Résumé des commandes à lancer pour installer un projet Symfony 4

1. INSTALLATION

Pour créer un nouveau projet de site développé avec Symfony 4

- Se placer dans le dossier dans lequel se trouve vos projets
    - par exemple : ***cd \xampp\htdocs*** ou ***cd \wamp\www***
- Ensuite lancer la commande suivante pour installer un projet Symfony :

composer create-project symfony/skeleton:"4.4.*" *nom_du_projet*
L'exécution de cette commande va créer un dossier *nom_du_projet*. Pensez bien à vous placer dans le dossier *nom_du_projet* avant de lancer les prochaines commandes : ***cd nom_du_projet***

2. SERVER

Pour visualiser son projet dans son navigateur, comme s'il était sur un serveur distant.
Installer **Symfony CLI** (cf. https://get.symfony.com/cli/setup.exe)

- Pour lancer le serveur en tâche de fond : **symfony server:start -d**
- Pour arrêter le serveur en tâche de fond : **symfony server:stop**

3. MAKER

Pour pouvoir créer plus facilement et automatiquement du code (des classes) utiles à votre projet.
composer require maker --dev

- Pour créer un fichier (controller, entity, …) : **bin/console make:*xxx***

⚠ Avant de pouvoir lancer **make:controller**, vous devez installer le composant ***Annotations***

4. ANNOTATIONS

Pour pouvoir utiliser la syntaxe *annotation* afin de déclarer des *routes* dans les fichiers *controller* (par exemple)
composer require annotations

5. DEBUG

Pour installer une barre de débug, qui donne beaucoup d'informations pour faciliter le débuggage
composer require debug --dev

La barre de débug s'affiche sur toutes les pages de votre projet. Elle ne sera visible que sur un environnement de développement (cf. fichier ***.env***)

6. DOCTRINE

Pour la base de données, Symfony utilise Doctrine, une bibliothèque PHP qui permet de gérer une base de données avec un projet PHP orienté objet
composer require orm

- Après l'installation, il faut modifier le fichier ***.env***, afin de définir les informations de connexion au SGBD utilisé par votre projet (bien souvent, MySql) :

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

Il faut remplacer les termes commençant par db_ par les valeurs correspondantes. S'il le faut, modifier aussi la version de votre serveur de base de données.

- Vous devez aussi modifier le fichier ***config/packages/doctrine.yaml*** en ajoutant les informations suivantes (⚠ à la syntaxe yaml, ne pas supprimer les tabulations du fichier)

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

- création de la base de données : **bin/console doctrine:database:create**
- création d'une table : **bin/console make:entity**
  - ↳ crée une classe Entity qui aura les caractéristiques de la table correspondante
  - ↳ répondre aux questions pour la création de la classe
- création d'une migration : **bin/console make:migration**
  - ↳ c'est un fichier contenant les requêtes de modification de la BDD
- exécution de la migration : **bin/console doctrine :migrations:migrate**

NB : chaque fois que vous créez/modifiez une entité, vous devez créer une migration. Les modifications dans la BDD ne seront effectives que lorsque vous aurez exécuter la migration

⚠ Ne modifiez jamais la structure de la base de données directement dans phpMyAdmin. Modifiez/supprimez les *Entity* et créez des migrations pour modifier la BDD

7. Fixtures

Pour enregistrer de fausses informations dans la BDD, on peut utiliser le composant Fixtures.
composer require orm-fixtures --dev

- création d'un fichier fixture : **bin/console make:fixture**
- installation de faker : **composer require fzaninotto/faker --dev**

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

- exécution des fixtures : **bin/console doctrine:fixtures:load**

8. SECURITY

Pour gérer automatiquement la connexion des utilisateurs et leurs droits (= *role*)
composer require security

- création de l'entity utilisateur : **bin/console make:user**

  ⚠ Si on utilise une autre entité que *User* pour la connexion (ex: Membre), il faut modifier la méthode ***UserRepository::upgradePassword***

  ↳ Dans l'instruction if (!$user instanceof User) , remplacer *User* par l'entité utilisée

- création de l'authentification (connexion) : **bin/console make:auth**

  ⚠ On peut créer plusieurs système d'authentification.

9. TWIG

Twig est un moteur de templates. Il permet des fichiers HTML sans utiliser de PHP
composer require twig

- cf. la documentation de Twig afin de retrouver toutes les fonctions et tous les filtres existants

10. WEBPACK ENCORE

Permet de gérer (et générer) les fichiers ressources utilisées pour le front
- composer require encore
S'il y a une erreur, il faudra peut-être installer node-sass avec **npm install node-sass**

- Pour lancer la commande qui permettra de créer les fichiers du front, il faut installer **yarn** sur votre ordinateur
  - Ensuite, installez yarn dans votre projet Symfony : **yarn install**
  - Si vous utilisez du Sass (avec un framework css ou pas) : **yarn add sass-loader@^7.0.1 node-sass --dev**
  - Pour utiliser la police d'icones FontAwesome : **yarn add @fortawesome/fontawesome-free**
    - Ajouter au début du fichier (s)css (ex : **assets/css/app.css**) :

