



# Introduction

Les frameworks

Le MVC

Installation de Symfony

Création d'un projet de démonstration

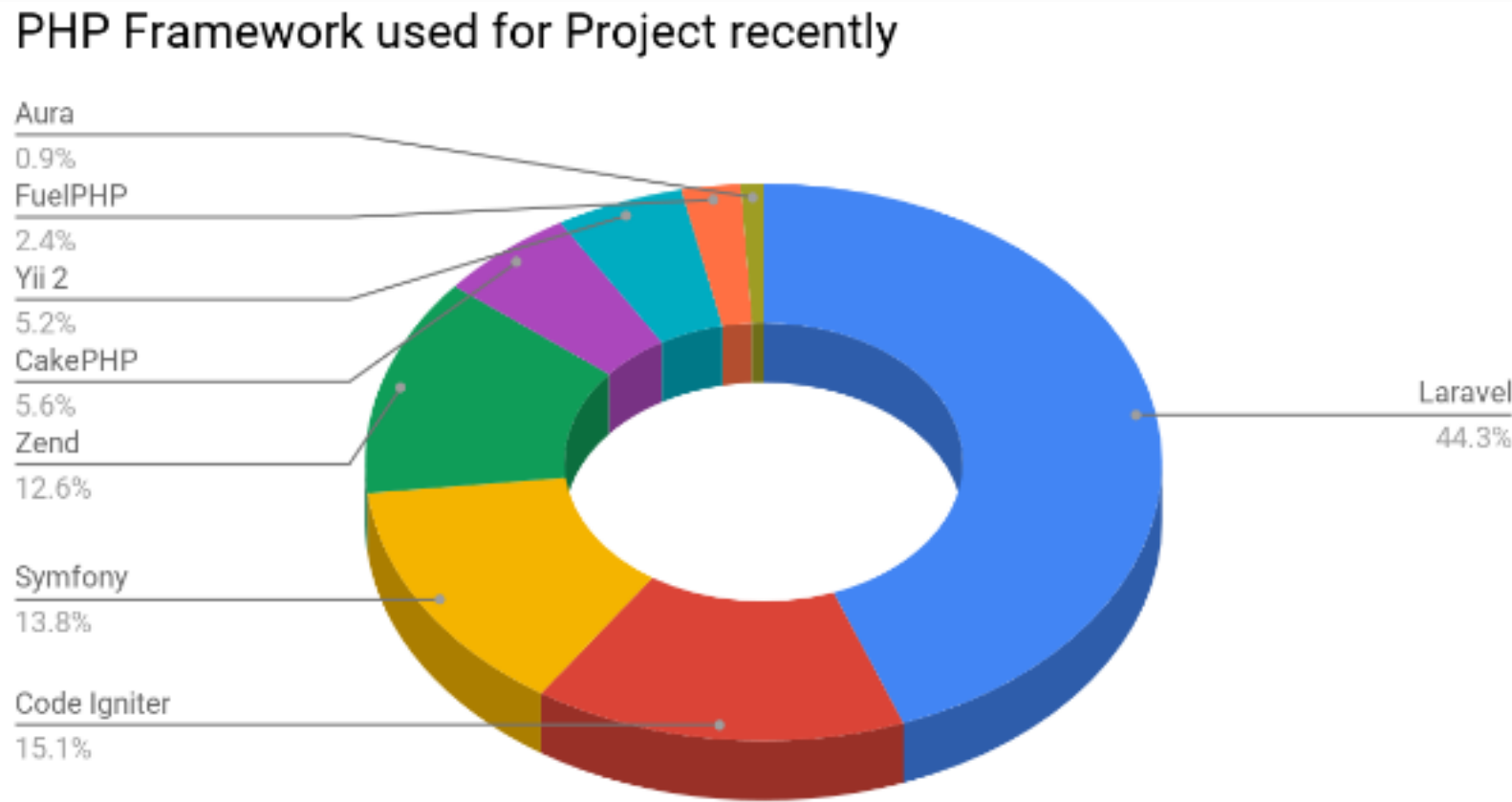
Les environnements de développement et de production

Notions supplémentaires

# Principaux frameworks 2020



# Principaux frameworks 2020



# 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.

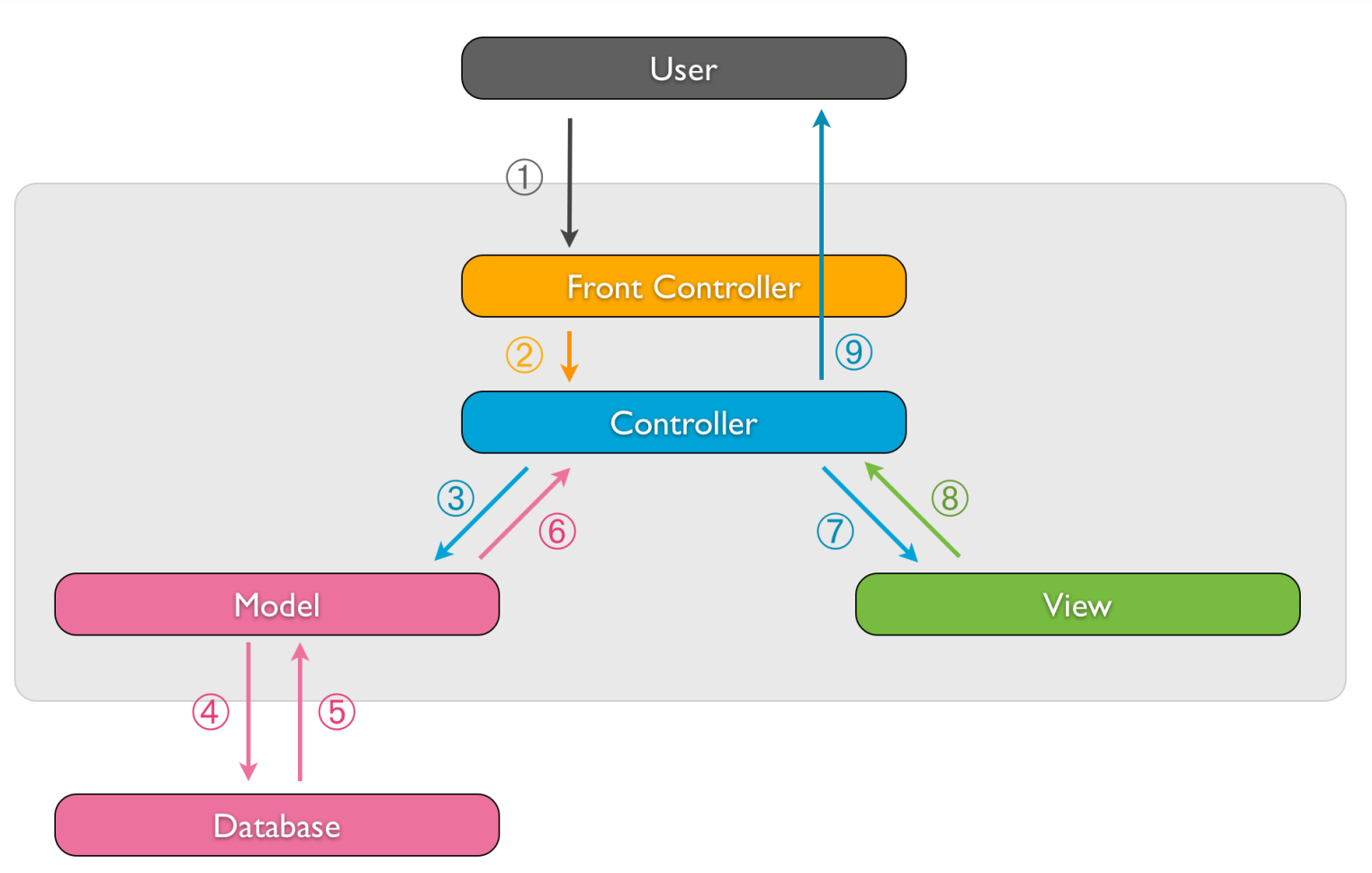
Il inclut généralement de nombreuses fonctionnalités prêtes à l'emploi dont les implémentations sont bien rodées et utilisent des modèles de conception standard et réputés. Le temps ainsi gagné sur les questions génériques pourra être mis à profit pour les parties spécifiques de l'application.

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 terme 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://knpbundles.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

# 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 officielle 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.

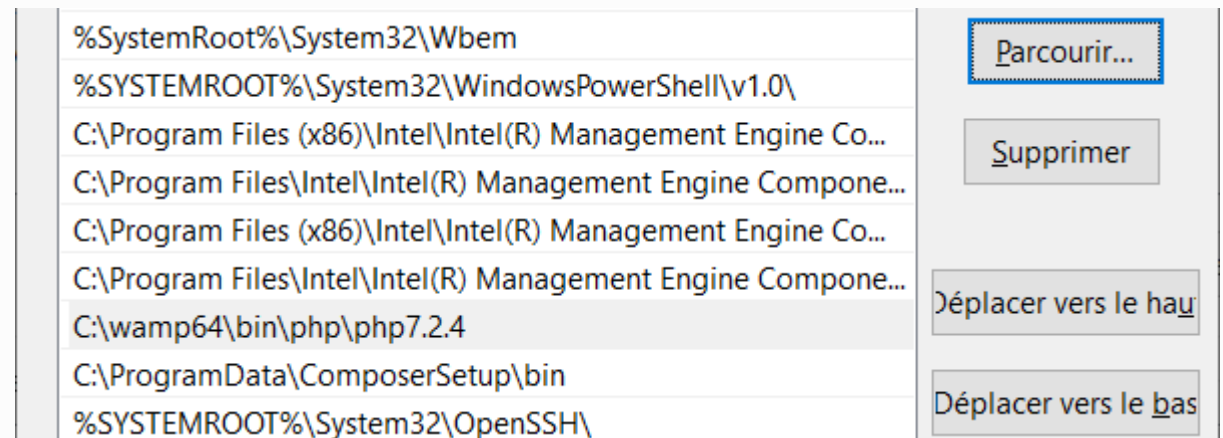
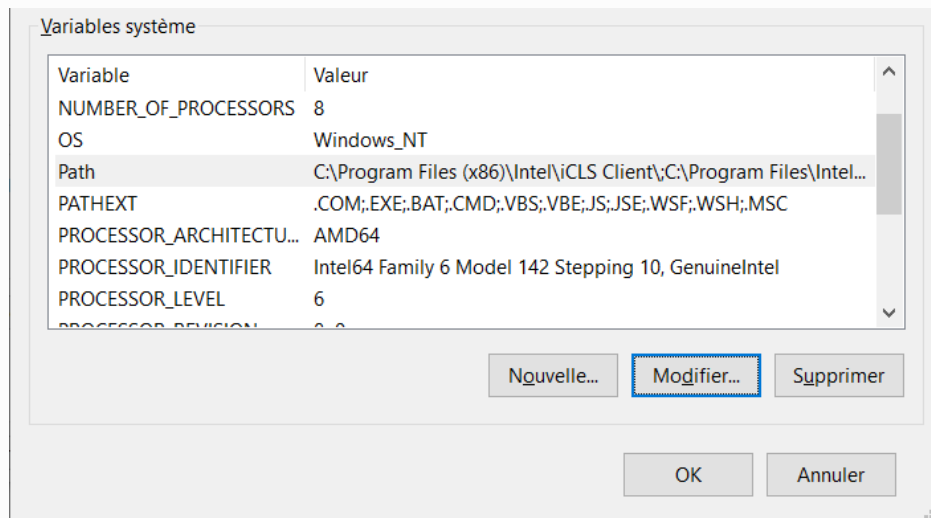
1. Avant l'installation, créez un dossier Symfony pour tous vos projets dans votre localhost (www).
2. Sur le site officiel (<https://symfony.com/download>) téléchargez “setup.exe” et choisissez votre système d'exploitation.
3. Exécutez l'application et vérifiez si l'option “Add application directory to your path” est bien activée.
4. Ouvrez une fenêtre PowerShell (invite de commande) à partir du dossier Symfony (Shift + clic droit).
5. Saisissez la commande: `symfony new --webapp practice` (practice 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 server 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.

