TP 1 – Premiers pas avec Symfony avec Docker

Docker est une plateforme logicielle qui permet de créer, déployer et exécuter des applications de manière isolée, appelée *conteneurisation*. Les conteneurs sont des environnements légers, portables et reproductibles qui contiennent tout le nécessaire pour faire fonctionner une application : le code, les dépendances, les bibliothèques et le système d'exploitation minimal requis. Docker est déjà installé sur les ordinateurs de l'ISTIC.

Étape 0 : Lancer le docker engine

Lancez docker via le Docker Desktop

Étape 1 : Création d'un projet docker

1) Créer un répertoire `mon_projet_docker`

A l'istic, ne le créez pas dans H : mais sur C:/ par exemple sur le bureau de votre PC et non, celui de votre compte étudiant **(donc pensez à tout récupérer en fin de séance!!!)**

2) placez le fichier `docker-compose.yml`

Une fois, le dossier pour ton projet créé, placez le fichier nommé `docker-compose.yml` à l'intérieur de ce dossier .

Le fichier docker-compose.yml définit quatre services : app, webserver, db et phpmyadmin. Chacun de ces services correspond à un conteneur Docker qui contient différentes applications ou composants nécessaires au bon fonctionnement de ton projet Symfony.

Les conteneurs contiennent les éléments suivants :

- **Conteneur** `app` : Exécute PHP pour gérer la logique de l'application Symfony.
- **Conteneur** `webserver` : Gère les requêtes HTTP avec Nginx et sert les fichiers de l'application.
- **Conteneur** `db` : Gère la base de données MySQL utilisée par l'application Symfony.
- **Conteneur** `phpmyadmin` : Gère PhpMyAdmin utilisé par l'application Symfony.

Tous ces conteneurs interagissent ensemble pour fournir l'environnement nécessaire au développement de ton application Symfony

3) Placer le fichier `de configuration Nginx`

Placer le fichier nommé `nginx.conf` dans le même répertoire que ton `docker-compose.yml` avec le contenu qui suit. Il va remplacer le fichier de configuration existant pour Nginx.

Le fichier `nginx.conf` est un fichier de configuration pour le serveur web Nginx. Il définit comment Nginx doit traiter les requêtes HTTP, comment il doit servir les fichiers de ton application, et comment il interagit avec d'autres services.

4) Placer le Dockerfile

Tout d'abord, placez le fichier nommé `Dockerfile` dans le même répertoire que le fichier `docker-compose.yml`.

Petit rappel : *Il faut d'abord faire CTRL+D pour sortir du shell …ou CTRL+C pour arrêter le serveur qui tourne dans le terminal…*

Étape 2 : Création d'un projet Symfony

Nous allons utiliser Symfony dans Docker, nous allons devoir démarrer les containers, utiliser Symfony puis arrêter les containers.

1) Lancer Docker Compose

Ouvre un terminal depuis le dossier où se trouve ton fichier `docker-compose.yml` et exécute :
`docker-compose up -d`

```
C:\Users\vsans\Desktop\mon_projet_docker>docker-compose up -d
[+] Building 0.0s (0/0)
[+] Running 4/4
 ✔Network mon_projet_docker_default  Created
 ✔Container symfony_db               Started
 ✔Container symfony_app              Started
 ✔Container symfony_web              Started
```

La commande `docker-compose up -d` est utilisée pour démarrer les services définis dans un fichier `docker-compose.yml`.

On peut voir les containers actifs dans docker Desktop.

On peut les voir également avec la commande : docker ps

```
C:\Users\vsans\Desktop\mon_projet_docker>docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED         STATUS         PORTS
14bb81463d45   nginx:alpine   "/docker-entrypoint.…"   6 minutes ago   Up 6 minutes   0.0.0.0:8080->80
497b7067095b   php:8.1-fpm    "docker-php-entrypoi…"   6 minutes ago   Up 6 minutes   9000/tcp
17a931844ec5   mysql:5.7      "docker-entrypoint.s…"   6 minutes ago   Up 6 minutes   3306/tcp, 33060/
```

2) Accéder au conteneur PHP

Exécute cette commande pour accéder au conteneur PHP : `docker exec -it symfony_app bash`

On accède au shell de PHP qui est dans docker :

```
C:\Users\vsans\Desktop\mon_projet_docker>docker exec -it symfony_app bash
root@497b7067095b:/var/www#
```

et on peut enfin travailler !

