

Démarrer un projet PHP avec Symfony : organisation et structure - Tuto Symfony - PHP - Partie 1

Symfony est un **Framework PHP** presque indispensable dans le développement d'un projet PHP aujourd'hui. Il offre énormément de possibilités tout en donnant (enfin) envie de coder aux plus réticents du [langage PHP](#) (et il y en a beaucoup).

Cet article sera la première partie d'un article divisé en trois sur la création d'un projet Symfony. Nous ferons donc dans celle-ci un tour d'horizon global du Framework à travers **sa définition**, nous verrons ensuite **l'installation d'un projet Symfony** et enfin nous apprendrons à en **maîtriser la structure** pour y entrer plus en détails dans de prochains articles.

Symfony, c'est quoi ?



Symfony

Symfony est un Framework qui représente un ensemble de composants **PHP** autonomes (aussi appelés librairies / packages / bundles). Il permet de réaliser des sites internet dynamiques de manière rapide, structurée, et avec un développement clair. Les développeurs peuvent travailler sur ce Framework très facilement, seul ou en équipe, grâce à la facilité de prise en main et de maintenance.

Ce **puissant Framework PHP** a été lancé en 2007 par l'agence web française Sensio Labs. Historiquement, Symfony est une évolution d'un Framework interne à l'agence Sensio Labs, qui a été rendu open source à la communauté PHP. Pour l'anecdote, le logo représente les lettres SF, selon la volonté des créateurs de garder les initiales du projet de base : Sensio Framework.

Démarrage d'un projet Symfony

Préparation de l'installation de l'application

Avant toute chose, il faut disposer au minimum d'un serveur web ([Apache](#) ou [Nginx](#)) et d'un **PHP** (version 7.3 de préférence).

Ensuite, la mise en place du gestionnaire de dépendances **Composer** est indispensable : voici le [lien](#) pour suivre l'installation. Il va nous permettre l'installation de Symfony ainsi que la gestion des packages et de leur dépendances (nous y reviendrons plus tard).

Il vous faudra également [télécharger](#) et installer l'**exécutable** fourni par Symfony. Cela vous donnera accès à la commande **symfony** qui vous permettra l'installation du projet ainsi que le lancement en local d'un serveur web de développement.

Enfin, la [documentation](#) Symfony est vraiment très complète et couvre l'ensemble des versions du Framework (v2.0 à v5.1 aujourd'hui) : elle vous sera très utile pendant votre développement (notamment pour l'installation).

Installation de l'application

Il suffit maintenant d'exécuter la commande suivante :

```
symfony new article_symfony5 --full
```

Toute une liste de bundles vont être installés et le projet Symfony sera initialisé automatiquement par Composer dans un dossier (**article_symfony5** dans cet exemple). Une fois le projet généré, rendez-vous dans le dossier.

Testons notre application Symfony

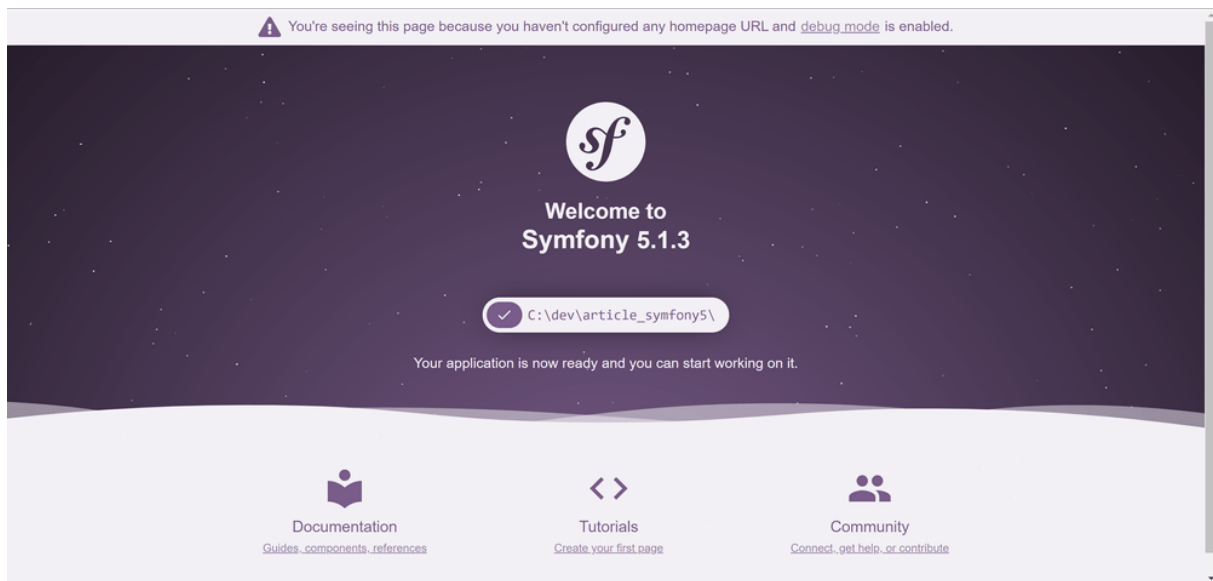
Pour tester le fonctionnement de votre application, vous pouvez démarrer le serveur de développement avec la commande suivante :

```
symfony server:start
```

