

# Symfony 4

Framework PHP

# Introduction

Les frameworks

Le MVC

Installation de Symfony

Création d'un projet de démonstration

Les environnements de développement et de production

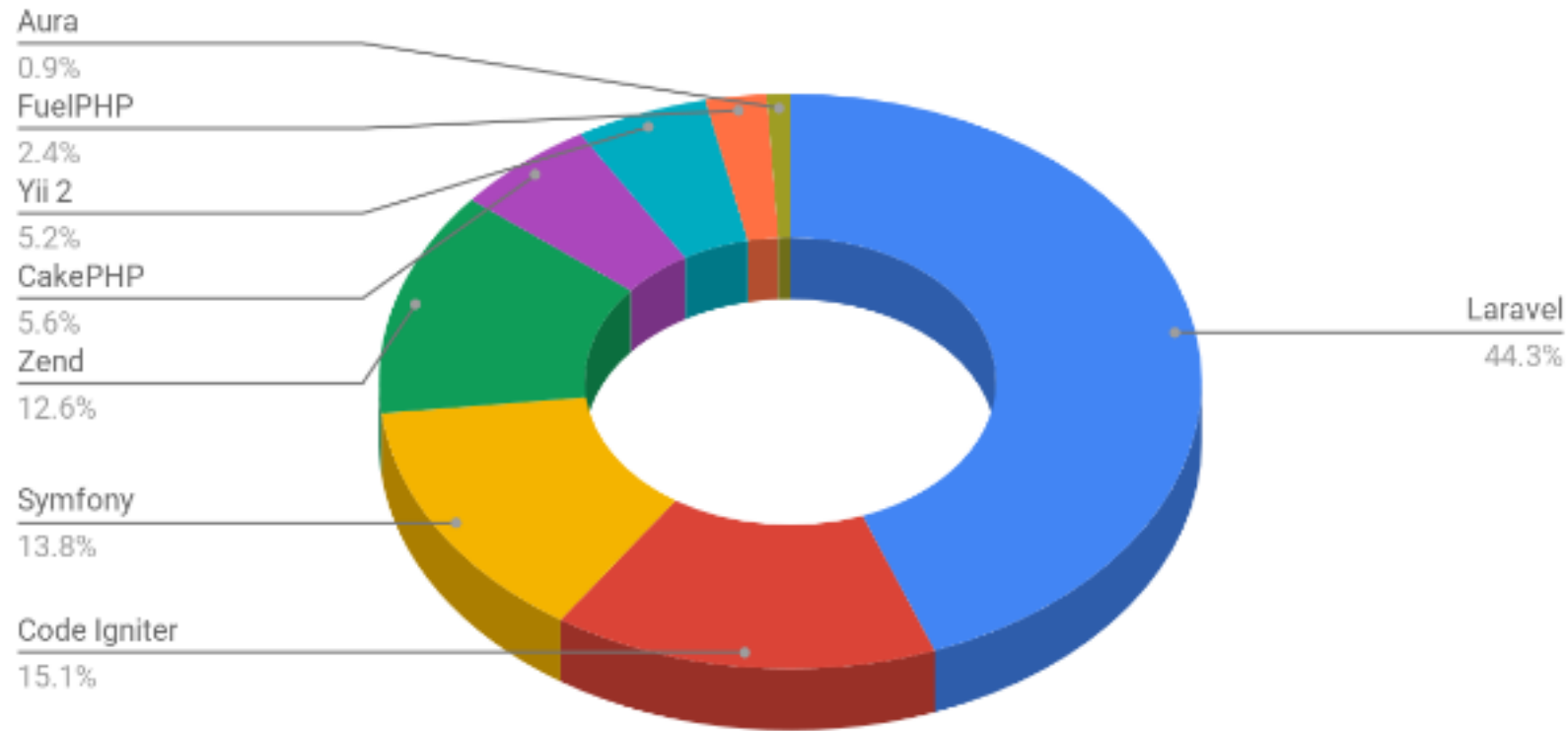
Notions supplémentaires

# Principaux frameworks 2018



# Utilisation des frameworks 2018

PHP Framework used for Project recently



# Qu'est-ce qu'un framework ?

Un Framework est une sorte de cadre applicatif structurant 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 dizaines 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.

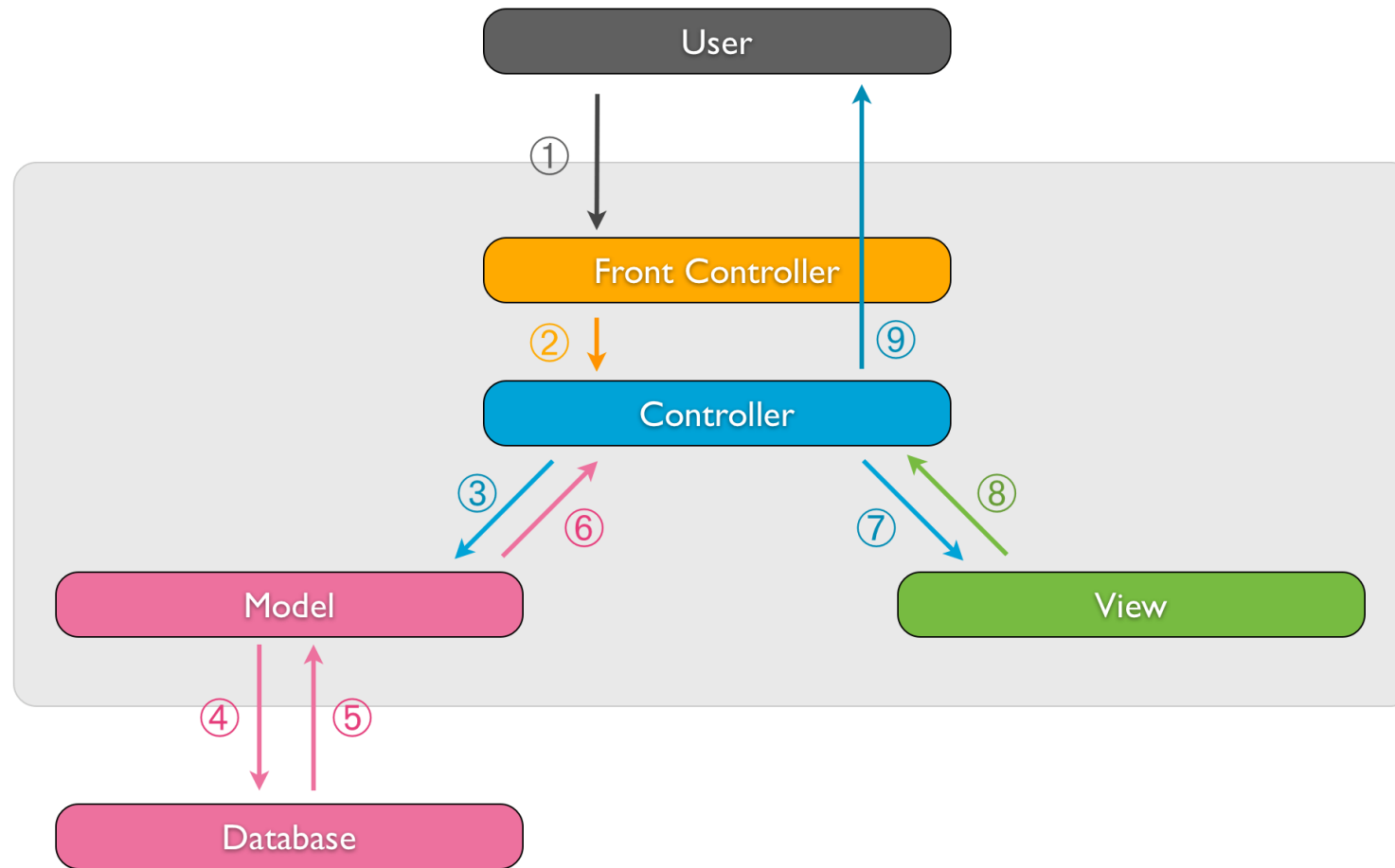
# Caractéristiques d'un framework

## PHP Objet et MVC

Fournir des bibliothèques éprouvées permettant de s'occuper des tâches usuelles (formulaire, validation, sécurité, mail, template,...).

