



MVC / Symfony

Développer une application avec le framework
Symfony afin d'assurer un code de qualité qui facilitera
son évolution et sa maintenance.

Vous avez dit Symfony ?



Qu'est ce que Symfony ?

Un framework Le 1er framework PHP utilisé pour créer des sites / applications web	Du code pré-existant Des fonctionnalités sont directement utilisables dès l'installation de symfony
Une communauté + 3 000 contributeurs + 600 000 développeurs Des millions de sites...	Une philosophie Des bonnes pratiques, un code standardisé facilement compréhensible, en toute flexibilité

Pourquoi Symfony ?

Utiliser un framework n'est pas obligatoire, mais permet de :

- Développer plus rapidement,
- Développer plus proprement,
- Organiser son code pour s'y retrouver facilement,
- Utiliser du code déjà existant / réutiliser son propre code,
- Avoir un cadre, tout en restant libre,
- Trouver beaucoup de réponses sur les forums (symfony a une grande communauté)

Une fois qu'on est à l'aise avec Symfony, on ne peut plus s'en passer !

Exemple : Requête MySQL et affichage des données

PHP procédural

```
// on se connecte à MySQL
$db = mysql_connect('localhost', 'login', 'password');

// on sélectionne la base
mysql_select_db('nom_de_la_base', $db);

// on crée la requête SQL
$sql = "SELECT * WHERE nom = 'Dupont' ORDER BY date ASC FROM famille_tbl";

// on envoie la requête
$req = mysql_query($sql) or die('Erreur SQL !<br>'. $sql. '<br>'. mysql_error());

// on fait une boucle qui va faire un tour pour chaque enregistrement
while($data = mysql_fetch_array($req))
{
    // on affiche les informations de l'enregistrement en cours
    echo '<b>' . $data['nom'] . ' ' . $data['prenom'] . '</b> (' . $data['statut'] . ')';
    echo ' <i>date de naissance : ' . date_format($data['date'], "d-m-Y"); . '</i><br>';
}

// on ferme la connexion à mysql
mysql_close();
```

Dupond Grégoire (Grand-père), date de naissance : 17-05-1932

Dupond Germaine (Grand-mère), date de naissance : 15-02-1939

Dupond Gérard (Père), date de naissance : 22-12-1959

Dupond Marie (Mère), date de naissance : 02-03-1961

Dupond Julien (Fils), date de naissance : 17-05-1985

Dupond Manon (Fille), date de naissance : 29-11-1990

Exemple : Requête MySQL et affichage des données

Avec Symfony

```
$personnes = $familleRepository->findBy(  
    array( 'nom' => 'Dupont' ),  
    array( 'date' => 'ASC' )  
) ;
```

```
{% for personne in personnes %}  
    <b>{{ personne.prenom }} {{ personne.nom }}</b> ({{ personne.statut }}),  
    <i>date de naissance : {{ personne.date|date('d-m-Y') }}</i> <br>  
{% endfor %}
```

Dupond Grégoire (Grand-père), date de naissance : 17-05-1932

Dupond Germaine (Grand-mère), date de naissance : 15-02-1939

Dupond Gérard (Père), date de naissance : 22-12-1959

Dupond Marie (Mère), date de naissance : 02-03-1961

Dupond Julien (Fils), date de naissance : 17-05-1985

Dupond Manon (Fille), date de naissance : 29-11-1990

Exemple : les liens

```
<a href="/a-propos-de-moi">À propos de moi</a>
```

Pour modifier cette URL, il faudra la modifier sur toutes les pages de notre site sur lesquelles ce lien est présent.

```
<a href="{{ path('a_propos') }}>À propos de moi</a>
```

Dans Symfony, on donne des noms à nos routes.

Si l'on veut changer cette URL, il suffira de le faire à un seul endroit

Exemple : Système de traductions

```
{# accueil.html.twig #}
<div class="col-12">
    <h1>{{ 'accueil.bienvenue' |trans }}</h1>
</div>
```

```
# messages.fr.yaml
accueil:
    bienvenue: Bienvenue sur mon site
```

```
# messages.en.yaml
accueil:
    bienvenue: Welcome on my website
```

Exemple : La sécurité des espaces membres

```
access_control:
    - { path: '^/staff/', role: ROLE_STAFF }
    - { path: '^/membre/', role: ROLE_MEMBRE }
    - { path: '^', role: IS_AUTHENTICATED_ANONYMOUSLY }
```

Un utilisateur non connecté qui se rendrait sur l'espace **/membre** sera automatiquement redirigé vers la page de connexion.

Un membre qui essaierait d'accéder à l'espace **/staff** sera automatiquement bloqué, tout simplement car il n'est pas staff.

Prérequis

Prérequis

Avant de commencer... Quelques notions de base :

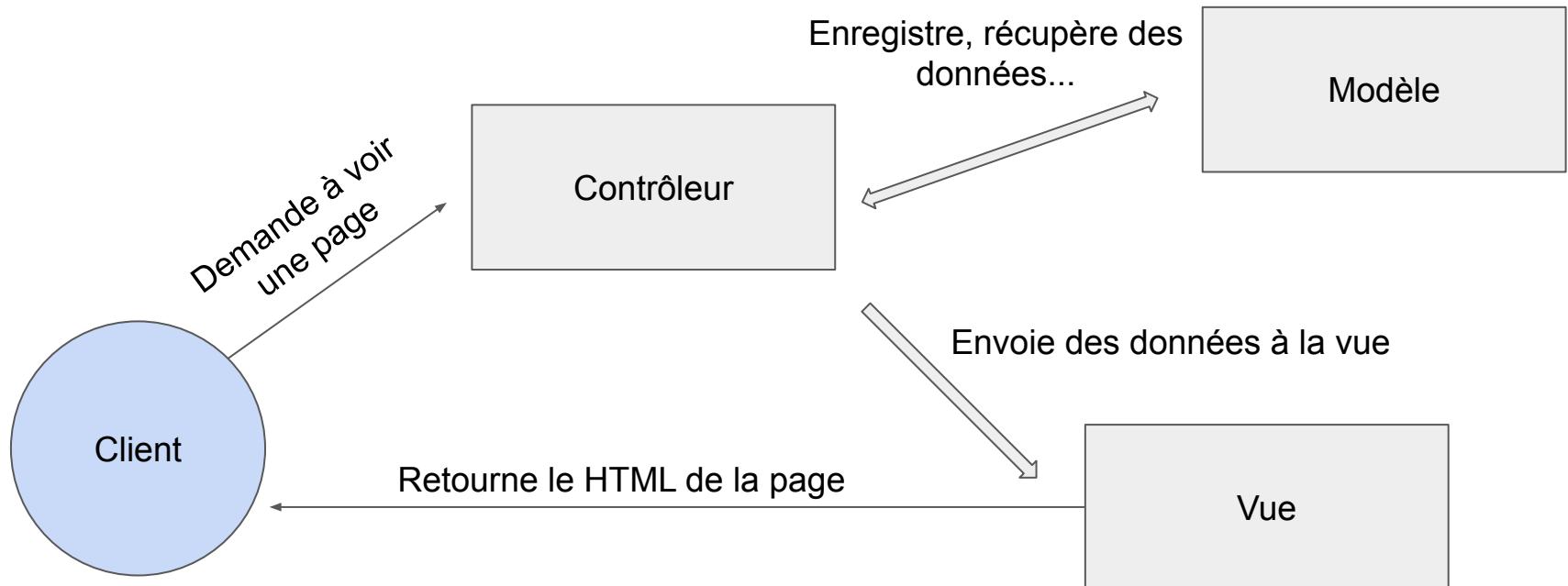
- Modèle-Vue-Contrôleur ou MVC
- Moteur de template
- Contrôleur frontal
- Les routes
- Composer
- Packagist
- Versions de Symfony
- Commandes Symfony (bin/console)

Modèle-Vue-Contrôleur ou MVC

Le motif (pattern) MVC permet de bien structurer son code, en séparant la logique de l'application en 3 parties :

- **Modèle** : Gère les données du site, on y retrouve entre autre le schéma de la base de données.
- **Vue** : Gère l'affichage, on y retrouve presque uniquement du HTML, avec un peu de PHP pour afficher les variables, faire des boucles..
- **Contrôleur** : Gère la logique, prend les décisions. Cette partie fait le lien entre le modèle et la vue, elle récupère les données, les traite, et renvoie le texte à afficher dans la Vue. On y retrouve uniquement du PHP.