# 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 recherchez l'exécutable de PHP.





# Le problème de version de PHP – Symfony server

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 > .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
```

# 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.

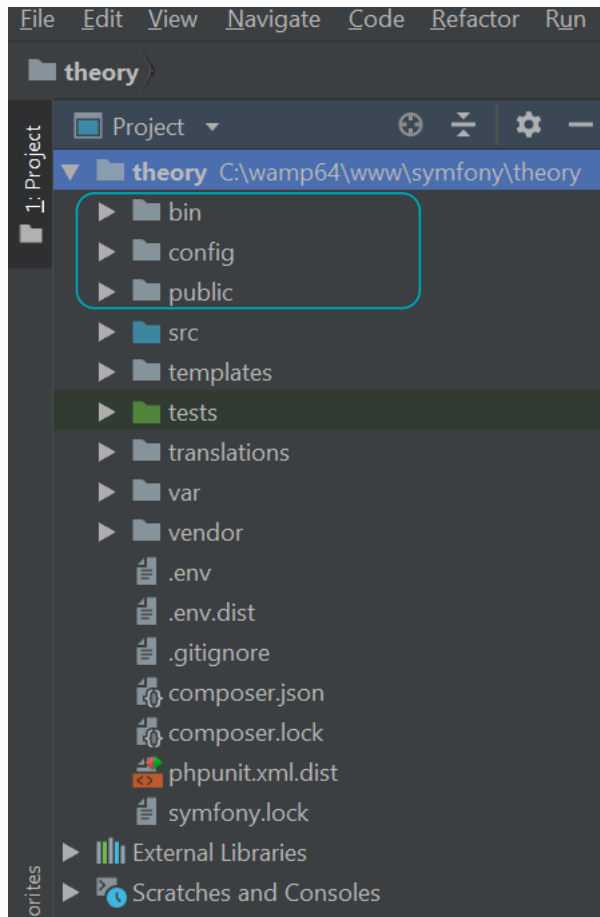
Si vous souhaitez en configurer une malgré tout, vous pouvez configurer et installer une base de données vide uniquement pour ce projet ou alors configurer une base de données existantes.

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

**DATABASE\_URL="mysql://root:@127.0.0.1:3306/weblearning?serverVersion=5.7"**

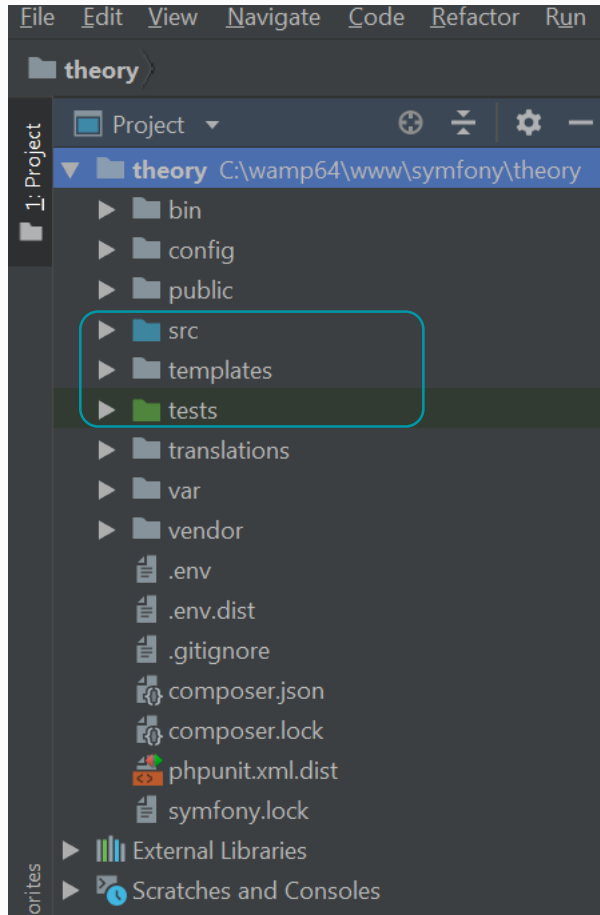
**Weblearning représente la base de données.**

# l'architecture d'un projet vide



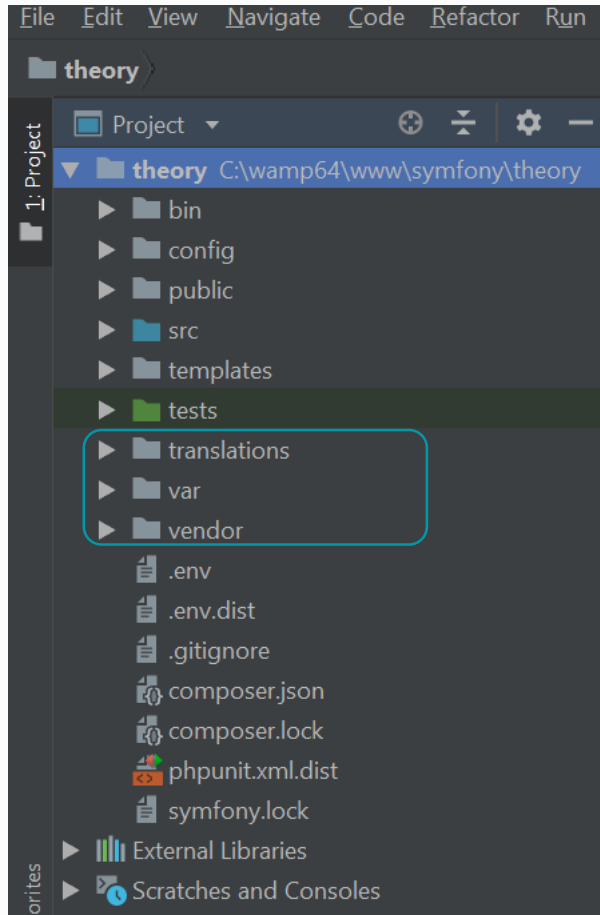
- **bin**: ce répertoire contient l'exécutable dont nous allons nous servir en mode console pendant le développement. La commande pour y accéder est la suivante: `php bin/console`. Vous avez également la commande pour les tests unitaires.
- **config**: contient tous les fichiers de configuration au format yaml (routing, security, configuration de développement ou de production,...).
- **migrations**: (non visible sur cette copie d'écran) 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 des dossiers d'un projet vide



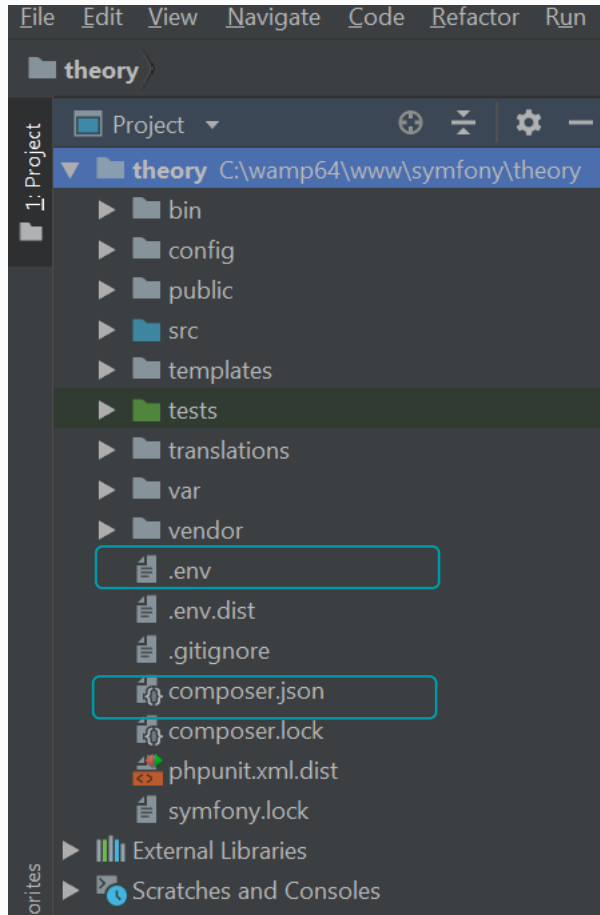
- **src**: ce dossier comporte toute l'architecture php de votre site web avec les Controllers, les Entités, les Repositories, les forms (que nous créerons par la suite). Il correspond au namespace **App** défini dans l'autoloader de Composer.
- **templates**: Ici, se trouvent toutes les vues, les templates 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 des dossiers 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 des dossiers 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 fichier composer.json**: il contient la liste des dépendances de votre projet. Utile pour le transfert, le partage et les mises à jour.



# 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

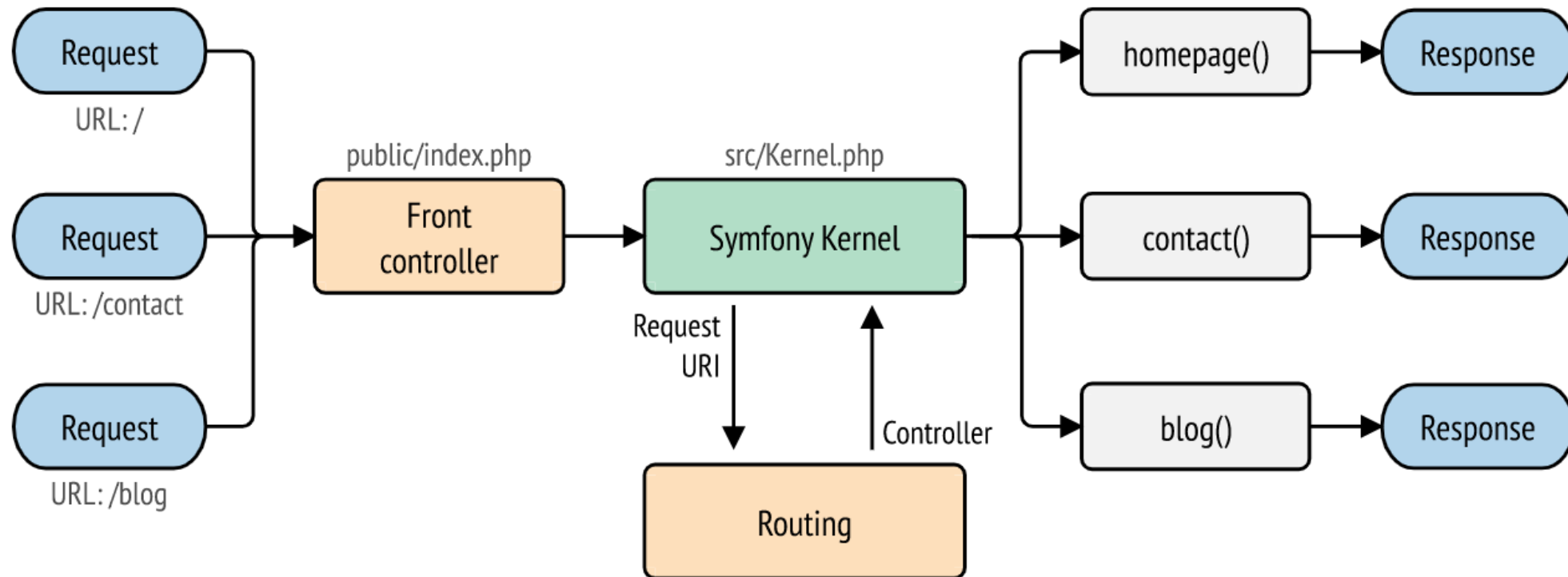
Syntaxe

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 orienté 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)

# 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 long 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'annotations:

```
/**  
 * @Route("path", name="nom")  
 */
```

```
/**  
 * @Route("/", name="home")  
 */
```

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\Annotation\Route;
```

# Routing, une autre pratique

- Il existe plusieurs techniques pour la création de vos routes. Vous pouvez les configurer en YAML, XML, PHP ou en utilisant des annotations ou des attributs. Symfony recommande l'une des deux dernières. Dans l'exemple précédant, c'est l'annotation qui a été introduite, mais à partir de Symfony 5.2, par défaut c'est maintenant les routes configurées sous la forme d'attributs qui sont privilégiées.
- Les exemples dans les notes sont encore largement utilisés avec les annotations (toujours largement utilisées) mais vous pouvez les adapter avec les attributs.
- Exemples

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

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

# 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 `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 action

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

# Contrôleur finalisé

```
namespace App\Controller;

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

class HomeController extends AbstractController
{
    /**
     * @Route("/", name="home")
     */
    public function index()
    {
        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 (Twig), 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>
    {% block stylesheets %}{% endblock %}
  </head>
  <body>
    {% block body %}{% endblock %}
    {% block javascripts %}{% 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 !
- Le contenu doit être insérer 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;

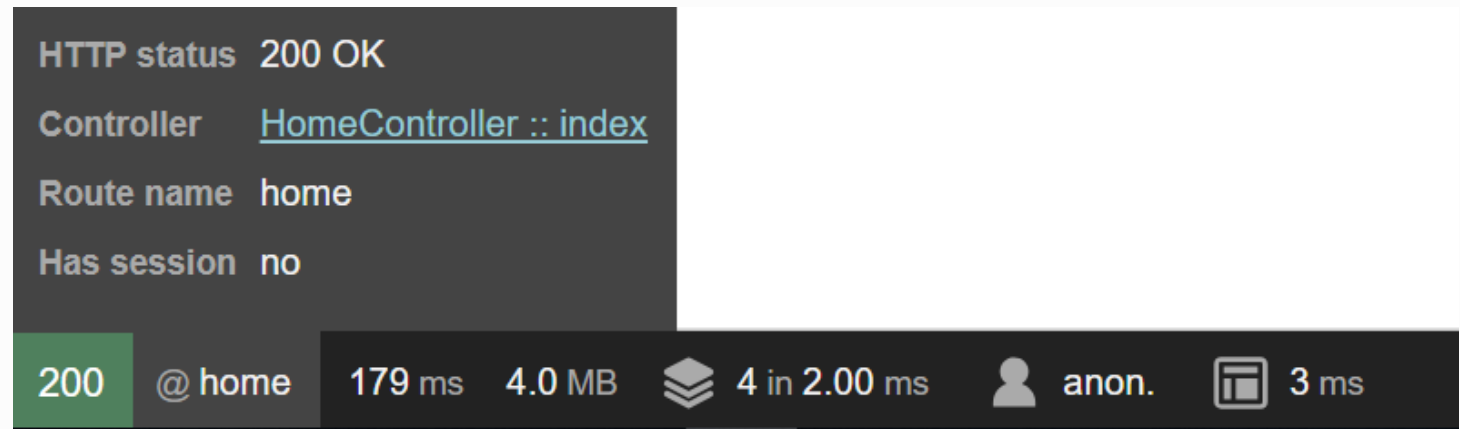
401 : utilisateur non authentifié;

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/'. A dark green bar below that contains 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 a search bar and buttons for 'Last 10', 'Latest', and 'Search'. Below these are icons and labels for various profiler sections: 'Request / Response' (selected), 'Performance', 'Validator', 'Forms', 'Exception', 'Logs', 'Events', 'Routing', and 'Cache'.

The main content area shows the 'HomeController :: index' action. Below the action name are tabs for 'Request', 'Response' (selected), 'Cookies', 'Session', and 'Flashes'. The 'Response Headers' section is displayed as a table:

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

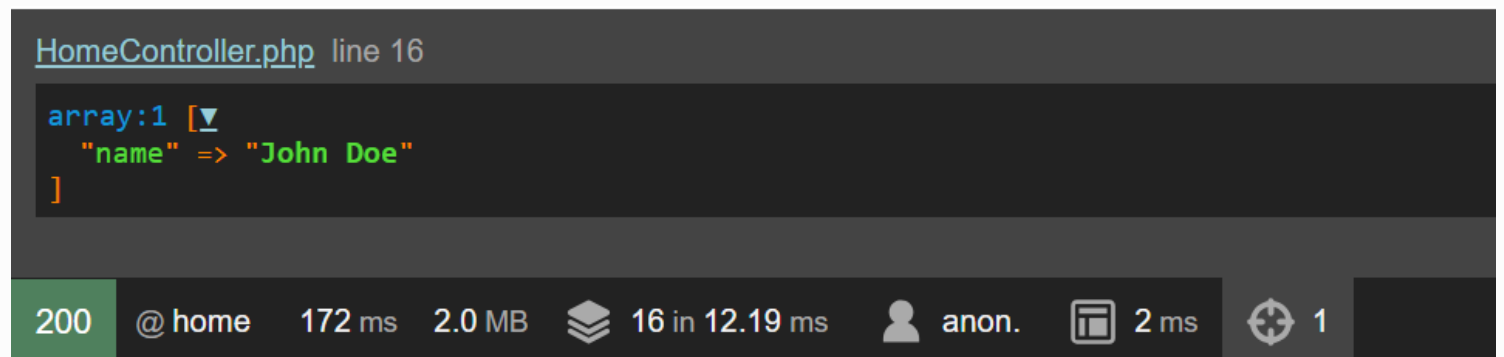
- Nous utilisons régulièrement la fonction `var_dump()` qui affiche les informations structurées d'une variable, y compris son type et sa valeur. Nous la faisons suivre d'une instruction d'arrêt pour afficher le contenu sans exécuter le reste du script.

**`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 this, 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, and other details like 16 in 12.19 ms, user anon., and a 2 ms execution time for the current request.

```
HomeController.php line 16  
array:1 [▼  
    "name" => "John Doe"  
]
```

200 @ home 172 ms 2.0 MB 16 in 12.19 ms anon. 2 ms 1



# 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("/", name="home")
```

Accès à la racine (index)

```
@Route("/products", name="products")
```

Ajout d'un niveau supplémentaire

```
@Route("/{category}", name="category")
```

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

```
@Route("/{category}", name="category", defaults={"category"=null})
```

Le paramètre n'est pas obligatoire

```
/**
 * @Route (
 *    ("/{id}",
 *     name="category",
 *     defaults={"category"=null},
 *     requirements={"id"="\d+"}
 * )
 */
```

Requirements permet de valider une expression régulière (valeur numérique de n'importe quelle longueur)

# Récupération du paramètre

```
<?php

namespace App\Controller;

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

class HomeController extends AbstractController
{
    /**
     * @route("/{login}", name="home")
     */

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

# 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', ['login' => $login]);
```

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: {{ login }}</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 tout 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 toutes 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. Il 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

# 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()
{
    $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.
  - Créez un dossier "Service" dans le répertoire "src"
  - Créez une classe portant le nom du service "Utils"
  - 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(Utils $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 averti 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 DocBlock.

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

# Installation d'un framework CSS (1)

1. Téléchargez Bootstrap (Compiled CSS and JS)
2. Créez deux dossiers dans public (css et js)
3. Copier les fichiers « bootstrap.min.css » et « bootstrap.min.js »
4. Créez les liens vers le framework Bootstrap dans le fichier « base.html.twig » (-> héritage)

## Le lien css

```
{% block stylesheets %}  
    <link rel="stylesheet" href="{{ asset('css/bootstrap.min.css') }}">  
{% endblock %}
```

# Installation d'un framework CSS (2)

Le lien JS

```
{% block javascripts %}  
    <script src="{{ asset('js/bootstrap.bundle.min.js') }}"></script>  
{% endblock %}
```

# 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")
```

# 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 les articles. Les publier sur la page d'accueil, permettre d'ajouter un nouvel article, de l'éditer et de le supprimer (CRUD).
- Nous ne gérerons pas dans cette étape les catégories (étape 02) et les utilisateurs (étape 03)



# Projet de départ

1. Installez symfony dans un nouveau projet: **webarticles**

**symfony new --webapp webarticles**

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/webarticles?serverVersion=8&char  
set=utf8mb4"**

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

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

4. Installez manuellement les fichiers Bootstrap et jquery dans le dossier **public**
5. Installez également une librairie d'icônes: icofont.com
6. Créez une feuille de style vide: main.css
7. Dans base.html.twig liez les feuilles de style et les fichiers JS

# bases.html.twig

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>{% block title %}Webarticles{% endblock %}</title>
    {% block stylesheets %}
      <link rel="stylesheet" href="{{ asset('css/icofont.min.css') }}">
      <link rel="stylesheet" href="{{ asset('css/bootstrap.min.css') }}">
      <link rel="stylesheet" href="{{ asset('css/main.css') }}">
    {% endblock %}
  </head>
  <body>
    {# Navbar #}
    {% block navbar %}{% endblock %}
    {% block body %}{% endblock %}
  </body>
  {% block javascripts %}
    <script src="{{ asset('js/bootstrap.bundle.min.js') }}"></script>
  {% endblock %}
</html>
```