Si vous êtes fainéant et que vous ne voulez pas vous rendre vous-même à l'adresse <http://localhost:8000/>, il y a aussi une commande qui le fera à votre place :

```
symfony open:local
```

Vous devriez avoir accès à la page d'accueil de Symfony :

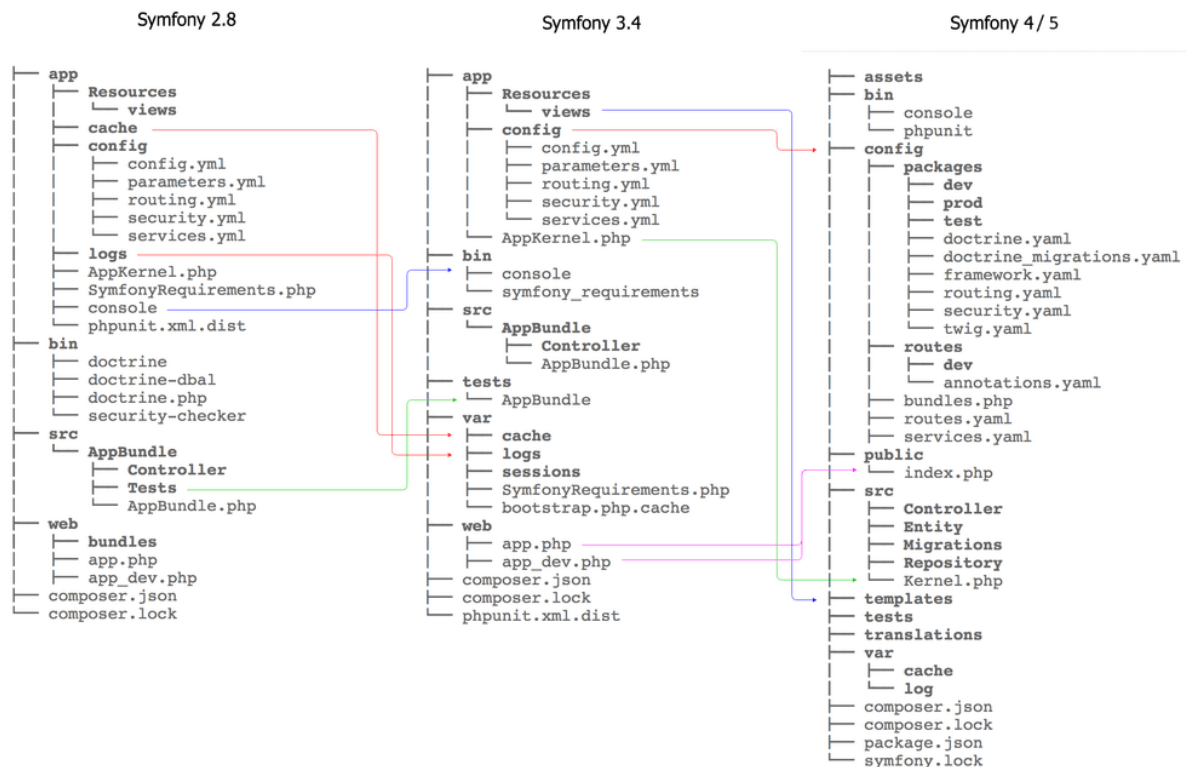


Vous êtes maintenant prêt à développer votre application !

Structuration du projet d'application Symfony

La structure d'un projet Symfony a fortement évolué depuis le début du Framework. Chaque nouvelle version apporte ses modifications et il est très facile de s'y perdre si l'on développe sur plusieurs versions.

Voici l'évolution en image (ça pourrait vous être utile) :



Nous allons maintenant faire un tour global des dossiers / fichiers importants à connaître pour s'y retrouver dans un projet Symfony 5. Nous reviendrons par la suite plus en détails à travers quelques exemples concrets.

Dossier [assets]

Comme son nom l'indique, ce dossier contient tous les assets nécessaires pour le front de notre application Symfony.

Les assets sont principalement des fichiers `.css` et `.js` qui seront par la suite compilés à l'aide du package **Webpack Encore**.

Dossier [bin]

Ce dossier contient les fichiers de commandes permettant, par exemple, de vider le cache Symfony, mettre à jour la base de données ou encore lancer nos tests unitaires.

