

Créer un site web avec Symfony

# Framework PHP Symfony

ded.formateur.web@gmail.com



---

## Table des matières

---

I.	PRÉSENTATION .....	3
A.	Qu'est qu'un framework ?.....	3
B.	Qu'est-ce que l'architecture MVC ? .....	3
II.	INSTALLATION.....	4
A.	Pré-requis .....	4
B.	Créer un nouveau projet Symfony 5 .....	4
C.	Arborescence des dossiers .....	5
D.	Afficher le site : démarrer un serveur web.....	6
1.	Démarrer le serveur avec .....	6
2.	Arrêter le serveur .....	6
3.	Serveur sécurisé : HTTPS .....	6
4.	Domaine et proxy .....	7
E.	Afficher la page web.....	8
III.	Créer un site web avec Symfony .....	9
A.	Aide à la création de code : composant MAKER .....	9
-	Commandes Maker .....	9
-	Liste des commandes possibles avec Maker .....	9
B.	Ajouter une classe Controller .....	9
C.	Le routage.....	10
1.	Les annotations .....	11
2.	Les attributs .....	11
3.	Syntaxe .....	11
4.	Définition d'une route .....	12
5.	Liste des routes d'un projet .....	12
IV.	VIEW : Générer les vues pour l'affichage avec Twig .....	14
A.	Filtres : modifier la valeur préposée .....	14
B.	Fonctions .....	14
C.	Tests : tester une variable .....	15
D.	Variable globale : app .....	15
E.	Configuration.....	15
V.	MODEL : gérer la base de données avec DOCTRINE .....	16
A.	Informations de connexion : fichier .env .....	16

B.	Configuration : config/packages/doctrine.yaml.....	16
C.	Création de la bdd .....	16
1.	Création d'une table .....	17
2.	Création d'une migration .....	17
3.	Exécution de la migration .....	17
4.	Annuler la migration .....	17
VI.	Authentification, inscription avec SECURITY .....	19
A.	Composant Security.....	19
B.	Entité utilisateur .....	19
1.	Le fichier Security/LoginFormAuthenticator.php :.....	19
VII.	Développement front-end avec WEBPACK ENCORE.....	22
A.	Gestionnaire de dépendances : YARN.....	22
B.	Configurer la compilation des fichiers ressources .....	23
1.	webpack.config.js .....	23
C.	Décommenter la ligne 69 pour éviter un bug lors de l'utilisation de jQuery .....	23
D.	Placer les images dans le dossier public/images :.....	23
VIII.	Formulaires, affichage et validation avec FORM VALIDATOR.....	26
IX.	ANNEXES.....	28
A.	Remplir la bdd : Fixtures   Faker .....	29
B.	Gérer un formulaire avec un input de type <i>file</i> .....	30
C.	JS – Afficher immédiatement les images téléversées .....	33
E.	Envoyer des e-mails avec XAMPP (PC) .....	34
F.	Générer des PDF avec Symfony.....	36
G.	TWIG : ajouter une nouvelle fonction .....	38
H.	Pages d'erreurs HTML.....	39
I.	.htaccess .....	40

# I. PRÉSENTATION

Selon le [site officiel](#), Symfony est un ensemble de composants PHP réutilisables ainsi qu'un framework PHP pour les projets web. La partie framework va plus nous intéresser, même s'il est vrai qu'avant tout un framework est un ensemble de composants réutilisables.

## A. Qu'est qu'un framework ?

Un framework, ou cadre de travail, fournit une structure et un ensemble d'outils pour faciliter la programmation. C'est donc une base de travail qui permettra de développer une application. L'architecture sera déjà définie mais aussi beaucoup de parties de l'application sont déjà fournies, ce qui permet de se concentrer sur ce qui fait que votre projet sera différent d'un autre projet. Le développement est donc facilité et accéléré puisqu'il n'y a pas besoin de réinventer la roue à chaque nouveau développement.

Le framework Symfony est développé en PHP Orienté Objet basé sur une architecture MVC.

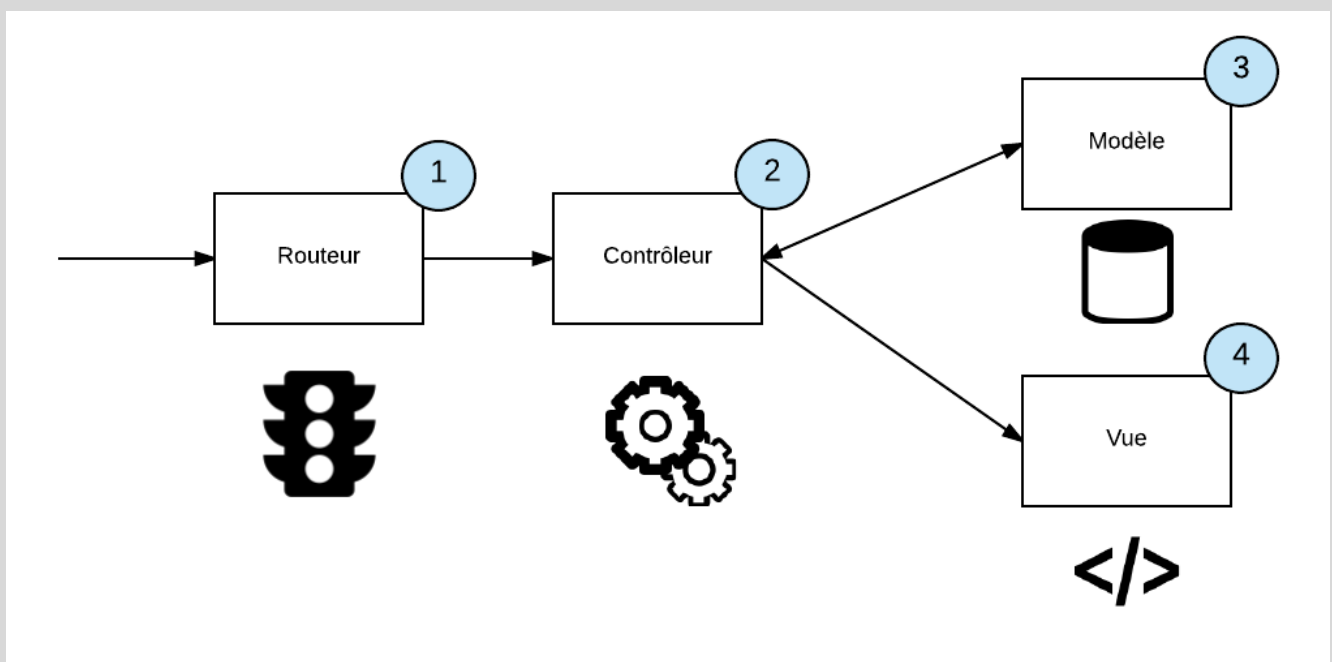
## B. Qu'est-ce que l'architecture MVC ?

Pour rappel, l'acronyme **MVC** (**M**odel **V**iew **C**ontroller, **M**odèle **V**ue **C**ontrôleur en VF) décrit une architecture de projet informatique. C'est donc une façon d'organiser son projet en séparant la gestion des données (la base de données, la partie Model), l'affichage (la partie View) et tout ce qui concerne le traitement des données, parfois appelé la logique métier (la partie Controller).

Dans un projet de site web MVC développé en Orienté Objet, il y a un unique point d'entrée, en général le fichier `index.php`. Ce sera le seul fichier PHP auquel les navigateurs externes accèderont. Il sera dans un dossier qui sera l'unique dossier accessible aux navigateurs web. Ce fichier peut aussi faire office de routeur.

Le routeur détermine, à partir de l'URL, quelle partie du code doit s'exécuter afin d'avoir la page HTML demandé par l'utilisateur par l'intermédiaire de son navigateur web.

Le routeur (1) reçoit la demande de l'utilisateur (par exemple : afficher la liste des articles). À partir de cette demande, le routeur détermine donc quel contrôleur (2) doit être appelé. Ce contrôleur va interroger la base de données en passant par le modèle (3). Le modèle renvoie les données demandées au contrôleur (2). Le contrôleur va générer l'affichage via la vue (4). C'est ce qui sera renvoyé au navigateur de l'utilisateur.



## II. INSTALLATION

### A. Pré-requis

Il y a quelques pré-requis nécessaire à l'installation d'un projet Symfony.

- [PHP](#) : version 7.2.5 minimum (la version 7.4 est recommandée).
- [Composer](#) : est un gestionnaire de dépendances qui va permettre d'ajouter des dépendances (ou composants) à ton projet. Ces composants sont des bibliothèques PHP ou des modules qui vont faciliter le développement de ton projet.
- [Symfony CLI](#) : est un utilitaire, utilisable en ligne de commande, qui fournit tous les outils nécessaires à la gestion d'un projet Symfony. Tu vas, par exemple, pouvoir exécuter un serveur pour afficher ton projet dans un navigateur web.
- [Yarn](#) : est un gestionnaire de dépendance mais pour le développement front-end. Ce composant n'est nécessaire que si tu dois installer Webpack pour gérer les fichiers de front (js, css, ...).
- [Git](#) : est un outil de versionnage, qui s'utilise principalement en ligne de commande mais il existe des logiciels proposant une interface graphique. Il faut installer et configurer Git avant d'installer un projet Symfony.

Pour ce cours :

- [MySQL](#) : (MariaDB). C'est le SGBD qui sera utilisé pour le présent cours. Vous pouvez choisir [xampp](#) pour installer MySQL, et PHPMyAdmin (utilisable avec Apache).
- [Visual Studio Code](#) : EDI totalement gratuit, développé par Microsoft. Il est modulable, tu peux ajouter des extensions qui sont développées par la communauté.