# Création du contrôleur et de la vue

- 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: **PostController**
- Symfony vient de créer le contrôleur dans le dossier src et la vue (index.html.twig) dans le dossier templates/post
- 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: PostController.php

```
<?php

namespace App\Controller;

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

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

# Vue: index.html.twig

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

{% block title %}WebArticles{% endblock %}

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

# L'ORM Doctrine

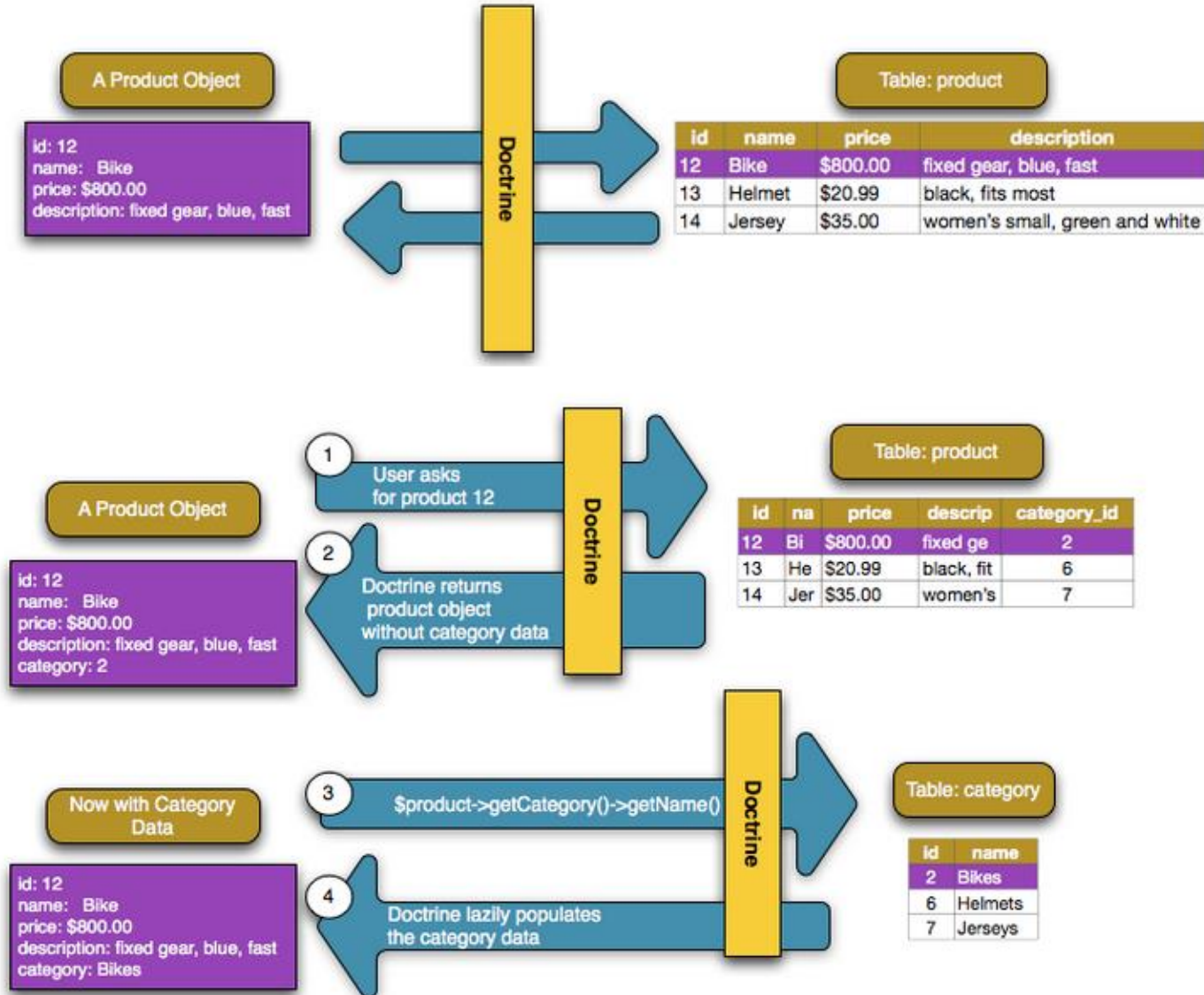
## Object Relation Mapper

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.

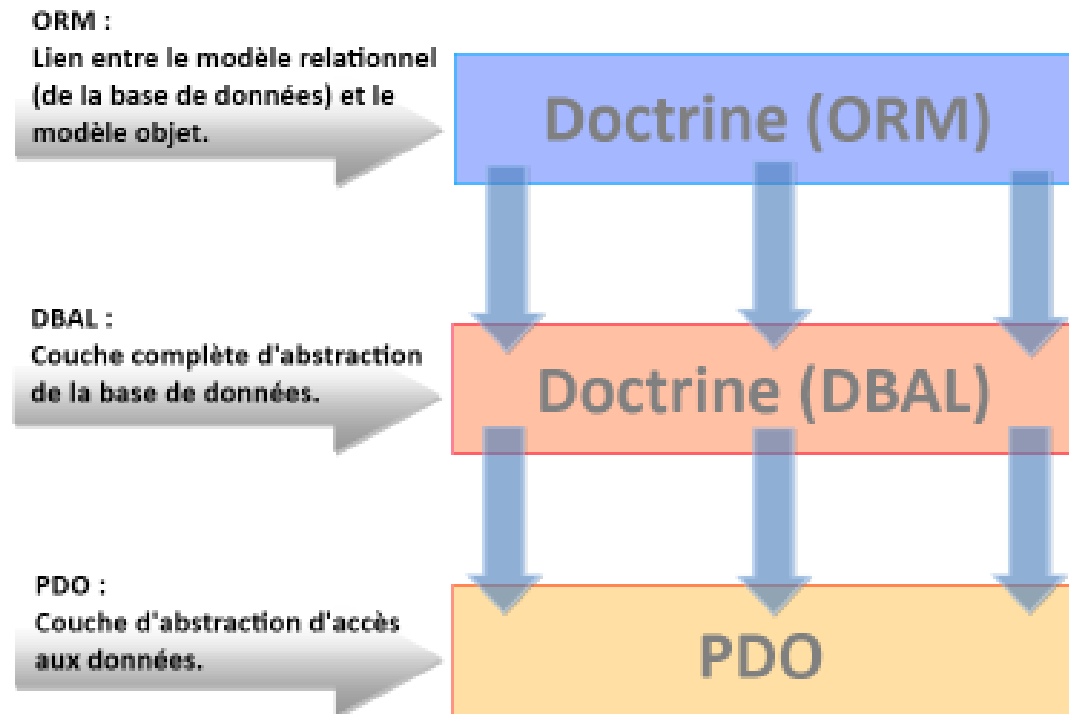
Son objectif est de se charger du traitement de vos données (CRUD: Create, Read, Update, Delete) sans manipuler la base de données et sans la création de requêtes SQL.

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

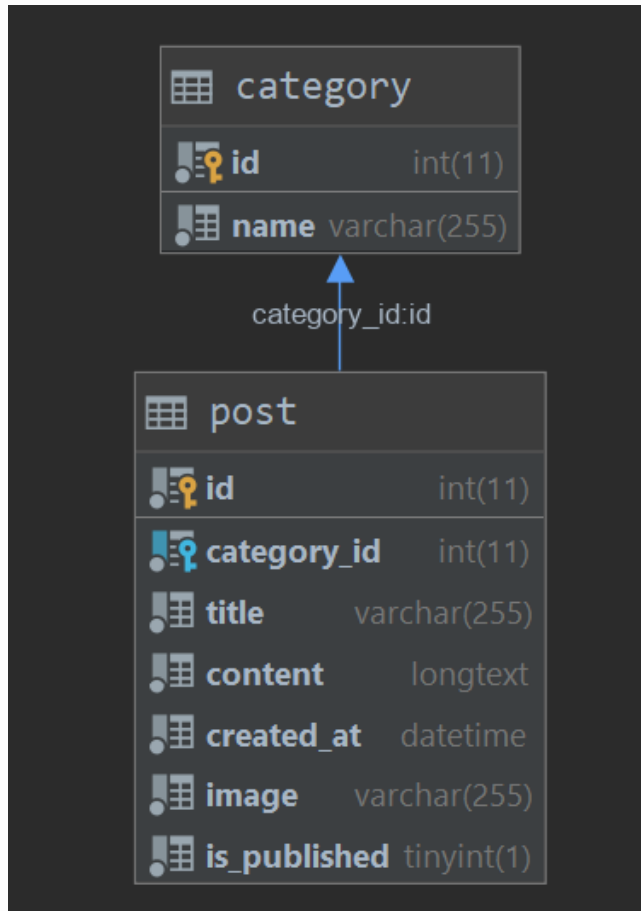


# Composants de Doctrine





# 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




- Pour obtenir la liste complète des commandes disponibles tapez la commande:
  - **php bin/console**
- Pour la création proprement dite de l'entité saisissez:
  - **php bin/console make:entity**
- Class name of the entity to create or update: **Category**
- 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 (id).
- Pour le choix du type de champ, vous pouvez saisir un point d'interrogation pour obtenir la liste complète.

# L'entité Category

Un seul attribut est nécessaire

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):
  - Terminez l'ajout des champs avec « **Enter** »

	category	
	id	int(11)
	name	varchar(255)

```

#[ORMEntity(repositoryClass: CategoryRepository::class)]
class Category
{
    #[ORMId]
    #[ORMGeneratedValue]
    #[ORMColumn]
    private ?int $id = null;

    #[ORMColumn(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): self
    {
        $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 manuellement un nouvelle propriété.

## Le repository CategoryRepository (classe PHP)

### Méthodes contenant les requêtes pour les catégories

Pour l'instant la classe contient qu'un constructeur, deux méthodes 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 [string]: 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]: 120
- 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	
 <b>id</b>	int(11)
 <b>category_id</b>	int(11)
 <b>title</b>	varchar(255)
 <b>content</b>	longtext
 <b>created_at</b>	datetime
 <b>image</b>	varchar(255)
 <b>is_published</b>	tinyint(1)

# L'entité Post

## Propriété category

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

```
#[ORMManyToOne(inversedBy: 'posts')]  
#[ORMJoinColumn(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): self  
{  
    $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.

```
#[ORMOneToMany(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

- ...

# 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. 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', '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:

On se sert d'un tableau contenant les catégories, dans la boucle, on instancie l'entité, on stocke la première donnée du tableau, on persiste l'objet à l'aide du Manager de Doctrine. 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
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;

class PostFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        $faker = Factory::create(); // Instanciation du générateur Faker
        $categories = $manager->getRepository(Category::class)->findAll();

        for($i = 1; $i <= 30; $i++) {
            $post = new Post(); // Instanciation de l'entité Post
            $post->setTitle($faker->words($faker->numberBetween(3, 5), true))
                ->setContent($faker->paragraphs(3, true))
                ->setCreatedAt($faker->dateTimeBetween('-30 days', 'now'))
                ->setImage($i.'.png')
                ->setIsPublished($faker->boolean(90))
                ->setCategory($categories[$faker->numberBetween(0, count($categories) -1)]);
            $manager->persist($post);
        }
        $manager->flush();
    }
}
```

# 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)
    {
        // Code de la méthode
    }

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

php bin/console doctrine:fixtures:load

# 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'EntityManager mais vous pouvez construire vos propres requêtes DQL à l'aide de la classe QueryBuilder

# 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, qui propose déjà quelques méthodes très utiles pour récupérer des entités. Nous n'écrirons rien pour le moment dans la classe Repository.

# Lister les enregistrements (le contrôleur)

## Principe

- Ajoutez une action **posts** et une route dans le contrôleur: PostController
- En premier, on récupère Doctrine dans une variable de type objet:
- **`$posts = $this->getDoctrine()`**
- Ensuite on récupère le Repository (permet de créer les requêtes ou d'utiliser des méthodes intégrées) créé par Doctrine:
- **`->getRepository('Post::class')`** // nom de la classe gérée par le Repository
- Après, on invoque une méthode de sélection du Repository:
- **`->findAll()`**;
- Enfin, on passe les données à la vue:
- **`return $this->render('post/index.html.twig', ['posts' => $posts]);`**



```
use App\Entity\Post;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;

class PostController extends AbstractController
{
    /**
     * @Route("/", name="posts")
     */
    public function posts()
    {
        $posts = $this->getDoctrine() // méthode permettant de communiquer avec Doctrine
            ->getRepository(Post::class) // méthode permettant d'utiliser ou de créer
des méthodes de type query. Il faut indiquer le nom de l'entité gérée par le Repository
            ->findAll(); // méthode du Repository permettant de récupérer tous les
articles
        return $this->render('post/index.html.twig', [
            'posts' => $posts
        ]);
    }
}
```

# 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 class="row">
    <div class="col-md-12">
      {% for post in posts %}
        <p>{{ post.title }} - {{ post.createdAt|date('d/m/Y') }}</p>
      {% endfor %}
    </div>
  </section>
</main>
```

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

# Refactoring du contrôleur

- Il est possible de simplifier le code du Contrôleur en utilisant l'injection de dépendance directement dans la définition de la méthode.
- Cette façon de procéder va récupérer automatiquement le service Doctrine et le Repository. Et c'est à partir de cet objet (\$repository) que je vais pouvoir invoquer les méthodes de sélection du Repository.

```
use App\Repository\PostRepository;
```

```
public function posts(PostRepository $repository)
{
    $posts = $repository->findAll();
    return $this->render('post/index.html.twig', [
        'posts' => $posts
    ]);
}
```

# Refactoring du contrôleur.

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

```
/**  
 * @Route("/", name="Posts")  
 * @param PostRepository $repository  
 * @return Response  
 */  
public function posts(PostRepository $repository) : Response  
{  
    $posts = $repository->findAll();  
    return $this->render('post/index.html.twig', [  
        'posts' => $posts  
    ]);  
}
```

- Ajouter le return à la méthode (Response)
- Ajoutez le use pour simplifier la propriété @param
- Ajouter les Doc Blocks avec les propriétés @param et @return

# 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

## Liste des Articles

Image	titre	Catégorie	date de publication
	quia aut nulla	Front End	02/06/2020
	ut fugit sit	Back End	02/06/2020
	ad architecto illo temporibus	Symfony	03/06/2020
	omnis et voluptatem	Laravel	04/06/2020
	placeat repellendus placeat quia	Back End	06/06/2020
	cupiditate possimus adipisci	Symfony	06/06/2020
	distinctio eos enim	Security	07/06/2020
	qui et vel eos et	Back End	08/06/2020
	quas distinctio quo qui aut	Back End	08/06/2020
	totam maiores laboriosam iure ab	PHP 7	10/06/2020
	consequatur porro ut dolore	Front End	10/06/2020
	dolorem iure ut sed	Back End	11/06/2020

