

Création du projet Symfony (version 4.1)

Introduction

Dans cette suite de TP nous apprendrons à installer un projet Symfony puis nous créerons ensemble un formulaire de création d'entités, puis à les afficher sur une page annexe.

Les outils utilisés dans ces TP seront les suivants :

- NetBeans 8.2 : <https://netbeans.org/downloads/> version PHP pour architecture 64 bits
- WampServer (PHP 7.2.4, Apache 2.4.33, MySQL 5.7.21) :
<https://sourceforge.net/projects/wampserver/files/WampServer%203/WampServer%203.0.0/>
- MySQL Workbench 6.3 : <https://www.mysql.com/products/workbench/>
- Composer : <https://getcomposer.org/download/>
- Cmdr : <http://cmdr.net/> J'ai choisi d'utiliser Cmdr plutôt que l'invite de commande de Windows ou encore PowerShell parce qu'il est plus joli esthétiquement parlant, mais surtout parce qu'il est plus facile de s'y retrouver dans les instructions grâce à sa coloration syntaxique. Mais libre à vous d'utiliser l'outil qui vous convient !
- Bootstrap 4.1 : <https://getbootstrap.com/>
- Font Awesome 5.2 : <https://fontawesome.com/>
- Twig : <https://twig.symfony.com/>
- Git : <https://git-scm.com/>
- Doctrine : <https://www.doctrine-project.org/>
- Un navigateur au choix (Firefox, Vivaldi, Chrome, Internet Explorer pour les plus courageux...)
- Spotify

Si vous êtes en BTS SIO au lycée Bonaparte de Toulon, ce sont les outils que vous devez normalement utiliser couramment lors de vos TP en développement web ;-)

Pourquoi Symfony et pas Laravel alors que ce dernier est plus utilisé ?

La phrase que je vais écrire n'est pas forcément valable dans tous les cas, mais qui peut le plus, peut le moins. Ensuite M. Roche vous le répétera sûrement, mais « Un développeur Symfony sait coder avec Laravel, mais l'inverse n'est pas forcément vrai ». Et la raison est assez simple, en fait Laravel a été développé à partir de Symfony. Je vous parlerai plus de Symfony à la toute fin de ce TP.

Plugins essentiels pour NetBeans :

- Symfony 2/3 Framework
- Twig Templates
- HTML5 Kit
- PHP

Ce TP n'est pas à faire d'un coup ! Il est préférable d'avancer étape par étape et comprendre ce que l'on fait plutôt que de faire un « rush », de terminer avant les autres et de penser que le professeur en sera satisfait ! Nous sommes là pour passer de l'état « consciemment incompétent » à l'état « consciemment compétent » (cf. SLAM5 avec M. Gil).

Installation du projet

Première étape : Se rendre sur le site de Symfony (<https://symfony.com/>), puis allez dans la documentation. Pour Symfony comme pour d'autres langages ou frameworks, ayez le réflexe d'aller voir la documentation dès que vous souhaitez apprendre quoi que ce soit ou dès que vous bloquez. En effet, celle de Symfony est très complète, et vous donne de petits tutoriels à **jour** vous permettant de visualiser le fonctionnement des formulaires, de l'ORM avec Doctrine, etc... La documentation **officielle** sera toujours plus fiable que ce que vous pourrez trouver sur YouTube, parce qu'elle est à jour et écrite par des contributeurs du projet Symfony.

Ensuite vous pouvez toujours vous aider du cours Symfony présent sur OpenClassrooms (<https://openclassrooms.com/fr/courses/3619856-developpez-votre-site-web-avec-le-framework-symfony>) qui a été écrit par Fabien POTENCIER, le créateur de Symfony. Alors oui le cours sur OpenClassrooms porte sur la version 2 du framework. Et c'est bien pour cela que j'insiste vraiment sur le fait qu'il faut consulter en premier lieu la documentation officielle. Mais depuis, ce qui a réellement changé dans Symfony, c'est la gestion des packages et l'architecture du projet. D'autres choses n'ont pas vraiment bougé, comme par exemple les formulaires.

Si vous voulez une approche complémentaire de Symfony, un cours vidéo est proposé dans la documentation : <https://knpuniversity.com/screencast/symfony> . Le cours est en anglais, mais largement compréhensible.

Maintenant que ce qu'il y avait à dire a été dit, nous pouvons enfin commencer.

Dans la version 4 de Symfony, l'installation se fait via Composer et pas avec la commande **Symfony** comme dans la version 3.

Pour installer le projet, ouvrez votre console, déplacez vous dans le dossier www de WampServer et entrez la commande suivante :

```
D:\wamp\www
λ composer create-project symfony/skeleton tp_symfony
Installing symfony/skeleton (v4.1.3.1)
Installing symfony/skeleton (v4.1.3.1): Downloading (100%)
Created project in tp_symfony
Loading composer repositories with package information
Installing dependencies (including require-dev) from lock file
Package operations: 21 installs, 0 updates, 0 removals
- Installing symfony/flex (v1.0.89): Loading from cache
- Installing psr/cache (1.0.1): Loading from cache
- Installing psr/container (1.0.0): Loading from cache
- Installing psr/simple-cache (1.0.1): Loading from cache
- Installing symfony/polyfill-mbstring (v1.8.0): Loading from cache
- Installing symfony/console (v4.1.3): Loading from cache
- Installing symfony/routing (v4.1.3): Loading from cache
- Installing symfony/http-foundation (v4.1.3): Loading from cache
- Installing symfony/event-dispatcher (v4.1.3): Loading from cache
- Installing psr/log (1.0.2): Loading from cache
- Installing symfony/debug (v4.1.3): Loading from cache
- Installing symfony/http-kernel (v4.1.3): Loading from cache
- Installing symfony/finder (v4.1.3): Loading from cache
- Installing symfony/filesystem (v4.1.3): Loading from cache
- Installing symfony/dependency-injection (v4.1.3): Loading from cache
- Installing symfony/config (v4.1.3): Loading from cache
- Installing symfony/cache (v4.1.3): Loading from cache
- Installing symfony/framework-bundle (v4.1.3): Loading from cache
- Installing symfony/yaml (v4.1.3): Loading from cache
- Installing symfony/dotenv (v4.1.3): Loading from cache
Generating autoload files
Symfony operations: 4 recipes (96f0d4f1b572d821b37441c223a3ab3b)
- Configuring symfony/flex (>=1.0): From github.com/symfony/recipes:master
- Configuring symfony/framework-bundle (>=3.3): From github.com/symfony/recipes:master
- Configuring symfony/console (>=3.3): From github.com/symfony/recipes:master
- Configuring symfony/routing (>=4.0): From github.com/symfony/recipes:master
Executing script cache:clear [OK]
Executing script assets:install public [OK]
Some files may have been created or updated to configure your new packages.
Please review, edit and commit them: these files are yours.

What's next?

* Run your application:
1. Change to the project directory
2. Create your code repository with the git init command
3. Execute the php -S 127.0.0.1:8000 -t public command
4. Browse to the http://localhost:8000/ URL.

Quit the server with CTRL-C.
Run composer require server --dev for a better web server.

* Read the documentation at https://symfony.com/doc
```

Nous pouvons voir que le projet a été installé à l'endroit où nous avons lancé la commande avec ses dépendances.

La console nous dit également que nous pouvons lancer un serveur web avec une commande PHP dans la rubrique **What's next** ?. Mais il faut d'abord se déplacer à la racine du projet.

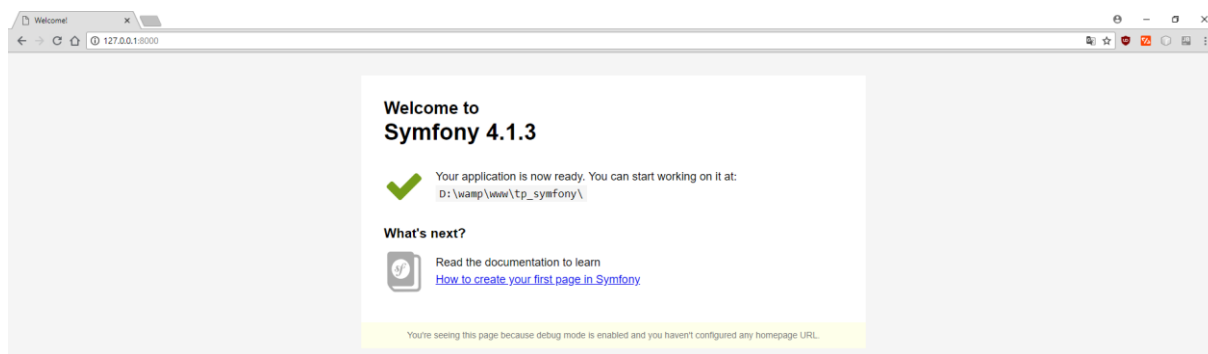
Essayons :

```
D:\wamp\www
λ php -S 127.0.0.1:8000 -t public
Directory public does not exist.

D:\wamp\www
λ cd tp_symfony\

D:\wamp\www\tp_symfony
λ php -S 127.0.0.1:8000 -t public
PHP 7.2.4 Development Server started at Sun Aug  5 18:52:33 2018
Listening on http://127.0.0.1:8000
Document root is D:\wamp\www\tp_symfony\public
Press Ctrl-C to quit.
|
```

Le serveur semble s'être lancé. Essayons ensuite dans le navigateur :



Notre projet est donc fonctionnel ! Nous pouvons maintenant arrêter le serveur avec **Ctrl+C**.

Les dépendances

On peut également observer que Composer a installé les dépendances supplémentaires nécessaires au projet comme `symfony/flex`.

Qu'est-ce que Flex ?

La réponse dans la documentation ! <https://symfony.com/doc/current/setup/flex.html>

C'est un plugin Composer dont les objectifs de Flex sont de faciliter la gestion des dépendances et d'avoir une configuration par défaut qui fonctionne immédiatement. Il requiert au minimum la version 7.1 de PHP (nous sommes dans la version 7.2.4).

Oui mais concrètement ?

Installation d'un package sans Flex :

- `composer require mon-package`
- Instancier le(s) package(s) dans le Kernel (un gros-mot)
- Créer la configuration dans `app/config/config.yml` (un autre)
- Importer le routing dans `app/config/routing.yml` (ô joie)

Et avec Flex :

- `composer require mon-package` (plus facile)

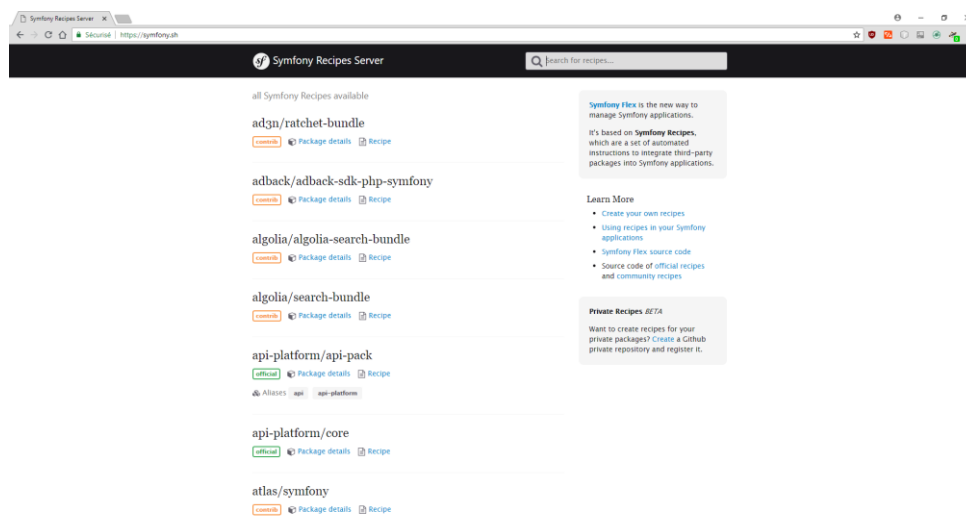
Flex automatise donc l'installation des packages et de leurs dépendances et permet d'utiliser des alias pour les installer.

Les alias permettent d'installer des packages en utilisant un nom plus court que leur nom complet.

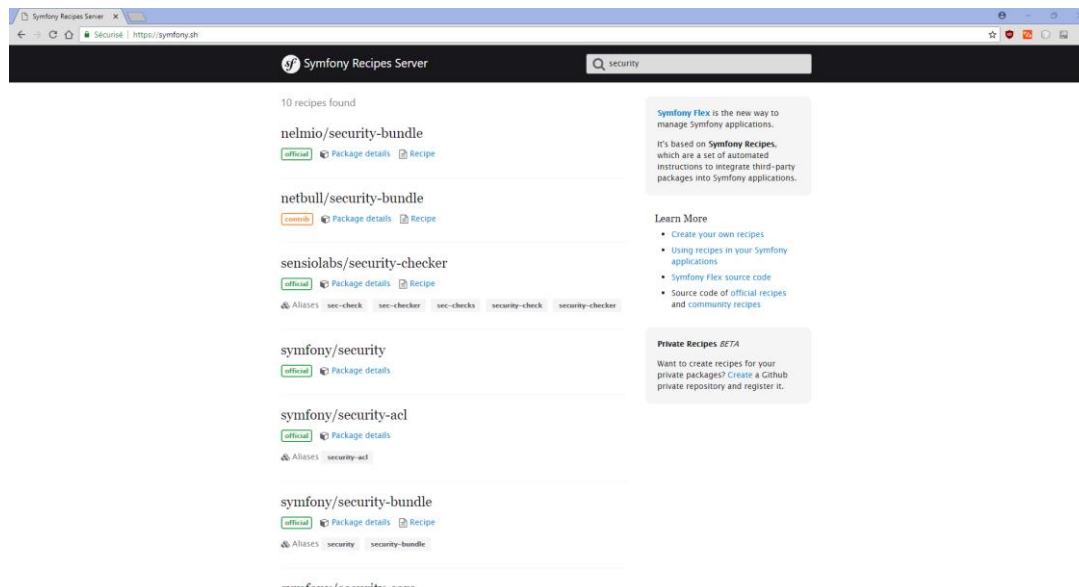
Par exemple : l'alias de package `api-platform/api-pack` est `api`. L'utilisation des alias permet de gagner du temps d'écriture et en ne se trompant pas dans le nom du package. C'est tout de même plus confortable.

Comment connaître le nom des packages et leurs alias ?

Il y a UN site à connaître : <https://symfony.sh/>.

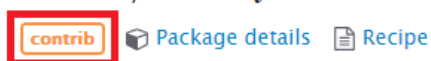


Par exemple, nous souhaitons installer dans notre projet un recipe (autre nom des packages) permettant de vérifier la conformité des autres recipes que nous installerons par la suite dans notre projet. Nous allons donc taper dans la barre de recherche du site [security](#).

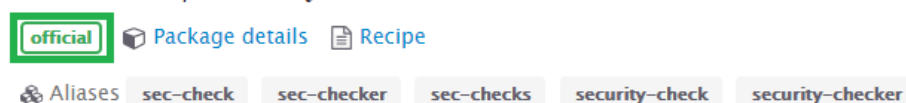


Nous nous retrouvons avec un certain nombre de packages... Quand nous ne sommes pas habitués, ça peut porter confusion. La première chose à regarder, ce sont les petites étiquettes suivantes :

netbull/security-bundle



sensiolabs/security-checker

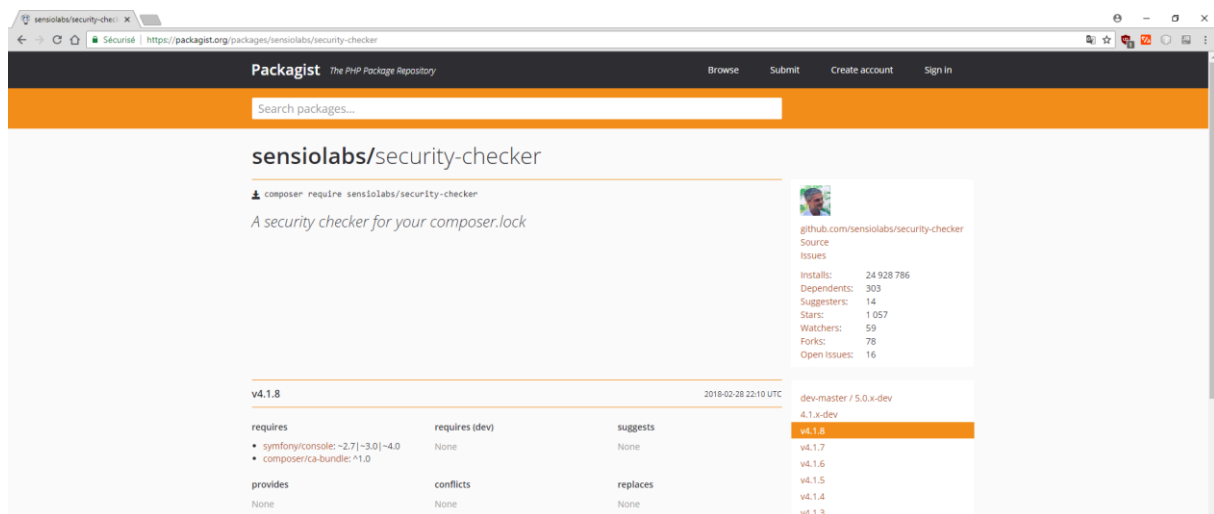


Toujours privilégier les recipes **officielles** par soucis de fiabilité.

Les recipes sont nommées de la manière suivante : [nom-auteur/nom-recipe](#).