On utilise généralement la commande `php bin/console` qui affiche toutes les commande Symfony disponibles.

Dossier [config]

Toute la configuration des packages, services et routes se fera dans ce dossier.

Cela permettra, entre autre, de configurer notre connexion à la base de données, mettre en place tout un système de sécurité, ou encore personnaliser les services que nous développerons.

Les fichiers de configuration sont par défaut en **YAML**, même s'il est tout à fait possible d'utiliser **PHP** ou **XML**.

Dossier [public]

C'est le point d'entrée de l'application : chaque requête / demande passe forcément par ce dossier et le fichier `index.php`.

Étant accessible par tous, il est généralement utilisé pour mettre à disposition des fichiers de ressources, principalement des images.

Dossier [src]

C'est le cœur du projet ! L'endroit où vous passerez le plus de temps à coder. Il regroupe tout le code PHP de votre application, c'est ici que vous mettrez en place toute la logique de votre application.

Les dossiers qui seront obligatoires à utiliser pour le fonctionnement de l'application sont :

- **[Controller]** : Définition des points d'entrée de votre application. Il se charge de rediriger vers les Manager / Service / Repository. Aucun traitement de données, accès à la BDD (base de données) ne doit se faire depuis un Controller (très important). Possibilité de choisir les méthodes d'entrée (GET, POST, PUT, DELETE, ...) ainsi que le type de réponse retournée (JSON, XML, ...).
- **[Entity]** : Définition de la structure de votre BDD (base de données) au travers de classes. Chaque Entity représente généralement une table en BDD. La commande `php bin/console doctrine:migrations` nous permettra de mettre à jour notre BDD à chaque modification de l'Entity.
- **[Repository]** : Un Repository est toujours rattaché à une Entity, il nous permet de créer nos fonctions qui iront requêter la table de notre Entity (ainsi que les tables liées). Symfony utilise l'ORM **Doctrine** qui permet de créer nos requêtes SQL à travers les **QueryBuilder** (très utile si l'on déteste faire du SQL).

Dossier [templates]

Symfony utilise depuis ses débuts le moteur de templates **Twig**.

Les fichiers de template Twig ont comme format `monfichier.html.twig` et viennent rajouter quelques fonctionnalités au HTML classique :

- `{{ ... }}` : appel à une variable ou une fonction PHP, ou un template Twig parent.
- `{% ... %}` : commande, comme une affectation, une condition, une boucle ou un bloc HTML.
- `{# ... #}` : commentaires.

Pour avoir plus d'infos et en apprendre plus sur Twig, vous trouverez ici une [documentation](#) assez complète.

Dossier [tests]

Les tests unitaires **PHPUnit** seront définis ici pour tester notre application.

La commande pour lancer nos tests :

```
php bin/phpunit
```

Il faut néanmoins s'assurer que le **package phpunit** soit installé en utilisant la commande suivante :

```
composer require --dev symfony/phpunit-bridge
```

Dossier [translations]

L'internationalisation des applications est très importante aujourd'hui. Il est donc nécessaire de mettre en place un système de traduction dès le début du projet. Cela reste néanmoins facultatif si vous êtes certain de ne jamais avoir à traduire votre application (je ne m'y essaierais pas)

Pour cela, on installe le **package translation** et on suit la [documentation](#) Symfony :

```
composer require symfony/translation
```

Dossier [var]

Ici seront stockés le **cache** et les fichiers de **log**.

Il est possible dans les fichiers de config de paramétrer la mise en cache et ce que l'on écrit dans les logs.

Fichier [composer.json]

Tous les **packages** sont enregistrés dans ce fichier.

Ils sont installés automatiquement dans le dossier **vendors** lors de l'initialisation du projet mais on peut utiliser la commande **composer install** pour les installer manuellement si besoin.

Pour mettre à jour les packages, on utilise la commande **composer update** et pour ajouter un package on utilise **composer require monpackage**

Configuration des packages et premiers développements Symfony - Tuto Symfony - PHP - Partie 2

Après une première partie introductive au Framework **Symfony**, on se retrouve aujourd'hui pour commencer à coder !

Nous verrons dans cette **deuxième partie** la **configuration des packages** de notre projet ainsi que la **mise en place** et l'**utilisation** des Controller / Managers / Service / Repository et Entity.

Configuration des packages d'un projet Symfony

La configuration des packages du projet se base sur le projet initialisé lors de la [partie 1](#) de cette série d'articles. Si vous rencontrez quelques difficultés, nous vous conseillons de vous y référer.

Étape 1 : Auto-génération des fichiers de configuration

Lors de l'ajout de certains packages à l'aide de la commande **composer require monpackage**, Symfony s'occupe de créer les fichiers **YAML** de configuration du package, d'activer des bundles et de modifier les variables d'environnement.