# Refactoring de la vue

```
<table class="table table-hover">
  <thead>
    <tr>
      <th>Image</th>
      <th>titre</th>
      <th>Catégorie</th>
      <th>date de publication</th>
    </tr>
  </thead>
  <tbody>
    {% for post in posts %}
      <tr>
        <td></td>
        <td>{{ post.title }}</td>
        <td>{{ post.category.category }}</td>
        <td>{{ post.createdAt|date("d/m/Y") }}</td>
      </tr>
    {% endfor %}
  </tbody>
</table>
```

# 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 action 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: detail.html.twig

```
/**
 * @Route("/post/{id}", name="post")
 * @param PostRepository $repository
 * @param $id
 * @return Response
 */
public function post(PostRepository $repository, int $id) : Response
{
    $post = $repository->find($id);
    return $this->render('post/detail.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 entitéPost 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="posts")
 * @param Post $post
 * @return Response
 */
public function post(Post $post) : Response
{
    return $this->render('post/detail.html.twig', [
        'post' => $post
    ]);
}
```

On passe l'entité 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()

# La vue pour un seul enregistrement

```
<main class="container">
  <section class="row">
    <div class="col-md-12">
      <h2>Détail de l'article</h2>
    </div>
    <div class="col-md-4">
      
    </div>
    <div class="col-md-8">
      <h3>{{ post.title upper }}</h3>
      <p>
        {{ post.createdAt|date('d/m/Y') }}<br>
        {{ post.category.category }}
      </p>
      <p>{{ post.content }}</p>
      <p><a href="{{ path('posts') }}" class="btn btn-dark">Retour aux articles</a></p>
    </div>
  </section>
</main>
```

# Lecture d'un article à partir de la liste

- Utilisation du Helper `path()` pour générer un lien vers la vue avec en paramètre l'id de l'article.
- Le lien correspond à la route de la méthode post (le name suivit de l'id)

```
<td>  
  <a href="{{ path('post', {id:post.id}) }}">{{ post.title }}</a>  
</td>
```

# La vue detail.html.twig

## Détail de l'article



### OPTIO QUI QUIA OMNIS A

29/06/2020

PHP 7

Qui omnis in veniam aliquam sapiente. Voluptates quis et itaque incidunt sit repellat consequuntur. Voluptatem velit facilis nulla consequatur dignissimos eaque. Et voluptas accusantium molestiae tenetur. Eligendi quia alias sit. Omnis et deserunt et autem. Odio quis modi et nihil harum voluptatem incidunt. Quia in quo nihil. Aspernatur ipsam ipsa optio in quo ut. Dolorem praesentium vitae mollitia fugit. Delectus delectus adipisci repellat corporis. Voluptas quis fuga corrupti tenetur ut odit ullam et.

[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.

# Ajouter des données

Introduction

La classe PostType (formulaire)



# Principe

- Pour bien organiser notre code et ne pas surcharger le contrôleur, nous allons séparer le formulaire de celui-ci. Externaliser son formulaire n'est pas obligatoire mais vivement conseillé surtout pour une réutilisation ultérieure (formulaire de mise à jour par exemple).
- 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.

# Procédé

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**
- Un 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)
{
    $builder
        ->add('title')
        ->add('content')
        ->add('createdAt')
        ->add('image')
        ->add('isPublished')
        ->add('category')
    ;
}
```

Symfony a ajouté pour chaque propriétés 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.

# La classe PostType

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,...)

Exemple:

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

# La classe PostType

Avec l'auto-complétion de PhpStorm le use nécessaire pour chaque type de champ est ajouté automatiquement (vérifier quand même). Je vous conseille de consulter la documentation pour en savoir plus sur les types: <https://symfony.com/doc/current/reference/forms/types.html>

## Le cas de la date de publication:

Vous avez plusieurs façon de procéder pour la gestion de la date. L'insérer automatiquement via le contrôleur, l'insérer dans l'entité Post (setCreatedAt) ou la proposer dans le formulaire. Nous choisissons cette dernière mais en remplissant les champs avec la date du jour (instanciation de la classe PHP DateTime()).

Depuis Symfony 5.3, les propriétés des entités de type **...At** sont automatiquement de type **DateTimeImmutable**. Vous devez dès lors adapter aussi la méthode avec la même classe PHP

```
->add('createdAt', \DateTimeImmutable::class, [  
    'label' => 'Date de création',  
    'data' => new \DateTime(),  
    'format' => 'dd MM yyyy'  
])
```

# La classe PostType

## Le cas des catégories:

Ici c'est différent, nous nous trouvons dans le cas d'une relation entre deux entités. Nous souhaitons afficher une liste déroulante avec les catégories disponibles. Pour ce faire, nous allons utiliser comme type une entité: **EntityType** et indiquez ensuite via la clé **class** le nom de notre entité **liée**. La clé **choice\_label** correspond à l'attribut de l'entité **Catégorie** devant être utilisé dans la liste déroulante.

```
->add('category', EntityType::class, [  
    'label'          => 'Sélectionnez une catégorie',  
    'placeholder'    => 'Sélectionnez...',  
    'class'           => 'App:Category',  
    'choice_label'    => 'category'  
])
```

```
$builder
->add('title', TextType::class, [
    'label' => 'Titre de l\'article'
])
->add('content', TextareaType::class, [
    'label' => 'Votre contenu'
])
->add('createdAt', DateType::class, [
    'label' => 'Date de création',
    'data' => new \DateTime(),
    'format'=> 'dd MM yyyy'
])
->add('image', TextType::class, [
    'label' => 'Votre image'
])
->add('category', EntityType::class, [
    'label' => 'Sélectionnez une catégorie',
    'placeholder' => 'Sélectionnez...',
    'class' => 'App:Category',
    'choice_label' => 'category'
])
->add('isPublished', ChoiceType::class, [
    'label' => 'Publication de l\'article',
    'choices' => ['Oui' => 1, 'Non' => 0]
])
->add('submit', SubmitType::class, [
    'label' => 'Enregistrer l\'article'
]);
```

La classe PostType et la méthode  
buildForm()

# Le contrôleur Post et la méthode addPost()

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->createView()
    ]);
}
```

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

Symfony 5.3

# 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 à 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 de 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 Post et la méthode addPost()

```
/**
 * @Route("/addpost", name="addpost")
 * @param Request $request
 * @param EntityManagerInterface $manager
 * @return Response
 */
```

```
public function addPost (Request $request, EntityManagerInterface $manager) : Response
{
    $post = new Post;
    $form = $this->createForm(PostType::class, $post);
    $form->handleRequest($request);
    if($form->isSubmitted() && $form->isValid()) {
        $manager->persist($post);
        $manager->flush();
        return $this->redirectToRoute('posts');
    }
    return $this->renderForm('post/add.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

# La vue avec le formulaire

## Ajouter un article

Titre de l'article

Votre contenu

Date de création

03

▼

07

▼

2020

▼

Votre image

Sélectionnez une catégorie

Sélectionnez...

▼

Publication de l'article

Oui

▼

ENREGISTRER L'ARTICLE

# Ajoutez Bootstrap aux formulaires

- Par défaut, bien qu'il soit installé, Bootstrap n'est pas appliqué au formulaire. Vous pouvez définir manuellement les classes Bootstrap sur les champs ou le configurer dans le fichier twig.yaml du dossier config/packages .
- Il suffit d'ajouter la clé «`form_themes`»

```
twig:  
    default_path: '%kernel.project_dir%/templates'  
    form_themes: ['bootstrap_5_layout.html.twig']
```

- Avec Symfony 5.1, le theme Foundation 6 a été introduit

```
form_themes: ['foundation_6_layout.html.twig']
```

# Fournir un lien vers le formulaire dans la page principale

- Dans la navbar du fichier base.html.twig remplacer le formulaire de recherche par un bouton.

```
<div class="form-inline my-2 my-lg-0">
  <a href="{{ path('addpost') }}" class="btn btn-dark">Ajouter un article</a>
</div>
```

- Le **helper path()** en Twig permet de rediriger vers un contrôleur. Il prend en paramètre le name de la route.



# 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()

Voilà ce à quoi ressemble la méthode en version longue

```
public function delPost($id)
{
    $manager = $this->getDoctrine()->getManager();
    $repository= $this->getDoctrine()->getRepository(Post::class);
    $post = $repository->find($id);
    $manager->remove($post);
    $manager->flush();

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

# Méthode delPost()

Avec l'injection de dépendance (EntityManagerInterface) et ParamConverter pour transformer l'id en un objet correspondant on va simplifier le code.

```
public function delPost(Post $post, EntityManagerInterface $manager) : Response
{
    $manager->remove($post);
    $manager->flush();
    return $this->redirectToRoute('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.

Ajoutez une classe Bootstrap et une icone Icofont. Pour les icones, installez Icofont dans les assets et ajoutez le lien css dans base.html.twig

# 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('/editpost/{id}', name: 'editpost')]

public function editPost(Post $post, EntityManagerInterface $manager, Request $request) : Response
{
    $form = $this->createForm(PostType::class, $post);
    $form->handleRequest($request);
    if($form->isSubmitted() && $form->isSubmitted()) {
        $manager->persist($post);
        $manager->flush();
        return $this->redirectToRoute('posts');
    }
    return $this->render('post/edit.html.twig', [
        'form' => $form->createView()
    ]);
}
```



# 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.

```
'data' => new \DateTime()
```

- Hors, dans le cas d'une mise à jour il faudrait récupérer la date de l'enregistrement voir ne pas pouvoir la modifier. Il existe plusieurs façons de procéder mais faisons ici au plus simple en souhaitant aussi utiliser le même formulaire (Type) pour l'insertion et la mise à jour.
  1. Ajouter automatiquement la date du jour lors de l'insertion des données dans le contrôleur et pas dans le `PostType`.
  2. Ne pas proposer de mise à jour de la date de publication. Ce qui me semble normal.

# Modification de la méthode addPost()

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




































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

- Retirer dans le PostType le champ de la date de publication. Comme il est retiré du formulaire, le champ ne sera jamais mis à jour.