### B. Créer un nouveau projet Symfony 5

Dans la suite du document, toutes les commandes ressemblant à

```
> cd nom_du_dossier
```

seront à taper dans une console (PowerShell ou Git Bash pour Windows, Terminal pour Mac, par exemple).

Se placer dans le dossier dans lequel se trouvent les projets web. Par exemple, le dossier htdocs de xampp.

```
> cd \xampp\htdocs
```

Ou, si vous utilisez wamp, `cd \wamp\www`. Ou dans n'importe quel dossier.

Ensuite tu peux installer un projet Symfony en utilisant l'utilitaire [Symfony CLI](#) installé précédemment, avec la commande suivante :

```
> symfony new nom_du_dossier --version="5.4.*" --webapp
```

Remplace `nom_du_dossier` par le nom de ton projet. Le dossier sera créé et le projet Symfony sera installé dans ce dossier.

L'option `--version` n'est à préciser que si l'on ne veut pas installer la dernière version de Symfony. La version dite LTS (Long-Term Support) est la version 5.4, utilisée pour ce cours.

`5.4.*` peut être remplacé par la version voulue.

L'option `webapp` est utilisée pour spécifier que l'on veut développer une application web. Plusieurs modules nécessaires au développement d'un site seront donc automatiquement installés. Si cette option n'est pas utilisée, l'installation du projet Symfony sera minimale. Ce genre d'installation est utilisé quand on veut développer une API ou une application console (parce que ce genre d'application ne requiert pas d'affichage).

Pour la suite du cours, ne pas oublier de changer de dossier (commande `cd`: *Change Directory*) et de se placer dans le dossier créé avant de lancer les prochaines commandes :

```
> cd nom_du_dossier
```

## C. Arborescence des dossiers

### ▼ SF5

- > assets
- > bin
- > config
- > migrations
- > node\_modules
- > public
- ▼ src
  - > Controller
  - > Entity
  - > EventListener
  - > Form
  - > Repository
  - > Security
  - 🐘 Kernel.php
- > templates
- > var
- > vendor
- ⚙️ .env
- ≡ .env.local
- 💎 .gitignore
- { } composer.json
- { } composer.lock
- { } package.json
- ≡ symfony.lock
- 📦 webpack.config.js
- 👤 yarn.lock

Voyons ce que contient chaque dossier installé dans le dossier du projet.

#### • bin/

`bin` pour binary, c'est-à-dire exécutable. Il n'y a qu'un fichier dans ce dossier : `console`, qui est un fichier PHP. Ce fichier sera appelé en ligne de commande.

#### • config/

Comme son nom l'indique, contient les fichiers de configuration. Plusieurs types de fichier possibles : php, yaml, xml. Le YAML sera plus souvent utilisé.

#### • src/

Dossier source. Une grande partie du code que tu vas écrire se trouvera dans ce dossier (les contrôleurs, les entités, ...)

#### • var/

Ce dossier va contenir les fichiers de logs et les fichiers de cache. Le contenu de ce dossier est différent pour chaque ordinateur sur lequel se trouve le projet. Son contenu n'est pas nécessaire au code source de ton projet (mais a une utilité pour le fonctionnement du site).

#### • vendor/

C'est dans ce dossier que se trouve le cœur de Symfony. Ce dossier ne sera jamais manipulé par le développeur. C'est ici qu'iront les codes sources des composants ou dépendances que l'on ajoutera au projet via Composer

Il y a aussi quelques fichiers notables :

- `.gitignore` : un projet Symfony est donc préparé pour être partagé sur git.
- `composer.json`, `.lock` : ces fichiers servent à garder la liste des composants installés avec Composer. Ces fichiers sont nécessaires pour la réinstallation du projet sur une autre machine.

#### D. Afficher le site : démarrer un serveur web

Maintenant que le projet existe, ce serait bien de voir à quoi il ressemble. On va pouvoir le visualiser dans un navigateur web, comme s'il était sur un serveur distant. Il faut donc :

##### 1. Démarrer le serveur avec Symfony CLI

```
> symfony server:start
```

Il y a un raccourci pour cette commande :

```
> symfony serve
```

Cette commande démarre donc un serveur PHP pour votre projet. Le terminal n'est plus accessible parce que le serveur « écoute » les connexions et affiche les *logs*. Pour lancer le serveur en tâche de fond (et donc ne pas bloquer le terminal), ajouter l'option `-d` à la fin de la commande.

##### 2. Arrêter le serveur

```
> symfony server:stop
```

La fermeture du terminal arrêtera aussi l'exécution du serveur s'il n'était pas démarré en arrière-plan.

##### 3. Serveur sécurisé : HTTPS

Le serveur Symfony peut simuler un serveur sécurisé (commençant par `https` au lieu de `http`). D'où le message suivant qui apparaît quand vous lancez la commande `server:start`

```
[WARNING] run "symfony.exe server:ca:install" first if you want to run the web server with TLS support, or use "--no-tls" to avoid this warning
```

Lancez la commande indiquée pour installer un *certificat d'autorité* et ainsi simuler un serveur sécurisé.

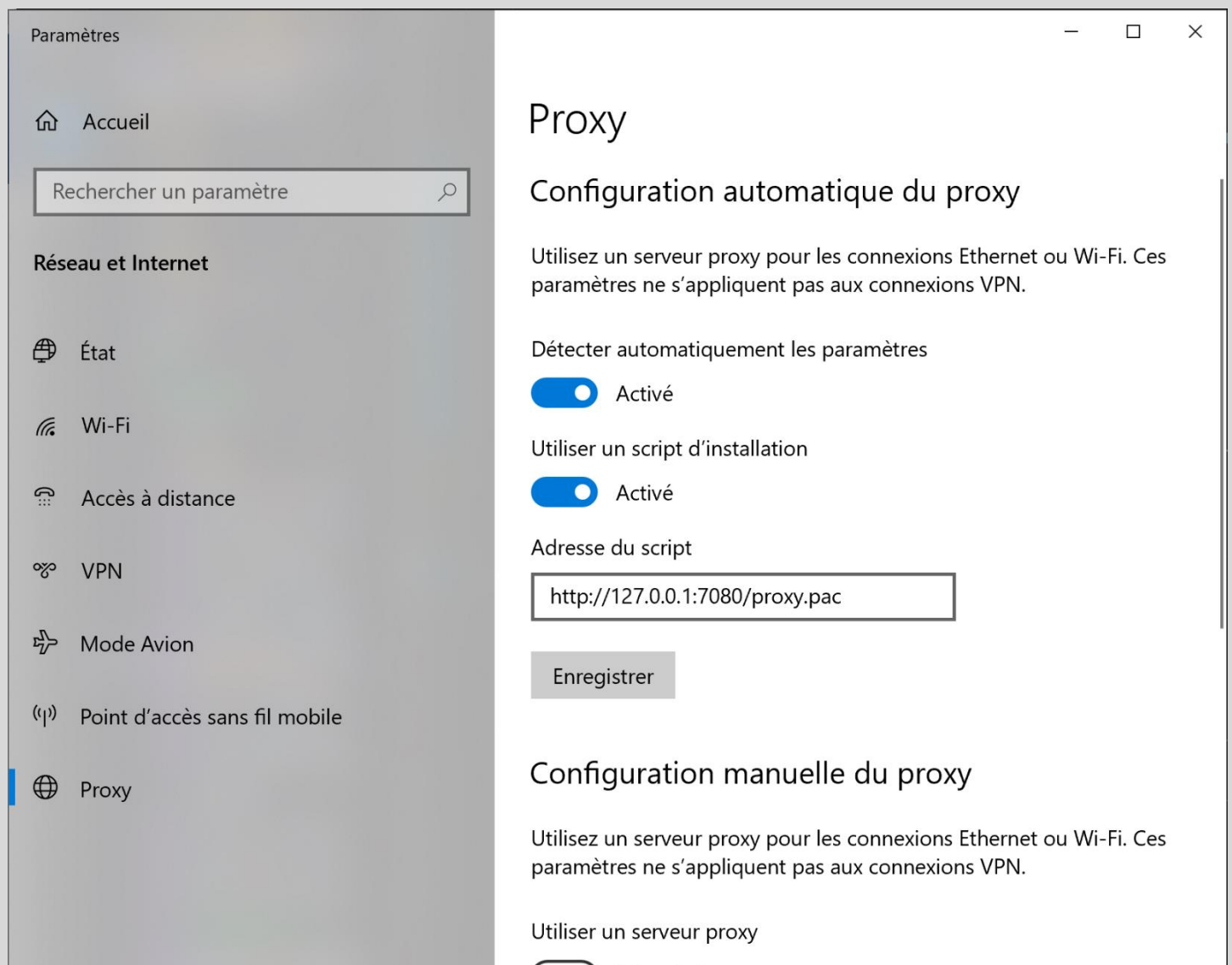
```
> symfony server:ca:install
```

Il se peut que ton navigateur ne reconnaisse pas le *certificat* comme étant légitime. Pas d'inquiétude, même si le message peut être anxiogène, tu peux accorder une autorisation en faisant confiance à ce certificat. Il n'y a aucun risque puisque les communications entre le navigateur et le serveur sont internes à ton ordinateur.

 Toutes les commandes commençant par : `symfony console` peuvent être lancées avec la commande : `php bin/console`.

#### 4. Domaine et proxy

On peut simuler un nom de domaine pour éviter d'avoir une URL du type `http://127.0.0.1:8000/`. Configurer un proxy pour l'adresse IP suivante : `http://127.0.0.1:7080/proxy.pac`



Ensuite exécutez la commande suivante :

```
> symfony proxy:start
```

Enfin, pour choisir votre nom de domaine fictif :

```
> symfony proxy:domain:attach nom-de-domaine-voulu
```

Attention, tu ne dois choisir que le nom de domaine sans l'extension. Symfony va rajouter **.wip** (pour *Work In Progress*).