Modèle-Vue-Contrôleur ou MVC



Moteur de template

Permet de séparer la logique PHP du code HTML.

Toutefois, il faut bien écrire des variables, faire des boucles, ajouter des conditions.. Twig permet d'ajouter du code dynamique très lisiblement et proprement via son propre langage.

Nous n'aurons donc pas de PHP dans nos vues, mais du Twig

Ex : Pour afficher une variable :

Il suffit d'écrire {{ variable }} au lieu de <?php echo \$variable; ?>

Contrôleur frontal

Un Contrôleur est une fonction PHP

Son rôle : interpréter une requête, retourner une réponse.

Entre les deux, il va généralement faire le lien entre des services, des modèles, puis appeler une vue qu'il retournera en réponse.

Il est le chef d'orchestre entre les différentes parties de notre application.

Dans Symfony, c'est en réalité le Kernel qui interprète les requêtes, et en fonction de la route (l'URL demandée), appelle le bon Contrôleur. Le Contrôleur fait ses opérations, puis appelle la Vue en lui passant des variables. C'est donc le HTML contenu dans la Vue qui sera retourné en réponse de la requête du client.

Les routes

Le rôle du routeur est de déterminer, à partir d'une URL, quel Contrôleur appeler et quels arguments lui passer.

URL : `www.monsite.fr/articles/mon-super-article`

Routeur : Appeler le Contrôleur ArticleSingle(`$slug`), passer : `slug = mon-super-article`

Nous dirons au routeur : Lorsqu'on te demande une URL du type `/articles/quelquechose`, tu appelleras le Contrôleur ArticleSingle, et tu lui passeras `quelquechose` en argument `$slug`.

Composer

Composer est un outil de gestion des dépendances PHP. Il permet d'installer rapidement et de gérer les dépendances PHP d'un projet.

Dans notre projet, nous aurons besoin d'utiliser des librairies, qui sont des fichiers de code préexistants qui nous permettent de ne pas réinventer la roue nous-même. Notre projet dépend de ces librairies, qu'on appelle des dépendances.

Ex : Nous voulons utiliser Stripe. Stripe nous propose une librairie PHP prête à l'emploi, nous aurons uniquement à modifier certaines variables, et à appeler des fonctions déjà prêtes.

Mais pour fonctionner, Stripe utilise d'autres librairies, dans des versions précises, que nous devons également inclure dans notre projet.

De plus, les développeurs de Stripe font évoluer leur librairie pour corriger des bugs, proposer de nouvelles fonctionnalités... De nouvelles versions de la librairie Stripe sont donc régulièrement proposées.

Nous aurons alors un fichier **composer.json**, à la racine de notre projet, dans lequel nous listerons nos dépendances. En utilisant la commande **composer install**, composer se chargera de toutes les télécharger dans un dossier **/vendor** avec les bonnes versions, compatibles les unes avec les autres.

Composer s'utilise en lignes de commandes, via un invite de commandes.

Composer - versions et contraintes

Les versions des librairies sont composées de 3 chiffres. Ex Symfony 4.4.2

En français : {Version Majeure}.{Version Mineure}.{Correctif}

En anglais : {Major}.{Minor}.{Patch}

Contraintes

4.4.2	Version Exacte
4.4.*	$\geq 4.4.0$ et $<4.5.0$ ou $<5.0.0$
$\sim 4.4.2$	$\geq 4.4.2$ et $<4.5.0$
~ 4.4	$\geq 4.4.0$ et $<5.0.0$
$\wedge 4.3.2$	$\geq 4.3.2$ et $<5.0.0$

Packagist

<https://packagist.org/>

Packagist est le répertoire principal pour les librairies PHP.

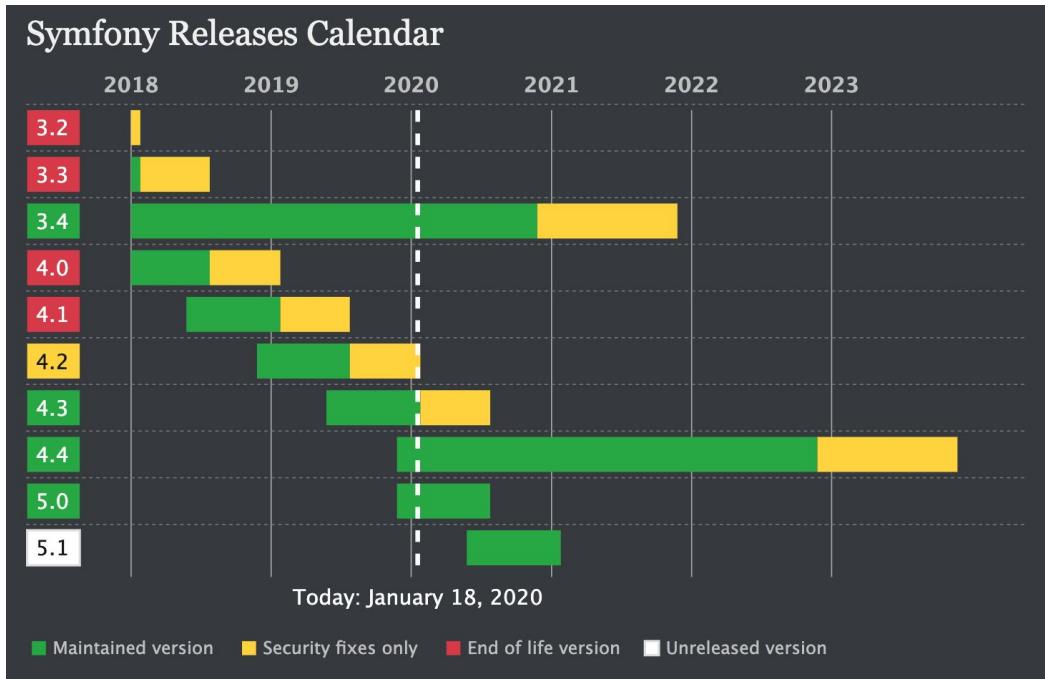
C'est sur Packagist que les développeurs publient leur code PHP, dans les différentes versions, que nous pourrons installer via Composer.

Nous pouvons nous rendre sur le site de Packagist pour rechercher une librairie en particulier, voir différentes informations concernant cette librairie, ses versions, une documentation, et une ligne de commande pour l'ajouter à notre projet :

`composer require package-name`

Versions de Symfony

<https://symfony.com/releases>



Symfony suit un agenda fixe : les versions mineures sont publiées tous les 6 mois en Mai et Novembre, les versions majeures tous les 2 ans.

Les versions en 4 (3.4 ou 4.4) sont appelées LTS (Long Term Support). Elles sont maintenues, mises à jour plus longtemps.

Commandes Symfony

Symfony propose des commandes par défaut, qui permettent d'effectuer rapidement différentes actions. Il faut les écrire dans l'invité de commandes.