# La vue index.html.twig

AJOUTER UN ARTICLE

Liste des Articles

Image	Titre	Catégorie	Auteur	Date de publication		
	iure qui expedita	JavaScript	Arthur Gallet	06/06/2020		
	quaerat minima vero	Security	Julien Rossi	08/06/2020		
	voluptatibus voluptate beatae	Back End	Thibault Roux	09/06/2020		
	qui aliquam id rerum enim	Front End	Audrey Fischer	10/06/2020		
	nulla inventore dolore voluptas incidunt	Back End	Thibault Roux	12/06/2020		
	harum voluptas labore nesciunt mollitia	Back End	Audrey Fischer	14/06/2020		
	ea aut dolore ea et	Symfony	Thibault Roux	15/06/2020		
	omnis vel est	PHP 7	Audrey Fischer	15/06/2020		
	repellendus qui nobis tenetur	Security	Audrey Fischer	16/06/2020		
	nesciunt sed dolorum doloreque	Laravel	Audrey Fischer	18/06/2020		
	quia molestiae alias consequuntur reprehenderit	JavaScript	Gilles Samson	18/06/2020		

# Les utilisateurs et le composant Security

Fonctionnement

L'entité User

Passwords

Fixtures

Inscription

Authentification

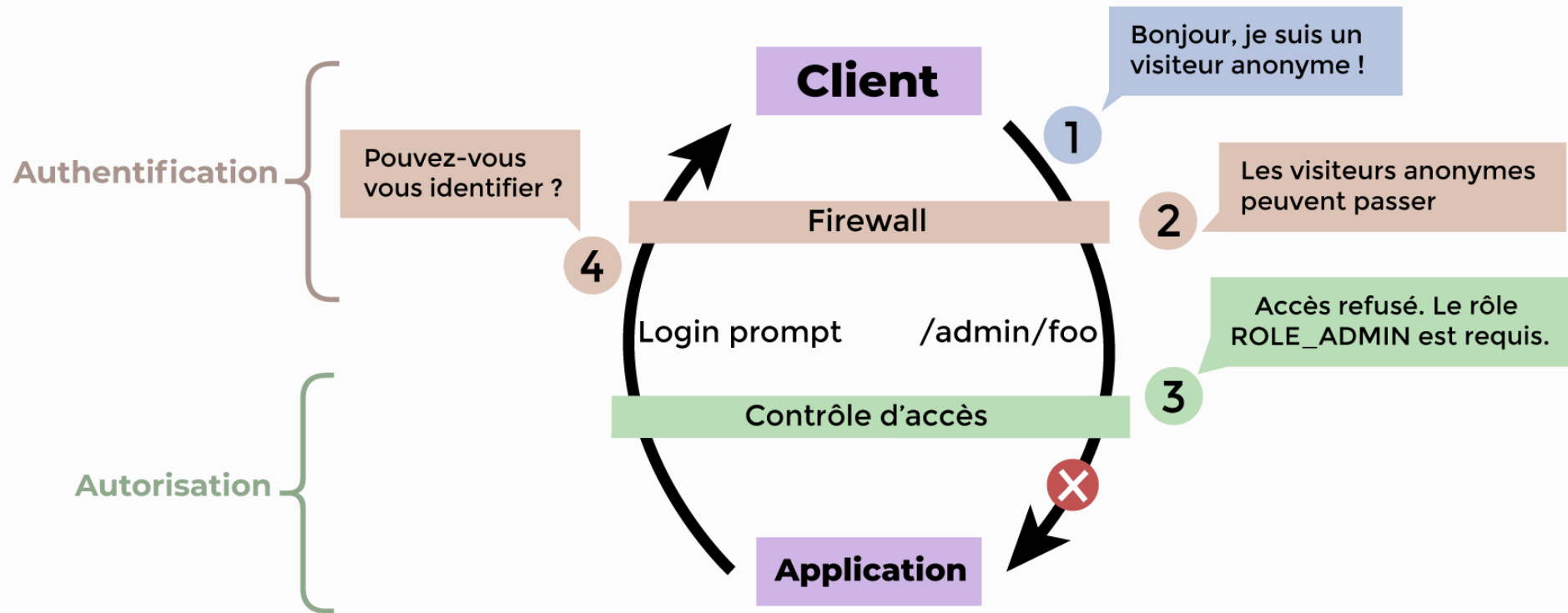
# Fonctionnement

- 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: `webarticles_users`

DB: `webarticles_users` (modifier le `.env`)

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

# 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 CLI vous permettant de gérer automatiquement dans Symfony les utilisateurs. Vous devrez configurer certains points comme l'Access Control mais cette solution est plus rapide.

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]

```
> username
```

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 attributs avec leurs getters et leurs setters: `$id`, `$username`, `$roles` et `$password`

L'attribut `$role` est un peu particulier car il s'agit d'un type json sous la forme d'un 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 le cas ici, ça sera toujours le rôle user. Cela permet d'avoir au moins le rôle de base et nous pourrons changer ça par après.

Deux autres méthodes `getSalt()` et `eraseCredentials()` proviennent de l'interface `UserInterface` implémentée dans la classe User. Selon l'architecture objet, quand une classe implémente une Interface toutes ses méthodes doivent obligatoirement être ajoutées. Ici il s'agit seulement de la signature des fonctions car elles sont vides et le resteront.

# 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
  - lastName
  - Email
- N'oubliez pas la migration:
  - `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

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

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

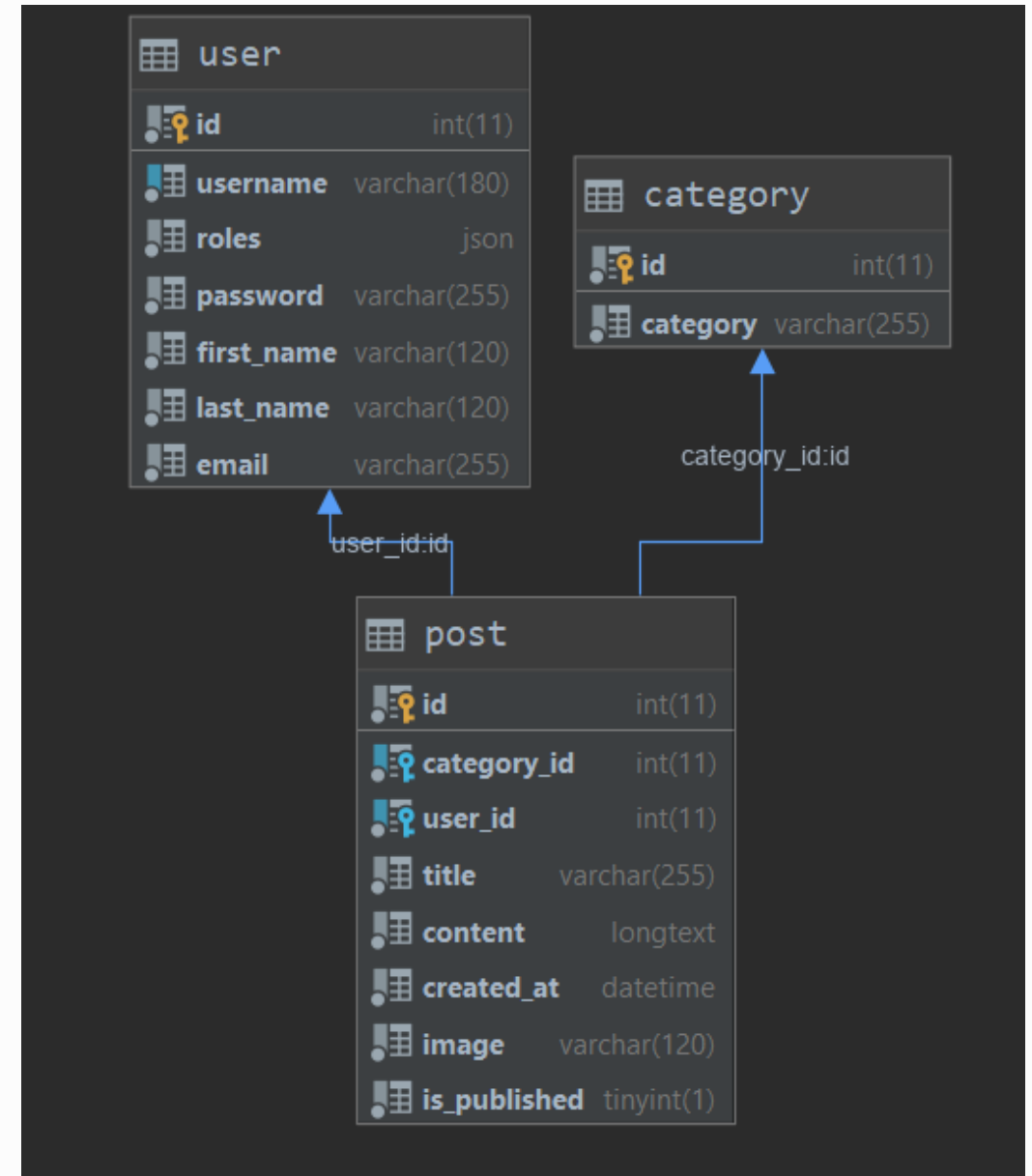
# La migration

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.





# 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.
- Lors de l'utilisation de la commande `make:user`, le fichier `security.yaml` a été modifié. Le `password_hashers` permettant de crypter les mots de passe a été ajouté pour le composant `security`.

```
password_hashers:  
  Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'  
  App\Entity\User:  
    algorithm: auto
```

- On y retrouve le nom de l'entité concernée par le cryptage et le type d'algorithme utilisé. Symfony utilise une valeur automatique 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 du champ `username` de l'entité `User`.
- Le firewall demande à un provider de lui fournir les utilisateurs provenant de la base de données.

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

# 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

```
public function load(ObjectManager $manager)
{
    $faker = Factory::create('fr_FR');
    for($i = 1; $i <= 30; $i++) {
        $user = new User();
        $user->setFirstName($faker->firstName());
        $user->setLastName($faker->lastName());
        $user->setUsername($user->getFirstName().$user->getLastName());
        $user->setEmail($user->getFirstName().$user->getLastName().'@gmail.com');
        $password = $this->hasher->hashPassword($user, 'password');
        $user->setPassword($password);
        $manager->persist($user);
    }
    $manager->flush();
}
```

# Modifications de PostFixtures

```
$users = $manager->getRepository(User::class)->findAll();  
for($i = 1; $i <= 20; $i++){  
    $post = new Post();  
    //  
    $post->setUser($users[$faker->numberBetween(0, count($users) -1)]);  
    $manager->persist($post);  
}  
$manager->flush();  
}
```

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

# Lier les articles aux utilisateurs

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

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

# Création du formulaire pour les inscriptions

**php bin/console make:form RegistrationType**

- The name of Entity or fully qualified model class name that the new form will be bound to (empty for none): **User**
- Dans les deux dias suivantes se trouve la vue du formulaire et le code de la méthode buildForm()
- N'oubliez pas de vérifier les **uses** et de rajouter le champ de confirmation du mot de passe.

# La vue «registration.html.twig»

S'enregistrer

Prénom

Nom

Email

Nom d'utilisateur

Mot de passe

S'ENREGISTRER



# RegistrationType

```
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('firstName', TextType::class, ['label' => 'Prénom'])
        ->add('lastName', TextType::class, ['label' => 'Nom'])
        ->add('email', EmailType::class, ['label' => 'Email'])
        ->add('username', TextType::class, ['label' => 'Nom d\'utilisateur'])
        ->add('password', TextType::class, ['label' => 'Mot de passe'])
    ;
}
```

- La propriété roles est gérée via l'entité (tous les inscrits ont le rôle USER)
- Le champ submit est ajouté directement dans le template

# Le contrôleur

- La gestion de la sécurité se fera via un contrôleur indépendant que nous nommerons **SecurityController**
- Le code de la méthode **registration()** ressemble à peu de chose près à la méthode **addPost()** du contrôleur **PostController**.
- **Créez le contrôleur avec la commande:** `php bin/console make:controller`
- Nommez le **SecurityController**

# Security Controller – Uses & Route

```
namespace App\Controller;
```

```
use App\Entity\User;
```

```
use App\Form\RegistrationType;
```

```
use Doctrine\ORM\EntityManagerInterface;
```

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

```
use Symfony\Component\HttpFoundation\Request;
```

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

```
use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;
```

```
/**
```

```
 * @Route("/registration", name="registration")
```

```
 */
```

# Security Controller - méthode

```
public function registration(Request $request, EntityManagerInterface $manager,
UserPasswordEncoderInterface $encoder)
{
    $user = new User();
    $form = $this->createForm(RegistrationType::class, $user);
    $form->handleRequest($request);
    if($form->isSubmitted() && $form->isValid()) {
        $hash = $encoder->encodePassword($user, $user->getPassword());
        $user->setPassword($hash);
        $manager->persist($user);
        $manager->flush();
        return $this->redirectToRoute('posts');
    }
    return $this->render('security/registration.html.twig', [
        'form' => $form->createView()
    ]);
}
```