Après avoir démarrer le serveur Symfony, navigue vers l'adresse `localhost:7080` (l'adresse du proxy).

Tu devrais voir la liste des serveurs démarrés, ainsi que le domaine correspondant. Tu n'as plus qu'à cliquer sur ce lien pour voir ton site internet.

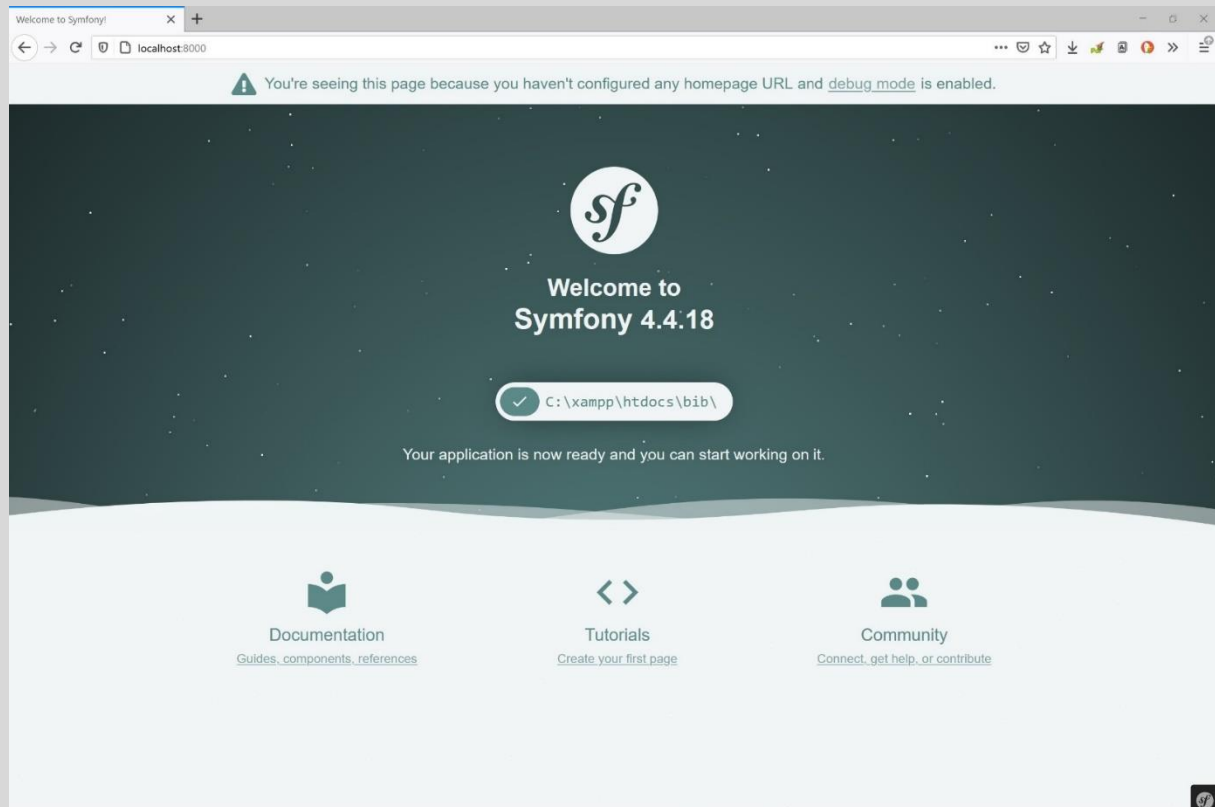


## E. Afficher la page web

À l'exécution de la commande pour lancer le serveur, tu trouveras l'adresse url à coller dans ton navigateur pour voir ce magnifique site en construction.

```
[OK] web server listening
The web server is using PHP CGI 7.4.12
https://127.0.0.1:8000
```

Cela revient à dire que le "*nom de domaine*" du projet est `127.0.0.1:8000`<sup>1</sup>.



---

<sup>1</sup> Pour information, l'adresse IP `127.0.0.1` est l'adresse IP de `localhost`. Aucun ordinateur dans le monde ne peut avoir cette adresse IP, ni une adresse commençant par `192.168` ou `10.0` (réservées aux adresses internes d'un réseau local).

### III. Créer un site web avec Symfony

Maintenant entrons dans le vif du sujet. On va taper du code.

Symfony est un framework développé en PHP orienté objet avec une architecture MVC. Dans un tel projet de site web, lorsque l'on veut ajouter une page au site, on n'ajoute pas un fichier PHP. mais une méthode dans une classe qu'on appelle un *contrôleur*. Pour afficher le résultat de cette action, il faut lier cette méthode à une URL. C'est ce qu'on appelle une *route*.

Le seul fichier PHP accessible aux navigateurs est le fichier `public/index.php`. Ainsi le dossier racine du projet est le dossier `public` et donc les seuls fichiers qui seront accessibles en dehors du serveur seront dans ce dossier.

Ajoutons une classe Controller à ton projet. Pour cela tu va utiliser le composant nommé Maker.

#### A. Aide à la création de code : composant MAKER

Maker est un composant déjà présent dans un projet Symfony installé avec l'option *webapp*. Ce composant permet de générer des fichiers, par exemple des entités, contrôleurs, le système d'authentification, le formulaire d'inscription... Chaque commande déclenchera un questionnaire qui permettra de définir les caractéristiques des fichiers qui seront générés. L'avantage à utiliser Maker est que les classes et autres fichiers seront créés dans les dossiers où ils doivent se trouver, avec les bons *namespaces*,... Cela permet d'éviter certaines erreurs.

##### - Commandes Maker

Les commandes seront toujours du type :

```
> symfony console make:commande
```

Avec *commande* qui correspond à ce que l'on veut créer

##### - Liste des commandes possibles avec Maker

```
> symfony console list make
```

#### B. Ajouter une classe Controller

```
> symfony console make:controller
```

Tu devras choisir le nom de la classe contrôleur. Rappel : le nom de la classe doit utiliser la casse CamelCase. Ne pas oublier de mettre une majuscule au début. Le mot *Controller* sera ajouté au nom de la classe (sauf si tu l'ajoutes toi-même).

Par exemple, ajoutons la classe `TestController` au projet.

```
web-2@DESKTOP-SKONFM7 MINGW64 /c/xampp/htdocs/sf5 (main)
$ symfony console make:controller

Choose a name for your controller class (e.g. OrangePopsicleController):
> Test

created: src/Controller/TestController.php
created: templates/test/index.html.twig
```

Success!

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

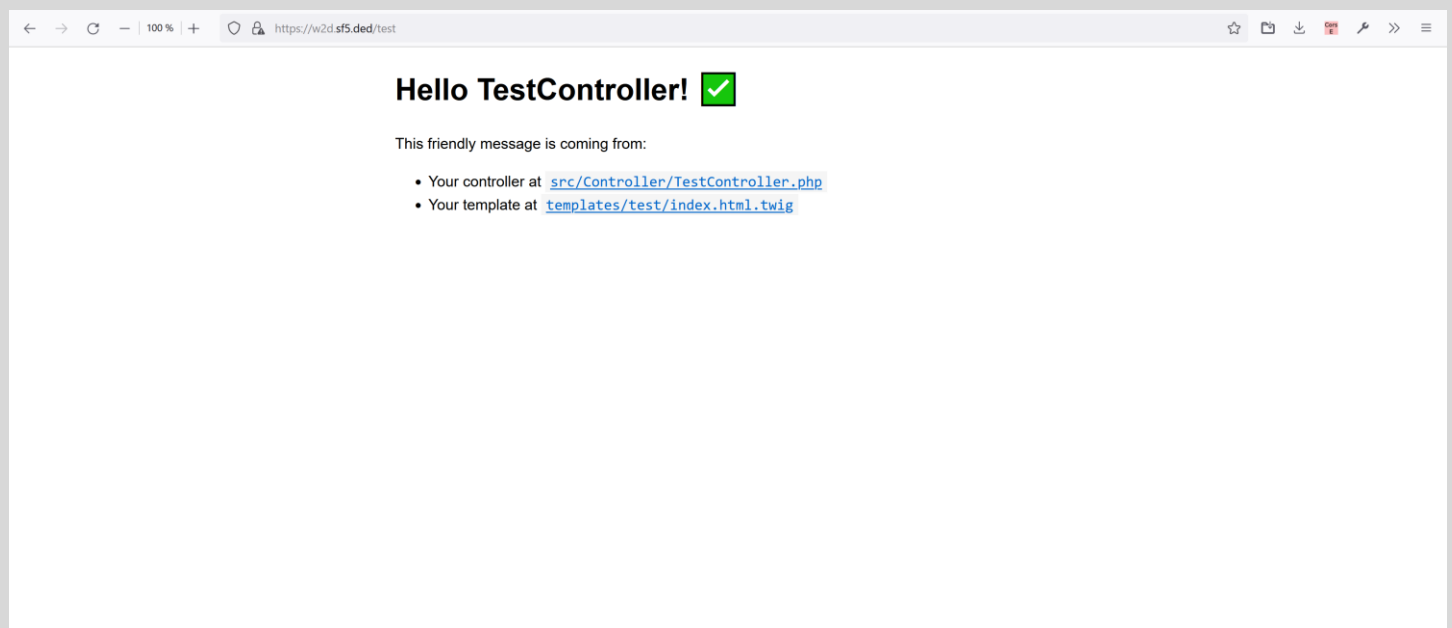
Après la commande, on voit que 2 fichiers ont été créés. Une classe `TestController` dans le dossier `src/Controller` et un fichier vue, dans un dossier du nom du contrôleur (lui-même dans le dossier `templates`) et nommé `index.html.twig`. On y reviendra plus tard. D'abord à quoi va nous servir la classe contrôleur.