Elles sont accessibles via le répertoire bin/console

Syntaxe : **bin/console command:command --argument**

Pour voir la liste de toutes les commandes disponibles, tapez : **bin/console**

Il est possible de créer ses propre commandes en faisant :

bin/console make:command

Plan du cours - le framework Symfony

- Installation et découverte de l'arborescence
- Création d'un Contrôleur et d'une Vue
- Découverte de Twig
- Création et injection de services
- Utilisation d'une base de données avec Doctrine
- Création de formulaires
- Upload de fichiers
- Création d'un espace membre sécurisé avec une hiérarchie
- Utilisation d'événements

TWIG

TWIG - Moteur de template

- Permet de séparer la logique de l'affichage
- Rend l'écriture du code plus lisible
- Dispose de fonctions qui simplifient l'écriture du code
- Permet de réutiliser facilement son code

Vue - Contrôleur

```
/**  
 * @Route("/test-url", name="test_url")  
**/  
public function test_url()  
{  
    $bonjour = "Bonjour à tous !";  
  
    return $this->render('twig/test_url.html.twig', [  
        'bonjour' => $bonjour,  
    ]);  
}
```

```
<!DOCTYPE html>  
<html>  
<head>  
    <meta charset="utf-8" />  
    <title>Test Url</title>  
</head>  
  
<body>  
  
    <h1>{{ bonjour }}</h1>  
  
</body>  
</html>
```

Dans le Contrôleur, on crée une fonction.
Dans les annotations, `@Route` donne des
infos comme l'URL et le nom de la route.
Notre fonction retourne le template twig
(`render`), auquel on aura éventuellement
passé des variables.

Notre template twig est globalement
une page html, dans laquelle nous
affichons nos variables, ou tout autre
élément twig
Un peu comme en php, mais avec
une autre syntaxe.

Créer un Contrôleur et une Vue

1. Taper la commande : **bin/console make:controller**
2. Entrer le nom du Controller (le format est toujours AbcController)
3. Symfony va créer automatiquement
 - a. Le fichier src/controller/AbcController.php
 - b. Le fichier templates/abc/index.php
4. Dans le Controller, modifier les paramètres de la route (url + name)
5. Depuis le navigateur, accéder à l'URL choisie

Dans le Contrôleur

namespace indique le répertoire de notre classe PHP AbcController dans l'arborescence de notre projet. En interne, Symfony charge nos Contrôleurs en faisant use App\Controller\AbcController;

use Ma\Super\Dependance; Charger les dépendances de notre projet via leur namespace

class AbcController extends AbstractController Nous avons créé une classe qui étend de la classe AbstractController, donc hérite des fonctions PHP de cette classe.

À l'intérieur de notre classe AbcController, nous trouvons une première fonction **index** (que nous pouvons renommer), qui retourne un template en lui passant des variables.

L'annotation **@Route** permet de définir une route, donc lorsque cette URL sera demandée, cette la fonction **index** sera appelée, et retournera le template **abc/index.html.twig**

Dans la Vue

{% extends 'base.html.twig' %} indique que base.html.twig est le template parent de notre template actuel.

{% block title %}...{% endblock %} permet de faire remonter l'information au block title défini dans le template parent. Le contenu éventuel du block title du template parent sera écrasé par le contenu du block title du template enfant.

{% block body %}...{% endblock %} même principe... Le template parent base.html.twig contient un block title, qui sera rempli par ce que nous mettons dans le block title du template enfant (abc/index.html.twig)

TWIG - Structure

La structure HTML de notre page (doctype, html, head, body, ...) restera la même sur toutes les pages de notre site. Nous la définissons donc dans le template base.html.twig. Ce template sera le parent de toutes les autres pages de notre site.

Nous pouvons également ajouter au base.html.twig les fichiers CSS et JS que dont nous aurons besoin sur toutes les pages. Par exemple les fichiers Bootstrap.

Puisque toutes nos vues étendent du fichier base.html.twig, elles seront toujours incluses dans cette structure, et les assets CSS et JS seront toujours incluses.

TWIG - Héritage

templates/base.html.twig

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>{% block title %}Mon super site{% endblock %}</title>
</head>

<body>

    {% include 'base/header.html.twig' %}

    {% block body %}{% endblock %}

    {% include 'base/footer.html.twig' %}

</body>
</html>
```

base.html.twig est le template parent.

On le crée une seule fois et il sera utilisé sur toutes nos pages. C'est la structure du site.

Il n'est appelé par aucun Contrôleur directement.

templates/dossier/page.html.twig

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

{% block title %}Ma super page{% endblock %}

{% block body %}

    <h1>Bienvenue sur ma page</h1>
    <p>Ici nous allons parler de ...</p>

{% endblock %}
```

Ce template est appelé par un Contrôleur. Il étend de base.html.twig et hérite donc de sa structure.

Le block title écrase son parent, et le block body en fait de même.

TWIG - Héritage

Depuis un template enfant...

Le contenu du block parent est rendu par défaut.

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

Écrase le contenu du block parent, et le remplace

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

{{ parent() }} permet de rendre le contenu du block parent, puis d'ajouter notre script à la suite

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

Permet de rendre le contenu du block parent, après avoir ajouté notre script

Le principe est le même, pour tous les blocks. C'est la ligne `{% extends ... %}` qui permet de définir le template parent.

TWIG : Les filtres

Les filtres permettent de modifier rapidement des variables. Ils s'appliquent en utilisant le caractère pipe |

```
 {{ (-10)|abs }}                      {# 10 #}
 {{ 'mon titre'|capitalize }}          {# Mon titre #}
 {{ 'mon titre'|title }}               {# Mon Titre #}
 {{ 'mon titre'|upper }}              {# MOPN TITRE #}
 {{ 'now'|date('Y-m-d') }}            {# 2020-01-27 #}
 {{ [1,2,3]|join('-') }}             {# 1,2,3 #}
 {{ [1,2,3]|join(', ', ' et ') }}    {# 1, 2 et 3 #}
```

TWIG : Les tags

Les tags permettent d'accéder rapidement et joliment à différentes fonctions

```
% set users = ['Victor', 'John', 'Paul'] %# Définit la valeur d'une variable #

{# Boucle sur les valeurs d'un tableau #}
{% for user in users %}
    L'utilisateur s'appelle {{ user }}

    {# Condition if - elseif - else #}
    {% if user == 'Victor' %}
        et il est super !
    {% elseif user == 'John' %}
        {# ... #}
    {% else %}
        {# ... #}
    {% endif %}

    <br>
{% endfor %}
```

Exercice - Twig

--- CODE PHP FOURNI ---

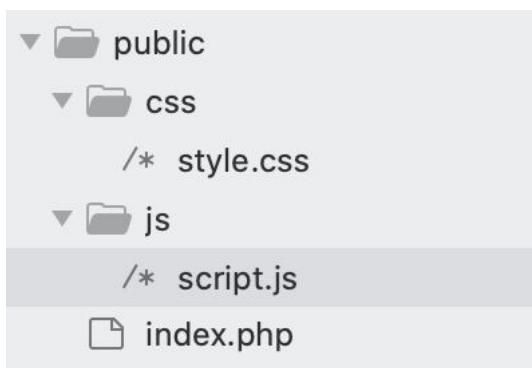
1. Affichez tous les membres selon l'exemple vu précédemment :

Dupond Grégoire (Grand-père), *date de naissance : 17-05-1932*

2. Affichez uniquement les membres de la famille Dupont
3. Affichez les 3 premières lignes, les noms de familles en MAJUSCULES
4. Affichez uniquement les membres nés APRÈS 1950
5. Affichez-les des plus vieux aux plus jeunes
6. Affichez la première ligne en bleu, la dernière en rouge
7. Sautez une ligne toutes les 2 lignes

Twig : Inclure des assets (css, js, images...)