1. Assurer une couche d'abstraction avec les données (ORM).
2. 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
3. Améliore la façon dont vous travaillez et facilite le travail équipe.
4. 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 les 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 fait 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 en collision.

# Installation de Symfony

# Installation de Composer

Depuis l'arrivée de Symfony 4, la seule façon d'installer le framework est de passer par le gestionnaire de dépendances Composer.

Installez composer:

Url : <https://getcomposer.org/download/>

Pour Windows : télécharger et exécuter [Composer-Setup.exe](#)

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)

Exécutez la commande suivante dans un dossier de projet du serveur  
([www/symfony](http://www.symfony.com))

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

Composer installe la dernière version stable de Symfony. La commande create-project crée une nouvelle application Symfony dans le dossier spécifié en fin de ligne. J'utilise pour la démonstration "bases" comme nom de dossier.

Composer récupère les fichiers (les packages), crée l'arborescence du framework et copie l'ensemble des fichiers nécessaires (bibliothèques et dépendances).

Les informations d'installation sont reprises sur le site officiel de Symfony (<https://symfony.com>).

# Installation de Symfony (2)

Installation de dépendances complémentaires

!!! Ces installations doivent être exécutées dans le dossier du projet en cours

## Security Checker

Symfony fournit un utilitaire appelé "Security Checker" pour vérifier si les dépendances de votre projet contiennent des failles de sécurité connues.

Cet utilitaire sera exécuté automatiquement chaque fois que vous installez ou mettez à jour une dépendance dans l'application. Si une dépendance contient une vulnérabilité, un avertissement sera affiché.

```
composer require sensiolabs/security-checker --dev
```

# Installation de Symfony (3)

Apache – Pack

Composant nécessaire pour l'utilisation du routeur interne.

`composer require symfony/apache-pack`

# Installation de Symfony (3) – non obligatoire

## Requirements Checker

Symfony 4.0 nécessite l'exécution de PHP 7.1.3 ou supérieur, en plus d'autres exigences mineures. Pour simplifier les choses, Symfony fournit un outil permettant de vérifier rapidement si votre système répond à toutes ces exigences. Exécutez cette commande pour installer l'outil:

```
composer require symfony/requirements-checker
```

L'outil de vérification crée un fichier appelé `check.php` dans le répertoire public/ de votre projet. Ouvrez ce fichier avec votre navigateur pour vérifier les exigences.



# Installation de Symfony (4) – non obligatoire

Une fois que vous avez corrigé tous les problèmes signalés, désinstallez Requirements Checker pour éviter de divulguer des informations internes sur votre application aux visiteurs:

```
composer remove symfony/requirements-checker
```

Ouvrez éventuellement le fichier php.ini pour résoudre les problèmes les plus connus:

```
realpath_cache_size = 5M
```

```
zend_extension = php_opcache.dll
```

# Affichage après installation

Une fois l'installation terminée, testez le bon fonctionnement du Framework en vous rendant sur l'url suivante :

<http://localhost/symfony/bases/public>

Ou

Via PhpStorm si l'interpréteur PHP est configuré

Le fichier affiché dans le navigateur est `app/Ressources/views/default/index.html.twig`

Nous le remplacerons ou le modifierons par la suite.

## Welcome to Symfony 4.1.1



Your application is now ready. You can start working on it at:

`C:\wamp64\www\symfony04\bases\`

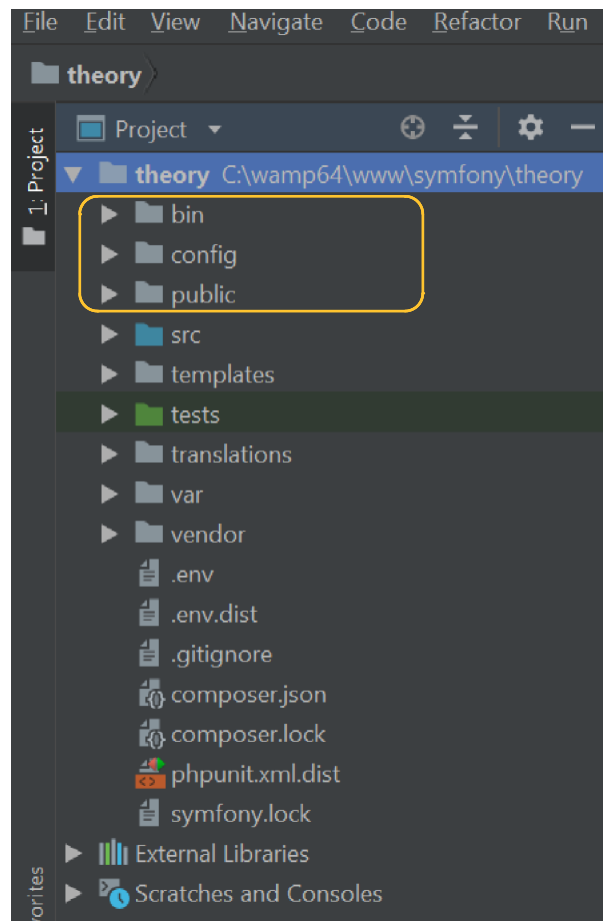
### What's next?



Read the documentation to learn

[How to create your first page in Symfony.](#)

# l'architecture des dossiers d'un projet vide

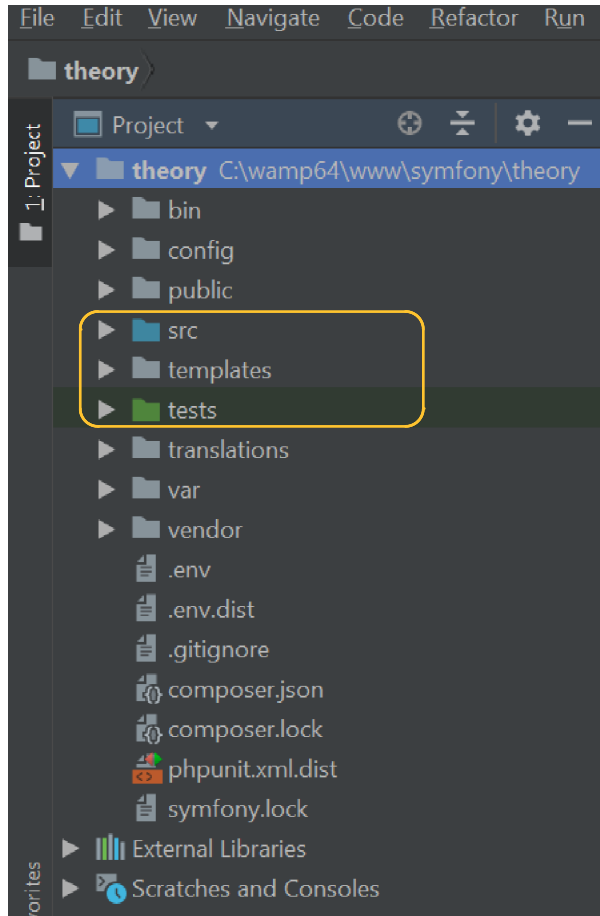


**bin:** ce répertoire contient l'exécutable dont nous allons nous servir en mode console pendant le développement. La commande pour y accéder est la suivante: `php bin/console`

**config:** contient tous les fichiers de configuration au format yaml (routing, security, configuration de développement ou de production,...).