### C. Le routage

Le routage (*routing* en anglais) est la façon dont un framework définit ses routes. Chaque nouvelle classe `Controller` créée avec `Maker` contient déjà une route.

```
/**
 * @Route("/test", name="app_test")
 */
public function index(): Response
```

Pour tester, il suffit de mettre le chemin `/test` dans la barre d'adresse de ton navigateur, après le nom de domaine.

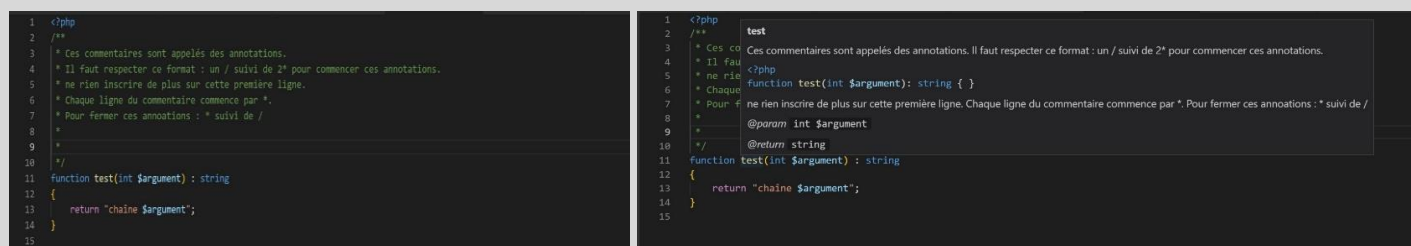


Dans les commentaires (annotations) situés juste avant la méthode *index*, tu peux voir la déclaration de la première route du projet. Ce « constructeur » n’a qu’un seul argument obligatoire, le chemin (l’adresse URL relative) pour lequel la méthode *index* de ce contrôleur *TestController* sera déclenchée.

Les arguments suivants seront “introduit” par leur nom. Par exemple *name*, qui permet de donner un nom à une route. Ce nom sera utilisé, par la suite, pour générer des liens ou faire une redirection, par exemple.

## 1. Les annotations

Les annotations sont un format spécial de commentaires de PHP utilisés, principalement, pour générer une documentation automatique des fonctions, méthodes ou classes. Ci-dessous, un exemple dans l’EDI<sup>2</sup> Visual Studio Code, avec l’affichage de la documentation d’une fonction au survol de la souris :



Symfony va utiliser ces annotations pour générer les routes du projets. Dans un contrôleur, toutes méthodes précédées par des annotations contenant l’appel `@Route` sera une nouvelle route du projet.

## 2. Les attributs

Avec PHP 8, Symfony utilise les *attributs*, notion introduite dans PHP 8 pour gérer les métadonnées des fonctions, classes... En résumé, pour faire ce que Symfony (et d’autres frameworks) faisait déjà avec les *annotations*, alors que ce n’était pas une utilisation standardisé de PHP.

La syntaxe est différente mais le principe reste le même.

## 3. Syntaxe

### a) Les annotations (comme un bloc de commentaires) :

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

- Commencent par `/**`, se terminent par `*/`, chaque ligne du bloc annotation doit commencer par `*`
- Les mots précédés de `@` seront traités comme des mots-clés des annotations.
- Symfony utilise cette syntaxe pour les classes qui seront utilisées dans les annotations, par exemple `Route`. `@Route()` est le constructeur de la classe `Symfony\Component\Routing\Annotation\Route`. Donc si la classe n’est pas importé au début du fichier (avec la clause `use`), il y aura une erreur.
- Les annotations permettent l’utilisation de plusieurs classes, sur une ligne différente à chaque fois.
- Les annotations seront aussi utilisés pour la gestion de la base de données par le composant Doctrine.
- On ne peut utiliser que les `"` comme délimiteur de valeur (pas de `'`).

<sup>2</sup> Environnement de Développement Intégré (IDE : Integrated Development Environment en anglais)

#### b) Les attributs (comme un commentaire sur une ligne)

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

- Les attributs sont introduits par le caractère `#`, donc comme un commentaire sur une seule ligne. Directement suivi d'un `[` qui sera fermé à la fin de la ligne `]`.
- Pour utiliser plusieurs classes dans les attributs, il faut une nouvelle ligne commençant par `#`.
- On ne doit utiliser que l'apostrophe `'` comme délimiteur de valeur (pas de `"`)

#### 4. Définition d'une route

La 1<sup>ère</sup> route défini dans le contrôleur Test est donc caractérisée par :

- le chemin (`path`) qui est le chemin relatif affiché dans la barre d'adresse du navigateur après le nom de domaine. C'est la page demandée par l'internaute.
- le nom (`name`) : c'est l'identifiant de la route. Il sera utilisé comme argument de la fonction `path` pour générer un lien dans l'affichage, par exemple (cf. Twig). Ou lorsque l'on voudra faire une redirection, dans un contrôleur.

ex : `$this->redirectToRoute("app_test");`

On peut ajouter d'autres arguments

- `methods` : précise les méthodes http acceptées (GET, POST, ...) par la route. Une tentative de requête HTTP avec une méthode non acceptée redirigera vers une page d'erreur
- `requirements` : permet d'assigner un modèle regex sur un paramètre d'une route. (cf. Route paramétrée)
- 

#### 5. Liste des routes d'un projet

Pour afficher la liste des routes définies actuellement dans un projet :

```
> symfony console debug:route --show-controllers
```

```
MINGW64/c:/xampp/htdocs/_cours_formation/symfony/2208_sf5_php8
web-2@DESKTOP-SKONFM7 MINGW64 /c:/xampp/htdocs/_cours_formation/symfony/2208_sf5_php8 (main)
$ symfony console debug:route --show-controllers
```

Name	Method	Scheme	Host	Path	Controller
app_admin_abonne_index	GET	ANY	ANY	/admin/abonne/	App\Controller\Admin\AbonneController::index()
app_admin_abonne_new	GET POST	ANY	ANY	/admin/abonne/new	App\Controller\Admin\AbonneController::new()
app_admin_abonne_show	GET	ANY	ANY	/admin/abonne/{id}	App\Controller\Admin\AbonneController::show()
app_admin_abonne_edit	GET POST	ANY	ANY	/admin/abonne/{id}/edit	App\Controller\Admin\AbonneController::edit()
app_admin_abonne_delete	POST	ANY	ANY	/admin/abonne/{id}	App\Controller\Admin\AbonneController::delete()
app_admin_auteur	ANY	ANY	ANY	/admin/auteur	App\Controller\Admin\AuteurController::index()
app_admin_auteur_ajouter	ANY	ANY	ANY	/admin/auteur/ajouter	App\Controller\Admin\AuteurController::ajouter()
app_admin_categorie_index	GET	ANY	ANY	/admin/categorie/	App\Controller\Admin\CategorieController::index()
app_admin_categorie_new	GET POST	ANY	ANY	/admin/categorie/new	App\Controller\Admin\CategorieController::new()
app_admin_categorie_show	GET	ANY	ANY	/admin/categorie/{id}	App\Controller\Admin\CategorieController::show()
app_admin_categorie_edit	GET POST	ANY	ANY	/admin/categorie/{id}/edit	App\Controller\Admin\CategorieController::edit()
app_admin_categorie_delete	POST	ANY	ANY	/admin/categorie/{id}	App\Controller\Admin\CategorieController::delete()
app_admin_emprunt_index	GET	ANY	ANY	/admin/emprunt/	App\Controller\Admin\EmpruntController::index()
app_admin_emprunt_new	GET POST	ANY	ANY	/admin/emprunt/new	App\Controller\Admin\EmpruntController::new()
app_admin_emprunt_show	GET	ANY	ANY	/admin/emprunt/{id}	App\Controller\Admin\EmpruntController::show()
app_admin_emprunt_edit	GET POST	ANY	ANY	/admin/emprunt/{id}/edit	App\Controller\Admin\EmpruntController::edit()
app_admin_emprunt_delete	POST	ANY	ANY	/admin/emprunt/{id}	App\Controller\Admin\EmpruntController::delete()
app_admin_livre_index	GET	ANY	ANY	/biblio/livre/	App\Controller\Admin\LivreController::index()
app_admin_livre_new	GET POST	ANY	ANY	/biblio/livre/ajouter	App\Controller\Admin\LivreController::new()
app_admin_livre_show	GET	ANY	ANY	/biblio/livre/{titre}	App\Controller\Admin\LivreController::show()
app_admin_livre_edit	GET POST	ANY	ANY	/biblio/livre/{id}/modifier	App\Controller\Admin\LivreController::edit()
app_admin_livre_delete	POST	ANY	ANY	/biblio/livre/{id}/supprimer	App\Controller\Admin\LivreController::delete()
app_admin_livre	ANY	ANY	ANY	/admin/livres/liste	App\Controller\Admin\LivreController2::index()
app_admin_livre_ajouter	ANY	ANY	ANY	/admin/livres/ajouter	App\Controller\Admin\LivreController2::ajouter()
app_home	ANY	ANY	ANY	/	App\Controller\HomeController::index()
app_recherche	ANY	ANY	ANY	/search	App\Controller\RechercheController::index()
app_register	ANY	ANY	ANY	/inscription	App\Controller\RegistrationController::register()
app_login	ANY	ANY	ANY	/connexion	App\Controller\SecurityController::login()
app_logout	ANY	ANY	ANY	/deconnexion	App\Controller\SecurityController::logout()
app_test	ANY	ANY	ANY	/test	App\Controller\TestController::index()
app_test_test	ANY	ANY	ANY	/nimportequoi	App\Controller\TestController::test()
app_test_bonjour	ANY	ANY	ANY	/bonjour	App\Controller\TestController::bonjour()
app_test_accueil	ANY	ANY	ANY	/accueil	App\Controller\TestController::accueil()
app_test_calculatrice	ANY	ANY	ANY	/calculatrice/{a}/{b}	App\Controller\TestController::calculatrice()
app_test_calculatrice_1	ANY	ANY	ANY	/calculatrice/{a}/{b}	App\Controller\TestController::calculatrice()
app_test_tableau	ANY	ANY	ANY	/tableau	App\Controller\TestController::tableau()
app_test_tableau2	ANY	ANY	ANY	/tableau2	App\Controller\TestController::tableau2()
app_test_objet	ANY	ANY	ANY	/objet	App\Controller\TestController::objet()
_preview_error	ANY	ANY	ANY	/error/{code}.{_format}	error_controller::preview()
_wdt	ANY	ANY	ANY	/wdt/{token}	web_profiler.controller.profiler::toolbarAction()
_profiler_home	ANY	ANY	ANY	/_profiler/	web_profiler.controller.profiler::homeAction()
_profiler_search	ANY	ANY	ANY	/_profiler/search	web_profiler.controller.profiler::searchAction()
_profiler_search_bar	ANY	ANY	ANY	/_profiler/search_bar	web_profiler.controller.profiler::searchBarAction()
_profiler_phpinfo	ANY	ANY	ANY	/_profiler/phpinfo	web_profiler.controller.profiler::phpinfoAction()
_profiler_search_results	ANY	ANY	ANY	/_profiler/{token}/search/results	web_profiler.controller.profiler::searchResultsAction()