Configurez enfin vos infos git

`git config – global user.email « votreemail@mail.com »`

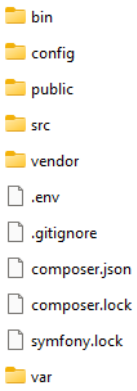
`git config – global user.name « votrenom »`

3) Créer le projet Symfony

Ensuite, crée un nouveau projet Symfony avec la commande
Petit rappel : *Il faut d'abord faire CTRL+D pour sortir du shell …ou CTRL+C pour arrêter le serveur qui tourne dans le terminal…*

```
symfony new apptuteur
```

Cela va créer l'arborescence suivante du projet :



Ton projet Symfony est dans ce répertoire apptuteur, tu peux créer autant d'applis symfony que tu veux avec les containers présents. Chaque application Symfony aura son répertoire dans lequel on pourra lancer le serveur Web.

Les répertoires et fichiers principaux d'un projet Symfony sont les suivants :

1. `bin/`
    - Contient les scripts exécutables, notamment `console`, qui est utilisé pour exécuter des commandes Symfony.
2. `config/`
    - Contient les fichiers de configuration de l'application.
    - `packages/` : Configurations spécifiques pour différents bundles.
    - `routes/` : Définitions des routes de l'application.
    - `services.yaml` : Configuration des services et dépendances.
3. `public/`
    - Dossier accessible publiquement, où se trouve le point d'entrée de l'application (`index.php`).
    - Contient également des fichiers statiques comme CSS, JavaScript, et images.
4. `src/`
    - Contient le code source de l'application.
    - `Controller/` : Contrôleurs qui gèrent la logique des requêtes HTTP.
    - `Entity/` : Modèles d'entité qui représentent les données dans la base de données.
    - `Repository/` : Repositories pour interagir avec les entités.
    - `Form/` : Formulaires pour la gestion des données utilisateur.
    - `Service/` : Services contenant la logique métier.

Petit rappel : *Il faut d'abord faire CTRL+D pour sortir du shell …ou CTRL+C pour arrêter le serveur qui tourne dans le terminal…*

5. `templates/`
    - Contient les fichiers de template Twig, utilisés pour générer des pages HTML.
6. `translations/`
    - Contient les fichiers de traduction pour gérer l'internationalisation de l'application.
7. `var/`
    - Contient des fichiers générés par Symfony, tels que les caches, logs, et fichiers temporaires.
    - `cache/` : Cache de l'application.
    - `log/` : Fichiers de log pour le débogage et la surveillance.
8. `vendor/`
    - Contient les dépendances installées via Composer, y compris les bibliothèques et bundles tiers.
9. `composer.json`
    - Fichier de configuration de Composer, qui définit les dépendances de l'application et d'autres métadonnées.
10. `symfony.lock`
    - Fichier qui verrouille les versions des dépendances installées pour garantir la cohérence entre les installations.
11. `README.md`
    - Fichier de documentation du projet, généralement utilisé pour expliquer comment installer, configurer et utiliser le projet.

Étape 4 : Démarrage le serveur avec PHP intégré et accès à l'application

Assure-toi d'être dans le répertoire de ton projet symfony puis démarre le serveur Symfony avec :

```
symfony server:start
```

Partie 1 : A la découverte des routes et des controllers

1. Créez une classe *MainController* dans le répertoire *src/Controller* avec une méthode index avec le code suivant :

Petit rappel : *Il faut d'abord faire CTRL+D pour sortir du shell …ou CTRL+C pour arrêter le serveur qui tourne dans le terminal…*

```php
<?php
// src/Controller/MainController.php
namespace App\Controller;

use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Attribute\Route;

class MainController
{
    #[Route('/index')]
    public function index(): Response
    {
        return new Response(
            '<html><body>Votre première page</body></html>'
        );
    }
}

?>
```

2. Testez la route suivante http://localhost:8000 . Que constatez-vous ? Quelle est l'URL à taper pour voir l'exécution de la méthode *index* de *MainController* ?
3. Rajoutez une méthode *indexbis* dans votre controlleur, accessible via la route */bonjour*
4. Ajoutez `use Symfony\Component\HttpFoundation\Request` en haut de votre fichier *MainController.php*
Cette librairie va nous permettre d'accéder aux classes `Request` et `Response` (Vous souvenez vous de ce à quoi cela correspond ?)

2. La méthode indexbis fait un affichage du message *"Bonjour X"*, cette méthode va créer un objet `$request` de la façon suivante :

```
$request= Request::createFromGlobals();
```

Puis appelez la méthode query sur cet objet request et la méthode get de la façon suivante :

```
$nom=$request->query->get('nom', 'Inconnu');
```

3.Testez les routes */bonjour* et */bonjour?nom=John*.

A quoi sert `Request::createFromGlobals();` ? A quoi cela équivaut `$request->query->get('nom', 'Inconnu');` si vous aviez utilisé du PHP *"classique"* ?