**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

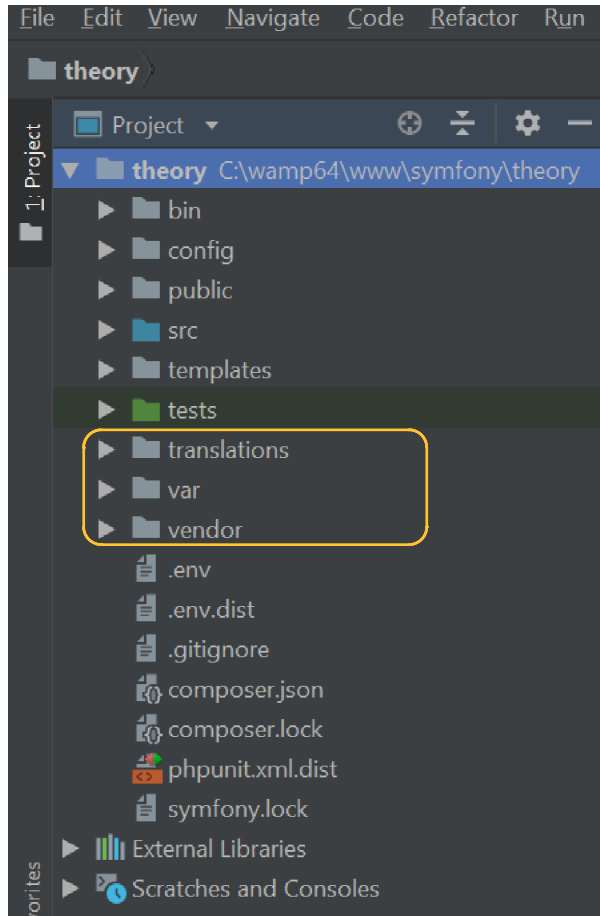


**src:** ce dossier comporte toute l'architecture php de votre site web avec les Controllers, les Entités, les forms (que nous créerons par la suite).

**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 pour la création de toutes nos vues.

**tests:** dossier réservé aux tests fonctionnels et unitaires.

# l'architecture des dossiers d'un projet vide

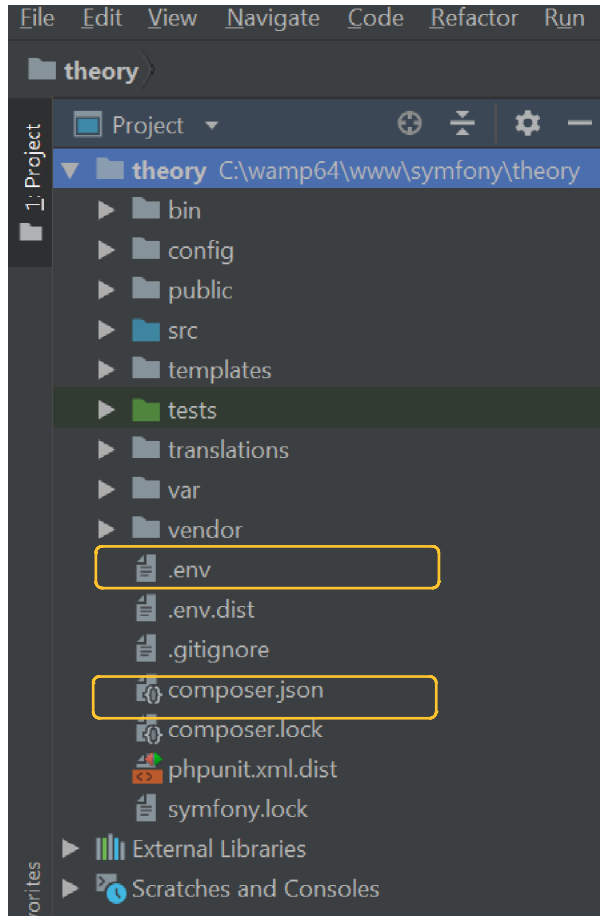


**translations:** contiendra les fichiers de traductions en Json ou xml.

**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).

**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

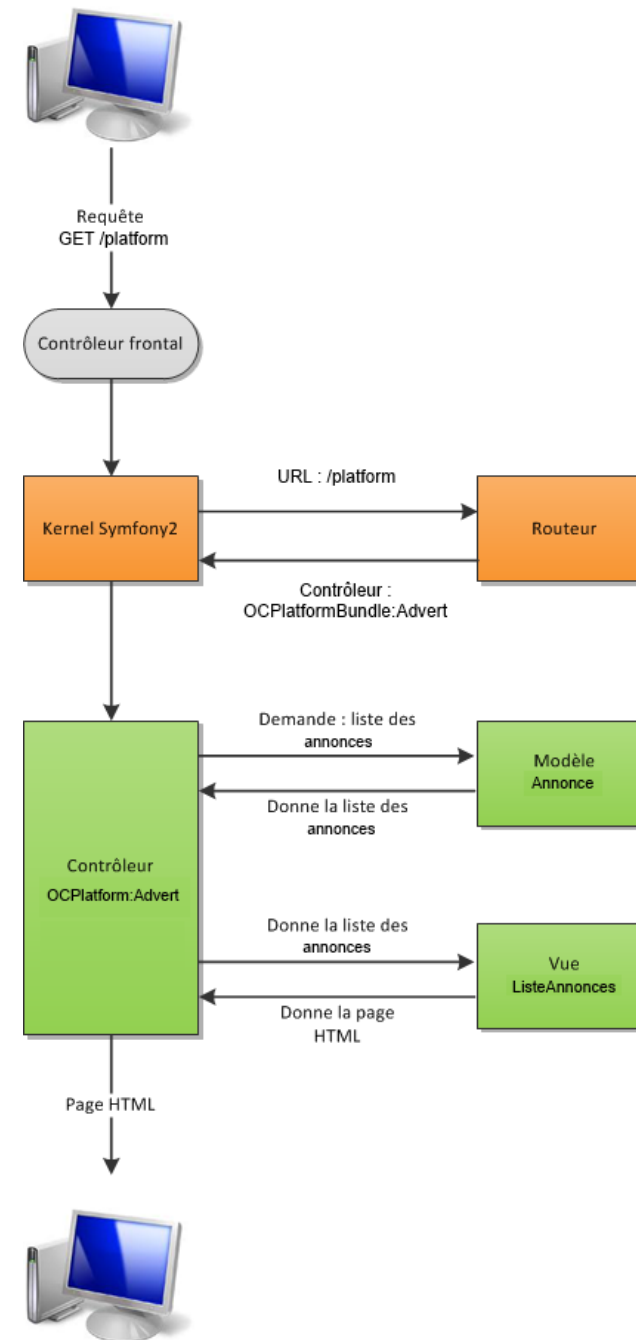
Namespace

La route

Action du contrôleur



# Une requête Symfony



# Principes de base

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

1. Créer une route : une route est l'URL (ex : /about) pour votre page et pointe sur un contrôleur;
2. Créer un contrôleur : un contrôleur est une fonction PHP que vous écrivez pour 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.

