**Installation**

Pour utiliser Symfony il faut installer plusieurs composants en respectant l’ordre suivant :

Composer

Télécharger le fichier d’installation sur le site <https://getcomposer.org/download/>

Executer le fichier.

Scoop

Dans le Powershell, executer la commande suivante :

Invoke-Expression (New-Object System.Net.WebClient).DownloadString('https://get.scoop.sh')

Symfony

Dans le bash, exécuter :

scoop install symfony-cli

Doctrine

Dans le Bash, exécuter :

composer require doctrine/doctrine-migrations-bundle

**Création d’un nouveau projet**

Pour une installation standard

symfony new --webapp (ou --full pour les versions plus anciennes) **my\_project**

Pour une installationlts (Long Time Support)

symphony new --full **projet-lts**--version=lts

Pour installer une connexion sécurisé (https) exécuter en plus :

symfony server:ca:install

**Configuration de l’application**

* Il faut vérifier si tous les prérequis sont installé et activés. Pour cela exécuter :

symfony check:requirement

* + Pour activer des modules il faut les décommenter en enlevant le **;** dans le fichier php.ini
  + Pour l'accélérateur (opcache) : ajouter la ligne : Zend\_extension="C:\xampp\php\ext\php\_opcache.dll" sous ;zend\_extension=opcache **et** décommenter opcache.enable=1
* Ré-enregistrer le fichier **.env** en **.env.local** (A la racine de l’application)
* Pour l'utilisation de Mysql: commenter la ligne de postgresql et décommenter la ligne de Mysql, puis Changer l'utilisateur et le mot de passe (db\_user:db\_password). Ici **root**.
* Remplacer db\_name par le nom de notre base de données. Ici **ecommerce\_sf6**
* Indiquer la version de serveur de Mysql. Ici **10.4.18**

Pour information

* Lors du passage en production, dans le fichier **.env.local**, il faudra passer l’attribut **APP\_ENV** de **dev**à **prod**.
* **APP\_SECRET** : clé qui génère les jetons CRSF utilisé pour la validation des formulaires. Devra être modifié lors du passage en production.
* **MESSENGER\_TRANSPORT\_DSN** : sert à la gestion de la file d’attente des mails.
* **MAILER\_DSN** : a décommenter et configurer si on veut intercepter les mails envoyés.

**Création et peuplement de la base de données**

Création de la BDD

symfony console doctrine:database:create

Création de la table utilisateurs

symfony console make:user

**Fichier créé** **:** src/Entity → **nomdufichier**.php

Création ou modification des entités

symfony console make:entity

**Fichiers créés** **:** src/Entity → **nomdelatable**.php pour chacune des entités.

Création du fichier avec les requêtes SQL

symfony console make:migration

**Fichier créé** **:** migrations → Version**xxxxxxxxx**.php

Synchronisation de Doctrine et Symfony

php bin/console doctrine:migrations:sync-metadata-storage

Migration des informations vers la BDD

symfony console doctrine:migration:migrate

**Important**

* Pour la création des clés étrangères il faut, lors de la création de la propriété, définir le type à relation.
* Pour la création de clés composites, supprimer la propriété Id et son geteur de la table, et ajouter **#[ORM\Id]** devant les clés étrangères qui constituent la clé composite.
* Dans les fichiers php générés, il faut manuellement préciser les propriétés unique en ajoutant : **unique: true**.
* Pour les dates, il faut ajouter : **options: ['default' => 'CURRENT\_TIMESTAMP']** pour que ce soit la date courante qui soit prise en compte.

**Création et gestion des templates**

**Avant de commencer.**Il faut désinstaller le composant Webpack de symfony. Ce dernier peut entrainer des erreurs par la suite.

composer remove webpack

* Une fois désinstallé, supprimer le dossier **assets**.
* Dans le fichier **base.html.twig** : vider les blocks stylesheet et javascript dont les contenus sont en fait des reliquats de Webpack.