L'option --show-controllers

## IV. VIEW : Générer les vues pour l’affichage avec Twig

Twig est un moteur de templates. Il permet d’écrire des fichiers HTML dynamiques sans utiliser de balises PHP. Twig peut s’utiliser sans Symfony (comme Symfony peut s’utiliser sans Twig).

```
> composer require twig
```

Tout de suite après l’installation, tu peux remarquer qu’un nouveau dossier a été créé à la racine du projet : `templates/`. C’est ce dossier qui contiendra les vues du projet. Ces fichiers seront toujours suffixés par `.html.twig` dossier

cf. la [documentation de Twig](#) afin de retrouver toutes les fonctions et tous les filtres existants. Quelques exemples :

### A. Filtres : modifier la valeur préposée

C’est ainsi que sont nommés certaines fonctions dans Twig dont le but est de changer une valeur ou variable variables. La syntaxe est : `valeur|filtre`. Les filtres s’utilisent en réalité comme des fonctions, mais avec une syntaxe particulière. Par contre, le but est toujours de modifier une valeur existante.

- RAW : afficher une chaîne contenant des balises html qui seront interprétées.

```
{{ message|raw }}
```

- LENGTH : obtenir la taille d’un tableau ou la longueur d’une chaîne.

```
{{ tableau|length }}
```

- SLICE : couper une chaîne de caractères

```
{{ chaine|slice(0, 25) }}...
```

- DATE : afficher un objet DateTime avec le format voulu

```
{{ dateRetour|date('d/m/Y') }}
```

### B. Fonctions

Dans Twig, les fonctions sont... des fonctions. Cette fois la syntaxe est la même que pour une fonction PHP.

- INCLUDE : inclure un template

```
{{ include('vue.html.twig') }}
```

```
{# avec paramètres envoyés à la vue #}
```

```
{{ include('vue.html.twig', { variable: "valeur" }) }}
```

### C. Tests : tester une variable

- NULL : la variable a-t-elle la valeur NULL ?

```
{% if var is null %}
```

- ISSET : la variable est-elle définie ?

```
{% if var is defined %}
```

- EMPTY : la variable est-elle vide ? ⚠ vérifier que la variable existe avant de tester si la variable est vide

```
{% if var is set and var is empty %}
```

### D. Variable globale : app

La variable `app` permet de récupérer des informations sur le projet actuel. En particulier,

- `app.user` pour avoir les informations de l'utilisateur actuellement connecté.

```
<a class="nav-link" href="{ path("profil") }">{{ app.user.username }}</a>
```

- `app.request.pathinfo` pour connaître l'URL relative

```
{{ app.request.pathinfo == path("app_login") ? "active" : "" }}
```

- `app.flashes` : un array d'arrays qui contient les messages flashes

```
{% for type, messages in app.flashes %}
```

- `app.request.server.get` pour récupérer la valeur d'une variable globale

```
{{ app.request.server.get("APP_ENV") == "dev" ? "En maintenance" : "En production" }}
```

- `app.environment` pour connaître l'environnement du serveur
- `app.debug` pour connaître la valeur de la variable `APP_DEBUG`

### E. Configuration

Le fichier de configuration de Twig `twig.yaml` se trouve dans le dossier `config/packages` :

On peut y définir des constantes que l'on pourra utiliser dans les fichiers twig.

On peut définir le format de date par défaut que l'on veut afficher. Dans ce cas, pas besoin de préciser le format lorsque l'on utilise le filtre `date`

On y définit aussi le template qui sera utilisé pour afficher les formulaires.

cf. <https://symfony.com/doc/current/reference/configuration/twig.html>



## V. MODEL : gérer la base de données avec DOCTRINE

Symfony utilise Doctrine, une bibliothèque PHP qui permet de gérer une bdd avec un projet PHP orienté objet (Doctrine peut s'utiliser sans Symfony). Pour l'installer dans votre projet :

```
> composer require orm
```

### A. Informations de connexion : fichier .env


Après l'installation, il faut modifier le fichier `.env` (à la racine du projet), afin de définir les informations de connexion au SGBD utilisé par votre projet (ici, MySQL) :

```
DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=5.7
```

Il faut remplacer les termes commençant par `db_` par les valeurs correspondantes. S'il le faut, modifier aussi la version du serveur de SGBD et/ou le serveur de la bdd, si vous n'utilisez pas MySQL. Si la bdd n'est pas installé localement, il faut aussi modifier le serveur hôte et le port utilisé.

### B. Configuration : config/packages/doctrine.yaml

Cette étape n'est pas obligatoire. Mais si tu rencontres des difficultés à exécuter les commandes qui modifient la bdd, il faudra essayer de modifier le fichier `config/packages/doctrine.yaml` en ajoutant les informations suivantes :

 Attention à la syntaxe yaml : ne pas supprimer l'indentation (les tabulations) du fichier.

```
dbal:
  driver: pdo_mysql
  server_version: 5.7
  charset: utf8mb4
  default_table_options:
    charset: utf8mb4
    collate: utf8mb4_unicode_ci
  engine: InnoDB
  url: '%env(resolve:DATABASE_URL)%'
```

Bien entendu, ceci n'est qu'un exemple, il faut mettre les valeurs qui correspondent à ton environnement. Les commandes suivantes permettent de créer et modifier la bdd de votre projet. Une suite de question vous permettra de définir le nom des tables, des champs, ...

### C. Création de la bdd

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

Une bdd du nom défini dans la variable d'environnement `DATABASE_URL` sera créée. Ne pas lancer cette commande si votre bdd existe déjà.

## 1. Création d'une table

```
> symfony console make:entity
```

Cette commande va lancer un questionnaire pour définir une classe Entity dont les propriétés correspondront aux champs de la table correspondante.

Pour ajouter des champs à une table, il faut donc ajouter des propriétés à une entité. Si besoin, il est possible de lancer à nouveau la commande `make:entity` en spécifiant la classe entité à modifier.

NB : la commande ne peut pas servir pour supprimer un champ de la table. Il faut modifier directement le fichier de l'entité. Pour supprimer la table, il faut supprimer le fichier de l'entité.

## 2. Création d'une migration

```
> symfony console make:migration
```

Chaque fois que vous créez/modifiez une *entité*, vous **devez** créer une migration. Cette commande va créer un fichier contenant les requêtes SQL de modifications pour que la bdd correspondent aux *entités*.

Les modifications dans la bdd ne seront effectives **que lorsque vous aurez exécuté la migration**.

## 3. Exécution de la migration

```
> symfony console doctrine:migrations:migrate
```

La bdd va être mise à jour, les tables vont correspondre aux *entités*.

## 4. Annuler la migration

```
> symfony console doctrine:migrations:execute "classeVersion" --down
```

La méthode *down* de la classe précisée sera exécutée. Le nom de la classe ressemble à :

`DoctrineMigrations\VersionYYYYMMDDHHMMSS`

```
namespace DoctrineMigrations;

use Doctrine\DBAL\Schema\Schema;
use Doctrine\Migrations\AbstractMigration;

/**
 * Auto-generated Migration: Please modify to your needs!
 */
final class Version20220825140151 extends AbstractMigration
{
    public function getDescription(): string
    {
        return '';
    }

    public function up(Schema $schema): void
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->addSql('CREATE TABLE partage (id INT AUTO_INCREMENT NOT NULL, sessid VARCHAR(50) NOT NULL, chemin VARCHAR(255) NOT NULL)');
    }

    public function down(Schema $schema): void
    {
        // this down() migration is auto-generated, please modify it to your needs
        $this->addSql('DROP TABLE partage');
    }
}
```



## VI. Authentification, inscription avec SECURITY

### A. Composant Security

Le composant *security* va permettre de mettre en place l'authentification, l'inscription, et les droits d'accès à certaines pages.

```
> composer require security
```

### B. Entité utilisateur

Pour garder les informations des utilisateurs, il faut une table en bdd et donc une entité dans le projet. Mais cette entité sera créée avec la commande :

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

Le questionnaire qui suit demande le nom de la classe. Par défaut, cette entité est nommée User, mais rien n'empêche de changer le nom de cette entité.