# Objectif de cette partie

Créer sa première page et découvrir les composants essentiels du framework:

1. Le contrôleur 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 fixés par le framework: le nommage, la structure des dossiers et le positionnement des fichiers.

# 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. Le contrôleur doit toujours porter le même nom que le fichier.

La ligne use... s'ajoute toute seule 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 bibliothèques 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 celle 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

La route va, à partir d'une URL, déterminer quel contrôleur appeler et avec quels arguments (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

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



# Création de l'action du contrôleur

La méthode intercepte la requête et retourne une réponse. La méthode `return()` renvoi un rendu. Ici une vue (un template Twig) mais ça peut-être un Json ou autre chose.

Créez un dossier « home » dans templates

Créez y un fichier « index.html.twig » et écrivez un contenu HTML.

Ajoutez la méthode (action) 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');
    }
}
```

Pour visualiser la page saisissez l'url suivante dans le navigateur: <http://localhost/symfony/bases/public>

# 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 (la fonction PHP) qui devra être exécuté ;

la méthode correspondante à la route est exécuté : 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, 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.

Grace à son système de cache le rendu 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 %}
```

Etendre 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

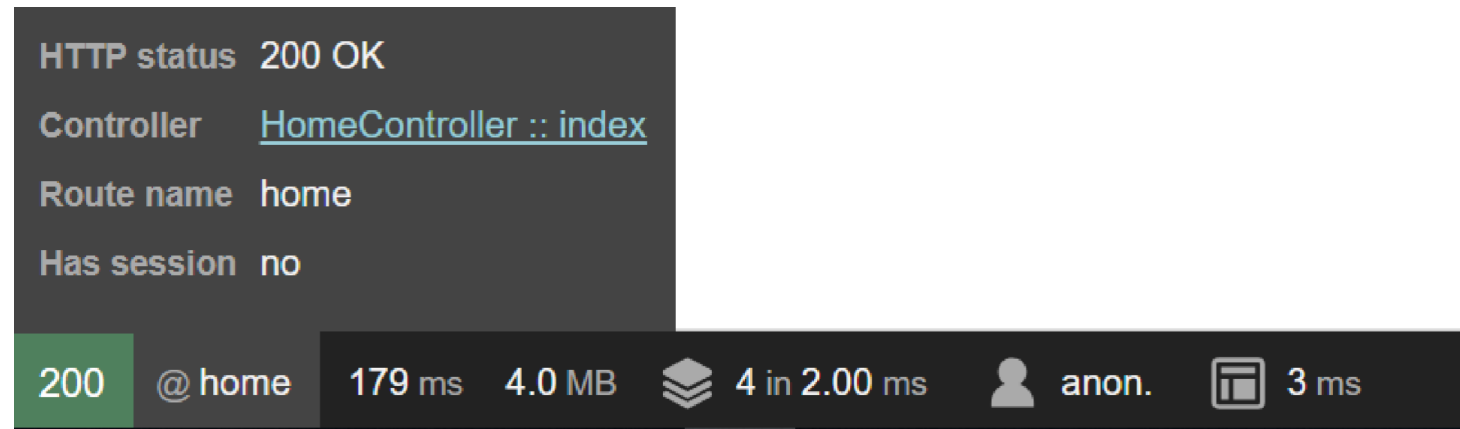
Profiler

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/'. Underneath the green bar, a dark green bar displays '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' (gear icon), 'Performance' (clock icon), 'Validator' (checkmark icon), 'Forms' (document icon), 'Exception' (bug icon), 'Logs' (book icon), 'Events' (radio tower icon), 'Routing' (cross icon), and 'Cache' (stack icon).

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, showing a table with the following data:

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

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

# La fonction dump

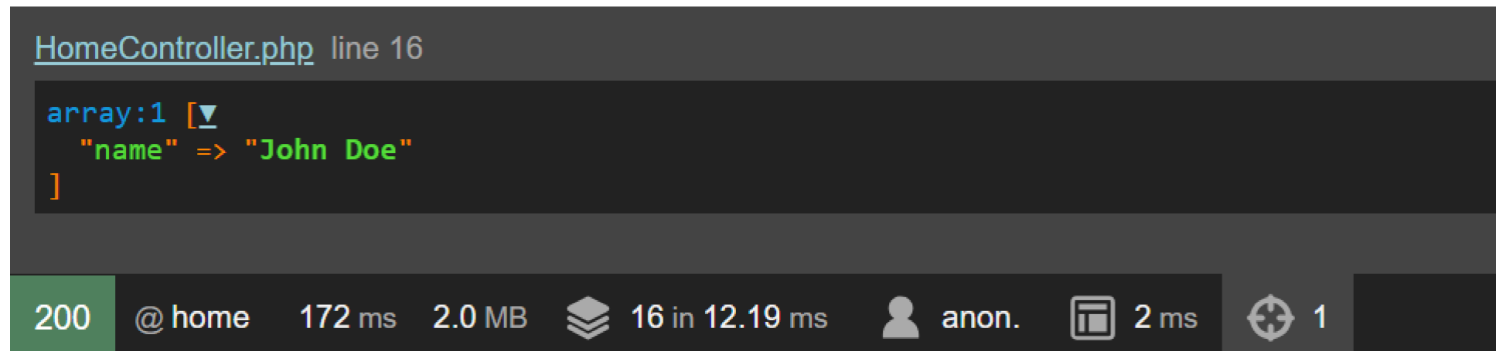
Nous utilisons régulièrement la fonction `var_dump()` qui affiche les informations structurées d'une variable, y compris son type et sa valeur. Nous la faisons suivre d'une instruction d'arrêt pour afficher le contenu sans exécuter le reste du script.

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

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

```
dump($var)
```

```
dump($request)
```



The screenshot shows a Symfony Debug Toolbar. The top bar indicates the file `HomeController.php` at line 16. The main panel displays a dump of an array: `array:1` containing a single element `"name" => "John Doe"`. The bottom bar shows the status: `200`, `@ home`, `172 ms`, `2.0 MB`, `16 in 12.19 ms`, `anon.`, `2 ms`, and `1`.

# Notions de base supplémentaires

Les routes

Les contrôleurs

Les vues

Les services

Les frameworks CSS

L'utilitaire de lignes de commandes

# Les routes

Symfony propose plusieurs techniques vous permettant de créer vos routes:

1. Sous la forme d'annotations (Symfony 3)  

```
/**  
 * @Route("/blog", name="bloglist")  
 */
```
2. Sous la forme d'un fichier YML
3. Sous la forme d'un fichier XML (moins fréquent)
4. Sous la forme d'un fichier PHP (rare)

## YML

```
# app/config/routing.yml
```

```
blog_list:  
    path:    /blog  
    defaults: { _controller: App:Blog:list }
```

```
blog_show:  
    path:    /blog/{slug}  
    defaults: { _controller: App:Blog:show }
```

Path: l'URL à capturer

Defaults: les paramètres de la route, notamment le nom du contrôleur appeler.