Dans le cas des deux recipes précédentes, nous avons donc deux auteurs distincts : Netbull et SensioLabs. Sachant que ce dernier est l'entreprise qui a créé Symfony, il serait préférable de choisir celui-ci.

Ensuite, cliquez sur [Package details](#) du second recipe, nous sommes redirigés sur Packagist qui est le site où sont stockés tous les packages pour PHP que nous pouvons installer via Composer.



Cette page renseigne donc sur le nom du package, son auteur, ses dépendances, sa version, etc... et sa **description**.

A security checker for your composer.lock

D'après la documentation de Composer (<https://getcomposer.org/doc/01-basic-usage.md#composer-lock-the-lock-file>), le fichier `composer.lock` permet de garder les dépendances dans un « état connu ». C'est-à-dire que lorsque vous versionnez votre projet, les packages et leurs dépendances ne sont pas versionnés (imaginez le temps que vous mettriez à récupérer votre projet si vous utilisez un VCS distant comme GitLab ou GitHub !). Mais le `composer.json` et le `composer.lock` le sont eux. Et ils permettent, lorsque vous venez de télécharger votre projet depuis un VCS distant, de télécharger toutes les dépendances que vous aviez installées en conservant la version qu'elles avaient sur votre projet (pour éviter les problèmes liés à la compatibilité) en lançant simplement la commande `composer update`.

Le package que nous avons trouvé est donc celui que nous cherchions !

Essayons donc de l'installer sans l'alias :

```
D:\wamp\www\tp_symfony
λ composer require sensiolabs/security-checker
Using version ^4.1 for sensiolabs/security-checker
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 2 installs, 0 updates, 0 removals
  - Installing composer/ca-bundle (1.1.1): Loading from cache
  - Installing sensiolabs/security-checker (v4.1.8): Loading from cache
Writing lock file
Generating autoload files
Symfony operations: 1 recipe (dd8aa2330280372f25817218fcff7788)
  - Configuring sensiolabs/security-checker (>=4.0): From github.com/symfony/recipes:master
Executing script cache:clear [OK]
Executing script assets:install public [OK]
Executing script security-checker security:check [OK]

Some files may have been created or updated to configure your new packages.
Please review, edit and commit them: these files are yours.
```


Il s'est bien installé. Maintenant nous allons le désinstaller à l'aide de la commande `composer remove sensiolabs/security-checker` :

```
D:\wamp\www\tp_symfony
λ composer remove sensiolabs/security-checker
Dependency "symfony/console" is also a root requirement, but is not explicitly whitelisted. Ignoring.
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 0 installs, 0 updates, 2 removals
  - Removing sensiolabs/security-checker (v4.1.8)
  - Removing composer/ca-bundle (1.1.1)
Writing lock file
Generating autoload files
Symfony operations: 1 recipe (e0fcdedc193debce5030027f991a385b)
  - Unconfiguring sensiolabs/security-checker (>=4.0): From github.com/symfony/recipes:master
Executing script cache:clear [OK]
Executing script assets:install public [OK]
```

Maintenant installons-le avec un de ses alias :

```
D:\wamp\www\tp_symfony
λ composer require sec-check
Using version ^4.1 for sensiolabs/security-checker
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 2 installs, 0 updates, 0 removals
  - Installing composer/ca-bundle (1.1.1): Loading from cache
  - Installing sensiolabs/security-checker (v4.1.8): Loading from cache
Writing lock file
Generating autoload files
Symfony operations: 1 recipe (c9d042b79963993824051ff542cfffbc)
  - Configuring sensiolabs/security-checker (>=4.0): From github.com/symfony/recipes:master
Executing script cache:clear [OK]
Executing script assets:install public [OK]
Executing script security-checker security:check [OK]

Some files may have been created or updated to configure your new packages.
Please review, edit and commit them: these files are yours.
```

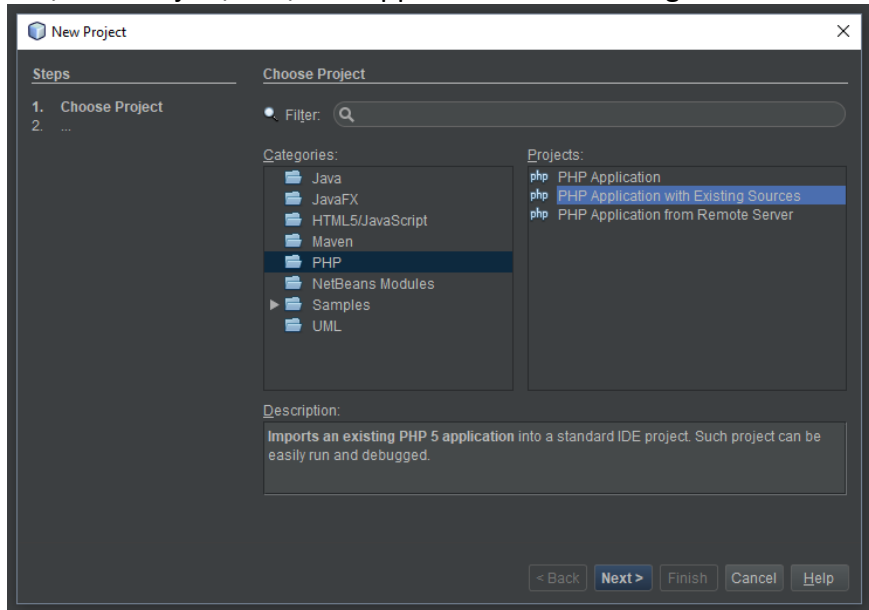
Nous pouvons même observer qu'il s'est lancé à la fin de l'installation :

```
Executing script security-checker security:check [OK]
```

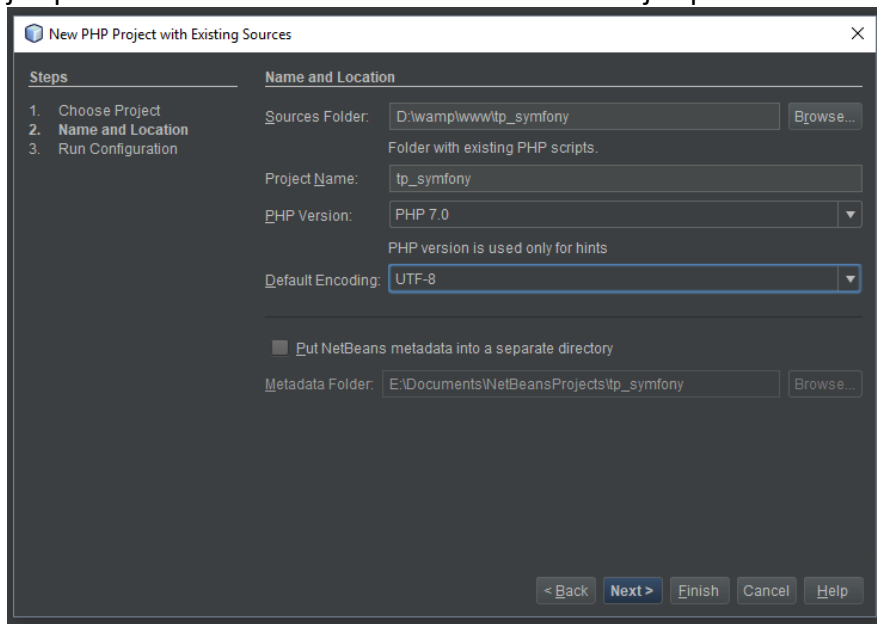

Ouverture du projet

Maintenant que vous en savez plus sur les dépendances et sur Symfony, je vous propose d'ouvrir votre projet dans NetBeans.

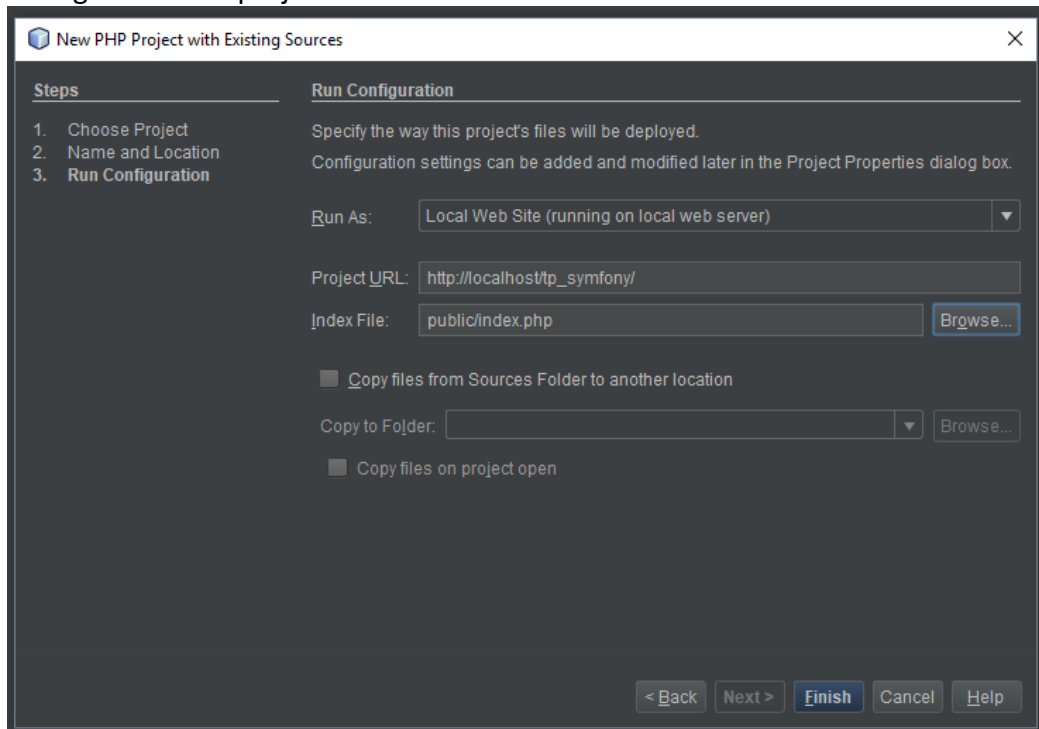
1. Ouvrir NetBeans
2. File/New Project/PHP/PHP Application with Existing Sources



3. Importer le projet et choisir la version de PHP à appliquer (NetBeans 8.2 ne va que jusqu'à la version 7.0 de PHP et NetBeans 9.0 ira jusqu'à la version 7.1)

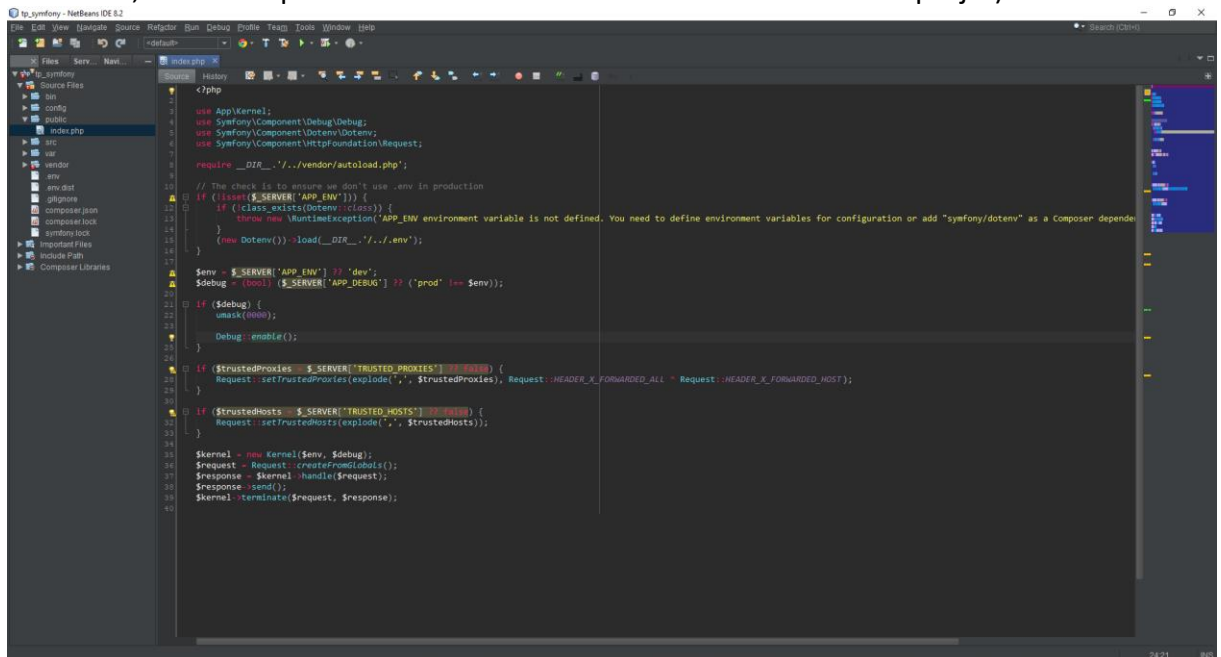


4. Changer l'URL du projet



Si vous ouvrez le fichier `public/index.php`, vous verrez qu'il ne contient pas le code de la page que nous avons vu précédemment. Il s'agit du contrôleur frontal (vous entendrez souvent parler de front controller) ! Plus d'explications sur ce fichier plus bas.

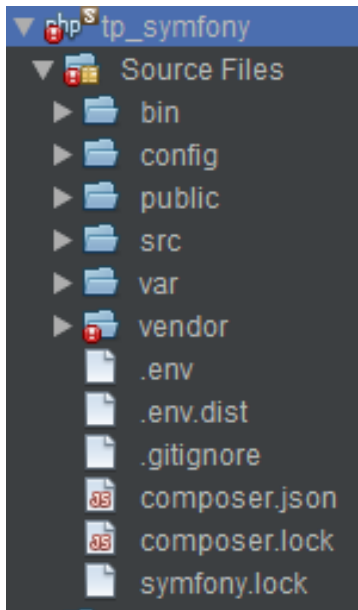
5. Votre projet s'ouvre dans NetBeans (ne pas faire attention aux erreurs levées par NetBeans, elles n'empêchent en aucun cas le bon fonctionnement du projet)



Le projet Symfony

Je vais vous donner des explications sur ce que vous avez à l'écran pour que vous puissiez par la suite vous repérer dans votre projet. Cette partie du TP constitue un rappel de ce qui a été vu avec M. Roche.

L'arborescence :



- **/bin** : Contient les fichiers exécutables, notamment bin/console pour exécuter les commandes PHP
- **/config** : Le dossier /config/packages contient les fichiers de configuration individuels des bundles que l'on installera. Un fichier par paquet et par environnement. Les fichiers de configuration peuvent être écrits en PHP, XML ou YAML. Le fichier **bundles.php** contient la référence de tous les bundles installés.
- **/public** : Il s'agit du dossier racine du site. Il contient le fichier **index.php**, le contrôleur frontal de l'application. C'est dans ce dossier que nous mettrons les fichiers CSS, JavaScript, images, fonts, etc... Pour faire simple, les ressources du site.
- **/src** : Contient le code source de l'application, mais aussi le noyau (**kernel.php**). Il contiendra également l'ensemble des contrôleurs, entités, repositories, etc...
- **/var** : Contient les fichiers de cache et les logs. On peut y créer un dossier tmp pour stocker des fichiers temporaires.
- **/vendor** : Contient tous les packages installés pour l'application.
- **.env** : Contient la définition des variables d'environnement, notamment les paramètres de la base de données liées à l'application. C'est ici que nous définissons si nous sommes dans un environnement de développement (dev) ou de production (prod).
- **.env.dist** : Exemple de fichier .env.
- **.gitignore** : Contient la liste des éléments du projet à ne pas versionner (cf : le cours de M. GIL de première année sur le versionning).

- `composer.json` : Contient les packages installés avec leur version.
- `composer.lock` : Ce fichier verrouille les dépendances de votre projet à un état connu.
- `symfony.lock` : C'est le bon fichier de verrouillage pour les recettes Symfony au lieu d'essayer de deviner via l'état de `composer.lock`. In verra que Flex conserve dans ce fichier les pistes des recettes qu'il a installées.

Le contrôleur frontal : C'est le point d'entrée de l'application. Toutes les demandes de page passent par ce fichier. Sa mission sera de charger le kernel (noyau) qui se chargera de trouver la bonne route pour renvoyer la bonne page.

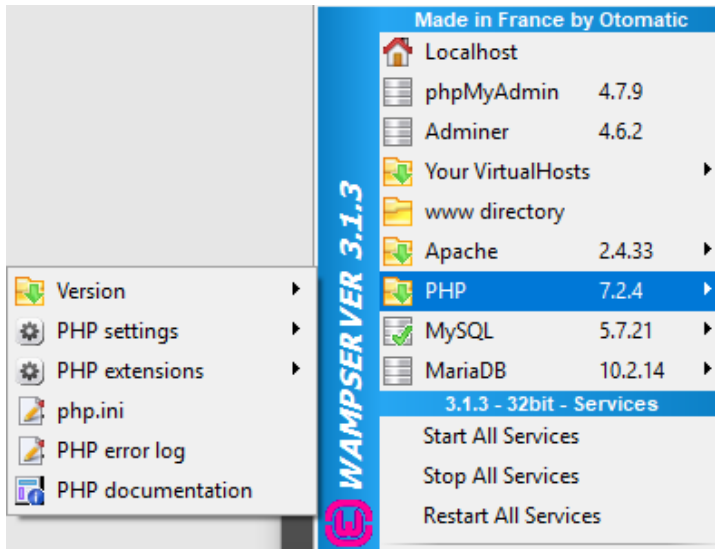
Doctrine : Doctrine est un ORM (Object-Relational Mapping) pour PHP. Pour Java EE, nous avons JPA. Un ORM peut être interprété comme étant une interface entre une application et une base de données relationnelle pour simuler une base de données orientée objet. Il définit donc des correspondances entre les schémas de la base de données et les classes de l'application (vous comprendrez plus tard de quoi je parle si ce n'est pas déjà le cas).