Il faut ensuite choisir le champs qui permettra d'identifier de manière unique les utilisateurs. Par défaut, ce champ est email

↳ Création de l'authentification (connexion) :

```
> php bin/console make:auth
```

⚠ A noter, tu peux créer plusieurs systèmes d'authentification

La commande va créer plusieurs fichiers. Par exemple :

1. Le fichier `Security/LoginFormAuthenticator.php` :

Après une connexion, une page d'erreur va s'afficher. Ce n'est pas un bug, c'est le comportement prévu dans la méthode `LoginFormAuthenticator::onAuthenticationSuccess` qui est exécutée à la suite d'une authentification validée. L'instruction

```
throw new \Exception('TODO: provide a valid redirect inside '.__FILE__);
```

qui déclenche une exception. En commentaire, se trouve le code pour une redirection après une authentification :

```
return new RedirectResponse($this->urlGenerator->generate('some_route'));
```

Evidemment, il faut remplacer *some\_route* par le nom d'une route existant dans votre projet.

Il peut être judicieux de rediriger selon le rôle de l'utilisateur connecté (par exemple : l'espace client pour un utilisateur commun, l'espace gestion du site pour un administrateur).

- Tout d'abord, il faut utiliser la classe

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

C'est une classe qui ne peut être instanciée directement, il faut utiliser l'injection de dépendance. La modification de la déclaration de la méthode *onAuthenticationSuccess* n'est pas possible. Donc pour pouvoir utiliser un objet de cette classe, il sera intégré à la classe *LoginFormAuthenticator* par le biais d'une propriété.

- Ajouter une propriété :

```
private $security;
```

- Instancier cette propriété dans le constructeur de la classe :

```
public function __construct(Security $security, EntityManagerInterface $entityManager, ...)
{
    ...
    $this->security = $security;
}
```

- Cette propriété peut être utilisée pour tester le *rôle* de l'utilisateur connecté :

```
if($this->security->isGranted("ROLE_ADMIN")){
    return new RedirectResponse($this->urlGenerator->generate("admin"));
} else {
    return new RedirectResponse($this->urlGenerator->generate("profil"));
}
```



## VII. Développement front-end avec WEBPACK ENCORE

Permet de gérer (et générer) les fichiers ressources utilisées pour le front (js, css). Les fichiers seront compilés et placés dans le dossier `public/build`. Dans la suite, tu vas savoir comment ajouter les technologies suivantes à ton projet : jQuery, Bootstrap, FontAwesome.

```
> composer require encore
```

### A. Gestionnaire de dépendances : YARN

Pour utiliser webpack, il va falloir installer **yarn** sur ta machine. Tu trouveras l'installateur pour Windows [ici](#). Tu pourras aussi trouver les commandes pour l'installer sur un Mac sur le site officiel de Yarn. Quoiqu'il arrive, il faut que **npm** (= **nodejs**) soit auparavant installé.

Une fois **yarn** installé sur la machine, il va permettre d'installer les dépendances nécessaires à la génération des fichiers ressources.

1. Préparer l'utilisation de yarn dans le projet Symfony :

```
> yarn install
```

2. Pour compiler le Sass (requis pour les frameworks développés en Sass comme **Bootstrap**) :

```
> yarn add node-sass
```

```
> yarn add sass-loader
```

3. Pour utiliser la police d'icônes FontAwesome :

```
> yarn add @fortawesome/fontawesome-free
```

4. Pour utiliser jQuery, Popper.js<sup>3</sup> :

```
> yarn add jquery
```

```
> yarn add popper.js
```

---

<sup>3</sup> Bibliothèque JavaScript requise par Bootstrap. On peut n'installer que jQuery si Bootstrap n'est pas utilisé.

5. Pour utiliser Bootstrap (framework CSS) :

```
> yarn add bootstrap
```

6. Pour utiliser la fonction copyFiles (cf. ) :

## B. Configurer la compilation des fichiers ressources

Pour utiliser ces bibliothèques js/css, il va falloir modifier certains fichiers :

1. webpack.config.js

a) *Décommenter la ligne 69 pour activer la compilation du SASS*

```
// enables Sass/SCSS support
.enableSassLoader()
```

## C. Décommenter la ligne 69 pour éviter un bug lors de l'utilisation de jQuery

```
// uncomment if you're having problems with a jQuery plugin
.autoProvidejQuery()
```

## D. Placer les images dans le dossier public/images :

```
.copyFiles({
  from: './assets/img,

  // optional target path, relative to the output dir
  to: '../images/[path][name].[ext]',

  /*
  // if versioning is enabled, add the file hash too
  //to: 'images/[path][name].[hash:8].[ext]',

  // only copy files matching this pattern
  //pattern: /\. (png|jpg|jpeg)$/
  */
})
```

Remarque le [path]... Cela signifie que l'arborescence du dossier assets/images sera respectée dans le dossier public/images.



Bien entendu, rien ne t'empêche de renommer les dossiers soulignés dans l'exemple ou de décommenter les autres options.

Le dossier de copie doit être donné relativement au dossier défini comme dossier de sortie des fichiers ressources.

```
.setOutputPath('public/build/')
```

Pour pouvoir utiliser la fonction *copyFiles*, il faut installer *file-loader* :

```
> yarn add file-loader
```

#### assets/styles/app.css

- Modifie l'extension du fichier par `.scss` pour pouvoir écrire du SASS dans ce fichier.
- Pour utiliser FontAwesome, ajoute au début du fichier :

```
@import "~@fortawesome/fontawesome-free/css/all.min.css";
```

- Pour utiliser Bootstrap, ajoute :

```
@import "~bootstrap/scss/bootstrap";
```

- S'il tu veux redéclarer les variables SASS, il faut le faire avant d'importer le framework SASS. Par exemple, pour modifier la valeur *primary* de Bootstrap

```
$primary: #5887ff;
```

#### assets/app.js

- Modifie *app.css* par *app.scss* (seulement si tu as dû renommer ce fichier, bien entendu).
- Pour jQuery, ajouter :

```
const $ = require('jquery');
```

- Pour éviter les risques d'erreurs en utilisant jQuery, ajouter :

```
// ⚠ fixe le problème d'utilisation tardive de jQuery  
global.$ = global.jQuery = $;
```

- Pour utiliser le js de Bootstrap, ajoute :

```
require('bootstrap');
```

#### Les fichiers templates

Pour créer les liens vers les fichiers css et js créés, il faut ajouter dans un template les fonctions twig qui vont générer les balises correspondantes (link, script) :

- Lien pour les fichiers CSS :

```
{{ encore_entry_link_tags("app") }}
```

- Lien pour les fichiers JS :

```
{{ encore_entry_script_tags("app") }}
```

Générer les fichiers ressources :

Les fichiers à modifier se trouvent dans le dossier `assets`. Dès qu'un de ces fichiers est modifié, il faut lancer la commande pour compiler et générer les fichiers :

```
yarn encore dev
```

- ↳ La dernière partie de la commande correspond à l'environnement sur lequel les fichiers vont être compilés. Sur un serveur de production, remplacer `dev` par `prod`.

Pour éviter de lancer continuellement cette commande lors du développement frontend, il faut lancer la commande :

```
yarn encore dev --watch
```

- ↳ Tant que le processus sera actif, dès qu'une modification sera enregistrée, les fichiers ressources seront générés.

Avec Symfony CLI, on peut lancer cette commande en tâche de fond :

```
symfony run -d yarn encore dev --watch
```

Mettre à jour une dépendance

Ponctuellement, il faudra peut-être mettre à jour une dépendance installée avec yarn. La commande suivante permettra d'installer la version désirée :

```
yarn upgrade dependance@^1.x
```

- ↳ Remplacer *dependance* par le nom de la dépendance à actualiser
- ↳ Remplacer *1.x* par la version minimum à installer

## VIII. Formulaires, affichage et validation avec FORM VALIDATOR

Pour la création automatisée de formulaire basé sur des entités et la validation de ces formulaires selon des contraintes définies par le développeur.

```
C:/> composer require form validator
```

- Appliquer un style automatique aux formulaires créés

Modifier le fichier `config/packages/twig.yaml` :

Ajouter la ligne :

```
form_themes: ['bootstrap_4_layout.html.twig']
```

⚠ Vous pouvez utiliser d'autres thèmes, d'autres versions du thème bootstrap

- Créer une classe formulaire :

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

↳ Cette commande va créer une classe dans le dossier `src/Form`. Dans cette classe, seront définis les types d'input à afficher, les contraintes sur les champs du formulaire...

Si la classe `FormType` est créée à partir d'une entité, le formulaire sera rempli avec toutes les propriétés de la classe. Par défaut, les champs seront affichés par des inputs de type text.

⚠ Toutes les propriétés de la classe entité qui ont un type de donnée qui ne peut pas être converti en string (array, objet...) généreront une erreur lorsque tu essaieras d'afficher le formulaire. Il faut donc modifier la classe `Form` pour corriger ces erreurs.

- Modifier la classe formulaire :
- Plusieurs façons d'intégrer le formulaire dans un template twig :

```
1. {{ form(variable) }}

2. {{ form_start(variable) }}
   {# possibilité d'ajouter des balises HTML dans le formulaire #}
   {{ form_end(variable) }}

3. {{ form_start(variable) }}
   {{ form_row(form.inputname) }} {# pour chaque champ du formulaire #}
   {{ form_end(variable) }}
```

Pour les méthodes n°1 et n°2, tous les champs ajoutés dans la classe `FormType` correspondante seront affichés. La façon n°2 permet d'ajouter des éléments HTML dans la balise `<form>`, comme un bouton par exemple. Mais tous les inputs seront ajoutés après.