# 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('home/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 argument de la méthode render().

```
return $this->render('home/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 }}

```
<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 les deux classes: Response et Request qui sont des représentations objet des concepts HTTP.

# 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

# Introduction aux services

Un service est simplement un objet PHP qui remplit une fonction et peut être utilisé n'importe où dans votre code.

Cette fonction peut être simple : envoyer des e-mails, vérifier qu'un texte n'est pas un spam, etc. Mais elle peut aussi être bien plus complexe : gérer une base de données (le service Doctrine !), etc.

Un service est donc un objet PHP qui a pour vocation d'être accessible depuis n'importe où dans votre code. Pour chaque fonctionnalité dont vous aurez besoin dans toute votre application, vous pourrez créer un ou plusieurs services (et donc une ou plusieurs classes et leur configuration). Un service est avant tout une simple classe.

# Introduction aux services

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

Pour éviter une surcharge du code métier dans le contrôleur, on doit le déporter et en créer un service.

# Introduction aux services

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

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

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



# Utilisation du service dans le contrôleur

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

Il suffit de passer un paramètre dans la méthode et de le typer avec le nom de la classe.

Ensuite, on invoque la méthode `clean()` sur l'objet `$clean` et on stocke le résultat dans `$login`.

Symfony s'occupe d'instancier l'objet à votre place.

# 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 DockBlock.

```
/**  
 * @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 web (css et js)
3. Copier les fichiers « bootstrap.min.css » et « bootstrap.min.js »
4. Téléchargez jQuery et ajoutez le fichier dans le dossier « js »
5. 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)

## Les liens JS

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

# Création de liens dans les vues

Utilisation du helper path() de Twig

```
<a class="nav-link" href="{{ path('ex03') }}">About Us</a>
```

Path('name'): indiquez la valeur du name="ex03" 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("/ex03/{id}", name="ex03")
```

# 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) publié 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 et les utilisateurs (Etape 2)

# Projet de départ

1. Installez symfony dans un nouveau projet: `posts`  
    `composer create-project symfony/website-skeleton posts` (dans le dossier avec le projet précédent)  
    `composer require sensiolabs/security-checker --dev` (dans le dossier `posts`)  
    `composer require symfony/apache-pack` (dans le dossier `posts`)
2. Configurer le fichier `.env`  
    `DATABASE_URL=mysql://root:"@127.0.0.1:3306/articles`
3. Créez une base de données vide: `posts`  
    `Php bin/console doctrine:database:create`
4. Installez manuellement les fichiers Bootstrap et jquery
5. Créez une feuille de style vide: `main.css`
6. 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 %}Welcome!{% endblock %}</title>
    {% block stylesheets %}
      <link rel="stylesheet" href="{{ asset('css/bootstrap.min.css') }}">
      <link rel="stylesheet" href="{{ asset('css/main.css') }}">
    {% endblock %}
  </head>
  <body>
    {% block body %}{% endblock %}
    {% block javascripts %}
      <script src="{{ asset('js/jquery.js') }}"></script>
      <script src="{{ asset('js/bootstrap.min.js') }}"></script>
    {% endblock %}
  </body>
</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 a été ajouté. 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 index()
    {
        return $this->render('post/index.html.twig');
    }
}
```

# Vue: index.html.twig

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

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

{% block body %}
    <div class="jumbotron">
        <h1 class="display-4">WebArticles</h1>
        <p class="lead">Check out many articles on web technologies.</p>
        <a class="btn btn-primary btn-lg" href="#" role="button">Login
to write </a>
    </div>
{% endblock %}
```

# L'ORM Doctrine

## Object relation mapper (lien objet-relation)

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. Pour ce faire, vous devez créer (automatiquement) les entités (entity) qui se chargeront de faire l'interface entre la DB et le traitement sous la forme d'objets.

# Doctrine en mode console

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: **PostEntity**

L'entité et son repository viennent d'être créés. Vous devez maintenant saisir les champs (attributs) de la future table et de l'entité. Vous devrez fournir au fur et mesure les propriétés (type, length,...). La clé primaire a été ajoutée automatiquement (id).

# Les attributs

## title

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

Field type [string]:

Field length [255]:

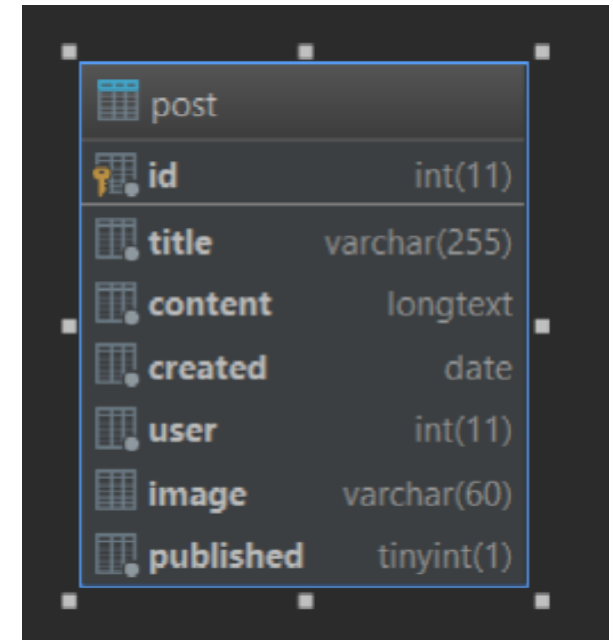
Can this field be null in the database (nullable) (yes/no) [no]:

## 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]:



A screenshot of a database table definition interface. The table is named 'post'. It contains seven fields: 'id' (int(11) with a primary key icon), 'title' (varchar(255)), 'content' (longtext), 'created' (date), 'user' (int(11)), 'image' (varchar(60)), and 'published' (tinyint(1)).

post	
 id	int(11)
 title	varchar(255)
 content	longtext
 created	date
 user	int(11)
 image	varchar(60)
 published	tinyint(1)

# Les attributs

## created

New field name (press <return> to stop adding fields): created

Field type [string]: date

Can this field be null in the database (nullable) (yes/no) [no]:

## user (auteur)

New field name (press <return> to stop adding fields): user

Field type [string]: integer

Can this field be null in the database (nullable) (yes/no) [no]:

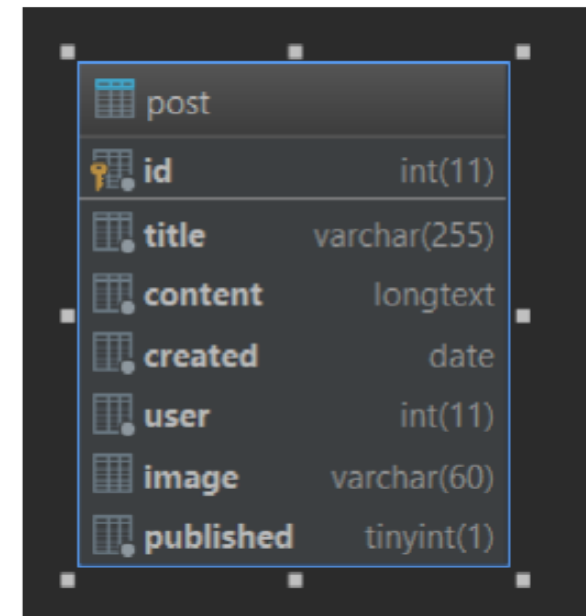
## Image

New field name (press <return> to stop adding fields): image

Field type [string]:

Field length [255]: 60

Can this field be null in the database (nullable) (yes/no) [no]: yes



A screenshot of a database schema editor window titled 'post'. It displays a list of fields for a table named 'post'. The fields are: 'id' (int(11)) with a primary key icon, 'title' (varchar(255)), 'content' (longtext), 'created' (date), 'user' (int(11)), 'image' (varchar(60)), and 'published' (tinyint(1)).

post	
id	int(11)
title	varchar(255)
content	longtext
created	date
user	int(11)
image	varchar(60)
published	tinyint(1)

Terminez l'ajout des champs avec « Enter »



# La migration vers la base de données

Doctrine vient de terminer la mise à jour du modèle: Post

Ouvrez et inspectez le fichier Post.php dans src/Entity

Doctrine vient de vous créer automatiquement la classe avec les attributs et leurs propriétés (annotations) ainsi que leurs getters et setters.

