase qui elle-même hérite de Assert (classes de PHPUnit) nous donnant accès à toutes les méthodes assert...() dont assertTrue() par défaut. Cette méthode signifie que tous les paramètres doivent obligatoirement être vrais. C'est le cas ici.

Etapas:

Pour effectuer le test, lancez la commande php bin/phpunit

```
Testing
.
1 / 1 (100%)

Time: 00:00.107, Memory: 8.00 MB

OK (1 test, 1 assertion)
```

Le test a été effectué avec succès (forcément). Les informations retournées sont:

- Un point en dessous de testing qui signifie qu'un seul test a été réalisé (une seule méthode de test).
- 1/1 c'est la même chose.
- Temps et mémoire utilisée.
- Confirmation de réussite du test.

Changez true par false, testez à nouveau et vous verrez que le test a échoué:

```
FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

TEST UNITAIRE SUR METHODE

Nous allons créer une classe qui calcule la tva pour un panier par exemple. Il s'agit juste de tester une méthode qui ne possède aucune dépendance ou aucun service. C'est le cas le plus simple.

Ajoutez la classe `Tva` dans le dossier `src/Class`. Elle contiendra un constructeur et une méthode pour calculer la TVA (uniquement le calcul de la tva).

Dans la classe `SimpleTest`, ajoutez la méthode de test `testTva()` pour vérifier l'intégrité du résultat. Utiliser la méthode d'assertion `assertEquals()` pour votre test. Choisissez les valeurs que vous souhaitez pour la vérification.

Testez avec la commande `phpunit` ou faites un "run test" dans PhpStorm.

```
class Tva
{
    public function __construct(private readonly float $price, private readonly float $rate){
    }
    public function getTva(): float
    {
        return $this->price * $this->rate /100;
    }
}
```

```
public function testTva(): void
{
    $tva = new Tva(1000, 21);
    $this->assertEquals(210, $tva->getTva());
}
```


TEST UNITAIRE SUR METHODE

Ajoutons une condition dans le calcul de la tva afin de vérifier si les valeurs passées dans le constructeur sont bien positives.

Maintenant, la méthode `getTva()` possède deux returns ce qui implique que devrez pour la tester créer deux méthodes de test. Une pour chaque return.

Voici le résultat attendu:

```
Testing
..

Time: 00:00.038, Memory: 8.00 MB

OK (2 tests, 2 assertions)
PS C:\wamp64\www\symfony2024\profwebposts>
```

```
public function getTva()
{
    if($this->rate < 0 || $this->price <= 0) {
        return 'Les valeurs ne doivent pas être négatives !';
    }
    return $this->price * $this->rate / 100;
}
```

```
public function testTvaNegative(): void
{
    $tva = new Tva(-100, 21);
    $this->assertSame('Les valeurs ne doivent pas être négatives !', $tva->getTva());
}

public function testTvaResult(): void
{
    $tva = new Tva(100, 21);
    $this->assertEquals(21, $tva->getTva());
}
```

Etapas:

- Commande: `php bin/console make:test`
- Choisissez `WebTestCase`
- Pour l'exemple, entrez comme nom de classe `PageTest`
- Lors du lancement, le test va retourner une failure et le message d'erreur suivant:

```
<!-- An exception has been thrown during the rendering of a template ("An exception occurred in the driver: SQLSTATE[HY000] [1049] Base 'profwebposts_test' inconnue"). (500 Internal Server Error) -->
```
- Pour les tests fonctionnels, vous devez disposer d'une base de données de test. Il s'agira d'une copie de votre DB utilisée pour des raisons de sécurité uniquement en phase de test.

Copier la variable d'environnement `DATABASE_URL` de votre `.env` vers le fichier `.env.test`

Créer la base de données de test avec la commande:

`php bin/console d:d:c --env=test`

Maintenant votre base de données est créée (vide) et est suffixée `_test`.