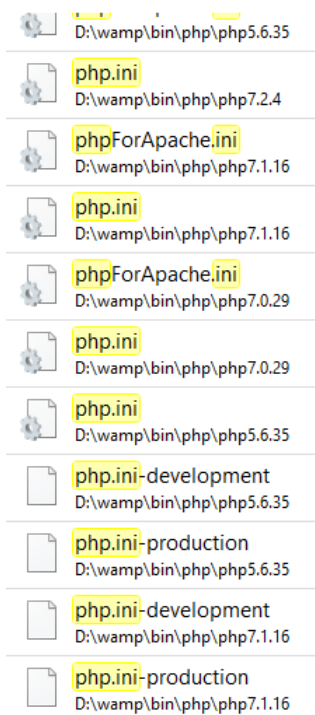
Il est important de maîtriser le vocabulaire si vous voulez coder correctement (cf. cours sur la POO de M. Roche). J'ajouterais même que c'est en forgeant qu'on devient forgeron ! Au terme de ce TP toutes les notions seront devenues naturelles pour vous ! Il n'est pas forcément indispensable de les apprendre par cœur maintenant, ça viendra tout seul en pratiquant.

Configuration de WampServer

La première étape consiste à démarrer WampServer, jusque-là pas de difficulté. Une fois celui-ci démarré, cliquez sur l'icône dans la barre des tâches, et allez dans **PHP/Version** puis cochez la version 7.2.4.

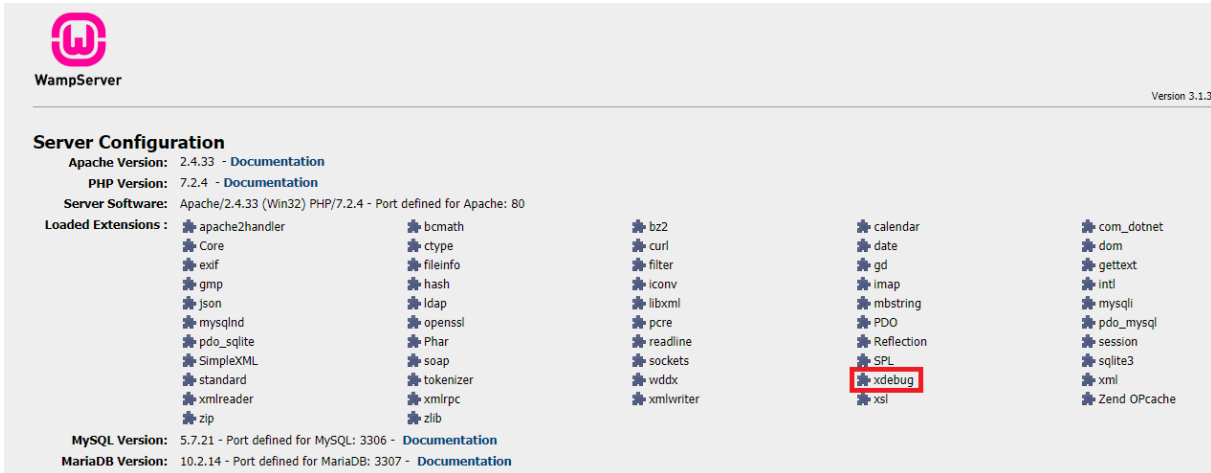


Ensuite, toujours dans le menu PHP de Wamp, cliquez sur php.ini. Je vous conseille de gérer la configuration de PHP de cette manière car il y a de nombreux fichiers php.ini dans les fichiers de Wamp :



Il faut configurer XDebug en suivant les instructions du lien suivant : [http://wiki.netbeans.org/HowToConfigureXDebug#How to configure xdebug with WAMP](http://wiki.netbeans.org/HowToConfigureXDebug#How_to_configure_xdebug_with_WAMP)

Une fois configuré, redémarrez Wamp, puis tapez **localhost** (ou **127.0.0.1**) dans la barre d'adresse de votre navigateur :



WampServer Version 3.1.3

Server Configuration

Apache Version: 2.4.33 - [Documentation](#)

PHP Version: 7.2.4 - [Documentation](#)

Server Software: Apache/2.4.33 (Win32) PHP/7.2.4 - Port defined for Apache: 80

Loaded Extensions :

- apache2handler
- Core
- exif
- gmp
- json
- mysqlnd
- pdo_sqlite
- SimpleXML
- standard
- xmlreader
- zip
- bcmath
- ctype
- fileinfo
- hash
- ldap
- openssl
- Phar
- soap
- tokenizer
- xmlrpc
- zlib
- bz2
- curl
- filter
- iconv
- libxml
- pcre
- readline
- sockets
- wddx
- xmlwriter
- calendar
- date
- gd
- imap
- mbstring
- PDO
- Reflection
- SPL
- xdebug
- xsl
- com_dotnet
- dom
- gettext
- intl
- mysqli
- pdo_mysql
- session
- sqlite3
- xml
- Zend OPcache

MySQL Version: 5.7.21 - Port defined for MySQL: 3306 - [Documentation](#)

MariaDB Version: 10.2.14 - Port defined for MariaDB: 3307 - [Documentation](#)

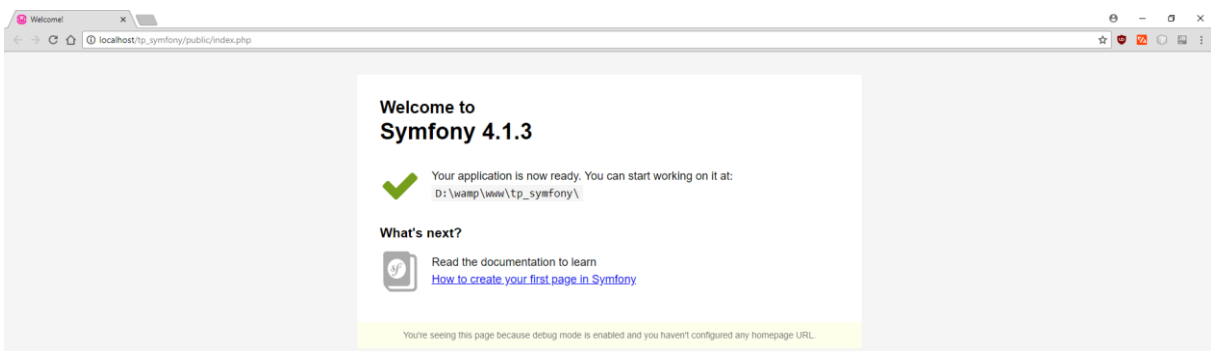
Puis vérifions en cliquant sur **phpinfo()** qui se trouve dans la rubrique « Tools » de la page localhost :

xdebug

| xdebug support | enabled | |
|---------------------------------|-------------|--------------|
| Version | 2.6.0 | |
| IDE Key | GL552\$ | |
| Supported protocols | | |
| DBGp - Common DeBuGger Protocol | | |
| Directive | Local Value | Master Value |
| xdebug.auto_trace | Off | Off |
| xdebug.cli_color | 0 | 0 |

Maintenant nous allons lancer le projet depuis WampServer plutôt que d'utiliser le serveur PHP. De cette manière nous nous rapprocherons plus d'une configuration réelle, et puis nous pourrons utiliser XDebug, plutôt que d'insérer des **var_dump()** dans le code qui ne sont pas très propres, et que vous risquez d'oublier par la suite.

Eh bien pour voir si cela fonctionne sur Wamp, vous n'avez qu'à lancer le projet depuis NetBeans en cliquant sur « Run Project » ou en appuyant sur la touche F6 :



Welcome to **Symfony 4.1.3**

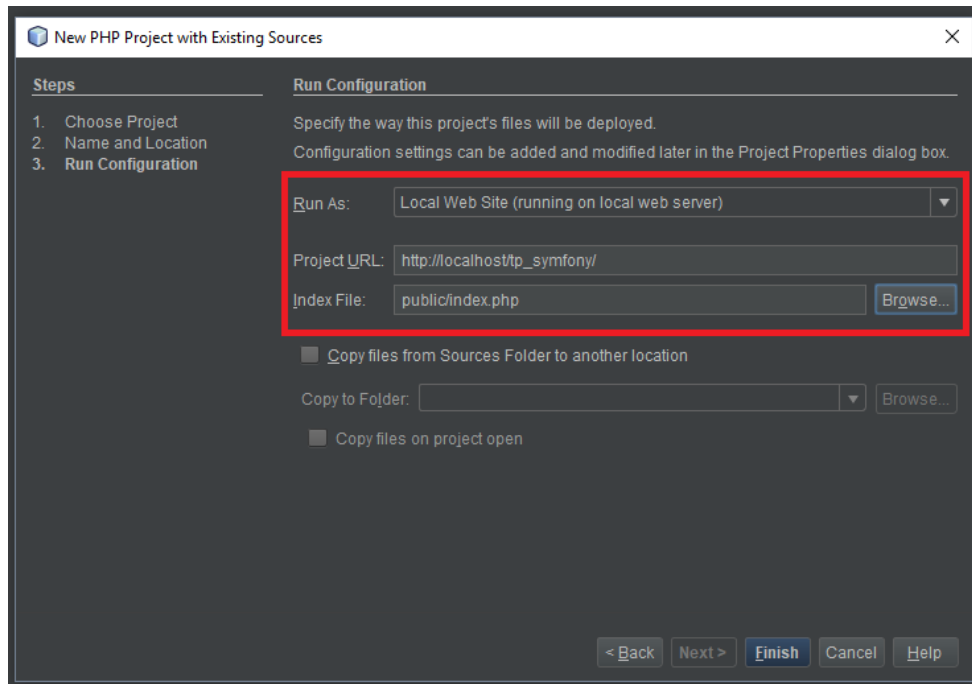
Your application is now ready. You can start working on it at:
D:\wamp\www\tp_symfony\

What's next?

Read the documentation to learn
[How to create your first page in Symfony](#)

You're seeing this page because debug mode is enabled and you haven't configured any homepage URL.

Eh oui ! Nous avons déjà configuré le lancement du projet depuis Wamp lorsque vous avez ouvert le projet sur votre IDE :



Création de votre première page

C'est ici que nous allons entrer dans le vif du sujet et que nous allons commencer à écrire du code.

Dans ce TP, nous créerons différents éléments :

- une page où nous pourrions créer des utilisateurs
- une page permettant de se connecter
- une page permettant d'afficher les informations de l'utilisateur connecté

Ça pourrait paraître bête est simple comme ça mais... oui, ça l'est. Symfony est là pour vous simplifier la vie, mais avant cela il faut passer un peu de temps dans sa documentation, ainsi que celle de Twig, de Composer, de Doctrine, etc... ;-) Mais après, tout sera vraiment plus simple pour vous. Faites-moi confiance.

Contrôleurs et routes

Pour créer notre premier contrôleur, il faut d'abord installer les packages (installez-les avec leur alias pour gagner du temps !) :

- `symfony/maker-bundle` : va nous aider à créer des contrôleurs, des entités, etc... en nous évitant d'écrire du code de manière répétitive.
- `sensio/framework-extra-bundle` : pour pouvoir travailler sur l'ensemble de notre projet avec des annotations.
- `symfony/profiler-pack` : il s'agit de l'outil qui affichera, en mode « dev », une barre d'outil en bas de la page qui nous servira notamment au niveau du débogage.
- `symfony/twig-bundle` : pour pouvoir utiliser le moteur de template traditionnellement associé à Symfony. De plus Twig protège vos variables contre les balises HTML malencontreuses.
- `symfony/asset` : permet de mieux gérer les chemins au sein de l'application

PHP est déjà un moteur de template non ?

Oui, mais dans le cas d'un framework MVC (Modèle Vue Contrôleur), il est préférable de ne pas mélanger du PHP avec du HTML. De plus nous verrons que Twig possède de nombreuses fonctionnalités qui nous seront d'une grande aide et nous permettront de gagner du temps.

Puis, dans votre console, saisissez :

```
D:\wamp\www\tp_symfony
λ php bin\console make:controller

Choose a name for your controller class (e.g. TinyPuppyController):
> |
```

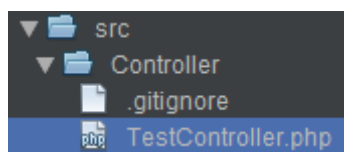
Puis nommez-le `TestController` :

```
Choose a name for your controller class (e.g. TinyPuppyController):
> TestController

created: src/Controller/TestController.php

Success!

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



Notre contrôleur a bien été créé !

En ouvrant votre nouveau contrôleur, vous pouvez voir que Symfony a déjà créé quelque chose sur la page. En fait, vous pouvez le tester par vous-même, la fonction `index()` va retourner un message lorsque vous allez ouvrir votre page à l'URL http://localhost/tp_symfony/public/index.php/test.

```
<?php

namespace App\Controller;

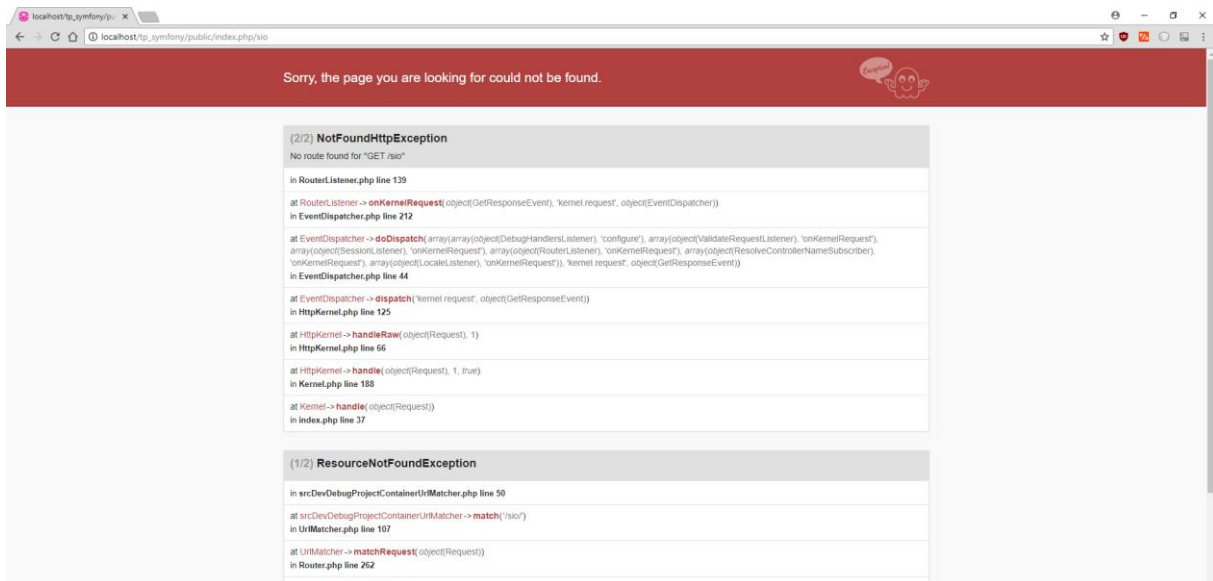
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class TestController extends Controller
{
    /**
     * @Route("/test", name="test")
     */
    public function index()
    {
        return $this->json([
            'message' => 'Welcome to your new controller!',
            'path' => 'src/Controller/TestController.php',
        ]);
    }
}
```

Pourquoi « /test » ?

C'est maintenant que nous allons parler d'annotations. Le bloc commenté au-dessus de la fonction comporte `@Route(...)`. Ceci est une annotation. Les annotations permettent de donner des informations sur les scripts sans modifier leur comportement propre. Ici par exemple l'annotation `@Route` permettra au kernel d'afficher la bonne page lorsque nous taperons `/test` dans la barre de recherche.

Si nous tapons quelque chose au hasard comme `/sio` :

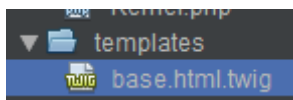


La route « sio » n'existe pas, le kernel ne peut donc pas afficher la page appropriée (qui n'existe pas non plus).

Nous allons donc créer une nouvelle fonction avec une nouvelle route et en utilisant Twig :

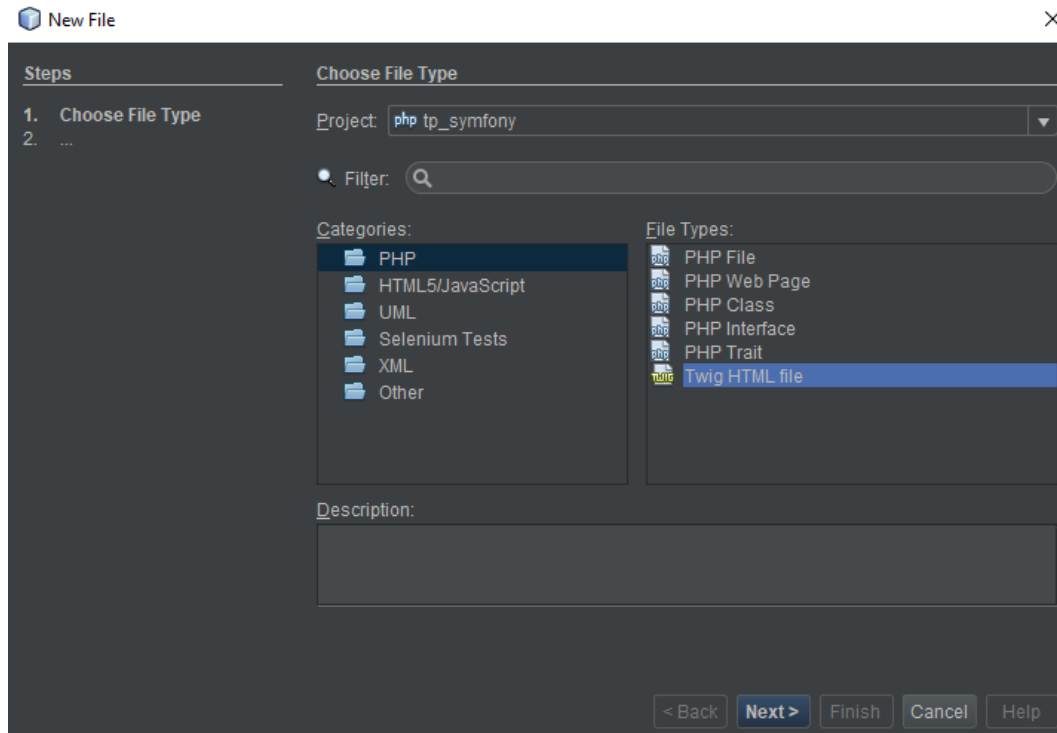
```
/**
 * @Route("/welcome", name="welcome")
 */
public function newWelcome()
{
    return $this->render('welcome.html.twig');
}
```

Ensuite rendons-nous dans le dossier `templates` du projet :



On voit qu'il y a déjà un fichier `.twig`. Si vous avez entendu parler d'héritage de classes dans les cours de Programmation Orientée Objet, ici nous allons aussi parler d'héritage de templates !

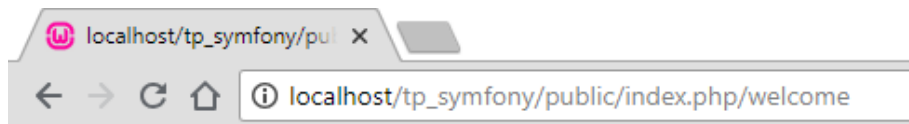
Je vais vous donner un exemple. Créez un fichier `welcome.html.twig` qui nous permettra d'afficher la page dont l'URL est `/welcome` :



Puis appelez ce fichier `welcome.html` (le « `.twig` » sera ajouté automatiquement à la création du fichier), et insérez-y le code suivant :

