Arborescence

`/bin`

Ce dossier contient deux fichiers, dont un qui va être très important car nous allons beaucoup l'utiliser : `console`. La console Symfony (**différente de l'outil Symfony CLI**) nous fournira des commandes utilitaires permettant de générer des fichiers au sein de notre arborescence, accélérant le développement de notre application, mais aussi garantissant une meilleure constance dans le style du code : la génération via des templates s'effectue toujours de la même façon, nous aurons uniquement à ajouter le code propre à notre application.

`/config`

Ce dossier, comme son nom le laisse facilement deviner, contient les fichiers de configuration des différents packages utilisés dans l'application.

Les fichiers de configuration utilisent le format `yaml`, basé sur l'indentation des différentes sections.

Se trouvent également 2 fichiers `bundles.php`, regroupant les composants applicatifs que l'on souhaite activer dans notre application, selon l'environnement, et `preload.php` pour le preloading introduit par PHP 7.4 pour le pré-chargement de scripts.

`/migrations`

Les migrations contiendront des classes PHP décrivant les changements de structure de notre base de données, quand nous travaillerons avec une BDD.

`/public`

Ce dossier contient uniquement un fichier `index.php`, qui va être le point d'entrée de notre application.

`/src`

Dans ce dossier, on retrouvera les classes de notre application.

On va avoir par exemple le dossier `Controllers` dans lequel se trouveront toutes les classes de contrôleurs permettant de gérer la navigation et le routage dans notre application.

Les entités (donc le modèle) se trouveront elles dans le dossier `Entity`.

Les repositories, dans le dossier `Repository`, seront notre couche de services, permettant de requêter nos modèles.

Nous n'aurons pas à toucher le fichier `Kernel.php`, c'est le noyau de Symfony.

`/templates`

Ce dossier contiendra tous nos templates, écrits avec `Twig`, un moteur de template que nous verrons plus tard.

`/tests`

On placera dans ce dossier tous les tests unitaires ou fonctionnels qu'on écrira.

Ces tests sont utiles pour disposer d'une suite automatisée vérifiant le bon comportement du code et/ou la bonne communication entre différents composants applicatifs.

`/translations`

Ce dossier contiendra des fichiers de traduction, si nous voulons travailler sur une application présentant des libellés multilingues, ou encore chaînes localisées.

`/var`

Ce dossier est destiné à recevoir des données de caches et de logs, il n'est pas intégré au gestionnaire de versions si on en utilise un.

On trouvera par exemple les fichiers issus de la compilation du conteneur applicatif, dans le dossier `cache`.

`/vendor`

Le dossier `/vendor` est géré par Composer, pour y inscrire la méthode d'autoloading ainsi que les sources des dépendances utilisées dans l'application.

On remarquera qu'il n'est pas versionné.

En effet, ce dossier est **entièrement** géré par Composer. Ainsi, lorsqu'on va clôner ou forker un dépôt de sources, on pourra utiliser la commande `composer install` pour le générer. Il contient souvent des milliers de fichiers issus des dépendances de notre application, il est donc inutile de commit et push tous ces fichiers sur un dépôt distant.

`.env`

Ce fichier contient les variables d'environnement de l'application.

Il contient en premier lieu la définition de l'environnement (`APP_ENV`).

Nous verrons plus tard l'utilité de définir plusieurs fichiers contenant des variables d'environnement et les stratégies de versioning associées.

`.env.test`

De la même façon que le fichier `.env`, on définira des variables d'environnement dans le fichier `.env.test`, mais ces valeurs seront utilisées uniquement lors de l'exécution de tests unitaires et/ou fonctionnels.

On pourra par exemple décider de changer les paramètres d'accès à la base de données pendant les tests, pour travailler avec une autre base de données que celle qu'on utilise en développement.

`.gitignore`

Tous les fichiers à ne pas intégrer au gestionnaire de versions.

`composer.json`

Le fichier Composer principal, qui contient toutes nos dépendances, et la méthode d'autoloading, entre autres.

On trouvera, entre autres, les dépendances de notre application dans 2 catégories : `require` et `require-dev`.

`require` regroupe les dépendances utilisées tout le temps.

Dans `require-dev`, on placera ce qu'on va appeler **des dépendances de développement**. Cela va concerner essentiellement les tests unitaires, ou utilitaires que l'on peut mettre en oeuvre lors de la phase de développement d'une application.

> En production par exemple, on ne voudra pas des dépendances de développement. On pourra ainsi demander à Composer de ne pas les intégrer au projet : `composer install --no-dev`

`composer.lock`

Ce fichier est celui consulté par Composer lorsque vous effectuez un `composer install`, pour installer toutes les dépendances préalablement définies.

On peut le voir comme l'équivalent du `package-lock.json` avec npm, qui permet de regrouper les versions installées.

Ainsi, n'importe qui de nouveau sur le projet peut faire un `composer install` après avoir clôné ou forké le projet : il aura exactement les mêmes versions que nous.

`docker-compose.override.yml & docker-compose.yml`

Ces 2 fichiers décrivent le ou les services à instancier pour l'outil de conteneurisation **Docker**, et plus précisément l'utilitaire `docker-compose`.

`phpunit.xml.dist`

L'outil qui exécute la suite de tests automatisés s'appelle PHPUnit.

Ce fichier est donc le fichier de configuration de PHPUnit.

`symfony.lock`

Ce fichier sert à un outil intégré avec la version 4 de Symfony : Symfony Flex.

Symfony Flex est un outil construit au-dessus de Composer. Il permet, dans un projet Symfony, en plus d'installer une dépendance, d'exécuter des **recettes**, comme des scripts de pré-configuration d'un package.

Vu que nous nous situons dans un framework, il adopte une certaine structure (dossier `config` pour la configuration, autoloading `PSR-4` avec `App` comme espace de nom racine, etc...).

Lorsqu'on installe une dépendance avec la commande `composer require`, Flex peut alors entrer en jeu en créant automatiquement un fichier de configuration à l'endroit adéquat, ou bien un template de classe PHP dans `src`, etc...**il va nous aider à l'intégration d'un package au sein d'une application Symfony**.

Ce fichier garde, en plus de la version du package, la version de la recette exécutée.