Pour créer des liens, on utilise **path()**, pour créer les liens vers les assets, on utilise **asset()**.
Les dossiers /css , /js , /images sont à créer dans le dossier /public



arborescence

base.html.twig
J'inclus Bootstrap +
mes propres styles et scripts

```
<!doctype html>
<html lang="fr">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

    {% block stylesheets %}
        <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/
        <link rel="stylesheet" href="{{ asset('css/style.css') }}"/>
    {% endblock %}

    <title>{% block title %}Welcome!{% endblock %}</title>
</head>
<body>
    {% block body %}{% endblock %}

    {% block javascripts %}
        <script src="https://code.jquery.com/jquery-3.4.1.slim.min.js" integrity="sha384-J6q/inzagPiJyF7ZgkPVtAjEJSe5lVfBQHnYqD8WZJGZLwqjZ" i
        <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js" i
        <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/js/bootstrap.min.js"
        <script src="{{ asset('js/script.js') }}"/></script>
    {% endblock %}
</body>
</html>
```

Twig - Inclusion de template

```
<body class="d-flex flex-column h-100">

{%
    include 'front/header.html.twig'
    with { posts: allPosts, variable2: 10 }
%}

<div class="container">
    {% include 'shared/alert.html.twig' %}

    {% block body %}{% endblock %}
</div>

{% include 'front/footer.html.twig' %}

{% block javascripts %}
    ...
{% endblock %}

</body>
```

Le tag **include** permet d'inclure un template. Cela me permet de séparer mon code proprement en différents fichiers.

Si mon header.html.twig a besoin de variables, je peux les passer grâce au tag **with {...}**

Twig - Inclusion de Contrôleur dans un template

```
 {{ render(controller(
    'App\\Controller\\FrontController::header',
    { variable1: true, variable2: 'Test' }
)) }}
```

Dans base.html.twig , nous souhaitons inclure notre header.

Notre header a peut-être besoin de récupérer les 3 derniers articles de blog.

Dans le cas d'un {% include %} , il faut lui passer ces 3 articles.

Une solution est de créer un Contrôleur pour le header, qui sera donc indépendant, et fera sa propre requête pour récupérer les articles

Nous pouvons inclure des Contrôleurs dans les Vues, et leur passer des variables si besoin.

Twig : Créer ses propres Filters

- php bin/console make:twig-extension → Lui donner un nom
- Adapter le code généré (Twig\XxxExtension.php)

```
namespace App\Twig;

use Twig\Extension\AbstractExtension;
use Twig\TwigFilter;

class FiltersExtension extends AbstractExtension
{
    public function getFilters(): array
    {
        return [
            new TwigFilter('prix', [$this, 'prix']),
        ];
    }

    public function prix($value)
    {
        return number_format($value, 2, ',', ' ');
    }
}
```

Je peux utiliser mon nouveau filtre dans ma vue

```
{% block body %}

    #{ Mon filter personnalisé #}
    {{ 10|prix }}

{% endblock %}
```

Twig : Créer des Functions

- php bin/console make:twig-extension → Lui donner un nom
- Adapter le code généré (Twig\XxxExtension.php)

```
namespace App\Twig;

use Twig\Extension\AbstractExtension;
use Twig\TwigFunction;
use Twig\Environment; // J'en ai besoin pour utiliser render() hors d'un Controller

class HeaderExtension extends AbstractExtension
{
    private $twig;

    public function __construct(Environment $twig) {
        $this->twig = $twig;
    }

    public function getFunctions(): array
    {
        return [
            // If your filter generates SAFE HTML, you should add a third
            // parameter: ['is_safe' => ['html']]
            new TwigFunction('displayHeader', [$this, 'displayHeader'], ['is_safe' => ['html']]),
        ];
    }

    public function displayHeader()
    {
        return $this->twig->render('front/header.html.twig');
    }
}
```

header.html.twig

```
<header>
    HEADER !!
</header>
```

Je peux utiliser ma nouvelle fonction dans ma vue

```
{# Ma fonction personnalisée #}
{{ displayHeader() }}
```

Exercice - MVC et inclusion de templates

Inclure Bootstrap sur toutes les pages de notre projet

Créer une structure de site propre avec : Header / Body / Footer. Header et Footer doivent être dans des fichiers séparés. Gardez à l'esprit que par la suite nous devrons faire des requêtes vers la BDD pour y afficher du contenu.

Créer les pages : Accueil / À propos / Articles (et Article single) / Contact avec des liens vers chaque page dans le menu du header. L'onglet Articles est un menu déroulant dans lequel nous mettrons plus tard les 3 derniers articles de blog. Ne perdez pas trop de temps sur le design, c'est la structure du site qui est importante 😊. Le formulaire de contact comporte les champs : Email + Message

Préparer les Controllers MembreController et AdminController avec une page principale pour chacun. On prépare simplement ces pages pour qu'elles soient prêtes pour la suite, nous sécuriserons les accès plus tard.

Service

Qu'est ce qu'un service ?

Un service est une classe PHP qui contient différentes fonctions, dont nous pouvons avoir besoin à différents endroits de notre site, et qu'il nous suffit d'appeler.

Le principe est le même que d'avoir un fichier **functions.php**, qu'on inclue lorsqu'on en a besoin, en PHP procédural.

Exemple : Nous avons souvent besoin d'envoyer des emails, nous pouvons créer un Service : EmailService, avec plusieurs fonctions : email_contact(), email_inscription(), email_reset_password(), email_notification()...

Où créer un Service ?

Service

```
<?php

namespace App\Service;

class TestService {

    private $bonjour;

    public function __construct() {
        $this->bonjour = 'Salut !';
    }

    public function dire_bonjour()
    {
        return $this->bonjour;
    }
}
```

Contrôleur

```
use App\Service\TestService;

class TestController extends AbstractController
{
    /**
     * @Route("/test", name="test")
     */
    public function test(TestService $testService)
    {
        return $this->render('test/index.html.twig', [
            'bonjour' => $testService->dire_bonjour(),
        ]);
    }
}
```

L'injection de Service dans un Contrôleur

```
use App\Service\TestService;

class TestController extends AbstractController
{
    /**
     * @Route("/test", name="test")
     */
    public function test(TestService $testService)
    {
        return $this->render('test/index.html.twig', [
            'bonjour' => $testService->dire_bonjour(),
        ]);
    }
}
```

use indique que nous avons besoin d'importer ce Service.

Nous l'injectons en argument, lorsque nous en avons besoin.

Il suffit ensuite d'appeler la fonction du service dont nous avons besoin. Il est possible de lui passer des arguments :

\$testService->maFonction(\$arg1, \$arg2);

L'injection de Service dans un Service

```
<?php  
  
namespace App\Service;  
  
use App\Service\TestService;  
  
class Test2Service {  
  
    private $testService;  
  
    public function __construct(TestService $testService) {  
        $this->testService = $testService;  
    }  
  
    public function dire_bonjour_2()  
    {  
        $bonjour = $this->testService->dire_bonjour();  
        return $bonjour;  
    }  
}
```

Je créer un 2ème Service : Test2Service

J'ai besoin d'appeler mon 1er Service : TestService

- Je le déclare avec **use**
- Je crée une variable privée **\$nomDuService**
- **J'injecte** le service dont j'ai besoin dans la fonction **__construct()**
- Je peux ensuite l'utiliser avec
\$this->nomDuService->fonction();

Exercice : Créer un Service d'envoi d'emails

- Sur une page /contact, créer un simple formulaire de contact en method="POST" avec les champs : Email et Message
- Lorsque le formulaire est validé, récupérer les variables \$email et \$messages
- Créer un Service **EmailService**, contenant une fonction **email_contact(\$email, \$message)**
- Dans cette fonction, ajouter le code qui permet d'envoyer des emails : Composant Mailer de symfony : <https://symfony.com/doc/current/mailer.html>
- Une fois l'email envoyé, rediriger sur la même page, et afficher un message Flash de confirmation (Bootstrap alert-success)

Adresse email de test, servant à envoyer les emails :

demo.wf3.victor@gmail.com

Demo.wf3!!