La méthode n°3 permet de réorganiser l'ordre des inputs en plus de pouvoir ajouter des éléments HTML. S'il y a des champs qui ne sont pas affichés explicitement (avec la fonction *form\_widget* ou *fom\_row*), ils seront ajoutés à la fin comme pour la façon n°2.

## **IX. ANNEXES**

## A. Remplir la bdd : Fixtures | Faker

En phase de développement du site, il peut être utile d'enregistrer de fausses informations dans la bdd. Cela permet de tester les requêtes pour récupérer des données mais aussi l'affichage de ces données, la pagination... Pour cela on utilise le composant Fixtures.

1. Installer le composant Fixtures

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

2. Créer un fichier fixture

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

3. Installer le composant Faker

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

Faker permet de générer des données aléatoires à utiliser avec les fixtures pour créer des données plus ou moins cohérentes (NB : les *fixtures* peuvent s'utiliser sans *faker*)

4. Exécution des fixtures :

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

Les fausses données vont être enregistrées en bdd.

Pour utiliser plus facilement les *fixtures*, en particulier si vous devez définir des relations entre différentes entités, tu vas pouvoir utiliser une classe BaseFixture dont vont devoir hériter toutes les classes *fixtures* que tu vas ajouter à ton projet.

## B. Gérer un formulaire avec un input de type *file*

Je veux créer dans mon formulaire un input de type file afin de pouvoir uploader des photos. Dans mon exemple, mon entité est **Annonce** et mon champ s'appelle *photo*. J'utilise **AnnonceType** (fichier généré par `make:form`) pour créer le formulaire. Ma route se trouve dans **AnnonceController**. Les images seront enregistrées dans le dossier `public/img`.

1. Dans le fichier `Form/AnnonceType.php`

a. Ajoutez (ou modifiez) le champ :

```
->add('photo', FileType::class, [ "mapped" => false, "help" => "* requis" ])
```

b. N'oubliez d'importer la classe à utiliser :

```
use Symfony\Component\Form\Extension\Core\Type\FileType;
```


c. L'option **"mapped"** avec la valeur **false** est importante. Elle précise que le champ ne doit pas être considéré comme propriété de la classe Entity liée au formulaire.

2. Dans le fichier `config/services.yaml`

Je vais créer un paramètre qui sera accessible à tout mon projet. C'est l'équivalent d'une constante globale au projet. Elle aura pour valeur le chemin du dossier image.

```
parameters:

    dossier_images: '%kernel.project_dir%/public/img'
```

 attention à la syntaxe YAML !

3. Dans le fichier `Controllers/AnnonceController.php`

Voici le code à placer dans la méthode (= route) qui va gérer le formulaire :

```

public function form(Request $rq, EntityManagerInterface $em){

    $nvAnnonce = new Annonce;
    // je crée un formulaire basé sur AnnonceType et je relie une entité à ce formulaire
    $form = $this->createForm(AnnonceType::class, $nvAnnonce);
    // le formulaire va gérer les informations de la requête HTTP
    $form->handleRequest($rq);

    // Si le formulaire a été soumis et si les valeurs respectent les règles de validation
    if($form->isSubmitted() && $form->isValid()){
        // je récupère la valeur du paramètre global "dossier_images"
        $destination = $this->getParameter("dossier_images");

        // je mets les informations de la photo téléchargée dans la variable $fichier
        // et je vérifie s'il y a bien une photo téléchargée (càd $fichier n'est pas nul)
        if( $fichier = $form->get("photo")->getData() ){

            // je récupère le nom du fichier original dans $nomPhoto
            $nomPhoto = pathinfo($fichier->getClientOriginalName(), PATHINFO_FILENAME);

            // j'utilise la classe AsciiSlugger pour supprimer les caractères interdits
            $nouveauNom = (new AsciiSlugger)->slug($nomPhoto);

            // je concatène une chaîne de caractères unique pour éviter d'avoir
            // 2 photos avec le même nom (sinon, la photo précédente sera écrasée)
            $nouveauNom .= "_" . uniqid() . "." . $fichier->guessExtension();

            // Le fichier est enregistré dans le dossier $destination avec le nouveau nom
            $fichier->move($destination, $nouveauNom);

            // je mets à jour l'objet Annonce en définissant la propriété photo
            $nvAnnonce->setPhoto($nouveauNom);

            // j'enregistre la nouvelle annonce dans la base de données
            $em->persist($nvAnnonce);
            $em->flush();

            // je définie le message qui sera affiché
            $this->addFlash("success", "Votre annonce a bien été enregistrée");

            // je redirige vers une route
            return $this->redirectToRoute("profil");
        }
    }

    $form = $form->createView();
    return $this->render("membre/annonce.html.twig", compact("form"));
}

```



4. Dans le fichier `config/packages/twig.yaml`

J'enregistre une variable globale pour Twig :

```
globals:
  ...
  dossier_images: "/img/"
```

5. Dans un fichier twig, lorsque je veux afficher les photos, à partir d'un objet `Entity\Annonce` :

```

```

### C. JS – Afficher immédiatement les images téléversées

```
function readURL(input) {  
  if (input.files && input.files[0]) {  
    var reader = new FileReader();  
  
    reader.onload = function (e) {  
      var idphoto = input.getAttribute("id") + "_img";  
      document.querySelector("#" + idphoto).setAttribute('src', e.target.result);  
    }  
    reader.readAsDataURL(input.files[0]);  
  }  
}
```

Il faut appeler cette fonction dans un écouteur d'évènement (EventListener) sur l'évènement *change* des éléments input de type *file*.

## E. Envoyer des e-mails avec XAMPP (PC)

Pour que ton serveur Apache puisse communiquer avec un serveur mail externe, il faut modifier le fichier `php.ini`. Dans l'exemple qui suit, tu vas utiliser ta boîte mail Gmail comme serveur mail (n'hésite pas à créer une boîte mail spécifique pour tester les envois d'e-mails avec ton projet Symfony. Cela évitera que ton adresse Gmail soit considérée comme spammeur).

### 1. Activer le module SSL

Donc dans le fichier `php.ini`, tu décommentes (c'est-à-dire qu'il faut supprimer le `;` au début de la ligne) la ligne suivante :

```
927 ;extension=oci8_12c ; Use with Oracle Database 12c Instant Client
928 ;extension=odbc
929 ;extension=openssl
930 ;extension=pdo_firebird
931 extension=pdo_mysql
```

L'extension peut être activé par une autre instruction :

```
966 magic_quotes_gpc=Off
967 magic_quotes_runtime=Off
968 magic_quotes_sybase=Off
969 extension=php_openssl.dll
970 extension=php_fpm.dll
971
972 [CLI Server]
973 ; Whether the CLI web server uses ANSI color coding in its terminal output.
974 cli_server.color=On
975
```

⚠ Il ne faut pas activer l'extension 2 fois. Une seule des 2 lignes doit être décommentée.

### 2. Entrer les informations du serveur e-mail

Toujours dans le fichier `php.ini`, dans la section `[mail function]` :

```
1084
1085 [mail function]
1086 SMTP=smtp.gmail.com
1087 smtp_port=587 465
1088 sendmail_from = ded.formateur.web@gmail.com
1089 sendmail_path = C:\xampp\sendmail\sendmail.exe -t
1090
```

### 3. Modifier les informations du server sendmail :

Dans le fichier `C:\xampp\sendmail\sendmail.ini` :

```
5
6 [sendmail]
7 smtp_server=smtp.gmail.com
8 smtp_port=465
9 debug_logfile=debug.log
10 auth_username=ded.formateur.web@gmail.com
11 auth_password=
12 force_sender=ded.formateur.web@gmail.com
13
```

NB : le mot de passe doit être écrit *en clair* dans le fichier

### 4. Tester l'envoi d'un e-mail

Pour tester l'envoi d'e-mail, tu peux utiliser un vrai-fausse adresse e-mail comme destinataire. Le site [yopmail.com](https://yopmail.com) permet d'utiliser n'importe quel adresse e-mail se terminant par `@yopmail.com`.