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

- Modifier le fichier **webpack.config.js**
  - décommenter la ligne 57 : .enableSassLoader()
  - renommer le fichier *app.css* en *app.scss*

- Installation **d'un** framework css (⚠ n'installez pas les 2)
  - **Bulma**
    - **yarn add bulma**
    - fichier **app.scss**

      ↳ A placer au début du fichier

      `@import "~bulma";`

      ↳ Si vous voulez utiliser une extension Bulm (ex : bulma-slider)

      - **yarn add bulma-slider**

- ajouter @import "~bulma-slider";

après l'import de bulma

↳ NB : Si besoin, il faut redéclarer les variables SASS avant d'importer Bulma

- : #5887ff;

- fichier **app.js**

↳ Modifier le contenu : import '../css/app.scss';

- **Bootstrap**
  - **yarn add bootstrap**
  - **yarn add jquery popper.js**
  - fichier **app.scss**

↳ A placer au début du fichier

@import "~bootstrap/scss/bootstrap";

↳ NB : Si besoin, il faut redéclarer les variables SASS avant d'importer Bootstrap

- : darken(#428bca, 20%);

  - fichier **app.js**

↳ Modifier le contenu : import '../css/app.scss';

  - fichier **app.js**

↳ import '../css/app.scss';

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

- Pour créer les liens vers les fichiers js et css créés, il faut ajouter dans un template :
  - Lien pour le fichier CS : {{ encore_entry_link_tags("app") }}
  - Lien pour le fichier JS : {{ encore_entry_script_tags("app") }}

- Pour générer les fichiers assets : **yarn encore dev**
  - Si vous devez lancer cette commande sur le serveur de production, remplacer **dev** par **production**
  - ⚠ à chaque modification d'un fichier ressource (css, js), vous devez lancer cette commande afin de générer les fichiers avec vos modifications. Pour ne pas devoir lancer la commande à chaque fois, vous pouvez lancer **yarn encore dev --watch**. Tant que le processus sera actif, dès qu'une modification sera enregistrée, cela génèrera les fichiers ressources.

11. FORM VALIDATOR

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

- composer require form validator

Pour choisir le style qui sera appliqué aux formulaires qui vont être créés automatiquement, il faut modifier le fichier **config/packages/twig.yaml :**

Ajouter :

avec Bootstrap : `form_themes: ["bootstrap_4_layout.html.twig"]`

avec Bulma : `form_themes: ['form_bulma_theme.html.twig']`

⚠ il faut créer le fichier **templates/form_bulma_theme.html.twig** (cf. le pdf cours_symfony_aymdev

créer un formulaire : **bin/console make:form**

intégrer le formulaire dans une template twig :

{{ form(variable) }}

{{ form_start(variable) }} … {{ form_end(variable) }}

…

Pour créer un formulaire avec un input file :

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

Dans le fichier **Form/AnnonceType.php**

Ajoutez (ou modifiez) le champ :

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

N'oubliez pas de rajouter, avant la classe :

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

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

Je vais créer un paramètre qui sera accessible à tout mon projet. C'est l'équivalent d'une constante globale

Dans le fichier **config/services.yaml** :

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

⚠ attention à la syntaxe YAML !

Dans le fichier **Controllers/AnnonceController**

Dans la méthode (route) qui va gérer le formulaire :

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

      // je crée un formulaire basé sur AnnonceType
      $form = $this->createForm(AnnonceType::class);

      // je lie le formulaire à la requête HTTP
      $form->handleRequest($rq);

      if($form->isSubmitted()){
        if($form->isValid()){
          // je récupère les informations de mon formulaire pour créer
          //une nouvelle entité Annonce
          $nvlAnnonce = $form->getData();

          // je récupère la valeur du paramètre global "dossier_images"
          // pour définir dans quel dossier va
          // être enregistré l'image téléchargée
          $destination = $this->getParameter("dossier_images");

          // je mets les informations de la photo téléchargée dans la
          // variable $photoTelechargee et s'il y a bien une photo téléchargée
          if($photoTelechargee = $form["photo"]->getData()){

            // je récupère le nom de la photo dans $nomPhoto
            $nomPhoto = pathinfo($photoTelechargee->getClientOriginalName(), PATHINFO_FILENAM
E);

            // je supprime les éventuels espace (au début et à la fin) du string $nomPhoto
            $nouveauNom = trim($nomPhoto);

            // je remplace les espaces par _ dans $nouveauNom (il vaut mieux éviter les espac
es dans les noms de fichier)
            $nouveauNom = str_replace(" ", "_", $nouveauNom);

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

            // Le fichier photo est enregistré dans le dossier $destination
            $photoTelechargee->move($destination, $nouveauNom);

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

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

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

            // je redirige vers une route
```

```
            return $this->redirectToRoute("profil");
          }
        }
        else{
          $this->addFlash("error", "Il manque des informations pour enregistrer votre annonce
");
        }
      }

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

Dans le fichier twig, lorsque je veux afficher les photos :

j'enregistre une variable globale pour twig. Dans le fichier **config/packages/twig.yaml** :

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

dans un template :

```
    <img src="{{ dossier_images_annonces ~ annonce.photo }}">
```