```
<h1>Hello World !</h1>
```

Maintenant lancez le projet, et ajoutez `/welcome` à la suite de l'URL :



Hello World !

Voilà votre première page créée pour ce projet ! Rien de bien impressionnant me direz-vous...

C'est maintenant que je vais vous présenter l'héritage de templates ! Modifier votre code :

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

{% block body %}
    <h1>Hello World !</h1>
{% endblock %}
```

Relancez votre projet. Vous ne voyez pas de changement ? Pourtant l'héritage a bien fonctionné ;-)

Maintenant faisons en sorte que vous puissiez observer des changements. Pour cela, rendez-vous sur le site de Bootstrap (<https://getbootstrap.com/>) et cliquez sur « Get started », et nous allons faire un petit copié-collé pour importer les CDN dans le `base.html.twig` :

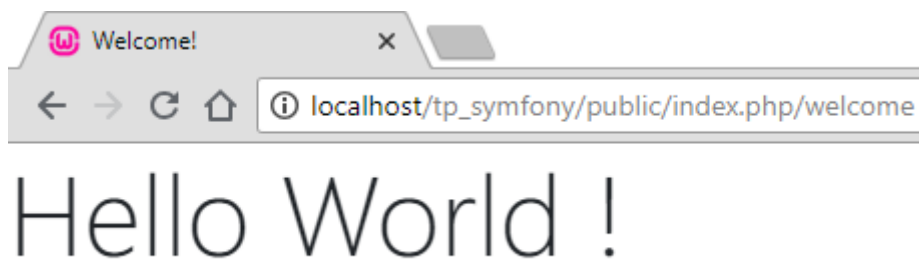
```
base.html.twig
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>{% block title %}Welcome!{% endblock %}</title>
{% block stylesheets %}
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css" integrity="sha384-q81/X+965Dz00rT7abK41JStQIAqVgRVzpbzo5"
{% endblock %}
</head>
<body>
{% block body %}{% endblock %}
{% block javascripts %}
<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="sha384-q81/X+965Dz00rT7abK41JStQIAqVgRVzpbzo5"
<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.3/umd/popper.min.js" integrity="sha384-ZMP7rVo3mIyKv"
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/js/bootstrap.min.js" integrity="sha384-ChfqqxuZUCnJSK3+
{% endblock %}
</body>
</html>
```

Et ajoutez la classe CSS `display-4` à votre texte :

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

{% block body %}
<h1 class="display-4">Hello World !</h1>
{% endblock %}
```

Relancez votre projet :



De plus, si vous inspectez le code de votre page dans le navigateur, vous verrez bien que les ressources sont bien importées...

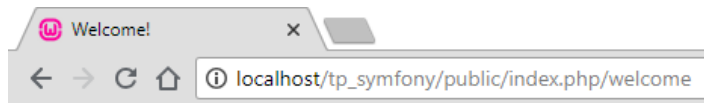
Autre subtilité de l'héritage de templates ! Dans le `block body` de votre `base.html.twig` :

```
{% block body %}
<h1 class="display-4">Hello</h1>
{% endblock %}
```

Et dans le `welcome.html.twig` :

```
{% block body %}
<h1 class="display-4">World !</h1>
{% endblock %}
```

Lancez le projet :



World !

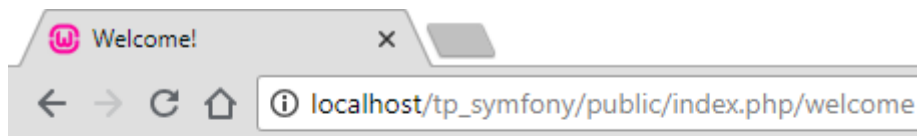


Je vais vous dire pourquoi vous n'avez pas le bon message...

En fait, lorsque vous écrivez dans un **block** Twig, vous écrasez le contenu du même bloc du fichier parent. Donc pour pouvoir afficher l'ensemble de la page, il suffit de rajouter la balise suivante à votre code (dans `welcome.html.twig`) :

```
{% block body %}
    {{ parent() }}
    <h1 class="display-4">World !</h1>
{% endblock %}
```

Résultat :



Hello World !

J'espère que maintenant vous comprenez mieux pourquoi Twig nous fera gagner du temps ! Si ce n'est pas le cas, je vous explique...

Dans le cas par exemple de votre portfolio, vous aurez sans doute une barre de navigation (navbar) dans laquelle vous aurez différents liens :

Accueil CV Formations ▾ Projets Veille Contact

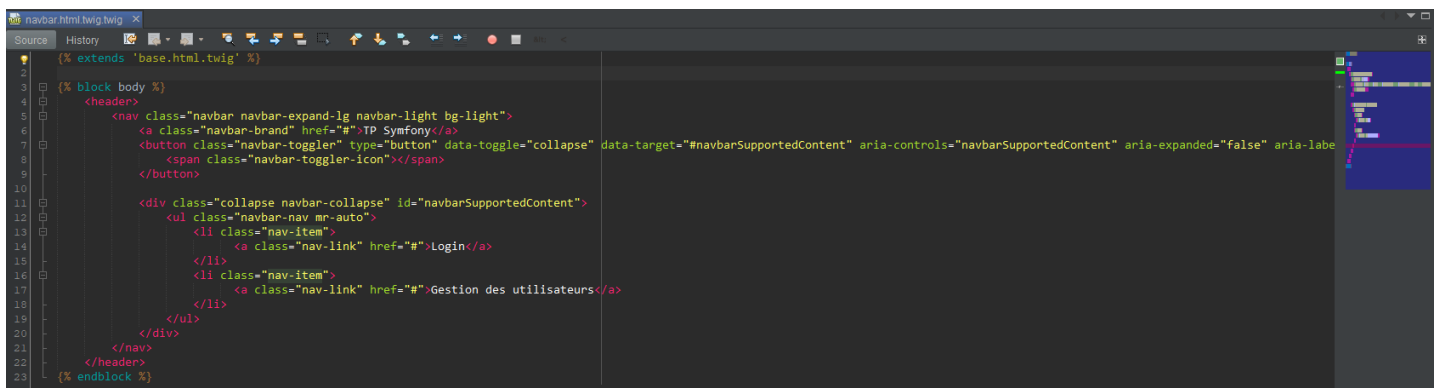
Rien que ça demande quelques lignes de code, et à chaque fois que vous aurez à modifier le code de votre navbar, vous le ferez sur tous vos fichiers HTML (ou PHP) ? Bien-sûr que non, l'idée est de factoriser le plus possible le code pour gagner du temps. Vous créez donc un fichier `navbar.html.twig` qui contiendra donc le code de votre navbar, et vous l'appellerez ensuite dans toutes vos pages ! Bilan : un seul fichier à modifier, pas d'oubli(s) → gain de temps.

Ajoutez le CDN de Font Awesome dans le `block stylesheets` de votre `base.html.twig`: <https://fontawesome.com/how-to-use/on-the-web/setup/getting-started?using=web-fonts-with-css>

Créons donc un fichier `navbar.html.twig` qui contiendra la barre de navigation dont nous aurons besoin (je vous fournis le code ce coup-ci) :

```
<header>
    <nav class="navbar navbar-expand-lg navbar-light bg-light">
        <a class="navbar-brand" href="#">TP Symfony</a>
        <button class="navbar-toggler" type="button" data-toggle="collapse"
data-target="#navbarSupportedContent" aria-controls="navbarSupportedContent" aria-
expanded="false" aria-label="Toggle navigation">
            <span class="navbar-toggler-icon"></span>
        </button>

        <div class="collapse navbar-collapse" id="navbarSupportedContent">
            <ul class="navbar-nav mr-auto">
                <li class="nav-item">
                    <a class="nav-link" href="#">Login</a>
                </li>
                <li class="nav-item">
                    <a class="nav-link" href="#">Gestion des
utilisateurs</a>
                </li>
            </ul>
        </div>
    </nav>
</header>
```

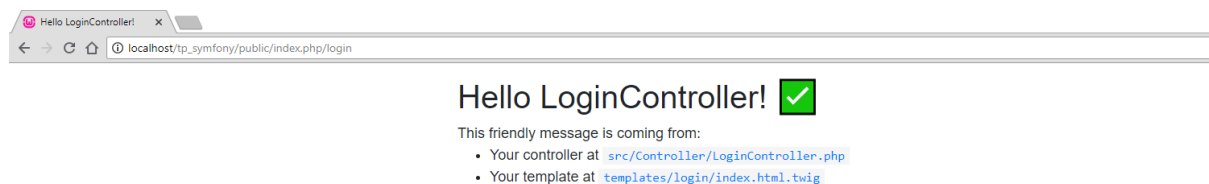


```
1  {% extends 'base.html.twig' %}
2
3  {% block body %}
4      <header>
5          <nav class="navbar navbar-expand-lg navbar-light bg-light">
6              <a class="navbar-brand" href="#">TP Symfony</a>
7              <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarSupportedContent" aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle navigation">
8                  <span class="navbar-toggler-icon"></span>
9              </button>
10
11              <div class="collapse navbar-collapse" id="navbarSupportedContent">
12                  <ul class="navbar-nav mr-auto">
13                      <li class="nav-item">
14                          <a class="nav-link" href="#">Login</a>
15                      </li>
16                      <li class="nav-item">
17                          <a class="nav-link" href="#">Gestion des utilisateurs</a>
18                      </li>
19                  </ul>
20              </div>
21          </nav>
22      </header>
23  {% endblock %}
```

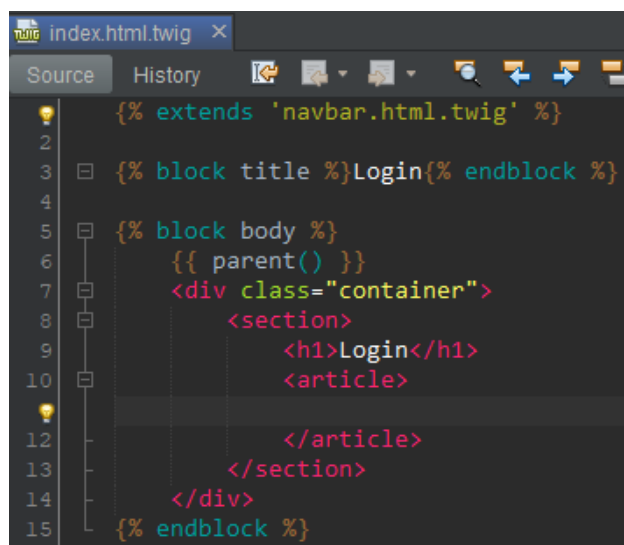

Les formulaires

Formulaire de login

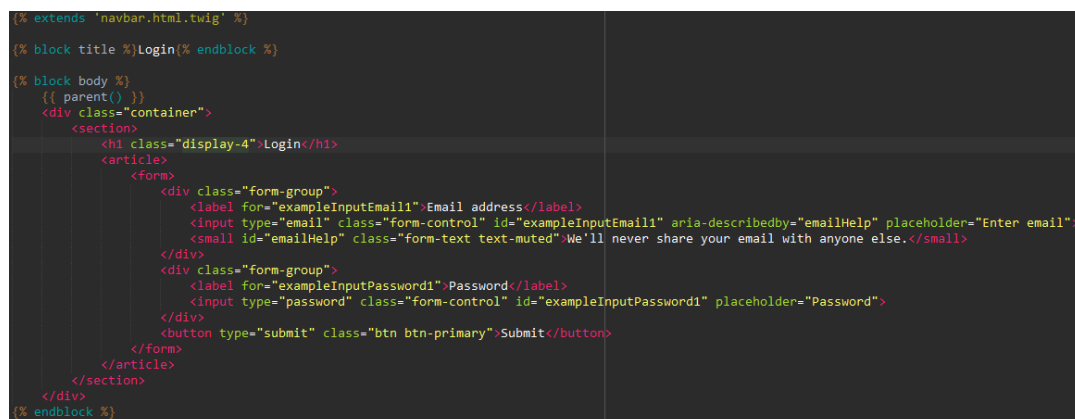
Maintenant vous allez créer un nouveau contrôleur que vous appellerez **LoginController**. Dans l'arborescence du projet on peut voir qu'un nouveau dossier a été créé dans le dossier **templates**. Il s'agit du dossier contenant la page liée au contrôleur créé :



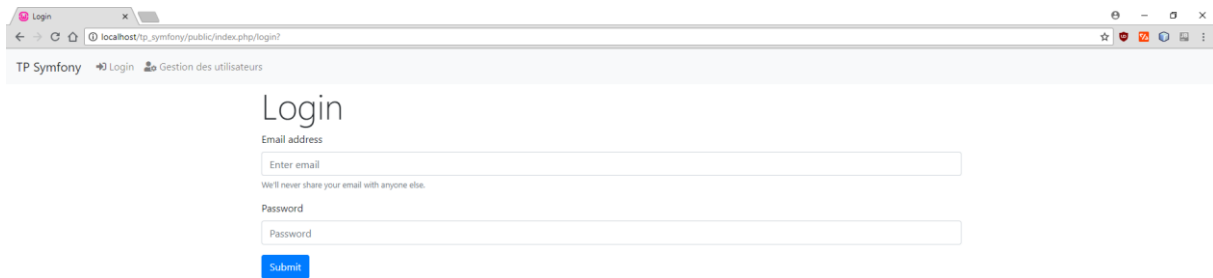
Et vous modifierez le code de la manière suivante dans le fichier **templates/login/index.html.twig**:



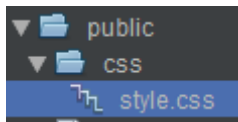
Et on ajoute le code du formulaire de login :



Résultat :



Nous pouvons améliorer un peu le rendu en rajoutant de nouvelles règles CSS dans un fichier à part. Il faut donc créer l'arborescence suivante dans le dossier `/public` : `resources/css/style.css`.



Et ajoutez-y le code suivant :

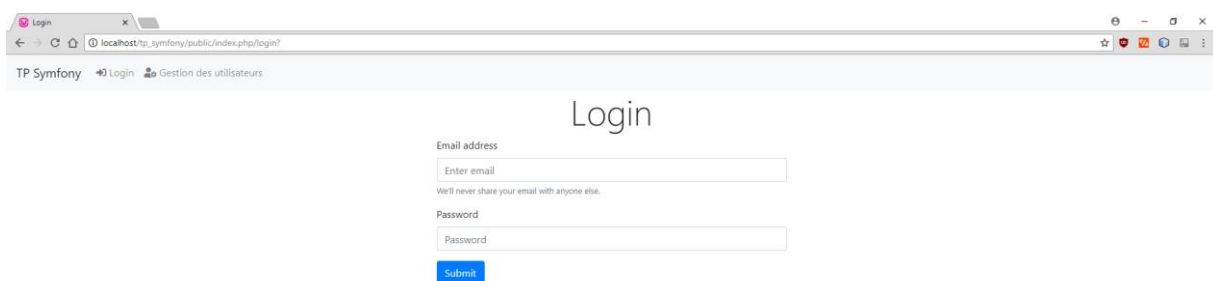
```
form {
    width: 50%;
    margin: auto;
}

.display-4 {
    text-align: center;
}
```

Et dans le fichier `base.html.twig`, en utilisant les assets :