4. Modifiez votre code pour que à la place de X s'affiche le prénom passé en paramètre.

Au fait, lorsque l'on appelle une URL directement dans le navigateur, de quelle méthode HTTP s'agit-il ?

Partie 2 : Routes paramétrables : valeurs par défaut et contraintes

Une façon plus propre est d'utiliser directement des paramètres dans la route. Par exemple, on souhaitera pouvoir taper la route *bonjour/John* au lieu de *bonjour?nom=John*

1. Modifier l'annotation de votre route et ajouter pour *indexbis* un paramètre au path *{nom}*

2. Testez à nouveau la route. Que constatez-vous ?

4. Testez la route */bonjour* . Que se passe t il ?

6. Rajoutez dans l'annotation de votre route une option defaults pour le nom qui vaut Inconnu grâce à `defaults: ['nom' => 'Inconnu']`

Petit rappel : *Il faut d'abord faire CTRL+D pour sortir du shell …ou CTRL+C pour arrêter le serveur qui tourne dans le terminal…*

7. Testez la route */bonjour/35* ? Que constatez vous ?

8. Dans l'annotation, definissez un pré-requis sur le paramêtre avec une expression régulière de la façon suivante `requirements: ['nom' => '[A-Za-z]+']`

9. Retapez la route suivante */bonjour/35* ? Que constatez-vous ?

10. Comment faire pour restreindre l'appel de cette route à un appel en méthode GET ?

11. Créez une méthode indexter toujours dans ce même controleur donc la route sera */calcul/18*

18 étant un paramètre de route, qui sera un chiffre et donc la valeur ne dépassera pas 100.
12. Retirez votre annotation et modifiez le fichier *apptuteur/config/routes.yaml* pour faire en sorte que *indexter* reste accessible, i.e la route est déclarée dans le fichier YAML et non grâce à des annotations.

Partie 3 : Premier template Twig

Twig est un langage de templates qui permet des affichages HTML plus simplement, mais pas seulement ! Nous pourrons produire d'autres formats de HTML
1. Créer un autre controlleur FormController.php avec le code minimal suivant :

```
public function hello($prenom="Bryan", Environment $twig){
return new Response ("Hello $prenom");
}
```

2. Dans le dossier *templates*, créez *hello.html.twig*. Ce fichier possèdera un titre au sens *title* et une balise *h1*
3. Dans *FormController.php*, créez la variable `$html=$twig->render('hello.html.twig');`
Remarque : Pas besoin de préciser à symfony de lui préciser que les templates sont dans le répertoire templates
4. Modifier votre return pour qu'il prenne en paramètre du `Response` cette variable et faites afficher votre page.
5. Modifiez maintenant votre code de *hello.html.twig* et indiquez `{{prenom}}` dans le *h1* En lançant la route, vous verrez que vous obtenez une erreur 500 car la variable n'est pas encore connue Pour cela, on va passer un tableau associatif comme deuxième paramètre de la fonction render

```
$html=$twig->render('hello.html.twig', ['prenom' => 'Sophie'] );
```

6. Testez cette possibilité en remplaçant cette valeur fixe de prénom par le paramètre reçu par votre route.
7. Rajoutez un filtre pour que le prénom s'affiche en majuscule
8. Créez une deuxième méthode *liste()* de route */tuteurs* qui déclare le tableau suivant

```
['tuteurs'=>[ ['nom'=>'Johnson', 'prenom'=>'Paul'], ['nom'=>'Walberg',
'prenom'=>'Mark'] ]]
```

Ce tableau est passé à un template twig que vous créérez pour afficher la liste des tuteurs dans une liste à puces.
9. Créez ensuite un formulaire accessible via la route /search_tuteur, celui-ci affichera un formulaire où l'on peut saisir un nom et il appelera en méthode POST le controlleur *FormController.php* via une méthode *verify* qui declarera le tableau précédent et vérifiera si le tuteur existe ou pas dans les tuteurs présents dans le tableau.
10. Améliorez avec un peu de CSS

Partie 4 : Vérification de vos routes

Testez la commande suivante

```
php bin/console debug:router
```

A quoi sert-elle ?

Petit rappel : *Il faut d'abord faire CTRL+D pour sortir du shell …ou CTRL+C pour arrêter le serveur qui tourne dans le terminal…*

Partie 5 : Arrêter les conteneurs

Lorsque tu as terminé, tu peux arrêter les conteneurs avec :

```
docker-compose down
```

Petit rappel : *Il faut d'abord faire CTRL+D pour sortir du shell …ou CTRL+C pour arrêter le serveur qui tourne dans le terminal…*