Exercice 2 : Formulaire de contact pro

Votre blog prend de l'ampleur, vous faites parler de vous. Vous décidez de créer un 2ème formulaire sur une page /contact-pro, dédié aux demandes professionnelles.

Sur votre page /contact, affichez un message : Pour toute demande professionnelle, utiliser **ce formulaire** (lien vers page /contact-pro)

Le formulaire /contact-pro a les champs : Nom, Prénom, Société, Sujet, Email, Message.

Lorsque la personne valide ce formulaire, vous envoyez 2 emails :

- 1 à vous-même pour recevoir son message
- 1 à la personne qui vous a contacté, en guise d'accusé de réception

Une fois les 2 emails envoyés, vous affichez un message Flash sous forme d'une Alert Bootstrap pour dire que vous avez bien reçu son message, et qu'une confirmation lui a été envoyée par email.

Entity

ORM Doctrine (Object Relational Mapper)

Une entité est une classe PHP, qui représente une table de notre base de donnée.

Dans Symfony, la base de donnée est gérée par l'ORM Doctrine, qui est une couche d'abstraction à la base de donnée.

Concrètement, nous allons créer notre base de donnée directement dans le code en créant des classes PHP, puis, à l'aide d'annotations, Doctrine va **mapper** les propriétés de notre classe PHP, les convertir en requêtes SQL (CREATE TABLE ...), puis **migrer** notre base de donnée.

Les Entités

Les entités sont des classes PHP, qui représentent nos tables en base de donnée.

Créer une entité : **php bin/console make:entity**

Se laisser guider en répondant aux questions

Préparer la migration : **php bin/console make:migration**

Du code SQL apparaît dans src/Migrations

Migrer le schéma : **php bin/console doctrine:migrations:migrate**

La base de donnée est à jour

```
<?php  
  
namespace App\Entity;  
  
use Doctrine\ORM\Mapping as ORM;  
  
/**  
 * @ORM\Entity(repositoryClass="App\Repository\PostsRepository")  
 */  
class Posts  
{  
    /**  
     * @ORM\Id()  
     * @ORM\GeneratedValue()  
     * @ORM\Column(type="integer")  
     */  
    private $id;  
  
    /**  
     * @ORM\Column(type="string", length=255)  
     */  
    private $title;  
  
    public function getId(): ?int  
    {  
        return $this->id;  
    }  
  
    public function getTitle(): ?string  
    {  
        return $this->title;  
    }  
  
    public function setTitle(string $title): self  
    {  
        $this->title = $title;  
  
        return $this;  
    }  
}
```

On déclare le namespace

On importe la classe Mapping de Doctrine, qu'on renomme ORM

Le Repository permettra de faire des requêtes en base de donnée, sur cette classe en particulier

On déclare la classe Posts (notre future table portera ce nom)

2 propriétés privées : id + title. Les annotations @ORM permettent le mapping avec la BDD

Les getters et setters nous permettront de définir/récupérer les valeurs des propriétés de nos objets, dans un Contrôleur :

```
$post = (new Posts())  
        ->setTitle('Mon titre');  
  
dump($post->getTitle()); die();
```

Persister un objet en base de donnée

```
$em = $this->getDoctrine()->getManager();  
  
$post = (new Posts())  
    ->setTitle("Mon titre")  
    ->setActive(1)  
;  
$em->persist($post);  
$em->flush();  
  
dump($post->getId()); die();
```

Persister un objet = créer une nouvelle entrée (ligne) dans notre table en base de donnée.

Nous avons besoin de l'EntityManager de Doctrine : \$em

persist(\$objet) indique que l'on souhaite créer cet objet en BDD

flush() indique que l'on enregistre tous les changement en BDD

Modifier un objet existant

```
$em = $this->getDoctrine()->getManager();
$postRepo = $em->getRepository(Posts::class);

$post = $postRepo->find(5);
$post->setTitle("Mon titre modifié");
$post->setActive(0);

$em->flush();
```

On récupère l'EntityManager

On récupère le Repository

On récupère l'objet en question

On le modifie

Comme il existe déjà en BDD, plus besoin de le persister, il suffit de faire flush() pour enregistrer les modifications.

Le Repository

Le Repository est un Répertoire qui correspond à une Entité. Il permet de faire des requêtes en langage DQL (Doctrine Query Language), directement sur la table qui correspond à l'Entité en question.

Repository

```
class PostsRepository extends ServiceEntityRepository
{
    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, Posts::class);
    }

    /**
     * @param smallint $active      0 = inactif | 1 = actif
     */
    public function getActivePosts($active = 1)
    {
        return $this->findBy(array(
            'active' => $active
        ));
    }

    public function getPostsWithTitleLike($string)
    {
        return $this->createQueryBuilder('p')
            ->andWhere('p.title LIKE :string')
            ->setParameter('string', '%' . $string . '%')
            ->getQuery()
            ->getResult();
    }
}
```

Controller

```
use App\Repository\PostsRepository;
class TestController extends AbstractController
{
    /**
     * @Route("/test", name="test")
     */
    public function index(PostsRepository $postsRepo)
    {
        // Retourne tous les posts actifs
        $postsActifs = $postsRepo->getActivePosts();
        dump($postsActifs); die();

        // Retourne les posts dont le titre contient 'titre'
        $posts = $postsRepo->getPostsWithTitleLike('titre');
        dump($posts); die();

        return $this->render('test/index.html.twig', [
            'postsActifs' => $postsActifs,
            'postsWithTitre' => $posts,
        ]);
    }
}
```

View

```
{% for post in postsWithTitre %}
<h2>{{ post.title|title }}</h2>
Ce post est {{ post.active == 1 ? 'actif' : 'inactif' }}.
{% if not loop.last %}<hr>{% endif %}
{% endfor %}
```

Les requêtes de base

Controller

```
/** * @Route("/test", name="test") */ public function index(PostsRepository $postsRepo) { $posts = $postsRepo->findAll(); // Retourne tous les Posts $post = $postsRepo->find(4); // Retourne l'ID 4 $posts = $postsRepo->findBy( array( 'id' => [1,4], // Critères de recherche ), array('id' => 'DESC'), // ORDER BY 10, // LIMIT 0 // OFFSET ); // Comme au dessus, mais retourne 1 seul résultat $post = $postsRepo->findOneBy(array(...)) } 
```

La plupart des Requêtes de base sont très simples à faire en Symfony.

findBy permet de faire facilement un système de pagination :

```
// 2 articles par page $limit = 2; // Calcul offset. Page est récupérée dans l'URL : ?page=3 $page = $request->query->get('page') ?? 1; $page = $page < 1 ? 1 : $page; $offset = ($page - 1) * $limit; $posts = $postsRepo->findBy( array('active' => 1), // Tous les articles actifs array('date' => 'DESC'), // Triés par date $limit, // LIMIT $offset // OFFSET ); 
```

Requêtes complexes : Doctrine QueryBuilder

Le Doctrine QueryBuilder permet de construire des requêtes complexes en langage DQL.

Repository

```
public function findPostsByTitle($query, $offset = 0, $limit = 10)
{
    return $this->createQueryBuilder('p')
        ->andWhere('p.active = 1')
        ->andWhere('p.title LIKE :query')
        ->setParameter('query', '%' . $query . '%')
        ->orderBy('p.id', 'DESC')
        ->setFirstResult( $offset )
        ->setMaxResults( $limit )
        ->getQuery()
        ->getResult()
    ;
}
```

Controller

```
// Autre façon de récupérer le repository
$em = $this->getDoctrine()->getManager();
$postRepo = $em->getRepository(Posts::class);

$posts = $postRepo->findPostsByTitle('titre');
dump($posts); die();
```

Base de donnée relationnelle

Les champs de type “relation” permettent de créer des relations entre les tables de notre BDD. Quand un utilisateur commente un article, le commentaire est stocké dans une table “commentaires”, mais il correspond à 1 utilisateur, et à 1 article.