Création du fichier contenant le code sql permettant la création de la table:

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

Maintenant, nous pouvons migrer vers la base de données en exécutant le code SQL généré par Symfony dans le fichier de migration:

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

# Structure d'une entité

Namespace

Use

Annotation de la classe

Classe

Annotation du premier attribut (id)

Attribut \$id en private

...

Annotation du premier Getter

Getters -> Accès

Annotation du premier Setter

Setters -> Modifications

...

# 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`

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

...

# Modifier une entité

## Ajout d'un champ (attribut)

Ajoutez l'attribut et l'annotation correspondante dans le fichier (Post.php)

Le champ booléen « published » permettra d'afficher l'article après validation. Ne vous occupez pas du getter et du setter, ils seront ajoutés par Doctrine.

```
/**
 * @ORM\Column(type="boolean")
 */
private $published;
```

# Modifier une entité

En mode console saisissez (générer les nouveau getters et setters):

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

Enter a class or namespace to regenerate [App\Entity]: <<enter>>

-> updated: src/Entity/Post.php

Créez le fichier de migration:

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

Migrer la modification (nouveau champ) en base de données. Si dans le dossier Migrations se trouvent dans anciennes versions, vous devez les effacer pour conserver uniquement le dernier.

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

Confirmer la migration.

Doctrine retourne la requête suivante:

```
ALTER TABLE post ADD published TINYINT(1) NOT NULL;
```

# 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 doctrine/doctrine-fixtures-bundle
```

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. La diapositive suivante est un exemple de la documentation officielle permettant de persister des objets dans une table `product`.

```
namespace App\DataFixtures;

use App\Entity\Product;
use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Common\Persistence\ObjectManager;

class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        // create 20 products!
        for ($i = 0; $i < 20; $i++) {
            $product = new Product();
            $product->setName('product ' . $i);
            $product->setPrice((10, 100));
            $manager->persist($product);
        }

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



# Plus loin avec Faker

Il s'agit d'un composant supplémentaire qui est capable de créer des données aléatoires de n'importe quels types sans devoir utiliser des fonctions ou des tableaux. Il se chargera de persister des milliers d'objets pour remplir temporairement des tables de produits, de personnes, des associations...

La documentation de la librairie est disponible à l'adresse suivante:

<https://github.com/fzaninotto/Faker#formatters>

Installation:

```
composer require --dev fzaninotto/faker
```

Migration en DB:

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

```
use App\Entity\Post;
use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Common\Persistence\ObjectManager;
use Faker;
class PostFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        $faker = Faker\Factory::create('fr_FR'); // initialize le générateur faker
        for($i = 0; $i < 20; $i++) {
            $post = new Post();
            $post->setTitle($faker->text(30));
            $post->setContent($faker->text(700));
            $post->setCreated($faker->dateTimeThisYear('now'));
            $post->setUser($faker->numberBetween(1, 10));
            $post->setImage($i'.png');
            $post->setPublished(1);
            $manager->persist($post);
        }
        $manager->flush();
    }
}
```

# 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.

# 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 et une route dans le contrôleur: `viewPosts()`

Récupérer le Repository (permet de créer les requêtes) créé par Doctrine:

```
$repository = $this->getDoctrine()->getRepository('Post::class');
```

Utilisation de méthodes magiques pour récupérer les enregistrements:

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

Passer les données dans la vue:

```
return $this->render('posts/viewpost.html.twig', ['posts' => $posts]);
```

# Le contrôleur et la méthode listing()

```
<?php
namespace App\Controller;
use App\Entity\Post;
use ...

class PostController extends AbstractController
{
    /**
     * @Route("/", name="posts")
     */
    public function viewPosts()
    {
        $post = $this->getDoctrine()
            ->getRepository(Post::class)
            ->findAll();
        return $this->render('post/index.html.twig', ['posts' => $post]);
    }
}
```

# Lister les enregistrements dans la vue

Dans la vue, vous devez ajouter une boucle pour afficher l'ensemble des enregistrements envoyés par le contrôleur. L'affichage des données est simple `post.attribut` entre accolades.

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

```
<section id="posts" class="container">
  <div class="col-12">
    {% for post in posts %}
      <p>{{ post.title }} - {{ post.created|date('d/m/Y') }}</p>
    {% endfor %}
  </div>
</section>
```



# Vue améliorée

Pour afficher les images en Twig, on utilise le helper `asset()`. Vous devez renseigner le chemin d'accès et le concatener ( `~` ) avec l'attribut `image` de la DB.

```
<tbody>
  {% for post in posts %}
  <tr>
    <td></td>
    <td>{{ post.title }}</td>
    <td>{{ post.created|date('d-m-Y') }}</td>
  </tr>
  {% endfor %}
</tbody>
```

# Afficher un seul article

Contrôleur

Créer la vue avec un seul article

Créer le lien dans la vue avec tous les articles

# Le contrôleur

Dans le contrôleur « PostControlleur »

1. Ajoutez une action (méthode) onePost
2. Créez la route avec un paramètre (id)
3. Utilisez la méthode find(\$id) et non findAll

```
/**
 * @Route("/article-{id}", name="onepost")
 * @param int $id
 * @return mixed
 */
public function viewOnePost($id)
{
    $post = $this->getDoctrine()
        ->getRepository(Post::class)
        ->find($id);
    return $this->render('post/detail.html.twig', ['posts' => $post]);
}
```

# La vue pour un seul enregistrement

```
<div class="jumbotron">
  <h1 class="display-4">{{ posts.title }}</h1>
  <div class="row">
    <div class="col-md-3">
      
    </div>
    <div class="col-md-9">
      <h3 class="lead">{{ posts.created|date('d-m-Y') }}</h3>
      <p>{{ posts.content }}</p>
      <hr class="my-4">
      <a class="btn btn-primary" href="{{ path('posts') }}"
role="button">Liste</a>
    </div>
  </div>
</div>
```

# 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.

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

# Helper methods

**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(«roman»)`. 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.






# La vue index.html.twig

## WebArticles

Check out many articles on web technologies.