Création du contrôleur principal

symfony console make:controller **MainController**

**Fichier créé** **:** src/Controller → **MainController**.php

Pour informations

* Il est recommandé de séparer les fichiers communs en plusieurs partie telles que le footer, la nav, etc…. Les fichiers devront avoir l’extension **.html.twig**, et être placer dans un sous-dossier de template (par exemple \_partials). Ensuite, il faudra utiliser la méthode **include** pour les importer dans un fichier (généralement dans le <body> de **base.html.twig**).
* La méthode **asset()** va chercher les fichier directement dans le dossier **public**.
* Si dans le fichier **index.html.twig** on intègre un block déjà existant dans le fichier **base.html.twig**, il remplacera ce dernier.

En cas de besoin de fichiers css et js personnels

* Créer un dossier **assets** dans **public**, puis les sous-dossiers **css** et **js**. Y placer les fichiers **styles.css** et **script.js**.
  + Les fichiers devront être importés dans **base.html.twig** via la méthode **asset()**
    - <link rel="stylesheet" href="{{ asset('assets/css/styles.css') }}">
    - <script src="{{ asset('assets/js/scripts.js')}}" **defer**></script>
  + L’importation devra se faire en dessous des blocks correspondant afin de permettre la surcharge.
  + De plus, il faut ajouter manuellement l’attribut **defer** afin que le javascript ne démarre pas avant le chargement du DOM (les balises se trouvant dans le <head>).

En cas d’utilisation de Bootstrap

* Télécharger les fichiers bundle de Bootstrap et copier les fichiers suivants :
  + Dans **css** : **bootstrap.min.css** et **bootstrap.min.css.map**.
  + Dans **js** : **bootsrap.bundle.min.js** et **bootstrap.bundle.min.js.map**.
* Importer les fichiers de bootstrap dans base.html.twig en les plaçant au dessus des blocks correspondant.
  + - <link rel="stylesheet" href="{{ asset('assets/css/bootstrap.min.css') }}">
    - <script src="{{ asset('assets/js/bootstrap.bundle.min.js')}}" **defer**></script>
  + Penser à ajouter l’attribut **defer**. Cf plus haut.

**Les formulaires d’authentification**

Création du formulaire d’authentification

symfony console make :auth

Deux options sont proposées : **[0]** il faut créer le formulaire intégralement, **[1]** on utilise le formulaire prédéfini.

**Fichiers créés** **:** src/Controller/Security → **SecurityController**.php et **UserAuthenticator**.php

template/security → **login.html**.twig

* Dans le fichier **security.yaml** il est possible de décommenter et de compléter la ligne #**target: app\_any\_route** afin d’indiquer sur quelle page se rendre lors de la déconnection.
* Dans le fichier **SecurityController.php**, la fonction **logout** peut restée vide : elle est gérée directement par Symfony.
* Dans **UserAuthenticator**.php, le **LOGIN\_ROUTE** reprend le même nom que celui enregistré dans le fichier **SecurityController.php**.
* Le fichier **login.html.twig** contient le formulaire de connexion.

Création du formulaire d’inscription

symfony console make:registration-form

A la question : Do you want to send email to verify the user’s email after registration ?, si on répond oui il faut un module complémentaire à Symfony (Symfonycast verify email).

**Fichier modifié :** src/Entity → **Users**.php

**Fichiers créés** : src/Form → **RegistrationFormType**.php

src/Controller → **RegistrationController**.php

templates/registration → **register.htm**l.twig

Les modifications suivantes doivent être apportées :

* Dans **login.html.twig**, remplacer app.user.**username** par app.user.**userIdentifier**.
* Modifier le constructeur de **Users**.php ajouter **$this->created\_at = new \DateTimeImmutable();** afin qu’une nouvelle instance de la date courante soit bien créée lors de chaque appel de la méthode.
* Dans le fichier **RegistrationFormType** il faut ajouter des champs au formulaire dans la section **$builder**. **Il est impératif que toutes les colonnes de la table Users dans la BDD se retrouvent dans le formulaire.**

Il faut ajouter le Type, puis il est possible d’ajouter des attributs ‘attr’ comme des class Bootstrap. Enfin on peut assigner un nom aux champs en utilisant ‘label’. Exemple :