```
{% block stylesheets %}
    <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.11.2/css/all.css">
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">
    <link rel="stylesheet" href="{{ asset('css/style.css') }}">
{% endblock %}
```

Résultat :



Nous voici donc avec un formulaire de login classique. Vous pensiez que les formulaires s'écrivaient de cette manière avec Symfony ? Eh bien non !

Il faut passer par un contrôleur et y implémenter les éléments que nous souhaitons retrouver sur notre page de la manière suivante :

```
$form = $this->createFormBuilder($task)
    ->add('task', TextType::class)
    ->add('dueDate', DateType::class)
    ->add('save', SubmitType::class, array('label' => 'Create Task'))
    ->getForm();
```

Pour créer des formulaires avec Symfony, il faut importer les packages suivants :

- `symfony/form` : permet de créer des formulaires.
- `symfony/validator` : valide les données saisies dans les formulaires avant qu'elles soient enregistrées dans une base de données.

Dans le cas de notre formulaire de login, je vous invite à créer une nouvelle fonction `createFormLogin()` dans le `LoginController`. Donc nous voulons afficher une zone de saisie d'adresse mail, une zone de saisie de mot de passe, et un bouton pour soumettre les informations :

```
/**
 * @Route("/login", name="login")
 */
public function index()
{
    $formLogin = $this->createFormLogin();
    return $this->render('login/index.html.twig', [
        'controller_name' => 'LoginController',
        'formLogin' => $formLogin->createView()
    ]);
}

private function createFormLogin()
{
    return $this->createFormBuilder()
        ->add('mailAddress', EmailType::class)
        ->add('password', PasswordType::class)
        ->add('submit', SubmitType::class)
        ->getForm();
}
```

Il s'agit d'une méthode privée car elle ne sera utilisable que dans ce fichier.

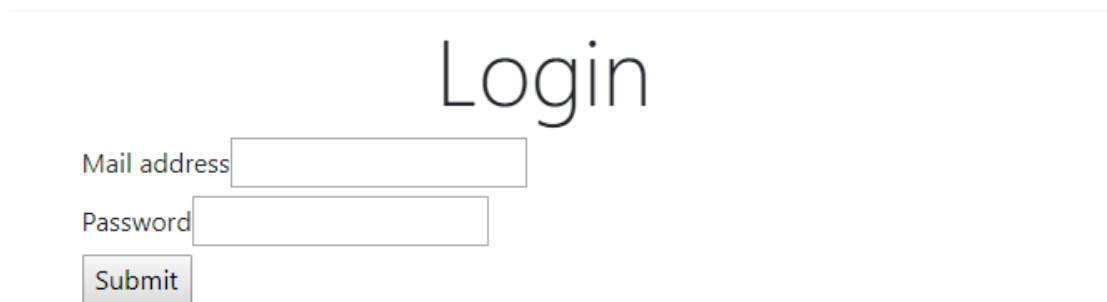
Dans le fichier `templates/login/index.html.twig` :

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

{% block title %}Login{% endblock %}

{% block body %}
    {{ parent() }}
    <div class="container">
        <section>
            <h1 class="display-4">Login</h1>
            <article>
                {{ form_start(formLogin) }}
                {{ form_end(formLogin) }}
            </article>
        </section>
    </div>
{% endblock %}
```

Résultat :



Non ce n'est pas très joli... Mais nous pouvons l'améliorer ! Il suffit de « décomposer » le formulaire dans le template, et de rajouter des paramètres à chacun des éléments :

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

{% block title %}Login{% endblock %}

{% block body %}
    {{ parent() }}
    <div class="container">
        <section>
            <h1 class="display-4">Login</h1>
            <article>
                {{ form_start(formLogin) }}
                <div class="form-group">
                    {{ form_widget(formLogin.mailAddress, { 'attr': { 'class': 'form-control' } }) }}
                </div>
                <div class="form-group">
                    {{ form_widget(formLogin.password, { 'attr': { 'class': 'form-control' } }) }}
                </div>
                {{ form_widget(formLogin.submit, { 'attr': { 'class': 'btn btn-primary' } }) }}
                {{ form_end(formLogin) }}
            </article>
        </section>
    </div>
{% endblock %}
```



Je n'ai rien inventé ! Tout cela sort tout droit de la documentation !

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

Résultat :

Login

C'est déjà plus propre. Mais il nous manque les labels...

Encore une fois la documentation est d'une grande aide :

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

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

{% block title %}Login{% endblock %}

{% block body %}
    {{ parent() }}
    <div class="container">
        <section>
            <h1 class="display-4">Login</h1>
            <article>
                {{ form_start(formLogin) }}
                <div class="form-group">
                    {{ form_label(formLogin.mailAddress, 'Email address') }}
                    {{ form_widget(formLogin.mailAddress, { 'attr': {'class': 'form-control'} }) }}
                    <small class="form-text text-muted">We'll never share your email with anyone else.</small>
                </div>
                <div class="form-group">
                    {{ form_label(formLogin.password, 'Password') }}
                    {{ form_widget(formLogin.password, { 'attr': {'class': 'form-control'} }) }}
                </div>
                {{ form_widget(formLogin.submit, { 'attr': {'class': 'btn btn-primary'} }) }}
                {{ form_end(formLogin) }}
            </article>
        </section>
    </div>
{% endblock %}
```

Résultat :

Login

Email address

We'll never share your email with anyone else.

Password

Submit

Vous venez donc de créer votre premier formulaire Symfony !

Maintenant que cette page est créée, il faut en compléter le lien dans la barre de navigation :

```
<li class="nav-item">
  <a class="nav-link" href="{{ path('login') }}"><i class="fas fa-sign-in-alt"></i> Login</a>
</li>
```

Le `path` contient le nom de la route correspondante à la page que nous souhaitons.

Formulaire de création d'utilisateurs

Il faut commencer par créer une nouvelle entité User. Mais avant cela, importez le package `symfony/orm-pack`. Les entités vont être plus tard mappées dans une base de données.

Pour créer l'entité :

```
D:\wamp\www\tp_symfony
λ php bin\console make:entity

Class name of the entity to create or update (e.g. DeliciousElephant):
> User
User

created: src/Entity/User.php
created: src/Repository/UserRepository.php

Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-running this command.

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

Ajouter les champs suivants :

- `name` (string, 255, not null)
- `firstname` (string, 255, not null)
- `mail` (string, 255, not null)
- `password` (string, 255, not null)

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

Field type (enter ? to see all types) [string]:
>

Field length [255]:
>

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

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

Lorsque vous avez terminé de saisir vos champs, tapez sur entrée :

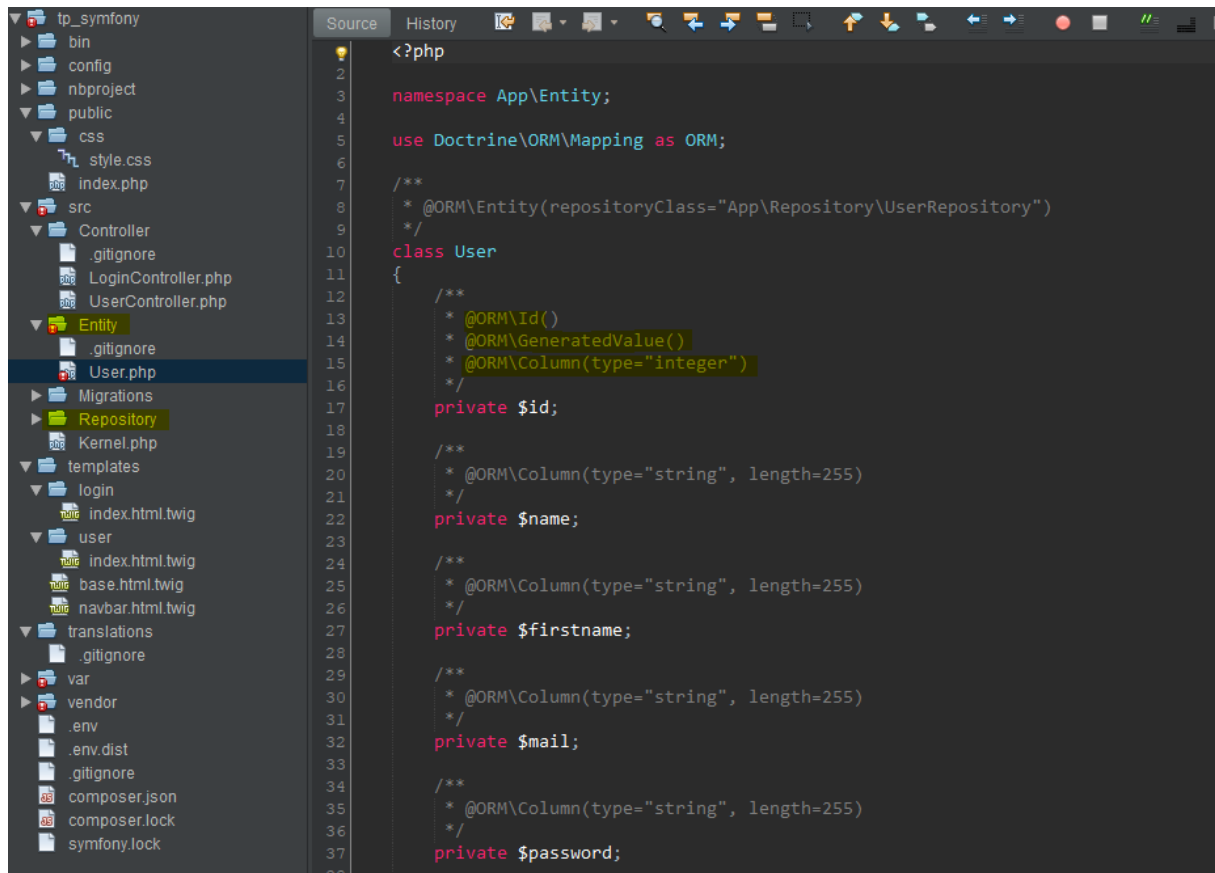
```
updated: src/Entity/User.php

Add another property? Enter the property name (or press <return> to stop adding fields):
>

Success!

Next: When you're ready, create a migration with make:migration
```

Si vous retournez dans NetBeans, vous verrez qu'un dossier **Entity** a été créé, ainsi qu'un dossier **Repository** qui va permettre de récupérer les utilisateurs en base de données :



```

<?php
namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

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

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

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

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

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

```

Tout à l'heure dans la console nous avons pu observer le message suivant :

```
Next: When you're ready, create a migration with make:migration
```

La commande en jaune est celle qui va nous permettre de mapper les entités créées en base de données. Mais pas de panique, celle-ci n'existe pas encore...

Pour créer la base de données, il faut configurer le fichier `.env` qui se trouve à la racine de notre projet :

```

###> doctrine/doctrine-bundle ###
# Format described at http://docs.doctrine-project.org/projects/doctrine-dbal/en
# For an SQLite database, use: "sqlite:///kernel.project_dir%/var/data.db"
# Configure your db driver and server_version in config/packages/doctrine.yaml
DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/db_name
###< doctrine/doctrine-bundle ###

```

Qu'il faut modifier avec les informations souhaitées :

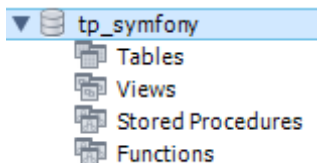
```
DATABASE_URL=mysql://root:@127.0.0.1:3306/tp_symfony
```

Ensuite lancez la commande :

```
D:\wamp\www\tp_symfony
λ php bin\console doctrine:database:create
Created database `tp_symfony` for connection named default
```

Si le message précédent n'a pas été obtenu, c'est que vous avez dû faire une erreur dans la configuration de la `DATABASE_URL`.

Si tout s'est bien passé, ouvrez MySQL Workbench :



Le schéma a bien été créé !

Nous pouvons maintenant mapper notre entité :

```
D:\wamp\www\tp_symfony
λ php bin\console make:migration
```

Success!

Next: Review the new migration "`src/Migrations/Version20180807102125.php`"
Then: Run the migration with `php bin/console doctrine:migrations:migrate`
See <https://symfony.com/doc/current/bundles/DoctrineMigrationsBundle/index.html>

Puis :

```
D:\wamp\www\tp_symfony
λ php bin\console doctrine:migrations:migrate
```

Application Migrations

WARNING! You are about to execute a database migration that could result in schema changes and data loss. Are you sure you wish to continue? (y/n)y

Migrating up to 20180807102125 from 0

++ migrating 20180807102125

--> CREATE TABLE user (id INT AUTO_INCREMENT NOT NULL, name VARCHAR(255) NOT NULL, firstname VARCHAR(255) NOT NULL, mail VARCHAR(255) NOT NULL, password VARCHAR(255) NOT NULL, PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci ENGINE = InnoDB

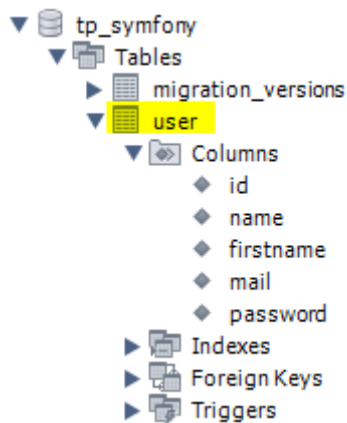
++ migrated (0.42s)

++ finished in 0.42s

++ 1 migrations executed

++ 1 sql queries

Retournons dans MySQL Workbench :



Entité mappée !

Maintenant nous allons pouvoir créer notre contrôleur `RegistrationController` :

```
D:\wamp\www\tp_symfony
λ php bin\console make:controller

Choose a name for your controller class (e.g. OrangeJellybeanController):
> RegistrationController

created: src/Controller/RegistrationController.php
created: templates/registration/index.html.twig

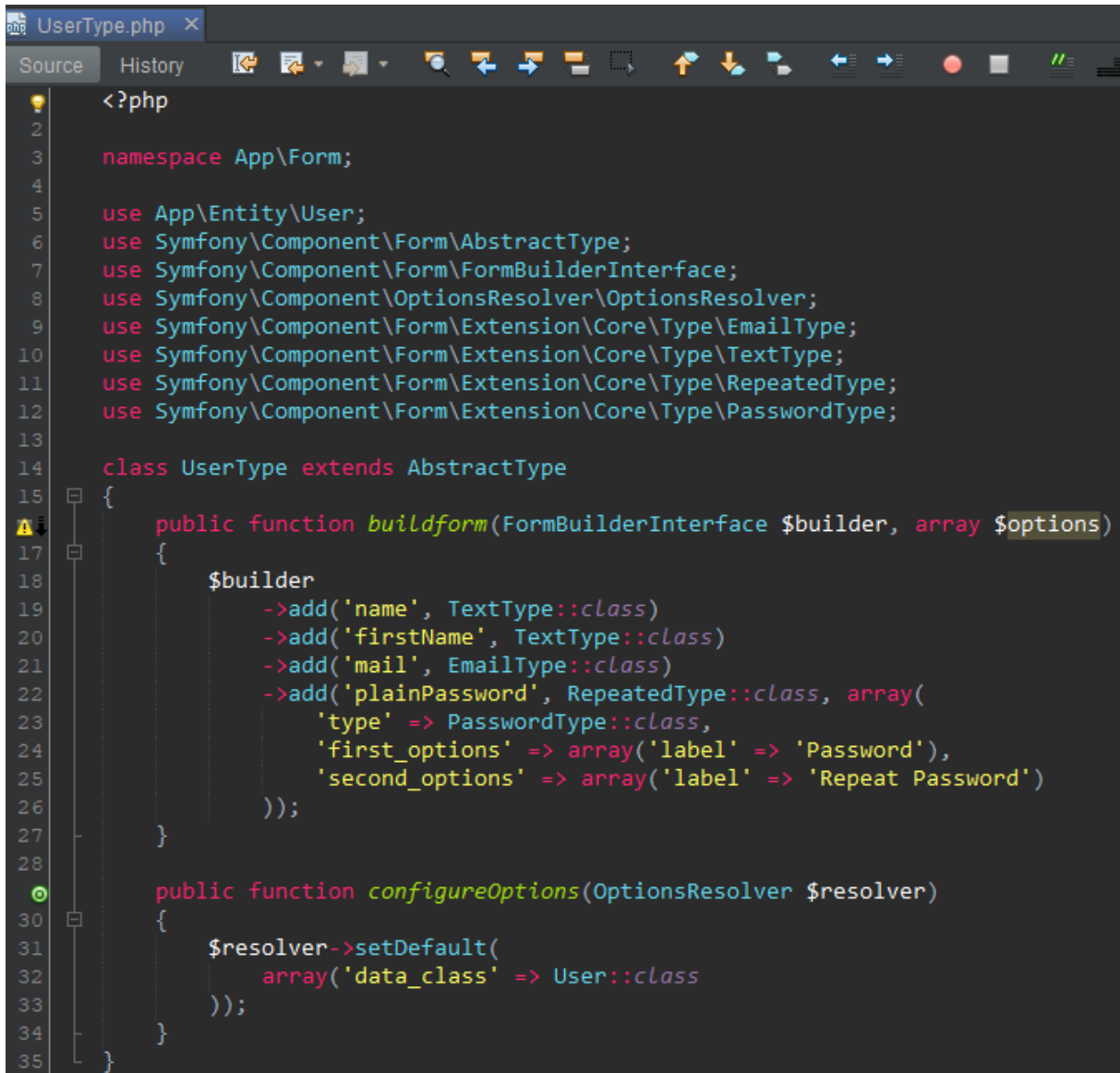
Success!