Étape 2 : Comprendre et terminer la configuration du package (si nécessaire)

Fichier [bundle.php]

Certains packages ajoutés nécessitent l'activation de bundles dans le fichier **bundles.php** dans le dossier **config** de notre projet.

Prenons comme exemple l'installation du package **profiler-pack** (qui vous sera extrêmement utile pendant votre développement).

Tout d'abord il faut ajouter le package au projet :

```
composer require symfony/profiler-pack
```

Une fois l'installation terminée, le fichier bundle.php a été modifié pour ajouter 2 nouveaux bundles :

```
<?php

return [
    Symfony\Bundle\FrameworkBundle\FrameworkBundle::class => ['all' => true],
    Symfony\Bundle\TwigBundle\TwigBundle::class => ['all' => true], // new
    Symfony\Bundle\WebProfilerBundle\WebProfilerBundle::class => ['dev' => true, 'test' => true], // new
];
```

On peut remarquer une différence entre le **TwigBundle** et le **WebProfilerBundle** :

- `['all' => true]` : Le TwigBundle est accessible dans n'importe quel environnement de développement. Il permet d'utiliser le **moteur de template Twig** et est donc toujours nécessaire pour afficher les pages web.
- `['dev' => true, 'test' => true]` : Le WebProfilerBundle est accessible uniquement pour les environnements de **test** et de **dev**. Ce bundle nous servant à déboguer notre application lors du développement, il ne doit pas être accessible pour l'environnement de **production**.

Il est donc possible de choisir l'activation du bundle en fonction de l'environnement désiré.

Fichier [.env]

Ce fichier, à la racine du projet, stocke les **variables d'environnement** nécessaires au fonctionnement de l'application. Certains packages modifient le fichier pour y ajouter leurs variables.

Prenons comme exemple l'installation du package **orm-pack** (qui nous permet la **connexion à la BDD** et l'installation de **Doctrine**).

Tout d'abord, il faut ajouter le package au projet :

```
composer require symfony/orm-pack
```

Une fois l'installation terminée, le fichier **.env** a été modifié pour ajouter la variable d'environnement **DATABASE_URL** :


```
...  
###> doctrine/doctrine-bundle ###  
# Format described at https://www.doctrine-project.org/p  
rojects/doctrine-dbal/en/latest/reference/configuration.  
html#connecting-using-a-url  
# For an SQLite database, use: "sqlite:///kernel.projec  
t_dir%/var/data.db"  
# For a PostgreSQL database, use: "postgresql://db_user:  
db_password@127.0.0.1:5432/db_name?serverVersion=11&char  
set=utf8"  
# IMPORTANT: You MUST configure your server version, eit  
her here or in config/packages/doctrine.yaml  
DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/  
db_name?serverVersion=5.7  
###< doctrine/doctrine-bundle ###  
...
```


Il vous restera ensuite à renseigner vos identifiants de connexion à la base de données.

Mais où est utilisée cette variable ? Il faut, pour le savoir, se rendre dans le dossier **packages** du dossier **config**.


Dossier [packages]


Pour chaque package ajouté nécessitant une configuration, un fichier **YAML** de config est créé dans ce dossier.

▼  config


▼  packages


▼  dev

 debug.yaml


 web_profiler.yaml


▼  prod


 doctrine.yaml


 routing.yaml


▼  test


 framework.yaml


 twig.yaml


 web_profiler.yaml


 cache.yaml

 doctrine.yaml

 doctrine_migrations.yaml

 framework.yaml

 routing.yaml

 twig.yaml

Lors de l'ajout du premier package **profiler-pack**, un fichier **web_profiler.yaml** a été créé.

Pour la package **orm-pack**, ce sont les fichiers **doctrine.yaml** et **doctrine_migration.yaml** qui ont été créés. Nous allons nous intéresser au fichier **doctrine.yaml** :

```
doctrine:
dbal:
url: '%env(resolve:DATABASE_URL)%'

# IMPORTANT: You MUST configure your server version,
# either here or in the DATABASE_URL env var (see .env file)
#server_version: '5.7'

orm:
auto_generate_proxy_classes: true
naming_strategy: doctrine.orm.naming_strategy.underscore_number_aware
auto_mapping: true
mappings:
App:
is_bundle: false
type: annotation
dir: '%kernel.project_dir%/src/Entity'
prefix: 'App\Entity'
alias: App
```

Nous retrouvons notre variable **DATABASE_URL** qui est utilisée pour se connecter à la BDD. On remarquera que pour récupérer une variable dans le fichier **.env**, il faut utiliser la syntaxe **%env(resolve:ma_variable)%**.