**->add('email', EmailType::class, [**

**'attr' => [**

**'class' => 'form-control'**

**],**

**'label' => 'E-mail'**

**])**

* Il faut également ajouter les mêmes champs dans le fichier **register.html.twig** et faire la mise en forme si nécessaire.
* Dans le fichier **UserAuthenticator**.php, dans la fonction publique **onAuthenticationSuccess**, il est possible d’effectuer la configuration suivante afin d’indiquer vers quelle pages rediriger l’utilisateur en cas de connexion :

return new RedirectResponse($this->urlGenerator->generate('**main**'));

// throw new \Exception('TODO: provide a valid redirect inside '.\_\_FILE\_\_);

**Création de Traits**

Lorsqu’un attribut est utilisé dans plusieurs fichiers, il est possible de simplifier le code en créant un Trait de cet attribut. Ensuite ce trait sera importé directement dans les classes. Dans notre exemple c’est l’attribut created\_at qui est optimisé.

* Créer un sous-dossier **Traits** dans **Entity**.
* A l’intérieur de celui-çi créer le fichier **CreatedAtTrait**.php.
* Dans ce fichier, définir le namespace : **App\Entity\Trait**, et ajouter dessous **use Doctrine\ORM\Mapping as ORM;**
* Créer le trait en lui-même : **trait** **CreatedAtTrait** **{}**
* Récupérer l’attribut et ses accesseurs et les copiers dans le trait.
* Pour importer le Trait dans une classe il faut intégrer le code : **use** **CreatedAtTrait;**
* il est ensuite possible de supprimer des classes les attributs et accesseurs de craeted\_at.

**Utilisation de dataFixtures**

Installation

composer require --dev orm-fixtures

L’option --dev permet de ne l’installer que pour le mode développeur car il est inutile en production.

Création d’une dataFixtures

symfony console make:fixtures **CategoriesFixtures**

**Fichier créé :** src/DataFixtures → **CategoriesFixtures**.php

Afin de simplifier la création des différentes catégories il est possible de faire une méthode complémentaire comme suit :

**public function createCategory(string $name, Categories $parent = null, ObjectManager $manager)**

**{**

**$category = new Categories(); // Création d’une nouvelle catégorie**

**$category->setName($name); // Attribution du nom de la catégorie**

**$category->setSlug($this->slugger->slug($category->getname())->lower()); // cf. plus bas**

**$category->setParent($parent); // Attribution d’une catégorie parente (par défaut null)**

**$manager->persist($category);**

**$this->addReference('cat-'.$this->counter, $category); // Créé référence**

**$this->counter++; // Au début de la classe, un compteur est initialisé à 1**

**return $category;**

**}**

Pour le slug, si on veut pouvoir le créer automatiquement en fonction du nom de la catégorie (tel que c’est codé plus haut), il faut intégrer un constructeur au début de la classe comme suit :

**public function \_\_construct(private SluggerInterface $slugger){}**

Execution des datatFixtures

symfony console doctrine:fixtures:load

**Important**

* Lors de l’éxecution, les tables sont automatiquement vidées avant d’être à nouveau remplies. Cela peut poser problème si il y a des clés étrangères. Pour y remédier, il faut dans l’entité concernée modifier la relation parent en ajoutant **#[ORM\JoinColumn(onDelete : ‘CASCADE’)]**. Générer ensuite un fichier migration et l’envoyer à la base de données.
* Les dataFixtures sont par défaut éxécutées par ordre alphabétique. Pour palier à cela il faut au niveau de la classe qui pose problème implémenter une interface comme suit :

**class ImagesFixtures extends Fixture implements DependentFixtureInterface**

Ensuite il faut lui définir la méthode getDependencies() en indiquant dans le tableau quelles sont les datasFixtures qui doivent s’éxécuter avant (ici ProductsFixtures) :

**public function getDependencies(): array**

**{**

**return [**

**ProductsFixtures::class**

**];**

**}**

**Utilisation de Faker**

Installation

composer require fakerphp/faker

Paramétrage

* Importe **use Faker;** au début du fichier.
* Intégrer le code suivant (dans ce cas ce sont des utilisateurs qui sont générés) dans la méthode **load** :

        $faker = Faker\Factory::create('fr\_FR'); **// Défini la langue**

        for($usr = 1; $usr <= 5; $usr++){ **// Boucle sur le nombre de users désiré**

            $user = new Users();