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

Ce contrôleur va implémenter le nécessaire pour enregistrer un utilisateur. Il va donc falloir créer un formulaire d'enregistrement. Mais cette fois nous n'allons pas implémenter la méthode de création de formulaire dans ce contrôleur. Eh oui ! Je vais vous montrer la deuxième manière de créer un formulaire. J'ai procédé comme ceci pour vous montrer qu'il existe plusieurs façons de créer de faire appel à des formulaires dans des contrôleurs, après il s'agira plus d'un choix technique pour vous. Si vous avez beaucoup de code dans votre contrôleur, il est préférable de gérer votre formulaire dans un fichier à part.

Créez d'abord un dossier `Form` dans le dossier `src`.

Puis créez une nouvelle classe PHP à l'intérieur de ce dossier que vous nommerez **UserType** :



```
<?php
2
3 namespace App\Form;
4
5 use App\Entity\User;
6 use Symfony\Component\Form\AbstractType;
7 use Symfony\Component\Form\FormBuilderInterface;
8 use Symfony\Component\OptionsResolver\OptionsResolver;
9 use Symfony\Component\Form\Extension\Core\Type\EmailType;
10 use Symfony\Component\Form\Extension\Core\Type\TextType;
11 use Symfony\Component\Form\Extension\Core\Type\RepeatedType;
12 use Symfony\Component\Form\Extension\Core\Type>PasswordType;
13
14 class UserType extends AbstractType
15 {
16     public function buildform(FormBuilderInterface $builder, array $options)
17     {
18         $builder
19             ->add('name', TextType::class)
20             ->add('firstName', TextType::class)
21             ->add('mail', EmailType::class)
22             ->add('plainPassword', RepeatedType::class, array(
23                 'type' => PasswordType::class,
24                 'first_options' => array('label' => 'Password'),
25                 'second_options' => array('label' => 'Repeat Password')
26             ));
27     }
28
29     public function configureOptions(OptionsResolver $resolver)
30     {
31         $resolver->setDefault(
32             array('data_class' => User::class)
33         );
34     }
35 }
```

Créez ensuite le contrôleur **RegistrationController** :

```

RegistrationController.php
Source History
<?php
2
3 namespace App\Controller;
4
5 use App\Form\UserType;
6 use App\Entity\User;
7 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
8 use Symfony\Component\HttpFoundation\Request;
9 use Symfony\Component\Routing\Annotation\Route;
10
11 class RegistrationController extends Controller
12 {
13     /**
14      * @Route("/register", name="user_registration")
15      */
16     public function register(Request $request)
17     {
18         // 1) build the form
19         $user = new User();
20         $form = $this->createForm(UserType::class, $user);
21
22         // 2) handle the submit (will only happen on POST)
23         $form->handleRequest($request);
24         if ($form->isSubmitted() && $form->isValid()) {
25
26             // 3) save the User!
27             $entityManager = $this->getDoctrine()->getManager();
28             $entityManager->persist($user);
29             $entityManager->flush();
30
31             return $this->render(
32                 'registration/register.html.twig',
33                 array('form' => $form->createView())
34             );
35         }
36         return $this->render(
37             'registration/register.html.twig',
38             array('form' => $form->createView())
39         );
40     }
41 }

```

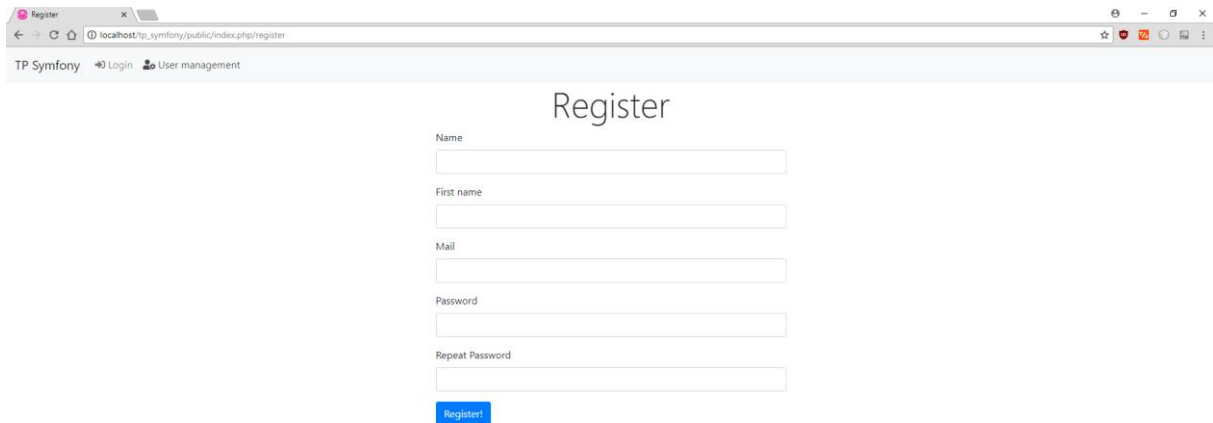
Puis dans **templates/registration/register.html.twig** :

```

register.html.twig
Source History
{% extends 'navbar.html.twig' %}
2
3 {% block title %}Register{% endblock %}
4
5 {% block body %}
6     {{ parent() }}
7     <div class="container">
8         <section>
9             <h1 class="display-4">Register</h1>
10            <article>
11                {{ form_start(form) }}
12                <div class="form-group">
13                    {{ form_row(form.name, { 'attr': { 'class': 'form-control' } }) }}
14                </div>
15                <div class="form-group">
16                    {{ form_row(form.firstName, { 'attr': { 'class': 'form-control' } }) }}
17                </div>
18                <div class="form-group">
19                    {{ form_row(form.mail, { 'attr': { 'class': 'form-control' } }) }}
20                </div>
21                <div class="form-group">
22                    {{ form_row(form.password.first, { 'attr': { 'class': 'form-control' } }) }}
23                </div>
24                <div class="form-group">
25                    {{ form_row(form.password.second, { 'attr': { 'class': 'form-control' } }) }}
26                </div>
27                <button class="btn btn-primary" type="submit">Register!</button>
28                {{ form_end(form) }}
29            </article>
30        </section>
31    </div>
32 {% endblock %}

```

Résultat :



Testons maintenant l'enregistrement d'un utilisateur :

| id | name | firstname | mail | password |
|----|-------|-----------|-----------------------|----------|
| 1 | Laure | Thomas | thomaslaure@gmail.com | password |



Le mot de passe est stocké en clair dans la base de données car je n'ai pas utilisé ici de moyen pour le chiffrer. De plus, le formulaire de login et le formulaire d'enregistrement ne comportent aucune sécurité. Il faut pour cela installer un certains package et configurer le fichier `config/packages/security.yaml`. Le but de ce TP est de se familiariser avec l'environnement de Symfony en manipulant les entités.

Pour plus d'informations sur la sécurité dans Symfony :

https://symfony.com/doc/current/security/entity_provider.html

https://symfony.com/doc/current/doctrine/registration_form.html

http://symfony.com/doc/current/security/form_login_setup.html

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

Page d'affichage des informations de l'utilisateur

Maintenant que nous avons notre formulaire de login et notre formulaire d'enregistrement, il nous faudrait une page où afficher les informations de l'utilisateur qui vient de s'identifier.

Dans le contrôleur `LoginController`, créez la méthode `fetchUser()` :

```
private function fetchUser(string $mail, string $password)
{
    $repository = $this->getDoctrine()->getManager()->getRepository(User::class);
    $result = $repository->findOneBy(
        array(
            'mail' => $mail,
            'password' => $password
        )
    );
    return $result;
}
```

Cette méthode va permettre de récupérer l'enregistrement en base de données pour lequel l'adresse mail et le mot de passe correspondent. Cette méthode va donc renvoyer une instance de la classe `User`.

Ensuite, modifiez la méthode `index()` du même contrôleur :

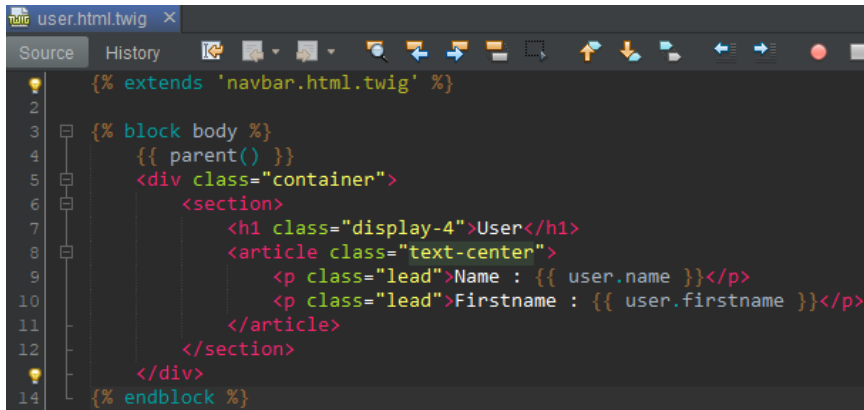
```
/**
 * @Route("/login", name="login")
 */
public function index(Request $request)
{
    $formLogin = $this->createFormLogin();
    $formLogin->handleRequest($request);
    if ($formLogin->isSubmitted() && $formLogin->isValid()) {
        $mail = $formLogin['mailAddress']->getData();
        $password = $formLogin['password']->getData();
        $user = $this->fetchUser($mail, $password);
        return $this->render('user.html.twig',
            array('user' => $user)
        );
    }
    return $this->render('login/index.html.twig',
        array('formLogin' => $formLogin->createView())
    );
}
```

Avec cette ligne : `$mail = $formLogin['mailAddress']->getData();` on va récupérer l'adresse mail saisie dans le formulaire (après sa validation).

Ensuite on retourne l'objet en plus de la page souhaitée pour afficher les données :

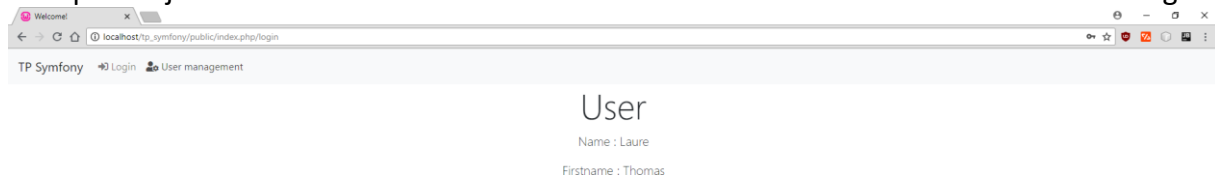
```
return $this->render('user.html.twig',
    array('user' => $user)
);
```


Créez le fichier `templates/user.html.twig` :



```
1 {% extends 'navbar.html.twig' %}
2
3 {% block body %}
4     {{ parent() }}
5     <div class="container">
6         <section>
7             <h1 class="display-4">User</h1>
8             <article class="text-center">
9                 <p class="lead">Name : {{ user.name }}</p>
10                <p class="lead">Firstname : {{ user.firstname }}</p>
11            </article>
12        </section>
13    </div>
14 {% endblock %}
```

Lorsque je saisis les informations dans le formulaire de login :



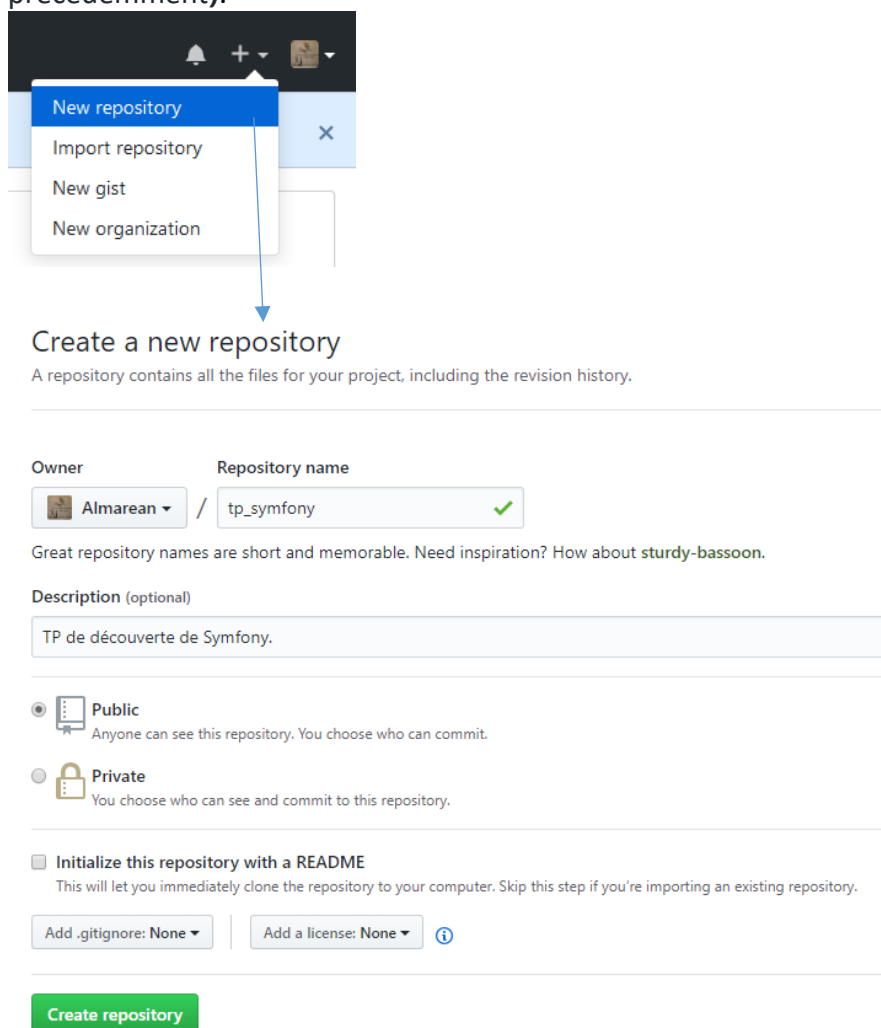
Deuxième partie

Maintenant que vous avez compris comment fonctionne Twig et la manipulation d'entités avec Symfony 4, nous pouvons aller plus loin en mettant en place un formulaire d'enregistrement ainsi qu'un formulaire de login sécurisés.

A partir de maintenant, nous allons utiliser le versionning avec Git.

Dans Cmder, tapez :

1. `echo « # nom_du_projet » >> README.md` : Crée le `README.md` du projet, c'est le fichier destiné à recevoir la description du projet.
2. `git init` : initialiser le dépôt Git local du projet, un dossier caché `.git` est créé.
3. `git add .` : ajoute tous les éléments modifiés (ici il s'agit de tous les éléments sauf ceux présents dans le `.gitignore` ainsi que les dépendances) du projet à l'index.
4. `git commit -m « Mon premier commit. »` : on fait un commit en ajoutant directement un message
5. (facultatif) Créer le répertoire distant qui permettra de stocker le fichier (sur GitHub, GitLab, etc...). La création se fait à la main sur le site souhaité (sans cocher la case **Initialize this repository with a README**, vu que nous l'avons déjà créé précédemment).



Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: Almarean / Repository name: tp_symfony

Great repository names are short and memorable. Need inspiration? How about sturdy-bassoon.

Description (optional): TP de découverte de Symfony.

☒ Public
Anyone can see this repository. You choose who can commit.

☐ Private
You choose who can see and commit to this repository.