```
class PageTest extends WebTestCase
{
    public function testAbout(): void
    {
        $client = static::createClient();
        $client->request('GET', '/about');

        $this->assertResponseSuccessful();
        $this->assertSelectorTextContains('h1', 'About Us');
    }
}
```

TEST FONCTIONNEL SUR UNE VUE

- Saisissez la commande `php bin/console d:m:m --env=test`
- La base de données contient maintenant les tables de votre application mais aucune données. Pour les pages "statiques" les données ne sont pas nécessaires.
- Relancez le test.
- Chargez les fixtures en mode test nous en aurons besoin pour la suite: `php bin/console d:f:l --env=test`

Explication sur la méthode:

- La classe `PageTest` hérite de `WebTestCase` qui hérite de `KernelTestCase` et elle-même de `TestCase`. De ce fait, vous pourrez utiliser les méthodes de cette dernière comme nous l'avons vu avec les tests unitaires.
- La méthode `createClient()` crée un objet de type `"KernelBrowser"` qui correspond à un navigateur virtuel qui permettra de gérer les données d'une page à tester.

- La méthode `request()` de l'AbstractBrowser récupère les données et prend deux paramètres (la méthode et l'url).
- La méthode `assertResponseSuccessful()` s'attend à recevoir le code http 200
- La méthode `assertSelectorTextContains()` vérifie des contenus par rapport à des sélecteurs (balises)

PageTest (ici vous pouvez tester vos différentes vues)

```
public function testAbout(): void
{
    $client = static::createClient();
    $client->request('GET', '/about');

    $this->assertResponseSuccessful();
    $this->assertSelectorTextContains('h1', 'About Us');
}
```

TESTER UNE ENTITE (Post)

Tester les insertions de données (setters) et les règles de validation

Utilisation du `KernelTestCase` pour accéder aux services de Symfony dont notamment le Kernel.

```
Php bin/console make:test  
post\PostEntityTest
```

La méthode `testPostValid()`

Dans vos tests d'intégration, vous devez récupérer le service depuis le conteneur de services pour appeler une méthode spécifique. Après le démarrage du noyau (`self::bootKernel()`), le conteneur est renvoyé par `static::getContainer()`. Dans cet exemple, le service appelé sur le container est "doctrine".

Ensuite on récupère les objets Category de la même manière que dans les fixtures et on instancie la classe Post.

Maintenant, on peut tester les setters de l'entité Post, vérifier à l'aide du service "validator" les éventuelles erreurs.

L'assertion `assertCount()` confirme le nombre d'erreurs attendues.

```
class PostEntityTest extends KernelTestCase  
{  
    public function testPostValid(): void  
    {  
        self::bootKernel();  
        $container = static::getContainer();  
        $manager = $container->get('doctrine')->getManager();  
        $categories = $manager->getRepository(Category::class)->findAll();  
        $post = (new Post())  
            ->setTitle('Lorem ipsum')  
            ->setContent('Lorem ipsum dolor sit amet')  
            ->setCreatedAt(new \DateTimeImmutable())  
            ->setUpdatedAt(new \DateTimeImmutable())  
            ->setPublished(true)  
            ->setCategory($categories[array_rand($categories)])  
            ->setimage('test.jpg');  
        $error = $container->get('validator')->validate($post);  
        $this->assertCount(0, $error);  
    }  
}
```

TESTER UNE ENTITE (PostEntity)

Dans la même classe de test (PostEntityTest) nous allons tester deux règles de validation (NotBlank et Length). Comme la pratique est similaire pour toutes les contraintes, deux exemples suffiront.

La méthode testTitleInvalid()

- Dans l'entité Post, nous avons ajouter deux contraintes sur la propriété \$title.

```
#[Assert\NotBlank]
#[Assert\Length(
    min: 5,
    max: 255,
    minMessage: 'Le titre doit contenir au moins {{ limit }} caractères',
    maxMessage: 'Le titre doit contenir au maximum {{ limit }} caractères'
)]
```

- Nous allons créer un test qui en principe doit nous retourner deux erreurs dans la méthode assertError(). Si le titre est vide et s'il ne contient pas entre 5 et 255 caractères.

```
public function testTitleInvalid(): void
{
    self::bootKernel();
    $container = static::getContainer();
    $manager = $container->get('doctrine')->getManager();
    $categories = $manager->getRepository(Category::class)->findAll();
    $post = (new Post())
        ->setTitle('')
        ->setContent('Lorem ipsum dolor sit amet')
        ->setCreatedAt(new \DateTimeImmutable())
        ->setUpdatedAt(new \DateTimeImmutable())
        ->setPublished(true)
        ->setCategory($categories[array_rand($categories)])
        ->setimage('test.jpg');
    $error = $container->get('validator')->validate($post);
    $this->assertCount(2, $error);
}
```

Cette méthode est quasi identique à la précédente sauf dans le cas du `setTitle('')` et dans la méthode `assertCount(2, $error)`. Une refactorisation du code s'impose ou d'un changement de stratégie.

TESTER UNE ENTITE (Post)

Refactorisation de la classe `PostEntityTest`.

On constate deux blocs de code similaires:

La gestion du Kernel et du container.

L'instanciation de l'entité Post et de ses setters.

Refactorisation:

Nous allons ajouter trois méthodes qui seront réutilisées pour tous les test de la classe: `getKernel()`, `getEntity()` et `assertError()`. Commençons par ajouter deux propriétés dans notre classe de test:

```
private object $manager;  
private object $container;
```

Ensuite, la méthode `getKernel()`

```
public function getKernel(): Object  
{  
    self::bootKernel();  
    $this->container = static::getContainer();  
    return $this->manager = $this->container->get('doctrine');  
}
```

La méthode `getEntity()`

```
public function getEntity(): Post  
{  
    $this->getKernel();  
    $categories = $this->manager->getRepository(Category::class)->findAll();  
    $post = new Post();  
    $post->setTitle('Lorem ipsum')  
        ->setContent('Lorem ipsum dolor sit amet')  
        ->setCreatedAt(new \DateTimeImmutable())  
        ->setUpdatedAt(new \DateTimeImmutable())  
        ->setPublished(true)  
        ->setCategory($categories[array_rand($categories)])  
        ->setimage('test01.jpg');  
    return $post;  
}
```

La méthode `assertError()`

```
public function assertError (int $numberError, Post $post): void  
{  
    $error = $this->container->get('validator')->validate($post);  
    $this->assertCount($numberError, $error);  
}
```

TESTER UNE ENTITE (Post)

Maintenant, nous allons pouvoir éviter du code redondant dans les méthodes de test.

```
public function testPostValid(): void
{
    $this->getKernel();
    $this->getEntity();
    $this->assertError(0, $this->getEntity());
}
```

```
public function testTitleInvalid(): void
{
    $this->getKernel();
    $post = $this->getEntity()->setTitle("");
    // var_dump($post);
    $this->assertError(2, $post);
}
```

```
public function testTitleUniqueInvalid(): void
{
    $this->getKernel();
    $post = $this->getEntity()->setTitle('beatae consequuntur sit. ');
    // var_dump($post);
    $this->assertError(1, $post);
}
```

Généralement, les développeurs vont tester pour chaque méthode les setters(). Mais comme, ils ont déjà été effectués dans la méthode testPostValid, il n'est plus obligatoire de les tester sur les méthodes de validation (constraints). Seule la propriété concernées devrait être testée (gain de temps, moins de répétitions)

Les requêtes DQL

Utilisation d'un repository pour la création des requêtes

Si on prend le cas du `PostController` et de sa méthode `posts()`, on constate que nous avons utiliser la méthode magique `findBy()` pour récupérer tous les articles dont la propriété `IsPublished` est `true` et le tout dans l'ordre du plus récent au plus ancien.

Cela ne me pose pas de problème dans le cas d'un `findAll()` mais dans le cas du `findBy()` vous allez devoir écrire une série de paramètres pour affiner votre requête. L'exemple ici est encore assez simple.

```
$posts = $repository->findBy(  
    ['isPublished' => true],  
    ['createdAt' => 'DESC'],  
);
```

Deux problème se pose dans ce cas:

1. Si vous devez utiliser la même requête pour plusieurs vue elle devra être réécrite dans chaque méthode.
2. Le rôle d'un controller dans le modèle MVC est d'envoyer les données à la vue mais pas d'y écrire les requêtes. C'est le rôle du model et dans Symfony plus particulièrement du Repository.

Dans mon controller, je préféreraifaire appel à une méthode (requête) personnelle.

```
$posts = $repository->findPublished();
```

Utilisation d'un repository pour la création des requêtes

Ouvrez le PostRepository pour y ajouter la méthode findPublished()

Voici son implémentation de base:

La méthode createQueryBuilder(string \$alias, ?string \$indexBy = null) prend ici un seul parametre qui correspond à l'alias de l'entité. Elle permet de créer la requête personnalisée.

Les autres méthodes proviennent du QueryBuilder:

- >where (condition)
- >orderBy (tri)
- >getQuery (va générer la requête)
- >getResult (retourne les résultats de la requête)

```
public function findPublished(): array
{
    return $this->createQueryBuilder('p')
        ->where('p.isPublished = true')
        ->orderBy('p.createdAt', 'DESC')
        ->getQuery()
        ->getResult();
}
```