Pour afficher les commentaires sous l'article, nous allons boucler sur
\$article->getCommentaires()

Dans chaque commentaire, pour écrire le nom de l'auteur, nous écrirons
\$commentaire->getUser()->getNom()

Les 4 types de relations

Type	Exemple
ManyToOne	Plusieurs commentaires peuvent être liés à 1 seul utilisateur
OneToMany	1 seul utilisateur peut être lié à plusieurs commentaires
ManyToMany	Plusieurs catégories peuvent être liées à plusieurs articles
OneToOne	Une seule adresse de facturation peut être liée à 1 seul utilisateur

Enregistrer une relation

```
$article = $articleRepo->find(3); // On récupère l'article

// On enregistre le commentaire
$commentaire = (new Commentaires())
    ->setNom("Victor Weiss")
    ->setDate( new \DateTime() )
    ->setCommentaire("Blablabla")
    ->setArticle($article) // On lie l'article et le commentaire
;
$em->persist($commentaire);
$em->flush();
```

Lorsqu'on enregistre un commentaire, si l'on a correctement ajouté un champ de type ManoToOne à notre Entité Commentaires,

Il suffit de faire `->setArticle($article)` pour les lier.

Cela revient à dire : Ce commentaire est lié à l'article ID = 3

Récupérer les relations dans TWIG

```
{# On boucle sur les commentaires de l'article #}
{% for commentaire in article.commentaires %}
    <div class="my-3">
        <h4 class="h6">Commentaire de {{ commentaire.nom }}</h4>
        <p class="small">(Publié le {{ commentaire.date|date('d/m/Y à H:i') }})</p>
        <div>{{ commentaire.commentaire|nl2br }}</div>
    </div>
{% endfor %}
```

Notre `commentaires` est bien enregistré, et lié à notre article 3.

Dans notre Contrôleur, nous avons récupéré l'article 3 et nous l'affichons dans TWIG

Sous l'article, il suffit de boucler sur les commentaires de cet article, en faisant :

for commentaire in article.commentaires

Form

FormBuilder

Le FormBuilder de Symfony permet, en quelques lignes, de créer des formulaires qui marchent presque tout seuls.

Après avoir créé une Entité, une commande permet de créer un formulaire qui sera lié à cette Entité. Le formulaire est paramétrable dans un fichier PHP appelé FormType.

On y précise les différents inputs qui doivent s'afficher, leur type (Text, Email, Number, Textarea...) et leurs propriétés (required, min, max, maxlength...)

Symfony va se charger de convertir automatiquement ce fichier PHP en un formulaire HTML.

Les formulaires

Créer un formulaire : `php bin/console make:form`

FormType

```
<?php

namespace App\Form;

use App\Entity\Contacts;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\TextareaType;
use Symfony\Component\Form\Extension\Core\Type>EmailType;

class MyContactType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('nom', TextType::class)
            ->add('prenom', TextType::class)
            ->add('email', EmailType::class)
            ->add('sujet', TextType::class)
            ->add('message', TextareaType::class, [
                'attr' => [
                    'rows' => 10
                ]
            ])
    }

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

Controller

```
/**
 * @Route("/my-contact", name="myContact")
 */
public function myContact()
{
    $contact = new Contacts();
    $form = $this->createForm(MyContactType::class, $contact);

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

View

```
{{ form_start(form) }}
    {{ form_widget(form) }}

    <button type="submit" class="btn btn-primary">Valider</button>
{{ form_end(form) }}
```

Afficher les Forms dans Twig

```
{# Afficher tout le formulaire d'un coup #}
{{ form(form) }}
```

```
{# Afficher le formulaire, en ajoutant manuellement un bouton Valider #}
{{ form_start(form) }}
  {{ form_widget(form) }}

  <button type="submit" class="btn btn-primary">Valider</button>
{{ form_end(form) }}
```

```
{# Afficher chaque input l'un après l'autre #}
{{ form_start(form, {'attr': {'novalidate': 'novalidate'}}) }}

<div class="row">
  <div class="col-sm-6">{{ form_row(form.prenom) }}</div>
  <div class="col-sm-6">{{ form_row(form.nom) }}</div>
</div>

<div class="row">
  <div class="col-sm-6">{{ form_row(form.email) }}</div>
  <div class="col-sm-6">{{ form_row(form.sujet) }}</div>
</div>

{{ form_row(form.message) }}

<button type="submit" class="btn btn-primary float-right">Envoyer</button>
{{ form_end(form) }}
```

Ici, le champ “message” est en col-12,
donc pas besoin de le mettre dans un
“row”

Forms : Thème Bootstrap

Pour ajouter automatiquement le thème Bootstrap à l'ensemble de nos formulaires, il suffit d'ajouter une ligne dans le fichier :

config/packages/twig.yaml

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

Récupérer et enregistrer les données du formulaire

```
/*
 * @Route("/my-contact", name="myContact")
 */
public function myContact(
    Request $request
) {
    $contact = new Contacts();
    $form = $this->createForm(MyContactType::class, $contact);

    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()) {
        $data = $form->getData();
        $data->setDate( new \DateTime() );

        $em = $this->getDoctrine()->getManager();
        $em->persist($data);
        $em->flush();

        return $this->redirectToRoute('myContact');
    }

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

- On crée un nouvel objet Contacts
- On récupère le Form MyContactType, créé à l'étape précédente
- Si le formulaire est soumis, et est valide...
- On récupère ses data dans \$data, qui est donc un objet de l'Entité Contacts
- On lui ajoute la date actuelle
- Pour finir, on l'enregistre et on redirige

Validation des formulaires

```
<?php

namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;

/**
 * @ORM\Entity(repositoryClass="App\Repository\ContactsRepository")
 */
class Contacts
{
    /**
     * @ORM\Id()
     * @ORM\GeneratedValue()
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=255)
     * @Assert\NotBlank(message = "contact.error.blank.nom")
     */
    private $nom;

    /**
     * @ORM\Column(type="string", length=255)
     * @Assert\NotBlank(message = "contact.error.blank.email")
     * @Assert\Email(message = "contact.error.format.email")
     */
    private $email;
```

Dans notre Entité, nous pouvons ajouter des Contraintes, qui permettent de vérifier la validité des informations entrées dans les formulaires, avant de les enregistrer en base de données.

On importe \Constraints as Assert

Le \$nom ne peut pas être vide (NotBlank), sinon on retourne une erreur.

L' \$email ne peut pas être vide, et doit être un email valide (Email), sinon on retourne le message correspondant à l'erreur.

message="..." peut être une chaîne de caractères, ou une clé dans fichier .yaml de traductions bien spécifique :
translations/validators.yaml

Les erreurs sont gérées automatiquement par le FormBuilder.

Exercice - Entités et Formulaires

Nous allons enregistrer en BDD les messages envoyés via le formulaire de contact PRO.

Supprimer le controller contactPro et son template, on le refait entièrement ;)

Créer une Entity ContactPro avec les champs : nom, prenom, sujet, email, message et date.

Migrer cette Entité vers la base de donnée

Créer le formulaire de contact via un FormType

Ajouter les contraintes pour la validation des champs du formulaire

Enregistrer les infos du formulaire de contact en base de donnée

Faire fonctionner l'envoi de l'email

Forms : Upload de fichiers

Entity

```
/**  
 * @ORM\Column(type="string", length=255, nullable=true)  
 */  
private $document;
```

FormBuilder

```
$builder  
// ...  
->add('document', FileType::class, [  
    'required' => false,  
    'label' => 'Pièce jointe',  
    'mapped' => false,  
])
```

services.yaml