Il vous est ensuite possible de configurer la connexion à la BDD selon vos besoins, et surtout spécifier où se situent vos **Entity** qui sont utilisés pour faire le lien avec les tables de votre BDD.

Enfin, on remarquera que la configuration des fichiers **YAML** peut être adapté selon votre environnement. Le fichier doctrine.yaml est utilisé une seconde fois dans le **dossier prod**, permettant ainsi d'ajouter une configuration spécifique à l'environnement de production.

Enfin, nous allons pouvoir coder un peu avec Symfony !

Dans l'exemple que nous avons préparé, nous allons voir comment récupérer une liste d'utilisateurs en BDD.

Entity : User.php

Nous allons créer notre classe **User** dans notre fichier **User.php** du dossier **Entity** :

```
<?php

namespace App\Entity;

use DateTime;
use Doctrine\ORM\Mapping as ORM;

/**
```

```
* @ORM\Entity
* @ORM\Table(name="user")
*/
class User
{
/**
 * @var integer
 *
 * @ORM\Id()
 * @ORM\GeneratedValue(strategy="AUTO")
 * @ORM\Column(name="id", type="integer")
 */
private $id;

/**
 * @var ?string
 *
 * @ORM\Column(name="first_name", type="string")
 */
private $firstName;

/**
 * @var ?string
 *
 * @ORM\Column(name="last_name", type="string")
 */
private $lastName;

/**
 * @var ?DateTime
 *
 */
```

```
* @ORM\Column(name="user_birthday", type="datetime")
*/

private $birthday;

/** @var ?string */
private $description;

/** @return int */
public function getId(): int {
return $this->id;
}

/** @param int $id */
public function setId(int $id): void {
$this->id = $id;
}

/** @return string|null */
public function getFirstName(): ?string {
return $this->firstName;
}

/** @param string|null $firstName */
public function setFirstName(?string $firstName): void {
$this->firstName = $firstName;
}

/** @return string|null */
public function getLastName(): ?string {
return $this->lastName;
}
```

```

/** @param string|null $lastName */
public function setLastName(?string $lastName): void {
$this->lastName = $lastName;
}

/** @return DateTime|null */
public function getBirthday(): ?DateTime {
return $this->birthday;
}

/** @param DateTime|null $birthday */
public function setBirthday(?DateTime $birthday): void
{
$this->birthday = $birthday;
}

/** @return string|null */
public function getDescription(): ?string {
return $this->description;
}

/** @param string|null $description */
public function setDescription(?string $description): void {
$this->description = $description;
}
}

```

Nous voyons que des décorateurs ont été utilisés pour la classe et nos 4 attributs :

Décorateur	Explication
@ORM\Entity	Spécifie que notre classe est une Entity . Il est possible d'y rattacher un Repository (nous
@ORM\Table(name="user")	Spécifie le nom de la table en BDD (facultatif).
@ORM\Id	Spécifie l'attribut comme PK de la table.
@ORM\GeneratedValue	Spécifie la PK comme étant un auto-incrément .
@ORM\Column	Spécifie principalement le nom et le type d'une colonne en BDD. Il est possible de rajouter un maximum et si le champ peut être NULL en BDD selon nos besoins.

Une fois notre UserEntity créé, la structure de notre BDD doit être mise à jour.

Pour cela, il nous faudra générer un fichier de migration et l'exécuter pour appliquer la création de la table user.

Générer le fichier de migration

La commande suivante permet la création d'un fichier de migration contenant le SQL appliqué en BDD :

```
php bin/console doctrine:migrations:diff
```

Un fichier Version[datetime].php est créé dans le dossier **migrations**. Tous les fichiers de migration sont triés par date pour pouvoir les exécuter les uns après les autres dans le bon ordre.

Pour appliquer le fichier de migration, il suffit d'exécuter la commande :

```
php bin/console doctrine:migrations:execute --up
DoctrineMigrations\Version[datetime]
```

La table user est **créée en BDD !**

Si vous voulez annuler l'insertion de la table, vous pouvez tout à fait utiliser la commande :

```
php bin/console doctrine:migrations:execute --down  
DoctrineMigrations\Version[datetime]
```

Alimenter la table user avec des fixtures

Il faut tout d'abord installer le package **doctrine-fixtures-bundle** :

```
composer require --dev doctrine/doctrine-fixtures-bundle
```

Ensuite, pour générer de fausses données utilisateur, un package existe et le fera à notre place :

```
composer require --dev fzaninotto/faker
```