[Post article](#)

### Liste des articles

Image	Titre de l'article	Création
	<a href="#">Enim modi laborum sed.</a>	04-05-2018
	<a href="#">Nam ea dolore velit.</a>	18-08-2018
	<a href="#">Qui omnis voluptatum vel ab.</a>	10-04-2018
	<a href="#">Rem est animi et vitae.</a>	22-04-2018
	<a href="#">Ea rem optio eaque molestiae.</a>	16-10-2017

# La vue detail.html.twig

## Est omnis eligendi ut fuga.

08-02-2018



Dolorum voluptas quam reprehenderit doloremque in. Repellat recusandae explicabo eum amet similique. Voluptatem iusto voluptatem eaque qui veritatis nihil sint. Sapiente cumque quia voluptatem tempore et officiis et. Quas suscipit maiores omnis eaque aut doloremque. Consectetur et necessitatibus iusto ipsam. Laboriosam dolorem laudantium autem ut nisi. Omnis et optio enim voluptas aut. Vero ea similique recusandae quisquam. Placeat doloribus aperiam voluptatibus. Dolorum et sit molestiae et in eos. Sed facere nihil fugit ullam et ducimus. Ea rerum et maiores repellat. Est delectus quia quos nam nobis veritatis. Inventore quia et provident praesentium a.

---

[Liste](#)



# 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**

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

```
<?php
```

```
class PostfType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array
$options)
    {
        $builder
            ->add('field_name')
        ;
    }
}
```

# La création du formulaire

Le principe est d'invoquer la méthode `add()` sur l'objet `$builder` pour chaque champ de formulaire correspondant à l'entité `Post`.

Syntaxe:

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

les paramètres:

- le nom de l'attribut: 'title'
- le type de champ: `TextType::class`

Avec l'auto-complétion de PhpStorm le use nécessaire pour chaque type de champ est ajouté automatiquement (vérifier quand même)

# La classe PostType

```
public function buildForm(FormBuilderInterface $builder, array
$options)
{
    $builder
        ->add('title', TextType::class)
        ->add('content', TextareaType::class)
        ->add('created', DateType::class)
        ->add('user', IntegerType::class)
        ->add('image', TextType::class, ['required' => false,
'empty_data' => 'default.png'])
        ->add('published', ChoiceType::class, ['choices' =>
['Oui' => 1, 'Non' => 0]])
        ->add('submit', SubmitType::class);
    ;
}
```

# Les types et les options de champs

Liste complète sur la doc officielle: <https://symfony.com/doc/current/forms.html>

Les options (les attributs html) sont définies dans un tableau associatif.

exemple pour un champ de type date:

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

Exemple pour générer le label:

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

# Le contrôleur pour afficher le formulaire

```
/**
 * @Route("/new", name="new")
 */
public function addPost(Request $request)
{
    $post = new Post();
    $form = $this->createForm(PostType::class, $post);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        $em = $this->getDoctrine()->getManager();
        $em->persist($post);
        $em->flush();
        return $this->redirectToRoute('posts');
    }
    return $this->render('post/new.html.twig', [
        'post' => $post,
        'form' => $form->createView(),
    ]);
}
```



# La vue avec le formulaire

```
{% block body %}
    <div class="jumbotron">
        <h1 class="display-4">Ajouter un article</h1>
        <div class="row">
            <div class="col-md-10">
                {{ form(form) }}
            </div>
        </div>
    </div>
{% endblock %}
```

# Apperçu de la vue

## Ajouter un article

Titre de l'article

Contenu de l'article

Date de création

25 ▾

09 ▾

2018 ▾

Auteur

Image

Publication

Oui ▾

# 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'
    debug: '%kernel.debug%'
    strict_variables: '%kernel.debug%'
    form_themes: ['bootstrap_4_layout.html.twig']
```

# La vue générale

Dans le bouton du Jumbotron, ajouter le path permettant d'accéder au formulaire d'insertion.

```
<a class="btn btn-primary btn-lg" href="{{ path('new') }}"  
role="button">Post article </a>
```



# Supprimer un article

Principe

La méthode delete()

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

# La méthode pour effacer un enregistrement

```
public function delete(Request $request, $id)
{

    $em = $this->getDoctrine()->getManager();
    $repository = $this-
>getDoctrine()->getRepository(Post::class);
    $post = $repository->find($id);
    $em->remove($post);
    $em->flush();

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

# Le lien dans la vue

```
<td>
    <a href="{{ path('delete', {id:post.id}) }}">
        <i class="fas fa-times text-danger"></i>
    </a>
</td>
```

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

Ajoutez une classe Bootstrap et une icône Font Awesome. Pour l'icône, vous pouvez créer le lien directement vers leur site ou télécharger le fichier CSS et les Webfonts.

# Mettre à jour un enregistrement

Principe

Contrôleur

Vue (formulaire)

Vue générale (index)



# Le contrôleur

```
/**
 * @Route("/edit-{id}", name="edit")
 */
public function edit(Request $request, Post $post)
{
    $form = $this->createForm(PostType::class, $post);
    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()) {
        $this->getDoctrine()->getManager()->flush();
        return $this->redirectToRoute('posts');
    }
    return $this->render('post/edit.html.twig', [
        'post' => $post,
        '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 %}
    <div class="jumbotron">
        <h1 class="display-4">Editer un article</h1>
        <div class="row">
            <div class="col-md-10">
                {{ form(form) }}
            </div>
        </div>
    </div>
{% endblock %}
```

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

# Etape 2 du projet - les relations

Ajouter des catégories

Gérer les erreurs

Ajouter des utilisateurs

# L'évolution du projet

A ce stade, nous avons terminer l'utilisation du CRUD dans un projet ne contenant qu'une seule table. Il est temps d'envisager des entités supplémentaires pour rendre ce projet plus performant en terme de fonctionnalités.

Nous allons ajouter la gestion des catégories et des utilisateurs mais nous ne nous limiterons pas qu'à la gestion plus avancée de la base de données. En effet, nous ajouterons la gestion des erreurs, l'upload de photos, la sécurité...

- Dupliquez le projet `posts` en `postsfinal` et ouvrez le dans PhpStorm.
- Corrigez le fichier `.env` pour configurer la nouvelle db: `articles_rel`.

```
DATABASE_URL=mysql://root:''@127.0.0.1:3306/articles_rel
```

- Créez la nouvelle base de données:

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

# Les relations avec Doctrine

Introduction

La classe PostType

# Introduction

L'objectif de cette partie est de pouvoir 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 de 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 un attribut correspondant 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`.

# Création de l'entité category

Pour illustrer la relation OneToMany, nous allons créer une nouvelle entité (category) permettant de gérer les catégories des articles.

Création de l'entité (category)

Nous allons la créer en mode console avec le générateur d'entité:

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

The Entity shortcut name: category

L'attribut "category" (string - 120) | l'id est automatiquement inséré

Ensuite, il faut migrer l'entité:

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

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



# Ajouter des objets dans l'entité

Ajoutez des catégories: manuellement ou via une fixture. Attention, l'upload des fixtures en DB concerne aussi l'entité Post. Si vous ne le souhaitez pas, déplacer PostFixtures.php temporairement.

Fixture: `CategoryFixtures`

```
class CategoryFixtures extends Fixture
{
    private $categories = ['PHP 7', 'Symfony', 'Laravel', 'Back
End', 'Security', 'Front End', 'Python'];
    public function load(ObjectManager $manager)
    {
        foreach($this->categories as $cat) {
            $category = new Category();
            $category->setCategory($cat);
            $manager->persist($category);
        }
        $manager->flush();
    }
}
```

# Modifier l'entité Post

Pour l'instant, nous ne pouvons pas créer la jointure entre les deux entités. Nous devons d'abord ajouter un attribut «category», ajouter les valeurs numériques correspondantes aux catégories et après modifier l'annotation pour réaliser la relation entre les deux attributs (post.category et category.id).