```
parameters:  
documents: '%kernel.project_dir%/public/documents'
```

Controller

```
$file = $form['document']->getData(); // On récupère le fichier entré par l'utilisateur  
if ($file) {  
    $repertoire = $this->getParameter('documents'); // Correspond au paramètre dans services.yaml  
    $nomDuDocument = 'document-' . rand(1, 99999) . '.' . $file->guessExtension(); // Renommer par un nom unique  
    $file->move($repertoire, $nomDuDocument); // Déplacer le fichier dans le bon répertoire  
    $data->setDocument($nomDuDocument); // Enregistrer le nom du fichier  
}
```

Ajouter un champ \$document de type string à l'entité. Il servira à stocker le nom du fichier.

Ajouter un input type File au FormBuiler

Ajouter un paramètre dans config/services.yaml pour stocker le répertoire pour enregistrer les documents

Exercice 2

Sous les articles de notre blog, nous souhaitons donner la possibilité de commenter.

Créer une Entité Commentaires, liée à l'Entité Articles, elle contiendra les champs : Article, Pseudo, Email, Commentaire, Date

Créer un Form CommentairesType, et afficher ce formulaire sous chaque article.

Enregistrer les commentaires lorsqu'on en publie

Afficher les commentaires sous les articles, entre l'article et le formulaire d'ajout de commentaires.

Security

La sécurité dans Symfony

La sécurité est un point très important, mais le principe est quasiment toujours le même : Une table utilisateur, des formulaires d'inscription et de connexion, un bouton “se souvenir de moi”, un encodage de mot de passe, un système de sessions, et un bouton déconnexion.

Symfony a tout prévu pour implémenter un système de sécurité solide, facile à dupliquer d'un projet à l'autre.

Système de sécurité : les étapes

1. Créer une Entité : **Users**, et ses différentes propriétés (nom, prenom, email, password, ...)
2. Créer un **Provider**, qui va :
 - a. Rafraîchir la session des utilisateurs à chaque chargement de page
 - b. Permettre l'utilisation de fonctionnalités (switch user, remember me)
3. **Encoder** les mots de passe
4. Créer un **Firewall**, qui définit le mode de connexion (formulaire de connexion, API token, ...), qui se charge d'authentifier les utilisateurs, de donner accès ou non à certaines parties du site
5. Authentifier les utilisateurs : formulaire de connexion
6. Définir des rôles hiérarchisés (SUPER_ADMIN, ADMIN, USER...), et des accès restreints à certaines partie du site en fonction du rôle
7. Créer un formulaire d'inscription, un lien de déconnexion

Créer l'Entity, le Repo, le Provider

En ligne de commandes : bin/console make:user

Cette commande crée :

- **Entity** : Users
- **Repository** : UserRepository
- **Provider et Encoder** dans : security.yaml

En ligne de commandes : bin/console make:entity Users

Ajouter les autres propriétés (nom, prenom, date_inscription, ...)

Puis migrer la base de données :

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

```
namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\UserInterface;

/**
 * @ORM\Entity(repositoryClass="App\Repository\UsersRepository")
 */
class Users implements UserInterface
{
    /**
     * @ORM\Id()
     * @ORM\GeneratedValue()
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=180, unique=true)
     */
    private $email;

    /**
     * @ORM\Column(type="json")
     */
    private $roles = [];

    /**
     * @var string The hashed password
     * @ORM\Column(type="string")
     */
    private $password;

    /**
     * @ORM\Column(type="string", length=255)
     */
    private $nom;

    /**
     * @ORM\Column(type="string", length=255)
     */
    private $prenom;

    /**
     * @ORM\Column(type="datetime")
     */
    private $date_inscription;
```

Créer le système de connexion

En ligne de commandes : bin/console make:auth

Cette commande crée :

- Un **Authenticator**, qui se charge d'authentifier les utilisateurs
- Un **SecurityController**, qui va gérer les pages en lien avec la sécurité (inscription, connexion, déconnexion, mot de passe oublié...)
- Un formulaire de connexion
- Modifie le **Firewall** dans security.yaml

Le système de connexion fonctionne, si nous ajoutons un utilisateur dans la base de donnée, il pourra se connecter.

Définir les ROLES hiérarchisés

security.yaml

```
# Easy way to control access for large sections of your site
# Note: Only the *first* access control that matches will be used
access_control:
    - { path: '^/a/staff', roles: ROLE_SUPER_ADMIN }
    - { path: '^/a/admin', roles: ROLE_ADMIN }
    - { path: '^/a/', roles: ROLE_USER }
    - { path: '^', role: IS_AUTHENTICATED_ANONYMOUSLY }

role_hierarchy:
    ROLE_USER:           IS_AUTHENTICATED_ANONYMOUSLY
    ROLE_ADMIN:          ROLE_USER
    ROLE_SUPER_ADMIN:    [ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH]
```

L' access_control gère les droits d'accès au site en fonction des URL, selon le rôle des utilisateurs.

Le role_hierarchy gère la hiérarchisation :

- ROLE_USER a également le rôle IS_AUTHENTICATED_ANONYMOUSLY
- ROLE_ADMIN a également le ROLE_USER
- ROLE_SUPER_ADMIN a également le ROLE_ADMIN ainsi que le ROLE_ALLOWED_TO_SWITCH

Le ROLE_SUPER_ADMIN a donc tous les rôles, donc théoriquement tous les accès.

Créer un formulaire d'inscription

En lignes de commandes : `bin/console make:registration-form`

Cette commande crée :

- Un **RegistrationController** (copier/coller le code dans le SecurityController)
- Un formulaire d'inscription : **RegistrationFormType** (à modifier selon les besoin)
- Un template de page d'inscription
- Modifie éventuellement l'Entity Users

Exemple de RegistrationTypeForm

```
$builder
    ->add('prenom', TextType::class, [
        'label' => 'Votre prénom',
    ])
    ->add('nom', TextType::class, [
        'label' => 'Votre nom',
    ])
    ->add('email', EmailType::class, [
        'label' => 'Votre email',
    ])
    ->add('agreeTerms', CheckboxType::class, [
        'mapped' => false,
        'label' => "J'accèpte les CGU",
        'constraints' => [
            new IsTrue([
                'message' => "Vous devez accepter les CGU.",
            ]),
        ],
    ])
    ->add('password', RepeatedType::class, [
        'type' => PasswordType::class,
        'invalid_message' => "Les mots de passes ne sont pas identiques.",
        'first_options' => [
            'label' => 'Mot de passe',
            'empty_data' => ' ',
        ],
        'second_options' => [
            'label' => 'Confirmez le mot de passe',
            'empty_data' => ' ',
        ],
    ]),
;
```

Le RepeatedType permet de demander deux fois le mot de passe, pour éviter les fautes de frappes.

La gestion des erreurs se fait automatiquement.

agreeTerms est en mapped => false car cette propriété n'existe pas dans notre Entité Users. Ce champ sert uniquement à valider le formulaire d'inscription.

Exemple de page d'inscription dans SecurityController

```
/*
 * @Route("/register", name="app_register")
 */
public function register(
    Request $request,
    UserPasswordEncoderInterface $passwordEncoder,
    GuardAuthenticatorHandler $guardHandler,
    Authenticator $authenticator
): Response
{
    $user = new Users();
    $form = $this->createForm(RegistrationFormType::class, $user);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        // encode the plain password
        $user->setPassword(
            $passwordEncoder->encodePassword(
                $user,
                $form->get('password')->getData()
            )
        );

        $user->setDateInscription( new \DateTime() ); // On ajoute manuellement la date d'inscription
        // Si besoin, on lui ajoute un rôle
        // Si c'est un simple ROLE_USER, pas besoin de le faire, il aura ce rôle par défaut
        $user->setRoles(['ROLE_ADMIN']);

        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->persist($user);
        $entityManager->flush();

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

        return $guardHandler->authenticateUserAndHandleSuccess(
            $user,
            $request,
            $authenticator,
            'main' // firewall name in security.yaml
        );
    }

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

Presque tout le code a été généré par la commande **make:registration-form**

Il ne nous reste qu'à ajouter la date d'inscription, et éventuellement un rôle.