Dans le fichier **AppFixtures.php** du dossier **DataFixtures**, nous allons créer 20 utilisateurs :

```
<?php  
  
namespace App\DataFixtures;  
  
use App\Entity\User;  
use Doctrine\Bundle\FixturesBundle\Fixture;  
use Doctrine\Common\Persistence\ObjectManager;  
use Faker\Factory;
```

```

use Exception;

class AppFixtures extends Fixture
{
/**
 * @param ObjectManager $manager
 * @throws Exception
 */
public function load(ObjectManager $manager)
{
    $faker = Factory::create();
    // création de 20 utilisateurs
    for ($i = 0; $i < 20; $i++) {
        $user = new User();
        $user->setFirstName($faker->firstName);
        $user->setLastName($faker->lastName);
        $user->setBirthday($faker->dateTime);
        $manager->persist($user);
    }

    $manager->flush();
}
}

```

Il nous reste plus qu'à insérer les fixtures en BDD :

```

php bin/console doctrine:fixtures:load
Repository : UserRepository.php

```

Nous allons générer un fichier **UserRepository.php** dans le dossier **Repository** avec un exemple de **queryBuilder** :

```
<?php

namespace App\Repository;

use App\Entity\User;
use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
use Doctrine\Common\Persistence\ManagerRegistry;

/**
 * @method User|null find($id, $lockMode = null, $lockVersion = null)
 * @method User|null findOneBy(array $criteria, array $orderBy = null)
 * @method User[] findAll()
 * @method User[] findBy(array $criteria, array $orderBy = null, $limit = null, $offset = null)
 */
class UserRepository extends ServiceEntityRepository
{
    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, User::class);
    }

    /**
     * @param \DateTime $datetime
     * @return User[]
     */
    public function findByBirthdayMoreThan(\DateTime $datetime) {
```



```

return $this->createQueryBuilder('u')
->where('u.birthday > :datetime')
->setParameter('datetime', $datetime)
->getQuery()
->getResult()
;
}
}

```

Il faut savoir qu'un Repository doit être le seul à accéder à la BDD au travers de queryBuilder ou requêtes natives.

Enfin, il ne faut surtout pas oublier de lier le Repository avec l'Entity User, sinon impossible d'aller requêter sur la table user :

```

<?php
/**
 * @ORM\Entity(repositoryClass="App\Repository\UserRepository") // ajouter le Repository
 * @ORM\Table(name="user")
 */
class User

```

Service : UserService.php

Nous allons créer un fichier **UserService.php** dans le dossier **Service** avec une fonction qui va modifier la description des utilisateurs :

```

<?php

namespace App\Service;

```

```

use App\Entity\User;

class UserService {

    /**
     * @param User $user
     * @return User
     */
    public function updateUserDescription(User $user) {
        $description = 'Je suis ' . ucfirst($user->getFirstName()) . ' '.
        mb_strtoupper($user->getLastName()) . ', mon anniversaire est le '.
        $user->getBirthday()->format('d/m/Y');
        $user->setDescription($description);
        return $user;
    }
}

```

Il faut savoir qu'un Service va s'occuper de tout le traitement des données reçues depuis un repository.

Manager : UserManager.php

Nous allons créer un fichier **UserManager.php** dans le dossier **Manager** avec des fonctions qui iront chercher un seul ou tous les utilisateurs avec leur description :

```

<?php

namespace App\Manager;

```

```
use App\Entity\User;
use App\Repository\UserRepository;
use App\Service\UserService;
use Doctrine\ORM\EntityManagerInterface;
use Exception;

class UserManager
{
    /** @var EntityManagerInterface */
    private $em;

    /** @var UserService */
    private $userService;

    function __construct(EntityManagerInterface $em, UserService $userService)
    {
        $this->em = $em;
        $this->userService = $userService;
    }

    /**
     * Rechercher d'un utilisateur avec sa description
     *
     * @param int $id
     * @return User|false
     * @throws Exception
     */
    public function findOneWithDescription(int $id) {
        $user = $this->getRepo()->find($id);
        if (!$user) {
```

```

throw new Exception('Utilisateur introuvable !', 422);
}
return $this->userService->updateUserDescription($user);
}

/**
 * Liste des utilisateurs avec leur description
 *
 * @return User[]
 */
public function findAllWithDescription() {
    $usersWithDescription = [];
    $users = $this->getRepo()->findAll();
    foreach ($users as $user) {
        $usersWithDescription[] = $this->userService->updateUserDescription($user);
    }
    return $usersWithDescription;
}

/**
 * Récupération de notre Repository UserRepository
 *
 * @return UserRepository
 */
public function getRepo(): UserRepository
{
    return $this->em->getRepository(User::class);
}
}

```

Il faut savoir qu'un Manager sert seulement à **rediriger depuis un Controller** vers un Service ou directement un Repository.

Controller : UserController.php

Dans cette seconde partie, nous verrons comment utiliser un Controller et renvoyer une réponse en **JSON**, à la manière dont le ferait l'application si celle-ci était une **API**. L'utilisation des templates **Twig** se fera dans une prochaine partie.