☐ Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

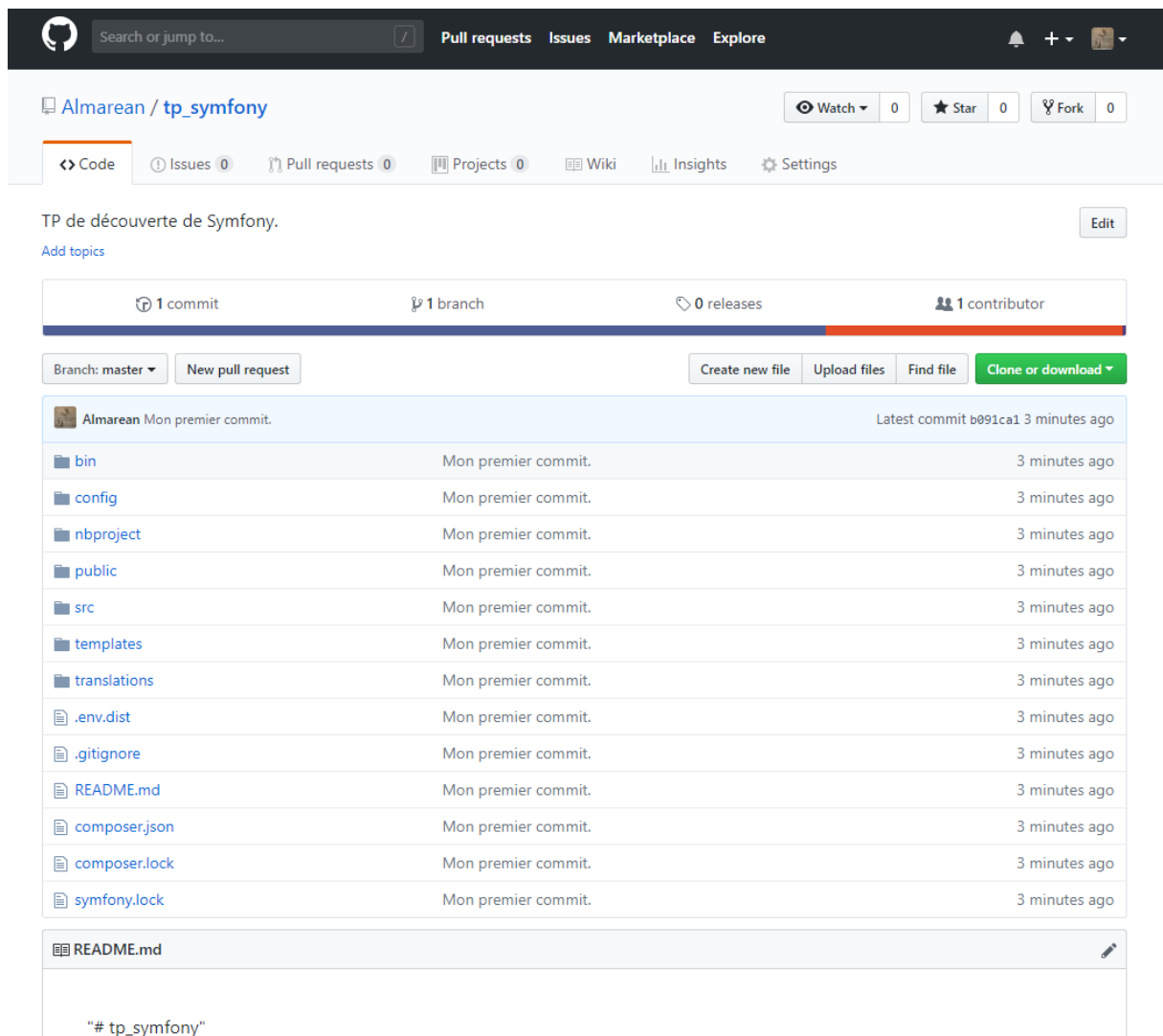
Add .gitignore: None Add a license: None

Create repository

6. Uniquement si vous avez un dépôt distant : `git remote add origin https://github.com/\[votre nom\]/tp_symfony.git`
7. Uniquement si vous avez un dépôt distant : `git push -u origin master`

```
D:\wamp\www\tp_symfony (master -> origin)
λ git remote add origin https://github.com/Almarean/tp_symfony.git

D:\wamp\www\tp_symfony (master -> origin)
λ git push -u origin master
Counting objects: 71, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (62/62), done.
Writing objects: 100% (71/71), 32.29 KiB | 2.48 MiB/s, done.
Total 71 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), done.
To https://github.com/Almarean/tp_symfony.git
* [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```



The screenshot shows the GitHub repository page for **Almarean / tp_symfony**. The repository has 1 commit, 1 branch, 0 releases, and 1 contributor. The main branch is **master**. The repository contains the following files and folders:

| File/Folder | Commit Message | Time |
|---------------|---------------------|---------------|
| bin | Mon premier commit. | 3 minutes ago |
| config | Mon premier commit. | 3 minutes ago |
| nbproject | Mon premier commit. | 3 minutes ago |
| public | Mon premier commit. | 3 minutes ago |
| src | Mon premier commit. | 3 minutes ago |
| templates | Mon premier commit. | 3 minutes ago |
| translations | Mon premier commit. | 3 minutes ago |
| .env.dist | Mon premier commit. | 3 minutes ago |
| .gitignore | Mon premier commit. | 3 minutes ago |
| README.md | Mon premier commit. | 3 minutes ago |
| composer.json | Mon premier commit. | 3 minutes ago |
| composer.lock | Mon premier commit. | 3 minutes ago |
| symfony.lock | Mon premier commit. | 3 minutes ago |

The **README.md** file content is as follows:

```
# tp_symfony
```

Sur GitHub, vous pouvez voir que le dossier `/vendor` n'a pas été importé dans le dépôt. Si vous le souhaitez, vous pouvez supprimer le projet de votre ordinateur, et à la racine du serveur (le dossier `www`) lancer la commande `git clone https://github.com/[votre nom]/tp_symfony.git`. Une fois le projet téléchargé, il n'est pas utilisable car ses dépendances n'ont pas été téléchargées. Pour cela, déplacez-vous à la racine du projet et lancez la commande `composer update`. A la suite de cela, vous pourrez voir que le dossier `/vendor` est créé, qu'il contient toutes vos dépendances et que le projet est de nouveau fonctionnel !

J'ai choisi de travailler avec GitHub parce que c'était une habitude, mais ça fonctionne **EXACTEMENT** de la même manière sur GitLab. Vous pouvez même importer vos dépôts GitHub sur GitLab gratuitement, et les dépôts privés sont gratuits nativement sur GitLab.

Formulaire d'enregistrement sécurisé

Premièrement, il faut installer le package `symfony/security-bundle`. Il s'agit du package qui nous permettra de gérer la sécurité dans Symfony, que ce soit au niveau des formulaires ou du chargement des entités depuis la base de données. C'est ce bundle qui nous permettra aussi d'implémenter l'interface `UserInterface` à notre entité `User`.

Rappel : Une interface définit le comportement d'une classe. Tous les éléments d'une interface doivent être implémentés dans la classe qui y fait appel.

Ici cette interface possède les méthodes suivantes :

- `getRoles()` : retourne les privilèges de l'utilisateur (ROLE_USER, ROLE_ADMIN, etc...)
- `getPassword()` : retourne le mot de passe utilisé pour authentifier l'utilisateur
- `getSalt()` : retourne le sel (oui oui, littéralement) utilisé pour chiffrer le mot de passe
- `getUsername()` : retourne l'identifiant de l'utilisateur
- `eraseCredentials()` : supprime les données sensibles de l'utilisateur

Lien vers la documentation de `UserInterface` :

<https://api.symfony.com/3.4/Symfony/Component/Security/Core/User/UserInterface.html>

Nous n'allons pas toutes les utiliser dans ce TP, mais nous allons tout de même les ajouter à notre code.



Le bundle précédemment installé fournit également une classe `User` prédéfinie, mais celle-ci contient des propriétés qui ne nous seront pas utiles dans notre cas, et il y a des propriétés dont nous avons besoin et qui ne sont pas présentes. Il est donc préférable de créer notre propre entité et de configurer ensuite le tout pour avoir une solution plus sur-mesure.

Lien vers la documentation de `User` :

<https://api.symfony.com/3.4/Symfony/Component/Security/Core/User/User.html>

Modifions notre entité **User** :

```
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;
use Symfony\Component\Security\Core\User\UserInterface;

/**
 * @ORM\Entity(repositoryClass="App\Repository\UserRepository")
 */
class User implements UserInterface
{
```

```
    /**
     * @Assert\NotBlank()
     * @Assert\Length(max=4096)
     */
    private $plainPassword;

    public function getPlainPassword(): ?string
    {
        return $this->plainPassword;
    }

    public function setPlainPassword(string $plainPassword): void
    {
        $this->plainPassword = $plainPassword;
    }

    public function getUsername(): string
    {
        return $this->mail;
    }

    public function getSalt()
    {
        return null;
    }

    public function getRoles(): array
    {
        return array('ROLE_USER');
    }

    public function eraseCredentials()
    {
    }
}
```



La propriété `$plainPassword` n'est pas à persister en base de données. Il s'agit d'une propriété qui accueillera le mot de passe en clair de l'utilisateur pour pouvoir ensuite le chiffrer et le stocker dans `$password`.

Et nous allons utiliser l'adresse mail en tant que « username ».

```
/**
 * @see \Serializable::serialize()
 */
public function serialize()
{
    return serialize(array(
        $this->id,
        $this->mail,
        $this->password
    ));
}

/**
 * @see \Serializable::unserialize()
 *
 * @param $serialized
 */
public function unserialize($serialized)
{
    list (
        $this->id,
        $this->mail,
        $this->password
    ) = unserialize($serialized, array('allowed_classes' => false));
}
```

Understanding serialize and how a User is Saved in the Session ¶

If you're curious about the importance of the `serialize()` method inside the `User` class or how the User object is serialized or deserialized, then this section is for you. If not, feel free to skip this.

Once the user is logged in, the entire User object is serialized into the session. On the next request, the User object is deserialized. Then, the value of the `id` property is used to re-query for a fresh User object from the database. Finally, the fresh User object is compared to the deserialized User object to make sure that they represent the same user. For example, if the `username` on the 2 User objects doesn't match for some reason, then the user will be logged out for security reasons.

Even though this all happens automatically, there are a few important side-effects.

First, the `Serializable` interface and its `serialize()` and `unserialize()` methods have been added to allow the `User` class to be serialized to the session. This may or may not be needed depending on your setup, but it's probably a good idea. In theory, only the `id` needs to be serialized, because the `refreshUser()` method refreshes the user on each request by using the `id` (as explained above). This gives us a "fresh" User object.

But Symfony also uses the `username`, `salt`, and `password` to verify that the User has not changed between requests (it also calls your `AdvancedUserInterface` methods if you implement it). Failing to serialize these may cause you to be logged out on each request. If your user implements the `EquatableInterface`, then instead of these properties being checked, your `isEqualTo()` method is called, and you can check whatever properties you want. Unless you understand this, you probably *won't* need to implement this interface or worry about it.

Il faut maintenant configurer le fichier `/config/packages/security.yml`. Vous n'aimez pas les fichiers de configuration ? Moi non plus, mais il faut bien y passer un jour ou l'autre pour faire le travail correctement ;-). Il y a des environnements de développement qui demandent beaucoup de configuration dans des fichiers de ce type, nous avons de la chance Symfony n'en fait pas partie !

```
security:
    encoders:
        App\Entity\User:
            algorithm: bcrypt
    providers:
        #in_memory: { memory: ~ }
        tp_provider:
            entity:
                class: App\Entity\User
                property: mail
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
    main:
        anonymous: true
        pattern: ^/
        http_basic: ~
        provider: tp_provider
```

On définit l'algorithme de chiffrement pour le mot de passe et par quelle entité il va être utilisé.

Le provider (que j'ai appelé `tp_provider`), ou fournisseur, va dire au pare-feu (celui de Symfony) qui est le fournisseur d'entités. Le pare-feu va donc aller chercher les utilisateurs dans la base de données. Et l'identifiant sera l'adresse mail.

Il s'agit du pare-feu principal de l'application. Le champ :

- `anonymous` signifie qu'on peut y accéder sans s'identifier
- le `pattern` est un masque qui définit sur quelle URL agit le pare-feu (ici il s'agit de la racine donc le pare-feu agit sur tout le site)
- et ensuite on définit le fournisseur (`provider`) du pare-feu

Maintenant nous allons apporter la légère modification à notre formulaire d'enregistrement dans le fichier `/src/Form/UserType.php`. Remplacez :

```
->add('password', RepeatedType::class, array(
```

Par

```
->add('plainPassword', RepeatedType::class, array(
```

Puis modifier le contrôleur `RegistrationController` en ajoutant le « use » qui va bien :

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

class RegistrationController extends Controller
{
    /**
     * @Route("/register", name="user_registration")
     */
    public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder)
    {
        // 1) build the form
        $user = new User();
        $form = $this->createForm(UserType::class, $user);

        // 2) handle the submit (will only happen on POST)
        $form->handleRequest($request);
        if ($form->isSubmitted() && $form->isValid()) {

            // 3) encode the password
            $password = $passwordEncoder->encodePassword($user, $user->getPlainPassword());
            $user->setPassword($password);

            // 4) save the User!
            $entityManager = $this->getDoctrine()->getManager();
            $entityManager->persist($user);
            $entityManager->flush();

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

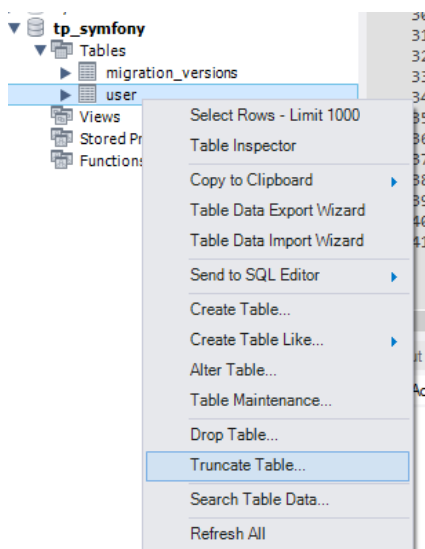
Nous avons rajouté le chiffrement du mot de passe !

Il faut aussi modifier notre vue par
 /templates/registration/register.html.twig (remplacer
 plainPassword):

```

1  register.html.twig
2  {% extends 'navbar.html.twig' %}
3  {% block title %}Register{% endblock %}
4
5  {% block body %}
6      {{ parent() }}
7      <div class="container">
8          <section>
9              <h1 class="display-4">Register</h1>
10             <article>
11                 {{ form_start(form) }}
12                 <div class="form-group">
13                     {{ form_row(form.name, { 'attr': { 'class': 'form-control' } }) }}
14                 </div>
15                 <div class="form-group">
16                     {{ form_row(form.firstName, { 'attr': { 'class': 'form-control' } }) }}
17                 </div>
18                 <div class="form-group">
19                     {{ form_row(form.mail, { 'attr': { 'class': 'form-control' } }) }}
20                 </div>
21                 <div class="form-group">
22                     {{ form_row(form.plainPassword.first, { 'attr': { 'class': 'form-control' } }) }}
23                 </div>
24                 <div class="form-group">
25                     {{ form_row(form.plainPassword.second, { 'attr': { 'class': 'form-control' } }) }}
26                 </div>
27                 <button class="btn btn-primary" type="submit">Register!</button>
28                 {{ form_end(form) }}
29             </article>
30         </section>
31     </div>
32 {% endblock %}
  
```

Maintenant, réinitialisez la table user en base de données en faisant un clic droit sur celle-ci puis « Truncate table... » :



Lorsque vous souhaitez réinitialiser le contenu d'une table, il est préférable d'utiliser l'instruction `TRUNCATE TABLE [nom_table]` plutôt que `DELETE FROM [nom_table]`, car la première réinitialise également les id tandis que la deuxième ne fait que supprimer les données.

Admettons que nous ayons une table contenant dix enregistrements, les id vont de 1 à 10. Lorsque que vous allez faire persister une nouvelle entité :

- avec l'instruction `DELETE FROM [nom_table]`, l'id de cette nouvelle entité sera 11
- avec l'instruction `TRUNCATE TABLE [nom_table]`, l'id sera 1.

Retournez dans votre navigateur à l'adresse http://localhost/tp_symfony/public/index.php/register

Et enregistrez un nouvel utilisateur :

12 • `select * from `user`;`

| id | name | firstname | mail | password |
|------|-------|-----------|-----------------------|---|
| 1 | Laure | Thomas | thomaslaure@gmail.com | \$2v\$13\$WqLx8/OOmhx0RY2KNz0O5OXiIEFPa.... |
| NULL | NULL | NULL | NULL | NULL |

Le chiffrement du mot de passe a bien fonctionné.

Faisons donc un petit commit pour entériner tout ça :

```
D:\wamp\www\tp_symfony (master -> origin)
λ git add . login correspond a la route du formulaire de connexion. En
warning: LF will be replaced by CRLF in composer.json.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in config/packages/security.yaml.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in src/Controller/RegistrationController.php.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in src/Entity/User.php.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in src/Form/UserType.php.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in symfony.lock.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in templates/registration/register.html.twig.
The file will have its original line endings in your working directory.
D:\wamp\www\tp_symfony (master -> origin)
λ git commit -m "Formulaire d'enregistrement sécurisé fonctionnel."
[master 5099210] Formulaire d'enregistrement sécurisé fonctionnel.
5 files changed, 87 insertions(+), 7 deletions(-)
```

Formulaire de login et chargement d'un utilisateur

C'est au tour de notre formulaire de login d'être modifié ! Mais cette fois il y aura quelque chose d'un petit peu nouveau : lors de la connexion d'un utilisateur, nous allons faire en sorte qu'il soit enregistré dans une session.

Nous allons commencer par modifier encore une fois le fichier `security.yml` en ajoutant les lignes suivantes :

```
main:
  anonymous: true
  pattern: ^/
  http_basic: ~
  provider: tp_provider
  form_login:
    login_path: login
    check_path: login
    default_target_path: user
```

Explications :

- `form_login` : c'est la méthode d'authentification utilisée par le pare-feu
 - o `login_path` : correspond à la route du formulaire de connexion, il s'agit de la route login que nous avons définie pour ce formulaire.
 - o `check_path` : correspond à la route de validation du formulaire de connexion, c'est sur cette route que seront vérifiés les identifiants saisis par l'utilisateur.
 - o `Default_target_path` : contient la route ciblée après qu'on se soit connecté.

Modification du contrôleur `LoginController` :

```
use Symfony\Component\Security\Http\Authentication\AuthenticationUtils;

class LoginController extends Controller
{
    /**
     * @Route("/login", name="login")
     */
    public function index(AuthenticationUtils $authenticationUtils)
    {
        $error = $authenticationUtils->getLastAuthenticationError();
        $lastUsername = $authenticationUtils->getLastUsername();
        return $this->render('login/index.html.twig', array(
            'last_username' => $lastUsername,
            'error' => $error
        ));
    }
}
```

Nous n'aurons plus besoin de celles que nous avons écrites.

- `$error` va contenir l'erreur de login s'il y en a une (erreur dans les éléments saisis)
- `$lastUsername` va contenir le nom d'utilisateur renseigné dans le formulaire de login

Maintenant modifions la vue associée :

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

{% block title %}Login{% endblock %}

{% block body %}
    {{ parent() }}
    <div class="container">
        <section>
            <h1 class="display-4">Login</h1>
            {% if error %}
                <div class="alert alert-warning text-center">{{ error.messageKey|trans(error.messageData, 'security') }}</div>
            {% endif %}
            <article>
                <form action="{{ path('login') }}" method="post">
                    <div class="form-group">
                        <label for="username">Email :</label>
                        <input type="text" class="form-control" id="username" aria-describedby="emailHelp" name="_username" value="{{ last_username }}">
                        <small id="emailHelp" class="form-text text-muted">We'll never share your email with anyone else.</small>
                    </div>
                    <div class="form-group">
                        <label for="password">Password :</label>
                        <input type="password" class="form-control" id="password" name="_password">
                    </div>
                    <button type="submit" class="btn btn-primary">Login</button>
                </form>
            </article>
        </section>
    </div>
{% endblock %}
```