Ajoutez un attribut:

```
/**
 * @ORM\Column(type="integer")
 */
private $category;
```

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

# Le composant Security

Introduction

# 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 de la base de données.

Projet: posts\_user

DB: posts\_user

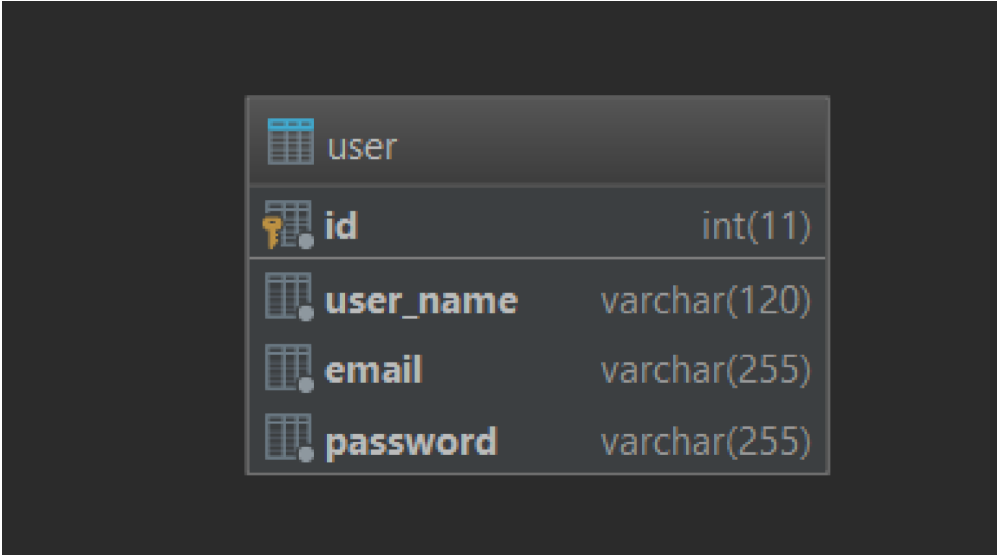
# Création de l'entité User

Les commandes ont déjà été expliquées plus en avant. Ici, je fais un listing fonctionnel de ces commandes.





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

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

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



A screenshot of a database schema diagram for a 'user' table. The table is represented as a box with a grid icon and the name 'user'. Inside the box, the columns are listed: 'id' (int(11)) with a primary key icon (a yellow key), 'user\_name' (varchar(120)), 'email' (varchar(255)), and 'password' (varchar(255)). Each column has a small grid icon next to its name.

user	
 <b>id</b>	int(11)
 <b>user_name</b>	varchar(120)
 <b>email</b>	varchar(255)
 <b>password</b>	varchar(255)

# 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 la dia suivante se trouve la vue du formulaire et 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»

## Inscription

Nom d'utilisateur

Votre Email

Votre mot de passe

Confirmation du mot de passe

Submit

```
public function buildForm(FormBuilderInterface $builder, array
$options)
{
    $builder
        ->add('userName', TextType::class, ['label' => 'Nom
d\'utilisateur'])
        ->add('email', EmailType::class, ['label' => 'Votre Email'])
        ->add('password', PasswordType::class, ['label' => 'Votre
mot de passe'])
        ->add('confirmPassword', PasswordType::class, ['label' =>
'Confirmation du mot de passe'])
        ->add('submit', SubmitType::class);
}

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



# Modification de l'entité User

Le champ confirm Password que vous ajouté dans ResistrationType doit avoir une correspondance dans l'entité User. Ajouté y l'attribut:

```
public $confirmPassword;
```

Sans indiquer l'annotation @ORM ni le getter et setter car il ne doit pas migrer dans la DB.

# Le contrôleur

Le code de la méthode `registration()` ressemble à peu de chose près à la méthode `addPost()` du contrôleur `PostController`.

```
/**
 * @Route("/inscription", name="inscription")
 */
public function registration(Request $request)
{
    $user = new User();
    $form = $this->createForm(RegistrationType::class, $user);
    $form->handleRequest($request);
    if($form->isSubmitted() && $form->isValid()) {
        $em = $this->getDoctrine()->getManager();
        $em->persist($user);
        $em->flush();
    }
    return $this->render('security/registration.html.twig', [
        'form' => $form->createView()
    ]);
}
```

# 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

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;
```

# 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;
```

# Configurer security.yaml

Nous devons ajouter dans le fichier une méthode de cryptage proposée par Symfony. Sous la rubrique security, ajoutez le type d'encodage (encoders), l'entité que vous souhaitez crypter et le type d'algorithme (bcrypt).

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

# 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)

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



# Modifier l'entité User

Pour pouvoir utiliser le cryptage et d'autres fonctionnalités sur les users, vous devez implémenter l'interface **UserInterface** dans l'entité.

Quand on implémente l'interface sur la classe User, PhpStorm indique qu'il manque un certain nombre de méthodes dans la classe (getRoles, getSalt...). Suivant l'architecture objet quand on implémente une interface, il faut déclarer ses méthodes.

Heureusement PhpStorm peut le faire pour nous. ALT + ENTER sur le nom de l'interface -> add. Les trois méthodes obligatoires sont ajoutées à la fin de l'entité. Pour l'instant, elles sont vides.

```
class User implements UserInterface
```

```
public function getRoles()
```

```
{
```

```
    return ['ROLE_USER'];
```

```
}
```

N'oubliez pas le return array avec un  
ROLE\_USER

```
public function getSalt()
```

```
{
```

```
}
```

```
public function eraseCredentials()
```

```
{
```

```
}
```

# 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"
 * )
 */
```

# La méthode login()

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

# la vue pour s'identifier: login.html.twig

```
<h2>Identification</h2>
<form action="{ path('login') }" method="post">
    <div class="form-group">
        <input type="email" class="form-control"
required name="_username" placeholder="Mail">
    </div>
    <div class="form-group">
        <input type="password" class="form-control"
required name="_password" placeholder="Password">
    </div>
    <div class="form-group">
        <button type="submit" class="btn-
success">Connexion</button>
    </div>
</form>
```

# Security.yaml: Firewalls et Providers

```
providers:
  in_memory: { memory: ~ }
  in_user:
    entity:
      class: App\Entity\User
      property: email
firewalls:
  dev:
    pattern: ^/(_(profiler|wdt)|css|images|js)/
    security: false
  main:
    anonymous: true

    provider: in_user

    form_login:
      login_path: login
      check_path: login
```

# Déconnexion

Pour terminer, nous devons ajouter une fonctionnalité permettant de se déconnecter. C'est assez simple, on ajoute une méthode `logout()` dans le contrôleur qui ne renvoie rien et c'est le core de Symfony qui se chargera de réaliser l'opération.

```
/**
 * @Route("/logout", name="logout")
 */
public function logout()
{
}
```

Après, il faut soit ajouter un bouton connexion soit un bouton logout. Tout dépendra si l'utilisateur est déjà ou pas connecté.



# 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:

```
<div>
    {% if not app.user %}
        <a class="btn btn-success" href="{{ path('login')
    }}">Connexion</a>
    {% else %}
        <a class="btn btn-danger" href="{{ path('logout')
    }}">Logout</a>
    {% endif %}
</div>
```

# 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 utilisateur.

# 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

Ici 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

# 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. 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 donc 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, donc 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;
```