            $user->setEmail($faker->email); **// Génère un mail**

            $user->setLastname($faker->lastName); **// Génère un nom**

            $user->setFirstname($faker->firstName); **// Génère un prénom**

            $user->setAddress($faker->streetAddress); **// Génère une adresse (rue)**

            $user->setZipcode(str\_replace(' ', '', $faker->postcode)); **// Génère un code postal**

            $user->setCity($faker->city); **// Génère un nom de ville**

            $user->setPassword(

                $this->passwordEncoder->hashPassword($user, 'secret') **// Génère un mot de passe**

            );

            $manager->persist($user);

        }

**Création des controllers**

symfony console make:controller **ProfileController**

**Fichiers créés** : templates/profiles → **index.html.twig**

src/Controller → **ProfileController**.php

* Placer la route avant la classe afin que toute ses méthodes soient accessibles.
* Dans la classe, modifier le nom (**name:)** présent dans la route afin d’être cohérent avec le nom de la méthode qui suit.
* Si on intègre un slug dans la route, il faut le placer entre accolades pour indiquer que c’est une propriété variable. Ensuite on fait une injection de dépendance dans les parmètres de la méthode en précisant l’entité visée (penser au **use**). Enfin il faut intégrer la méthode **compact()** au **render()**. Le résultat de **compact()** est un tableau. Exemple :

**#[Route('/{slug}', name: 'details')]**

**public function details(Products $product): Response**

    {

        return $this->render('products/details.html.twig', compact('product'));

    }

* Dans le template, si on veux afficher une données extraite du tableau précédement créé, il faut mettre **{{ nomdutableau.name }}** (par exemple pour afficher le nom).
* **Si le controller est créé manuellement** : penser à intégrer le **namespace** et les **use**.

**Intégration de données dans les Vues**

* Pour afficher les catégories produits sur la page d’acceuil , il faut commencer par injecter **CategoryRepository** dans la methode **index()** du fichier **MainController.php**.
* Il faut ensuite aller chercher les éléments avec la méthode **findby()** en lui passant deux arguments : quels éléments doivent être récupérés et dans quel ordre. Cela donne :

**public function index(CategoriesRepository $categoriesRepository): Response**

**{**

**return $this->render('main/index.html.twig', [**

**'categories' => $categoriesRepository->findBy([],**

**['categoryOrder' => 'asc'])**

**]);**

* Pour afficher dans la Vue, il faudra faire une boucle sur le tableau précédement créé.
* Pour pouvoir accéder aux sous-catégories via des liens, il faut commencer par faire un Controller. Ici **CategoriesController**. Ce dernier possède une méthode qui va nous permettre de récupérer la liste des produits :

**public function list(Categories $category): Response**

**{**

**//On va chercher la liste des produits de la catégorie**

**$products = $category->getProducts();**

**return $this->render('categories/list.html.twig', compact('category', 'products'));**

**}**

* Ensuite dans le lien href du bouton, il faut définir la destination via la propriété **path{{}}** comme suit :

**href="{{ path('categories\_list', {slug: category.slug}) }}"**

* Le résultat final de la boucle d’affichage donne ceci (des cards bootstrap on été utilisées pour la mise en forme) :

**{% for category in categories %} // On boucle sur le tableau généré par le findby()**

**{% if category.parent == null %} // On regarde si la catégorie possède un parent**

**<div class="col-12">**

**<h2>{{ category.name }}</h2> // On affiche juste le nom de la catégorie**

**</div>**

**{% else %}**

**<div class="col-4">**

**<div class="card h-100">**

**<div class="card-body">**

**<h3 class="card-title">{{ category.name }}</h3> // Sous-catégorie**

**<a href="{{ path('categories\_list', {slug: category.slug}) }}" class="btn btn-primary">Voir les produits</a> // Lien vers les produits des sous-catégories**

**</div>**

**</div>**

**</div>**

**{% endif %}**

**{% endfor %}**

**EasyAdmin Bundle**

Installation

composer require easycorp/easyadmin-bundle

Création du Dashboard

symfony console make:admin:dashboard

Création des Crud

symfony console make:admin:crud

Configuration des adresses

* Pour généréer automatiquement les routes, il faut implémenter la fonction urlGenerator. Dans le **DashBoardController**, créer une fonction construct dans la class **DashBoardController()** :

**public function \_\_construct(**

**private AdminUrlGenerator $adminGenerator**

**) {}**

* Ensiute il faut modifier la méthode **index()** comme suit :

**public function index(): Response**

**{**

**$url = $this->adminGenerator**

**->setController(ProductsCrudController::class) // Crud afficher par défaut**

**->generateUrl();**

**return $this->redirect($url);**

**}**

* Dans configureDashboard(), il est possible de changer le tittre affiché et de faire d’autre type de personnalisation (photo, etc…).