Voici un exemple de route pour tester l'envoi d'email :

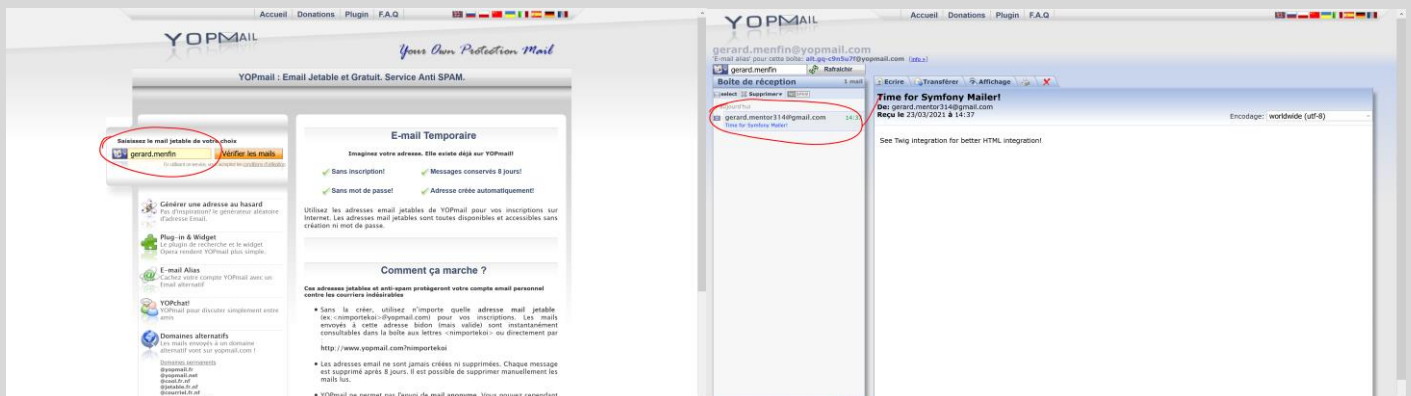
```
/**
 * @Route("/test/email")
 */
public function sendEmail(MailerInterface $mailer)
{
    $email = (new Email())
        ->from('hello@example.com')
        ->to('gerard.menfin@yopmail.com')
        //->cc('cc@example.com')
        //->bcc('bcc@example.com')
        //->replyTo('fabien@example.com')
        //->priority(Email::PRIORITY_HIGH)
        ->subject('Time for Symfony Mailer!')
        ->text('Sending emails is fun again!')
        ->html('<p>See Twig integration for better HTML integration!</p>');

    $mailer->send($email);
    $this->addFlash("success", "e-mail envoyé");
    // ...
    return $this->redirectToRoute("home");
}
```

N'oubliez pas d'inclure les *use* de toutes les classes que utilisées dans ce code :

```
use Symfony\Component\Mailer\MailerInterface;
use Symfony\Component\Mime\Email;
```

Cela permettra de savoir si la connexion à la boîte mail Gmail est bien configurée. Pour être sûr que l'e-mail a bien été reçu, un p'tit tour sur [yopmail.com](https://yopmail.com), en saisissant l'adresse voulue ([gerard.menfin](mailto:gerard.menfin@yopmail.com)), il sera possible de voir les e-mails reçus à l'adresse [gerard.menfin@yopmail.com](mailto:gerard.menfin@yopmail.com)



## F. Générer des PDF avec Symfony

### 1. Installer la bibliothèque DomPDF

DomPDF est la bibliothèque PHP la plus utilisée pour générer des PDF. Le plus gros inconvénient, c'est le manque de documentation.

Voici la commande pour ajouter la bibliothèque au projet :

```
> composer require dompdf/dompdf
```

### 2. Utiliser la classe DomPDF dans un contrôleur

N'oublie pas de préciser les classes que tu va utiliser.

```
use Dompdf\Dompdf;  
use Dompdf\Options;
```

Ensuite, tu ajoutes une nouvelle route au projet :

```
/**  
 * @Route("/pdf", name="pdf_upload")  
 */  
public function pdf()  
{  
    // Instant d'un objet Options pour configurer DomPDF  
    $pdfOptions = new Options();  
    $pdfOptions->set('defaultFont', 'Arial');  
  
    // Instantiation d'un objet Dompdf  
    $dompdf = new Dompdf($pdfOptions);  
  
    // On récupère du HTML à partir d'une vue twig  
    $html = $this->renderView('test/pdf.html.twig', [  
        'param' => 'valeur du paramètre'  
    ]);  
  
    // Ce code HTML servira de base à la création du PDF grâce à la méthode 'loadHtml'  
    $dompdf->loadHtml($html);  
  
    // Tu peux définir d'autres options avec la méthode 'setPaper' (format, disposition)  
    $dompdf->setPaper('A4', 'portrait');  
  
    // Enfin, la méthode 'render' génère le PDF  
    $dompdf->render();  
}
```

Que faire avec ce PDF ?

- Proposer le téléchargement du PDF par le navigateur :

```
$dompdf->stream("nom_fichier.pdf", [  
    "Attachment" => true  
]);
```

- Enregistrer le PDF sur ton serveur

```
// Récupérer le PDF au format binaire  
$output = $dompdf->output();  
  
// L'enregistrer dans le dossier public (il faut le chemin complet)  
$dossier = $this->getParameter('kernel.project_dir') . '/public';  
$cheminFichierPdf = $dossier . '/nom_fichier.pdf';  
  
// Write file to the desired path  
file_put_contents($cheminFichierPdf, $output);
```

- Ne pas oublier que la méthode doit retourner un objet Response

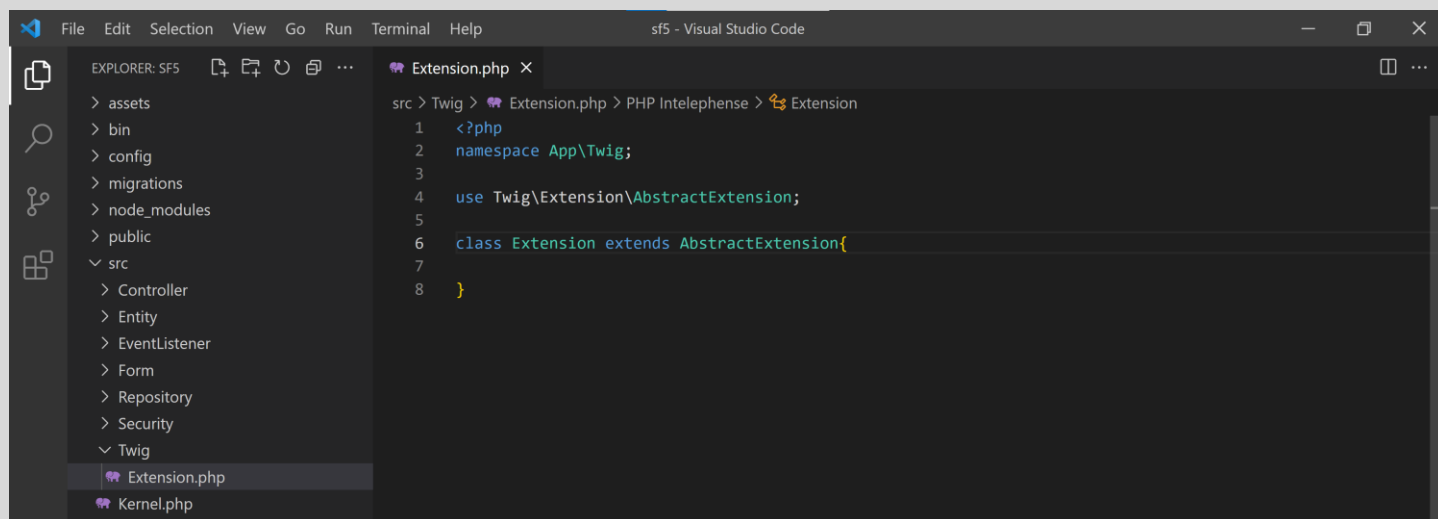
```
return new Response("The PDF file has been succesfully generated !");  
}
```

## G. TWIG : ajouter une nouvelle fonction

Comment ajouter une fonction qui sera utilisable dans n'importe quel fichier Twig ?

Par exemple, il n'existe de pas de test pour savoir si une variable contient une valeur numérique, ce qui existe en PHP. Voici comment ajouter ce test :

1. Créer une classe qui hérite de la classe **Twig\Extension\AbstractExtension** dans un fichier placé dans le dossier `src/Twig`. Donc le namespace de cette nouvelle classe sera **App\Twig**. Dans l'exemple la classe s'appelle **Extension**, le fichier sera donc `src/Twig/Extension.php`.



2. Selon le type de fonction twig que tu veux ajouter (filtre, fonction ou test), tu vas renseigner la classe :

```
use Twig\TwigFilter;  
use Twig\TwigFunction;  
use Twig\TwigTest;
```

3. Écrire le code de la fonction à ajouter
4. De la même façon, selon le type de fonction que tu veux ajouter, tu vas utiliser la méthode **getTypedefonctions** pour enregistrer la nouvelle fonction dans les fonctions utilisables.

Cette méthode **getTypedefonctions** doit renvoyer un array d'objet de la classe **TwigTypedefonction**.

5. Enregistrer l'extension comme service.

L'extension est enregistrée automatiquement comme service si le fichier de config `services.yaml` par défaut est utilisé (cf. section services-App). Dans Symfony 5, c'est le cas. Dans ce fichier, toutes les classes qui sont définis dans le dossier `src` sont enregistrées comme service.

6. Pour savoir si les fonctions/filtres créés sont bien reconnus par Twig, lancer la commande :

```
> php bin/console debug:twig
```

Votre fonction doit apparaître dans la liste des fonctions correspondantes.

## H. Pages d'erreurs HTML

### 1. Customiser une page d'erreur :

Dans le dossier `templates`, ajouter l'arborescence suivante :

**`bundles/TwigBundle/Exception`**

Il suffit de créer dans ce dossier des vues dont les nom seront formés ainsi :

**`errorxxx.html.twig`**

`xxx` étant le numéro de l'erreur HTML (403, 404, 500...)

Pour tester, il faut passer en environnement de production. En effet, tant que tu es dans un environnement de développement, Symfony t'affichera des erreurs détaillées, bien plus utile pour le développeur.

Rappel : pour passer en environnement de production, modifier la valeur de **APP\_ENV** dans le fichier `.env`.

Tu peux mettre *prod* ou *production*.

### 2. Rediriger vers une page d'erreur

- **404** : `throw $this->createNotFoundException('The product does not exist');`
- **403** : `createAccessDenied`
- **500** : `throw new \Exception('Something went wrong!');`



## I. `.htaccess`

Dans l'optique de placer son projet Symfony sur le serveur d'un hébergeur web, il faut paramétrer le serveur Apache afin que toutes les URL de votre site soient redirigées vers le dossier `public`.

Ce serait fait grâce à un fichier `.htaccess`, qui sera créé automatiquement en installant un nouveau module grâce à Composer :

```
> composer require symfony/apache-pack
```