Pour pouvoir transformer nos objets User en JSON, nous devons rajouter un **Serializer** à notre application. Il suffit d'utiliser la commande suivante :

```
composer require symfony/serializer symfony/property-access
```

Nous allons créer un fichier **UserController.php** dans le dossier **Controller** et ajouter 2 points d'entrée :

```
<?php

namespace App\Controller;

use App\Manager\UserManager;
use Exception;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\Serializer\Encoder\JsonEncoder;
```

```
use Symfony\Component\Serializer\Normalizer\DateTimeNormalizer;
use Symfony\Component\Serializer\Normalizer\ObjectNormalizer;
use Symfony\Component\Serializer\Serializer;

class UserController extends AbstractController
{
    /** @var Serializer */
    private $serializer;

    function __construct() {
        $this->serializer = new Serializer(
            [new DateTimeNormalizer(['datetime_format' => 'Y-m-d\TH:i:s.u\Z']), new
            ObjectNormalizer()],
            ['json' => new JsonEncoder()]
        );
    }

    /**
     * @Route("/users", name="get-users", methods={"GET"})
     *
     * @param UserManager $userManager
     * @return Response
     */
    public function getUsers(UserManager $userManager)
    {
        $users = $userManager->findAllWithDescription();

        return new Response($this->serializer->serialize($users, 'json'));
    }

    /**
```

```

* @Route("/users/{id}", name="get-user", requirements={"id"="\d+"}, meth
ods={"GET"})
*
* @param UserManager $userManager
* @param int $id
* @return Response
* @throws Exception
*/
public function getOneUser(UserManager $userManager, int $id)
{
    $user = $userManager->findOneWithDescription($id);
    return new Response($this->serializer->serialize($user, 'json'));
}
}

```

Testons nos services !

Il existe plusieurs façon de récupérer nos utilisateurs :

- Ouvrir le navigateur et taper l'url <http://localhost:8000/users> (fonctionne uniquement pour la méthode GET et si aucune authentification n'est nécessaire)
- Utiliser les logiciels **Postman** ou **Insomnia** qui permettront de créer une collection pour enregistrer tous les services de notre API (et plein d'autres choses très utiles).

Voici un exemple de JSON qui doit être retourné si l'on accède au service <http://localhost:8000/users/1> de notre application Symfony 5 :

```
{
```

```
"id": 1,  
"firstName": "Fidel",  
"lastName": "Sipes",  
"birthday": "2020-05-30T02:31:10.000000Z",  
"description": "Je suis Fidel SIPES. Mon anniversaire est le 30/05/2020"  
}
```


Affichage et administration des utilisateurs : assets, templates Twig et forms - Tuto Symfony - PHP - Partie 3

Le 07/12/2020 Par Corentin CHEVRET symfonyphpapplication-webutilisateurs symfony

Maintenant que [nous savons créer / peupler notre BDD et récupérer des données d'utilisateurs](#), nous allons voir comment les afficher et modifier.

Nous verrons dans cette **troisième partie** la **configuration des assets** et comment afficher nos utilisateurs à l'aide du **moteur de template Twig**. Enfin nous découvrirons les **Forms** qui permettront de modifier les informations utilisateurs.

Configuration des assets sur Symfony

Nous utiliserons le **bundle webpack-encore-bundle**, qui nous permettra de compiler nos assets en 1 seul fichier CSS et 1 seul fichier JS à travers un **serveur Node**. Cela fera gagner en performance pour charger tous nos assets (idéal pour les sites importants avec énormément de ressources à utiliser).

Installation

Tout d'abord, il faut ajouter le **bundle webpack-encore-bundle** :

```
composer require symfony/webpack-encore-bundle
```

Ensuite, comme nous utilisons un **serveur Node** pour compiler les assets, il est nécessaire [d'installer Node](#) pour pouvoir utiliser les commandes **npm**.

Une fois Node installé, nous pouvons installer les packages JS du fichier ****package.js**** (les développeurs Angular / React / Vue.js ne seront pas déployés) avec la commande :

```
npm install
```

À présent, nous avons accès à la commande **encore** : elle permet de lancer un serveur Node qui va compiler en direct nos assets à chaque modification (très pratique pendant le développement) :

```
encore dev-server
```

Un fichier **manifest.json** dans le dossier **public/build** est mis à jour pour indiquer sur quels liens sont accessibles nos assets :

```
{  
  "build/app.css": "http://localhost:8080/build/app.css",  
  "build/app.js": "http://localhost:8080/build/app.js",  
  "build/runtime.js": "http://localhost:8080/build/runtime.js",  
  "build/vendors~app.js": "http://localhost:8080/build/vendors~app.js"  
}
```

Les fichiers **app.css** et **app.js** sont disponibles dans le dossier **assets**. Ce sont ces fichiers qui vont contenir tout le CSS et JS de notre application. On peut

directement écrire dedans mais surtout faire des imports de fichiers pour que ce soit plus lisible.