Configuration du pannaux latéral

Nous allons pouvoir créer des section avec des actions. Par exemple nous allons créer une section pour les produits avec une fonction d’ajout de produit et une fonction d’affichage de la liste des produits. Le code ressemblera à ceci :

**yield MenuItem::section('Produits'); // Section globale des produits**

**yield MenuItem::subMenu('Actions', 'fas fa-bars')->setSubItems([**

**MenuItem::linkToCrud('Ajout Produit', 'fas fa-plus', Products::class)->setAction(Crud::PAGE\_NEW), // Bouton d’ajout de produits (ouvre une nouvelle page)**

**MenuItem::linkToCrud('Voir Produits', 'fas fa-eye', Products::class)**

**]); // Bouton d’affichage des produits**

* EasyAdmin permet d’utiliser les icones de Font Awesomme directement.
* Pour les MoneyField, penser à définir le **setCurrency()**
* **Penser à importer les ‘use’.**

Modification des formulaires

Dans le crudController correspondant, nous pouvont effectuer des modifications sur les formulaires. Cela permet entre autre :

* Ajouter un label aux champs. Ce dernier viendra remplacer le nom par défaut tiré de la BBD, et viens se placer après. Exemple :

**TextField::new('name', 'Nom'),**

* Pour masquer un champ afin qu’il n’apparaisse pas dans le formulaire, il faut lui ajouter :

**->hideOnForm()**

* Si on utilise un booléen pour activer/désactiver un produit ou autre, il est possible de filtrer à l’affichage uniquement ceux qui sont actif ou inversement avec **->setQueryBuilder()**

Ajout d’un bouton de duplication de produit

* En haut de la class il faut créer la constante :

**public const ACTION\_DUPLICATE = 'duplicate';**

* Ensuite on il afut créer le bouton et son action :

**public function configureActions(Actions $actions): Actions**

**{**

**$duplicate = Action::new(self::ACTION\_DUPLICATE) // Action appellé par le bouton**

**->linkToCrudAction('duplicateProduct') // Méthode qui effectue la duplication**

**->setCssClass('btn btn-info'); // CSS du bouton**

**return $actions->add(Crud::PAGE\_EDIT, $duplicate)**

**->reorder(Crud::PAGE\_EDIT, [self::ACTION\_DUPLICATE, Action::SAVE\_AND\_RETURN]);**

**} // Réorganise l’ordre d’affichage des boutons**

* Enfin il faut créer la méthode qui va réaliser la duplication :

**public function duplicateProduct(Admincontext $context, EntityManagerInterface $em, AdminUrlGenerator $adminUrlGenerator): Response**

**{**

**$product = $context->getEntity()->getInstance(); // Récupère les infos du produits qui doit être dupliqué**

**$duplicateProduct = clone $product; // Duplique le produit**

**parent::persistEntity($em, $duplicateProduct);**

**$url = $adminUrlGenerator->setController(self::class)**

**->setAction(Action::DETAIL) // Indique que l’on va ouvrir la page du nouveau produit après sa création**

**->setEntityId($duplicateProduct->getId())**

**->generateUrl();**

**return $this->redirect($url);**

**}**

Suppression d’une catégorie et des produits liés

Dans le **crudController** adéquat (ici **CategoriesCrudController**) :

**public function deleteEntity(EntityManagerInterface $em, $entityInstance): void**

**{**

**if (!$entityInstance instanceof Categories) return;**

**foreach ($entityInstance->getProducts() as $product) {**

**$em->remove($product);**

**}**

**parent::deleteEntity($em, $entityInstance);**

**}**