Nous allons modifier une petite chose dans la barre de navigation :

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

{% block body %}
    <header>
        <nav class="navbar navbar-expand-lg navbar-light bg-light">
            <a class="navbar-brand" href="#">TP Symfony</a>
            <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarSupportedContent" aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle navigation">
                <span class="navbar-toggler-icon"></span>
            </button>

            <div class="collapse navbar-collapse" id="navbarSupportedContent">
                <ul class="navbar-nav mr-auto">
                    <li class="nav-item">
                        <a class="nav-link" href="{{ path('user_registration') }}"><i class="fas fa-user-plus"></i> User registration</a>
                    </li>
                    {% if is_granted('ROLE_USER') %}
                        <li class="nav-item dropdown">
                            <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown" role="button" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
                                <i class="fas fa-user"></i> {{ app.user.firstname }} {{ app.user.name }}
                            </a>
                            <div class="dropdown-menu" aria-labelledby="navbarDropdown">
                                <a class="dropdown-item" href="{{ path('user') }}">Account</a>
                                <div class="dropdown-divider"></div>
                                <a class="dropdown-item" href="#"><i class="fas fa-sign-out-alt"></i> Disconnect</a>
                            </div>
                        </li>
                    {% else %}
                        <li class="nav-item">
                            <a class="nav-link" href="{{ path('login') }}"><i class="fas fa-sign-in-alt"></i> Login</a>
                        </li>
                    {% endif %}
                </ul>
            </div>
        </nav>
    </header>
{% endblock %}
```

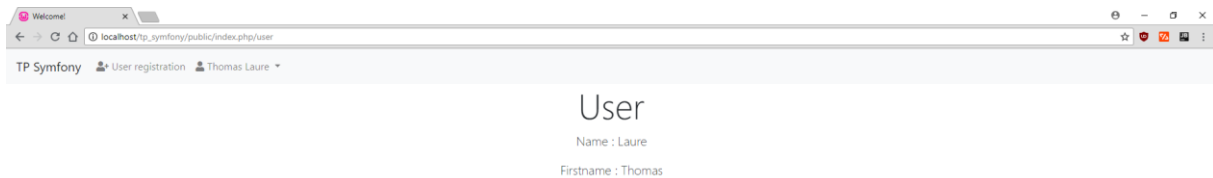
Ici nous vérifions que l'utilisateur possède bien les droits **ROLE_USER** (rôle qu'on lui a passé dans la méthode **getRoles()** de l'interface **UserInterface**). S'il possède bien les droits, on affiche le contenu souhaité.

Ensuite, retour dans le contrôleur **LoginController** pour écrire la méthode :

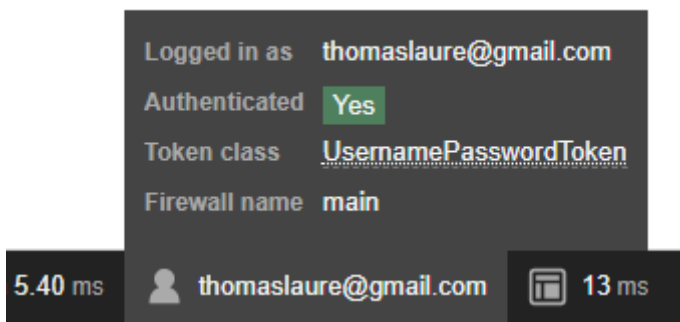
```
/**
 * @Route("/user", name="user")
 */
public function showUser(Security $security)
{
    $user = $security->getUser();
    return $this->render('user.html.twig', array(
        'user' => $user
    ));
}
```

Il s'agit de la route qui sera appelée après qu'on se soit connecté (cf. la précédente modification du `security.yml`).

Maintenant il ne nous reste plus qu'à essayer de nous connecter !



Ça fonctionne ! Observons de plus près la barre de débogage de Symfony :



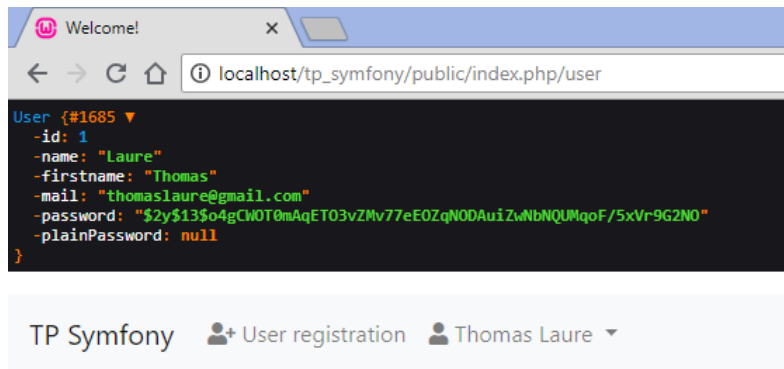
Ceci nous confirme bien que nous sommes bien connectés en tant qu'utilisateur.

Si vous souhaitez avoir le détail d'une ou de plusieurs variables, Symfony nous propose la fonction `dump()` (et non `var_dump()`), à utiliser comme suit :

```
/**
 * @Route("/user", name="user")
 */
public function showUser(Security $security)
{
    $user = $security->getUser();
    dump($user);

    return $this->render('user.html.twig', array(
        'user' => $user
    ));
}
```

En paramètre il suffit de passer toutes les variables dont vous souhaitez connaître le détail :



N'essayez pas de me contacter à cette adresse mail, c'en est une fausse (qui est probablement utilisée par quelqu'un d'autre par contre).

Déconnexion :

Ne pensez-vous pas qu'il serait bien de pouvoir nous déconnecter ? Bon, un petit détour rapide par le `security.yml` (c'est la dernière fois, je vous jure) :

```
main:
  anonymous: true
  pattern: ^/
  http_basic: ~
  provider: tp_provider
  form_login:
    login_path: login
    check_path: login
    default_target_path: user
  logout:
    path: logout
    target: login
```

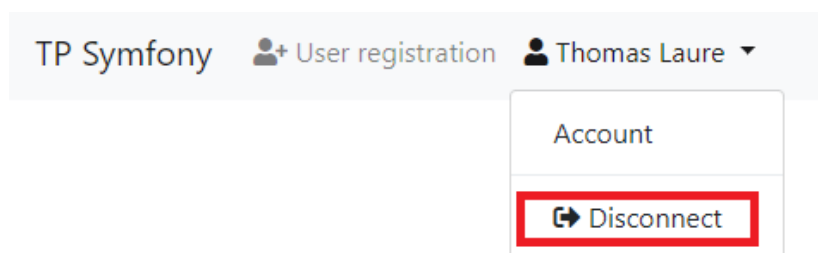
Dans `LoginController` :

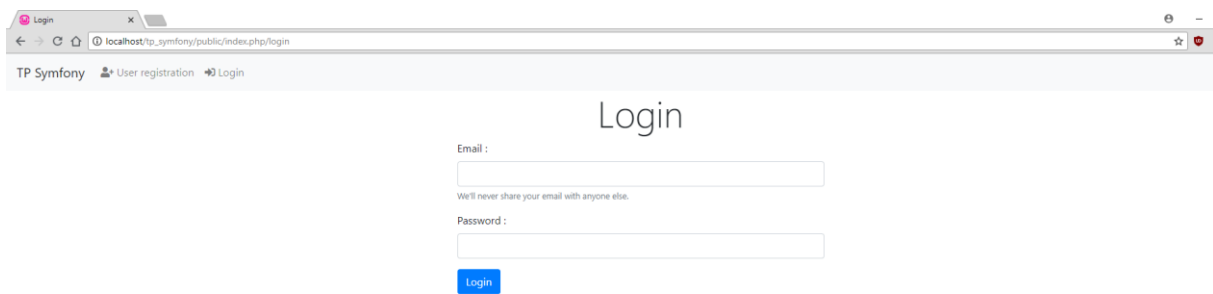
```
/**
 * @Route("/logout", name="logout")
 */
public function logout()
{
    return $this->render(
        'login/index.html.twig'
    );
}
```

Modifiez cette ligne dans la barre de navigation pour y ajouter le chemin :

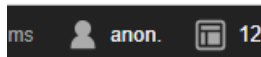
```
<a class="dropdown-item" href="{{ path('logout') }}"><i class="fas fa-sign-out-alt"></i> Disconnect</a>
```

A présent cliquez sur le bouton « Disconnect » :



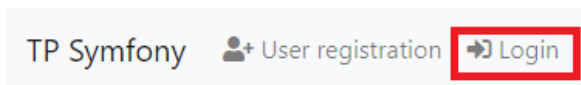


Puis :



Nous avons bien été déconnectés.

Et comme nous ne sommes plus connectés, la barre de navigation a également changé :



Maintenant nous pouvons faire de nouveau un commit pour la fin de notre formulaire de login :

```
D:\wamp\www\tp_symfony (master -> origin)
λ git add .
warning: LF will be replaced by CRLF in config/packages/security.yaml.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in src/Controller/LoginController.php.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in templates/login/index.html.twig.
The file will have its original line endings in your working directory.
warning: LF will be replaced by CRLF in templates/navbar.html.twig.
The file will have its original line endings in your working directory.
2018.1.2
D:\wamp\www\tp_symfony (master -> origin)
λ git commit -m "Amélioration du formulaire de login."
[master 621b46c] Amélioration du formulaire de login.
4 files changed, 83 insertions(+), 44 deletions(-)
rewrite templates/login/index.html.twig (73%)

D:\wamp\www\tp_symfony (master -> origin)
λ git push -u origin master
Counting objects: 27, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (24/24), done.
Writing objects: 100% (27/27), 4.55 KiB | 1.52 MiB/s, done.
Total 27 (delta 16), reused 0 (delta 0)
remote: Resolving deltas: 100% (16/16), completed with 11 local objects.
To: https://github.com/Almarean/tp_symfony.git
b091ca1..621b46c master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

Exercices bonus

1. Vous avez vu que nous n'avons rien pour vérifier qu'un utilisateur a bien été enregistré mis à part jeter un œil dans la base de données. Essayez de mettre en place un moyen d'afficher un message lorsqu'un utilisateur a réussi à s'enregistrer.
2. Comme il s'agit d'un système de login par adresse mail, essayez de mettre en place un moyen permettant de vérifier que l'adresse mail saisie dans le formulaire d'enregistrement n'est pas déjà utilisée pour un compte existant. Afficher un message si l'adresse mail est déjà utilisée.

Pourquoi utiliser un framework ?

Les développeurs Symfony qui maîtrisent le framework pourront facilement intégrer un projet développé à partir du framework, contrairement à un projet développé en PHP "maison", où il n'y a pas de normes ni règles imposées. Dans ce dernier cas, la phase d'apprentissage et reprise du code existant peut demander un effort conséquent pour le nouveau développeur intégrant le projet.

Les fichiers doivent respecter une syntaxe particulière et doivent se trouver au bon endroit dans l'arborescence du projet. Cela garantit une facilité de maintenance sur le long terme, les développeurs savent rapidement dans quel fichier il faut aller pour apporter les modifications nécessaires.

L'architecture MVC permet de découper le code représentant la logique métier de l'application et le code de présentation des vues. Ainsi un intégrateur web voire même un webdesigner n'aura aucun mal à intervenir sur la partie présentation du projet, sans avoir à intervenir sur des fichiers PHP complexes.

Un framework intègre des bundles/packages/recipes qui favorisent la réutilisation de code écrit par les développeurs du framework, qui peuvent intégrer des modules comme le bundle de sécurité comme nous l'avons vu, ou encore la création de formulaires, ce qui représente un gain de temps non négligeable. De plus ces éléments peuvent être retouchés pour produire des solutions sur mesure. Tout cela contribue donc à produire du code de qualité, ce qui entraîne souvent des sites et/ou applications dites haut de gamme.

Mon expérience sur Symfony :

Je ne peux guère prétendre d'avoir quinze ans d'expérience en développement web sous Symfony, mais je vais tout de même parler de mon expérience dans cet apprentissage ~~dans la douleur~~. Non, plus sérieusement j'ai eu du mal à m'y mettre au début dans les TP, mais les projets de deuxième année m'ont motivé à l'utiliser dans un contexte plus concret. Etant donné que nous avons eu des cours sur la version 3.3 du framework et que Symfony 4.0 venait de sortir, j'ai fait la très grande majorité de mon apprentissage en passant des heures (avec des pauses) dans la documentation. Ensuite les cours de M. Roche et celui d'OpenClassrooms ont été complémentaires, surtout du point de vue des formulaires et de l'utilisation des relations avec Doctrine.

Mais quand on se lance dans un projet d'apprentissage comme celui-ci, aussi petit soit-il, il ne faut pas s'arrêter en plein milieu, sinon ce sera plus dur de s'y remettre. J'ai mis un à deux mois avant de pouvoir écrire un simple formulaire permettant de laisser des avis sur des ateliers (donc deux entités), et encore aujourd'hui, à l'heure où j'écris ces lignes, je me rends compte que je n'ai pas respecté certains standards imposés par la communauté de Symfony, et que j'avais écrit pas mal de méthodes qui existaient déjà dans l'écosystème du framework.

Pour apprendre, il vaut mieux prendre son temps et passer deux semaines dans la documentation en se mettant petit à petit en tête les connaissances, plutôt que de faire un « rush » et faire du copier/coller de ce qu'on peut trouver sur Internet sans rien comprendre. C'est pour ça que j'ai bien insisté sur les sources officielles et à jour, plutôt que sur des tutoriels YouTube faits à moitié.

Ce n'est pas non plus la peine de rester jusqu'à 1 h ou 2 h du matin (je parle d'expérience) dans la documentation, parce qu'on ne s'en souvient pas au réveil étant donné qu'après minuit le cerveau est dans sa phase de récupération. Donc on perd pas mal d'heures de sommeil qui sont importantes, et en plus on n'est pas opérationnel la journée. Ensuite il faut rattraper le retard les autres jours.

Solution des exercices bonus

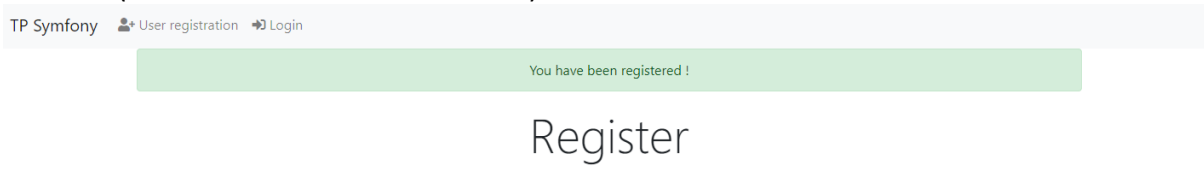
- 1) Dans le contrôleur `RegistrationController`, ajoutez dans le premier `return` :

```
return $this->render(
    'registration/register.html.twig',
    array(
        'form' => $form->createView(),
        'text_alert' => 'You have been registered !',
        'class_alert' => 'alert-success'
    )
);
```

Puis dans la vue associée (`/template/registration/register.html.twig`) :

```
<div class="container">
    {% if text_alert is defined and class_alert is defined %}
        <div class="alert {{ class_alert }}" text-center">{{ text_alert }}</div>
    {% endif %}
    <section>
        <h1 class="display-4">Register</h1>
        <article>
            {{ form_start(form) }}
```

Résultat (si on saisit des données valides) :



2) Toujours dans le même contrôleur, créez une méthode privée `findOneByMail()` :

```
private function findOneByMail(string $mail)
{
    $repository = $this->getDoctrine()->getManager()->getRepository(User::class);
    $result = $repository->findOneBy(array(
        'mail' => $mail
    ));
    return $result;
}
```

Cette méthode va demander à Doctrine de vérifier s'il existe une entité `User` possédant l'adresse mail passée en paramètre en passant par la méthode `findOneBy()`. Cette dernière va renvoyer un tableau de données avec une seule donnée, ou `null`.

L'index `'mail'` du tableau correspond à une propriété de

Et modifiez la méthode `register()` :

```
public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder)
{
    // 1) build the form
    $user = new User();
    $form = $this->createForm(UserType::class, $user);

    // 2) handle the submit (will only happen on POST)
    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()) {

        if ($this->findOneByMail($user->getEmail()) === null) {
            // 3) encode the password
            $password = $passwordEncoder->encodePassword($user, $user->getPlainPassword());
            $user->setPassword($password);



            // 4) save the User!
            $entityManager = $this->getDoctrine()->getManager();
            $entityManager->persist($user);
            $entityManager->flush();

            return $this->render(
                'registration/register.html.twig',
                array(
                    'form' => $form->createView(),
                    'text_alert' => 'You have been registered !',
                    'class_alert' => 'alert-success'
                )
            );
        } else {
            return $this->render(
                'registration/register.html.twig',
                array(
                    'form' => $form->createView(),
                    'text_alert' => 'You have been already registered !',
                    'class_alert' => 'alert-danger'
                )
            );
        }
    }

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

On vérifie si la méthode `findOneByMail()` renvoie un tableau vide (`null`). Si c'est le cas on peut créer le nouvel utilisateur et on renvoie un message de succès, sinon on renvoie simplement un message d'erreur.

Résultat (quand on saisit une adresse mail déjà enregistrée) :

TP Symfony  User registration  Login

You have been already registered !