# Crypter les mots de passe dans le contrôleur

- Pour hasher le mot de passe, on utilise l'interface `UserPasswordEncoderInterface` en injection dans la méthode.
- La méthode `encodePassword()` récupère le mot de passe du formulaire et le crypte dans la variable `$crypt`
- Ensuite c'est le mot de passe crypté qui est inséré dans la DB

```
$hash = $encoder->encodePassword($user, $user->getPassword());  
$user->setPassword($hash);
```

- Avec PhpStorm quand vous faites une injection de dépendance (ici `UserPasswordEncoderInterface`) vous pouvez générer le phpDocblock et le use en même temps avec la commande ALT + Enter)

# La vue registration.html.twig

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

{% block title %}Registration{% endblock %}

{% block body %}
    <main class="container">
        <section class="row">
            <div class="col-md-8 offset-md-2">
                <h2>S'enregistrer</h2>
                {{ form_start(form) }}
                {{ form_widget(form) }}
                <button type="submit" class="btn btn-outline-secondary">S'enregistrer</button>
                {{ form_end(form) }}
            </div>
        </section>
    </main>
{% endblock %}
```

# L'identification - Login

- Créez un contrôleur pour le formulaire de connexion :  
`php bin/console make:controller Login`
- Symfony retourne dans la console les messages suivants:
  - created: src/Controller/LoginController.php
  - created: templates/login/index.html.twig

Changez le dossier login en security et le template en login.html.twig

# la vue pour s'identifier

Dans le fichier login.html.twig, nous allons créer un simple formulaire HTML avec les champs username (name = “\_username”) et password (name = “\_password”). Vu que c’est Symfony qui gère lui-même le formulaire, vous devez respecter les names imposés.

! Ici le champs unique d'identification est le user Name. Dans d'autres projets, j'utilise le email

```
<main class="container">
  <section class="row">
    <div class="col-md-6 offset-3">
      {% if error %}
        <div class="alert alert-danger">{{ error.messageKey|trans(error.messageData, 'security') }}</div>
      {% endif %}
      <h4 class="display-3">Identification</h4>
      <form method="post">
        <div class="form-group">
          <label for="username" class="form-label">User Name</label>
          <input type="text" id="username" class="form-control" name="_username" value="{{ last_username }}">
        </div>
        <div class="form-group">
          <label for="password" class="form-label">Password</label>
          <input type="password" id="password" class="form-control" name="_password">
        </div>
        <div class="form-group">
          <button class="btn btn-outline-primary form-control" type="submit">S'identifier</button>
        </div>
      </form>
    </div>
  </section>
</main>
```



# Configurer le firewall

- Ouvrez `security.yaml` pour configurer le firewall et activer l'authentificateur de connexion par formulaire à l'aide du `form_login` paramètre. 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.

```
main:
  lazy: true
  provider: app_user_provider
  form_login:
    login_path: login
    check_path: login
```

`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

# Le contrôleur de connexion

Modifiez la méthode login() du LoginController pour afficher le formulaire de connexion:

```
class LoginController extends AbstractController
{
    #[Route('/login', name: 'login')]
    public function index(AuthenticationUtils $authenticationUtils): Response
    {
        $error = $authenticationUtils->getLastAuthenticationError();
        $lastUserName = $authenticationUtils->getLastUsername();
        return $this->render('security/login.html.twig', [
            'last_username' => $lastUserName,
            'error'         => $error
        ]);
    }
}
```

# Protection CSRF

- Le formulaire de login que vous avez créé manuellement n'est pas protégé contre la faille CSRF (cross-site request forgery) contrairement aux autres formulaires de Symfony.
- Pour se protéger, vous devez activer le **CSRF** dans **security.yaml**. Celui-ci permettra de générer un token (jeton) dans le formulaire qui sera ensuite comparé par le composant Security de Symfony.

```
form_login:  
  login_path: login  
  check_path: login  
  enable_csrf: true
```

# Le formulaire de login

- Utilisez la fonction `csrf_token()` de Twig pour générer un jeton CSRF et le stocker en tant que champ masqué du formulaire. Par défaut, le champ HTML doit être appelé `_csrf_token` et la chaîne utilisée pour générer la valeur doit être `authenticate`.

```
<div class="form-group my-3">
  <input type="hidden" name="_csrf_token" value="{{ csrf_token('authenticate') }}">
  <button class="btn btn-outline-primary form-control" type="submit">S'identifier</button>
</div>
```

- Vérifiez la présence du token dans le formulaire avec un view source. Si vous le modifiez, la connexion est refusée !

# Déconnexion

- Pour terminer, nous devons ajouter une fonctionnalité permettant de se déconnecter. Il faut modifier le config.yaml et insérer le paramètre **logout** dans le firewall.
- Dans le contrôleur LoginController on ajoute la route **/logout** et le name **app\_logout** qui déconnectera l'utilisateur et le redirigera.
- le contrôleur qui ne renvoi rien et c'est Symfony qui se chargera de réaliser l'opération.
- Après, il faut soit ajouter un bouton connexion soit un bouton logout dans la navbar. Tout dépendra si l'utilisateur est déjà ou pas connecté.

```
main:
  ...
  logout:
    path: app_logout
```

```
#[Route('/logout', name: 'app_logout', methods: ['GET'])]
public function logout()
{
    // controller can be blank: it will never be called!
}
```

# Le fichier base.html.twig

- Pour faire le test de connexion, on va utiliser en Twig une variable d'environnement (app).
- On ajoute les liens dans la navbar:

```
<a href="{{ path('addpost') }}" class="btn btn-light btn-sm">Ajouter un article</a>&nbsp;

{% if not app.user %}
<a href="{{ path('login') }}" class="btn btn-light btn-sm">S'identifier</a>
{% else %}
<a href="{{ path('logout') }}" class="btn btn-light btn-sm">Se déconnecter</a>
{% endif %}
```

# Restreindre les accès

- Comme l'utilisateur peut maintenant être identifié, vous pouvez sécuriser l'`index.html.twig` pour afficher seulement les liens (delete et update) s'il est connecté.
- Un simple test en Twig permettra d'afficher les icones (delete et update) seulement si l'utilisateur a le rôle «`ROLE_USER`».

```
{% if is_granted('ROLE_USER') %}
<td>
    <a href="{{ path('deletepost', {id:post.id}) }}" class="text-danger">
        <i class="icofont-ui-delete"></i>
    </a>
</td>
<td>
    <a href="{{ path('editpost', {id:post.id}) }}" class="text-primary">
        <i class="icofont-edit-alt"></i>
    </a>
</td>
{% endif %}
```

- Il est clair que ce rôle en réalité ne permettra pas de faire de telles opérations mais ça reste un exemple pour la démonstration.

# 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');  
}
```



# Sécurisé 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")  
 */
```

# Validation du formulaire

- Nous devons maintenant ajouter des contraintes de validation sur le formulaire:
  1. Password: un minimum de caractères (éventuellement des clés de sécurité: caractères spéciaux, chiffres...)
  2. Confirmation du mot de passe: identique au mot de passe
  3. Le username et l'email doivent être uniques
- Pour ce faire, nous allons ajouté des annotations de type **@Assert** dans l'entité User.
- N'oubliez pas d'ajouter le use du composant Validator
- `use Symfony\Component\Validator\Constraints as Assert;`

**Vous trouverez des informations complémentaires sur la validation plus bas dans [les notes](#)**

# L'entité User modifiée

```
/**
 * @ORM\Column(type="string", length=255)
 * @Assert\Length(
 *     min = 4,
 *     minMessage = "Le mot de passe doit contenir au minimum {{
limit }} caractères"
 * )
 */
private $password;

/**
 * * @Assert\EqualTo(propertyPath="Password", message="Le mot de
passe doit être identique")
 */
public $confirmPassword;
```

# Rendre un champ unique

- Il est important que le mail renseigné dans le formulaire soit unique. Pour ce faire, nous allons modifier l'entité pour ajouter une contrainte d'unicité.
- Il ne s'agit pas du même type de validation que nous avons déjà utilisé (@Assert). Ici l'unicité ce fait au niveau de la classe et pas au niveau de l'attribut.
- <https://symfony.com/doc/current/reference/constraints/UniqueEntity.html>

Votre Email

**ERROR** Cet Email est déjà utilisé

patrick.marthus@iepscf-namur.be

# L'entité User

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

/**
 * @ORM\Entity(repositoryClass="App\Repository\UserRepository")
 * @UniqueEntity(
 *     fields={"email"},
 *     message="Cet Email est déjà utilisé"
 * )
 * @UniqueEntity(
 *     fields={"userName"},
 *     message="Cet identifiant est déjà utilisé"
 * )
 */
```

# 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 récupérez.
- Cette fonctionnalité rend les messages "flash" particulièrement intéressants pour stocker tous 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, ajouter le code suivant entre la méthode flush() et la méthode redirectToRoute().

```
$this->addFlash(  
    'success',  
    'Article correctement ajouté'  
);
```

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



# La vue

- Pour récupérer le message dans la vue, on utilise une boucle for et la fonction app.flashes(). Elle prend en paramètre le type que vous avez mentionner dans la méthode addFlash(). Dans notre cas success.

```
{% block message %}
    {% for message in app.flashes('success') %}
        <div class="alert alert-success">
            {{ message }}
        </div>
    {% endfor %}
{% endblock %}
```

Dans cet exemple, vous devrez créer une boucle et une div par type de message. Ici sont seulement gérés les messages de succès.

# La vue

- Pour optimiser l'affichage et le traitement, j'utilise deux boucles for() pour récupérer aussi le nom du type que je peux concaténer dans la div.

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

# 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.
- En Symfony, le principe est simple. On définit des règles de validation que l'on va rattacher à une classe (Entité). Puis on fait appel à un service extérieur (composant Validator) pour venir lire un objet (instance de ladite classe) et ses règles, et définir si oui ou non l'objet en question respecte ces règles.

# L'entité Post

- Pour définir les règles de validation dans l'entité, nous allons utiliser les annotations. La première chose à savoir est le namespace des annotations 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;
```

- Ce use est à rajouter au début de l'objet que l'on va valider, notre entité Post en l'occurrence. En réalité, vous pouvez définir l'alias à autre chose qu'Assert. Mais c'est une convention adoptée par les développeurs.
- Ensuite, il ne reste plus qu'à écrire en annotations les règles de validation. Une fois de plus, la documentation officielle fournit les informations sur l'ensemble des propriétés disponibles.

```
/**
 * @ORM\Column(type="string", length=255)
 * @Assert\Length(
 *     min = 5,
 *     max = 50,
 *     minMessage = "Le titre de l'article doit contenir au moins {{
limit }} caractères",
 *     maxMessage = "Le titre de l'article ne peut pas dépasser {{ limit
}} caractères"
 * )
 * @Assert\Regex(
 *     pattern="/\d/",
 *     match=false,
 *     message="Votre article ne peut pas contenir de chiffres")
 */
private $title;
```

# Autres exemples

```
* @Assert\NotEqualTo(  
*     value = 0  
* )
```

```
* @Assert\GreaterThan(  
*     value = 2  
* )  
* @Assert\LessThan(10)
```

```
* @Assert\Url(  
*     message = "The url '{{ value }}' is not a valid url",  
* )
```

# Email avec Swift Mailer

Création de l'entité «Contact»

Création de la classe ContactType (formulaire)

Création ou modification du contrôleur PageController

Création du service ContactService

Création de la vue avec le formulaire



# Etapes a réaliser

- Installation de la librairie: `composer require symfony/swiftmailer-bundle`
- Installation d'une application pour récupérer les mails: `MailDev`
- Création d'une entité `Contact`
- Création du formType: `ContactType`
- Création d'un Contrôleur: `ContactController`
- Création d'un template: `contact.html.twig`
- Création d'un service pour l'envoi du mail: `Service/ContactService.php`
- Création de la vue du mail en twig: `contact/emailbasic.html.twig`
- Test et affichage du mail

## Nous contacter

Firstname

Lastname

Email

Subject

Message

Nous contacter

# Introduction

- Symfony dispose de la librairie «Swift Mailer» proposant de nombreux services permettant l'envoi et la gestion des emails. Pour permettre l'envoi des emails à partir du serveur local, vous disposez d'un outil qui se chargera d'intercepter et de lire le courrier.
- Nous allons installer et utiliser «Test Mail Server Tool» pour la capture des emails:
- <https://toolheap.com/test-mail-server-tool/users-manual.html>
- Il suffit de l'exécuter et la première fois, l'application va créer un dossier dans lequel elle va renvoyer tous les mails sortant (Mail Sent to Local Server). Le port par défaut est le 25.

# Introduction

- Une autre application permet également de capturer les mails sortant. Il s'agit de **MailDev**, qui a pour avantage de proposer une interface web pour l'affichage des mails:  
<https://www.npmjs.com/package/maildev>
- Utiliser une invite de commande pour l'installation: **npm install -g maildev**
- Ensuite exécutez la commande **maildev**
- Vous obtiendrez les adresses suivantes:
- MailDev webapp running at http://0.0.0.0:1080
- MailDev SMTP Server running at 0.0.0.0:1025
- Pour accéder à l'interface MailDev, saisissez dans votre navigateur: <http://localhost:1080/>
- !!! Pour éviter une erreur liée au protocole HTTPS utilisez la commande **maildev --hide-extensions STARTTLS**

# Création de l'entité «Contact»

- Nous allons définir l'ensemble des attributs 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.
- 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 dias précédentes.
- La dia suivante représente l'entité sauf les getters et setters générés automatiquement par PhpStorm.