Améliorations possibles

Nous avons maintenant un système de sécurité fonctionnel. Mais nous pouvons l'améliorer :

- Renommer les routes, gérer les erreurs dans des fichiers traductions...
- Créer les pages /admin pour chacun des rôles utilisés
- Envoyer un email à l'inscription, pour vérifier l'adresse email
- Ajouter des pages : mot de passe oublié, changer mot de passe / email
- Ajouter une propriété “active” à notre Entité Users, par défaut à 1, que nous pouvons passer à 0 pour bloquer un utilisateur. Il faudra donc vérifier s'il est “active” à chaque rechargement de page via le système d'événements.

TP Sécurité

- Ajouter la fonctionnalité : “Se souvenir de moi” sur la page de login
- À l’inscription, on envoie un email à l’utilisateur avec un lien pour valider son compte. Avant cela, il ne peut pas se connecter.
- Créer un système “mot de passe oublié”
- Améliorer le système mot de passe oublié pour en faire une page “Modifier mon mot de passe” accessible aux utilisateurs déjà connectés.
- Faire en sorte que seuls les PUBLISHERS puissent créer de nouveaux articles. Tous les visiteurs peuvent lire les articles. L’admin ne peut pas créer d’articles, mais il peut les modifier / supprimer.

Récupérer l'utilisateur dans Twig

```
{# app.user contient l'utilisateur en cours #}
{{ dump( app.user ) }}
{{ dump( app.user.email ) }}

{# afficher du contenu si l'utilisateur est connecté ou non #}
{% if app.user %}
    <a href="{{ path('app_logout') }}">Déconnexion</a>
{% else %}
    <a href="{{ path('app_login') }}">Connexion</a>
{% endif %}

{# Afficher du contenu selon le rôle de l'utilisateur #}
{% if is_granted('ROLE_SUPER_ADMIN') %}
    ROLE_SUPER_ADMIN
{% elseif is_granted('ROLE_ADMIN') %}
    ROLE_ADMIN
{% elseif is_granted('ROLE_USER') %}
    ROLE_USER
{% else %}
    ANONYME
{% endif %}
```

Events

Les Events dans Symfony

Symfony génère des événements à différents moments de l'exécution du code.

Il est possible d'écouter ces événements, ou de s'y abonner, pour déclencher une fonction lorsqu'un événement précis a lieu.

Des événements existent de base, mais il est possible d'en créer nous-mêmes.

Par exemple, à chaque fois qu'une page sera chargée, nous allons vérifier si notre utilisateur a toujours active = 1 , sinon nous le déconnecterons.

Créer un EventSubscriber

Lorsqu'une page est demandée (Requête), Symfony déclenche un événement : **kernel.request**. Nous allons nous abonner à cet événements, et à chaque fois qu'il se produit, nous allons vérifier :

- Y a-t-il une session en cours ? (Un utilisateur est-il connecté ?)
- Si oui, a-t-il active = 1 ?
 - Oui ? Ok, on ne fait rien
 - Non ? On le redirige sur la page de déconnexion

En ligne de commandes, créer un EventSubscriber :

- bin/console make:subscriber
- Lui donner un nom (ex : UsersSubscriber)
- Choisir l'événement kernel.request

```
<?php

namespace App\EventSubscriber;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\Security\Core\Authentication\Token\Storage\TokenStorageInterface;
use Symfony\Component\HttpFoundation\RedirectResponse;
use Symfony\Component\Routing\Generator\UrlGeneratorInterface;

use App\Entity\Users;

class UsersSubscriber implements EventSubscriberInterface
{
    private $tokenStorage;
    private $router;

    public function __construct(
        TokenStorageInterface $tokenStorage,
        UrlGeneratorInterface $router
    ) {
        $this->tokenStorage = $tokenStorage;
        $this->router = $router;
    }

    public function onKernelRequest(object $event)
    {
        $token = $this->tokenStorage->getToken(); // On récupère le token de la session en cours
        if ($token) {
            $user = $token->getUser(); // On récupère le User via le token de session
            if ($user instanceof Users) { // On vérifie que le User est bien connecté
                if ($user->getActive() != 1) { // S'il n'est pas active == 1 ...
                    $response = new RedirectResponse($this->router->generate('app_logout'));
                    $event->setResponse($response);
                }
            }
        }
    }

    public static function getSubscribedEvents()
    {
        return [
            'kernel.request' => 'onKernelRequest',
        ];
    }
}
```

EventSubscriber on kernel.request

Créer un nouvel Event

Afin de bien séparer son code, et de le réutiliser facilement, il est possible de créer ses propres Event et EventSubscriber.

Par exemple, lorsqu'un commentaire est publié sous un article, nous souhaitons déclencher un envoi d'email, mettre à jour l'article, etc...

Il est tout à fait possible de le faire via un système de fonctions PHP classique, c'est simplement une autre façon de faire, pour bien séparer son code.

Créer un Event

```
<?php

namespace App\Event;

use Symfony\Contracts\EventDispatcher\Event;
use App\Entity\Comments;

class NewCommentEvent extends Event
{
    public const NAME = 'comment.new';

    protected $comment;

    public function __construct(
        Comments $comment
    ) {
        $this->comment = $comment;
    }

    public function getArticle()
    {
        return $this->comment->getArticle();
    }
}
```

Nous créons notre propre événement.

Créer un dossier src/Event et y ajouter
NewCommentEvent.php

Nous lui donnons une constante NAME pour
pouvoir l'appeler plus facilement.

Cet évènement prendra le nouveau
commentaire en argument.

La fonction getArticle() n'est pas nécessaire,
elle permet simplement de récupérer l'article du
commentaire plus facilement par la suite.

Créer un EventSubscriber

```
<?php

namespace App\EventSubscriber;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;

class NewCommentSubscriber implements EventSubscriberInterface
{
    public function onCommentNew($comment)
    {
        $article = $comment->getArticle();
        dump($article); die();

        // Envoyer l'email ...
        // Modifier l'article ...
    }

    public static function getSubscribedEvents()
    {
        return [
            'comment.new' => 'onCommentNew',
        ];
    }
}
```

bin/console make:subscriber

On l'appelle NewCommentSubscriber

On choisir l'Event : comment.new

Nous aurons une nouvelle fonction
OnCommentNew, avec \$event en
paramètre, que nous pouvons
renommer en \$commentaire, car c'est
une instance de l'Entité Comments

Dispatcher l'Event

```
use Psr\EventDispatcher\EventDispatcherInterface;
use App\Event\NewCommentEvent;
use App\Entity\Comments;

class FrontController extends AbstractController
{

    /**
     * @Route("/nouveau-commentaire", name="nouveau_commentaire")
     */
    public function nouveau_commentaire(EventDispatcherInterface $dispatcher)
    {
        // ...
        // $comment = new Comments()...

        // On dispatch l'évènement pour dire qu'il vient de se produire
        $dispatcher->dispatch(
            new NewCommentEvent($comment), // On passe le commentaire en question
            NewCommentEvent::NAME // Le nom de l'évènement
        );
        // ...
    }
}
```

Depuis notre Controller, lorsqu'un nouveau commentaire est publié, nous pouvons dispatcher (=émettre) l'évènement NewComment, en passant le commentaire en question en argument.

Notre NewCommentSubscriber précédemment créé va se déclencher automatiquement lorsque cet événement sera émis.