```
namespace App\Entity;
```

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

```
class Contact {
```

```
/**
```

```
 * @var string|null
```

```
 * @Assert\NotBlank()
```

```
 * @Assert\Length(min=2)
```

```
 */
```

```
private $firstname;
```

```
/**
```

```
 * @var string|null
```

```
 * @Assert\NotBlank()
```

```
 * @Assert\Length(min=2)
```

```
 */
```

```
private $lastname;
```

```
/**
```

```
 * @var string|null
```

```
 * @Assert\NotBlank()
```

```
 * @Assert\Email()
```

```
 */
```

```
private $email;
```

```
/**
```

```
 * @var string|null
```

```
 * @Assert\NotBlank()
```

```
 * @Assert\Length(min=5)
```

```
 */
```

```
private $subject;
```

```
/**
```

```
 * @var string|null
```

```
 * @Assert\NotBlank()
```

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

```
 */
```

```
private $message;
```

```
// GETTERS & SETTERS
```

```
}
```

# Création de la classe ContactType

- Le contact type 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 uses en conformité avec les types des champs de formulaire.
- Dans la méthode configureOptions(), ajoutez l'entité **Contact** pour le data\_class permettant de récupérer les données.

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

```
class ContactType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array
$options)
    {
        $builder
            ->add('firstname', TextType::class)
            ->add('lastname', TextType::class)
            ->add('email', EmailType::class)
            ->add('subject', TextType::class)
            ->add('message', TextareaType::class)
            ->add('submit', SubmitType::class);
    }

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



# Le contrôleur ContactController et la méthode contact()

Ce contrôleur s'occupe de l'affichage de la page contact et de son formulaire. Nous allons y ajouter une méthode contact() permettant d'afficher le formulaire et par la suite d'envoyer le mail (incomplet pour l'instant).

```
/**
 * @Route("/contact", name="contact")
 */
public function contact(Request $request)
{
    $contact = new Contact();
    $form = $this->createForm(ContactType::class, $contact);
    $form->handleRequest($request);
    if($form->isSubmitted() && $form->isValid()){
        return $this->redirectToRoute('post');
    }

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

# La vue contact.html.twig

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

{% block title %}Contact{% endblock %}

{% block body %}
    <div class="jumbotron">
        <h1 class="display-4">Contact Us</h1>
        <h2>Web technologies Contact</h2>
        <p class="lead">This is a simple hero unit, a simple jumbotron-style
component for calling extra attention to featured content or information.</p>
        <hr class="my-4">
    </div>

    <div class="row">
        <div class="col-md-10 offset-md-1">
            {{ form(form) }}
        </div>
    </div>
{% endblock %}
```

# L'envoi du mail avec le service ContactService

- Pour éviter de surcharger la méthode contact() du contrôleur, nous allons écrire un service (classe) permettant la gestion et l'envoi du mail.
- Créez un dossier service dans le namespace App/ et ajoutez une classe ContactService.
- Modifiez le fichier .env pour l'envoi de mail en local.

```
MAILER_URL=smtp://localhost:1025
```

Pour la suite, nous allons utiliser la librairie «Swift Mailer» dont la documentation se trouve à l'adresse suivante:

<https://symfony.com/doc/current/email.html>

# ContactService

Le constructeur

```
public function __construct(\Swift_Mailer $mailer, Environment
$renderer)
{

}
```

- Paramètres: le service Swift\_Mailer et le service Environment qui permettra de générer le mail en html avec Twig. Attention aux uses.
- A l'aide de PhpStorm initialiser les champs pour le constructeur avec le raccourci ALT + ENTER (Initialize fields). Vous obtiendrez le code complet repris à la dia suivante.

# ContactService

```
/**
 * @var \Swift_Mailer
 */
private $mailer;

/**
 * @var Environment
 */
private $renderer;

public function __construct(\Swift_Mailer $mailer, Environment $renderer)
{
    $this->mailer = $mailer;
    $this->renderer = $renderer;
}
```

# ContactService

La méthode `sendMail()` permettant la gestion de l'envoi du mail.

```
public function sendMail(Contact $contact)
{
    $message = (new \Swift_Message())
        ->setFrom($contact->getEmail())
        ->setTo('contact@agence.be')
        ->setReplyTo($contact->getEmail())
        ->setSubject($contact->getSubject())
        ->setBody($this->renderer-
>render('contact/emailbasic.html.twig', [
            'contact' => $contact
        ]), 'text/html');
    $this->mailer->send($message);
}
```

# Terminer ContactController

```
public function contact(Request $request, ContactService $contactService)
{
    $contact = new Contact();
    $form = $this->createForm(ContactType::class, $contact);
    $form->handleRequest($request);
    if($form->isSubmitted() && $form->isValid()){
        // Gestion du mail dans la classe Service/contactService
        $contactService->sendMail($contact);
        $this->addFlash('success', 'Votre email a bien été envoyé');
        return $this->redirectToRoute('home');
    }

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


# email.html.twig


Conception simplifiée de la vue qui se chargera du contenu html du message:


```
<p>
    {{ contact.message }}
</p>
<p>
    de: {{ contact.firstname }} {{ contact.lastname }}
</p>
```





# Réception dans MailDev


 MailDev


 Search


 Refresh


 Clear Inbox


 Display


 Viewport

 Attachments

 Delete

 Relay

 Relay to

 Download EML

essai de contact	
contact@easylearning.be	11/26/19 11:19 AM
test premier	
contact@easylearning.be	11/26/19 11:07 AM

Votre message a bien été reçu

**De: Patrick Marthus**

# Bundles Symphony

# Paginer les articles

## KnplPaginator

# Installer le bundle KnpPaginator

- Un des nombreux avantages qu'offre Symfony est l'utilisation de Bundles prêt à l'emploi. Vous trouverez la liste complète sur <https://packagist.org/packages/symfony/> qui référence les librairies PHP disponibles pour Composer.
- Nous utiliserons knplabs/knp-paginator-bundle qui sera installé à l'aide de la commande **composer require knplabs/knp-paginator-bundle**
- Sur le dépôt GitHub du bundle, vous trouverez la documentation permettant la configuration et l'utilisation de la librairie: <https://github.com/KnpLabs/KnpPaginatorBundle>
- Il y est indiqué que vous devez modifier le fichier bundles.php pour y ajouter la nouvelle librairie. Ne modifier rien car depuis sa version 4, les bundles sont automatiquement reconnus et configurés.

- Sous la rubrique «Configuration example» un exemple de configuration vous est fourni. Créez le fichier **knp\_paginator.yml** dans config/packages et copiez le code proposé.
- Modifiez le contrôleur PostController pour y insérer le code permettant d'utiliser la classe PaginatorInterface
- ```
public function index(PaginatorInterface $paginator, Request $request)
{
    $post = $this->getDoctrine()
        ->getRepository(Post::class)
        ->findAll();
    $pagination = $paginator->paginate(
        $post,
        $request->query->getInt('page', 1),
        5);
    return $this->render('post/index.html.twig', ['posts' =>
    $pagination]);
}
```

- Dans la vue index.html.twig, ajoutez le code suivant après le tableau d'affichage:
- ```
<div class="navigation">
    {{ knp_pagination_render(posts) }}
</div>
```
- La pagination apparaît par défaut. Pour l'adapter à la sauce Bootstrap, modifiez votre configuration de template dans le fichier knp\_paginator.yml:
- ```
pagination:
  '@KnpPaginator/Pagination/twitter_bootstrap_v4_pagination.html.twig'
# sliding pagination controls template
```

# Traduire les boutons

- 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 **services.yml**:
- **parameters:**  
    **locale:** 'fr'
- 2. Dans le dossier translation, créez le fichier **KnpPaginatorBundle.fr.yml**
- 3. Ajoutez les traductions suivantes:  
    **label\_next:** Suivant  
    **label\_previous:** Précédent
- 4. Videz le cache

Créer des slugs pour les  
articles avec slugify



# Utilisation du package

- Dépôt officiel: <https://packagist.org/packages/cocur/slugify>
- Installation: composer require cocur/slugify
- Utilisez un générateur de slug pour obtenir des url plus cohérentes notamment pour les articles.  
Actuellement pour accéder à un article particulier, l'url se termine par article-num. Pour obtenir plus de sens vous souhaiteriez avoir le titre de l'article. Pour ce faire, ce titre doit être nettoyé (sanitize) pour correspondre aux normes des url (pas d'espace, maj, accents et caractères spéciaux).
- Il faut disposer dans la table des articles d'un champ appelé slug (string – taille du titre – not null).

# Création du slug via les fixtures

```
$faker = Factory::create('FR-fr');
$slugify = new Slugify();
for ($i = 1; $i <= 30; $i++) {
    $title = $faker->words(4, true);
    $advert = new Advert();
    $advert
        ->setTitle($title)
    ->setSlug($slugify->slugify($title))
    ...
    $manager->persist($advert);
}
$manager->flush();
}
```

# Création du slug via l'entité

- Lors de l'ajout ou de la mise à jour d'un article dans le site vous devez passer par l'entité pour ajouter le slug et non les fixtures. C'est pourquoi la gestion du slug doit être intégrée dans la classe. Pour ce faire nous allons ajouter une méthode et faire appel à deux événements du cycle de vie d'une entité.
- Ajoutez l'annotation suivante dans l'entité concernée:

```
/**  
 * @ORM\Entity(repositoryClass="App\Repository\AdvertRepository")  
 * @ORM\HasLifecycleCallbacks  
 */
```

# Création du slug via l'entité

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

### Webographie

- Symfony, <https://symfony.com>
- The Symfony PHP framework – GitHub, <https://github.com/symfony/symfony>
- SymfonyCasts - PHP and Symfony Video Tutorial Screencasts, <https://symfonycasts.com>
- Construisez un site web à l'aide du framework Symfony 5, <https://openclassrooms.com/fr/courses/5489656-construisez-un-site-web-a-l-aide-du-framework-symfony-5>
- Automatiser la documentation avec NelmioApiDocBundle, <https://zestedesavoir.com/tutoriels/1280/creez-une-api-rest-avec-symfony-3>
- Packagist, <https://packagist.org>

### Bibliographie

- **Fabien Potencier**, SYMFONY 5.2 : The Fast Track, 2019.
- **Yves Rocamora**, Apprendre à développer des applications web avec PHP et Symfony, Eni – 2020.