Finalement, nous devons importer les fichiers app.css et app.js dans notre **template base.html.twig** dans le dossier **templates** pour que tous les templates "enfants", que nous créerons plus tard pour les utilisateurs, puissent utiliser les assets :

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>{% block title %}{% endblock %}</title>
{% block stylesheets %}
<!-- Ajout du CSS -->
{{ encore_entry_link_tags('app') }}
{% endblock %}
</head>
<body>
{% block body %}{% endblock %}
{% block javascripts %}
<!-- Ajout du JS -->
{{ encore_entry_script_tags('app') }}
{% endblock %}
</body>
</html>
```

Prise en charge du SASS

Le **SASS** est presque indispensable. Il va nous permettre d'avoir un code beaucoup plus lisible et surtout utiliser `@import` pour importer le CSS d'autres packages (comme Bootstrap).

Le fichier **webpack.config.js** à la **racine du projet** doit être modifié pour activer le SASS :

```
...  
// enables Sass/SCSS support  
.enableSassLoader() // Décommenter cette ligne pour activer le SASS  
...
```

Enfin, il suffit d'installer les packages JS suivants :

```
npm install sass-loader@^8.0.0 node-sass --save-dev
```

On peut maintenant renommer le fichier **app.css** en **app.scss** et relance le serveur Node (commande `encore dev-server`).

Ajout de Bootstrap

Nous allons maintenant ajouter Bootstrap à notre application et donc dans nos assets. Pour cela, il faut d'abord ajouter le **package npm Bootstrap** :

```
npm install bootstrap jquery popper.js --save
```

Ensuite, dans le fichier **app.scss**, nous rajoutons le CSS de Bootstrap :

```
@import "~bootstrap/dist/css/bootstrap.min.css";
```

Enfin, on importe le JS de Bootstrap dans **app.js** :

```
// CSS
import '../css/app.scss';

// JS
import 'bootstrap';
```

Et voilà ! Nous pouvons maintenant commencer à faire de belles templates Twig.

Template Twig

Nous allons maintenant créer notre première page à l'aide du moteur de template Twig.

Liste des utilisateurs

Création du template Twig

Le nouveau template **users.html.twig** sera ajouté à un nouveau dossier **users** dans le dossier **templates**.

Il affichera la liste de nos utilisateurs :

```
// template users. sauvegardée dans le dossier templates
{% extends "base.html.twig" %}

{% block title %}Tableau utilisateurs{% endblock %}
{% block body %}
<div class="p-5">
<h2>Tableau utilisateurs</h2>

<table class="table table-responsive">
<tr>
<th>Prénom</th>
<th>Nom</th>
<th>Date anniversaire</th>
<th>Description</th>
</tr>
{% for user in users %}
<tr>
<td>{{ user.firstName }}</td>
<td>{{ user.lastName }}</td>
<td>{{ user.birthday|date('d/m/Y') }}</td>
<td>{{ user.description }}</td>
</tr>
{% endfor %}
</table>
</div>
{% endblock %}
```

On peut voir `{% extends "base.html.twig" %}` en première ligne du code. Cela va nous permettre de modifier les **block** renseignés dans **base.html.twig**.

Ainsi, on peut modifier le **block title** et le **block body** de base.html.twig.

Modification du Controller

Le Controller **UserController.php** doit être modifié pour retourner le template Twig à la place du JSON actuellement :

```
...  
public function getUsers(UserManager $userManager)  
{  
    $users = $userManager->findAllWithDescription();  
    return $this->render('users/users.html.twig', ['users' => $users]);  
}  
...
```

On utilise la fonction **render** pour **renseigner le template Twig** ainsi que la variable contenant **nos utilisateurs** accessibles dans la template.

Résultat

La page listant les utilisateurs doit ressembler à cela si tout se passe bien de votre côté (toujours accessible à l'adresse <http://localhost:8000/users>) :

Fiche utilisateur

[Retour](#)

Prénom : Fidel

Nom : Sapes

Né(e) le : 26/10/1983

Description : Je suis Fidel SAPES, mon anniversaire est le 26/10/1983



Fiche utilisateur

Création du template Twig

Le template **user.html.twig** sera également ajouté au dossier **users** dans le dossier **templates**.

Il affichera la fiche d'un utilisateur :

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

{% block title %}Fiche utilisateur{% endblock %}
{% block body %}
<div class="p-5">
<div class="row">
<div class="col">
<h2>Fiche utilisateur</h2>
</div>
<div class="col-auto">
```



```

<a class="btn btn-primary" href="{{ path('get-users') }}"> Retour</a>
</div>
</div>

<p>Prénom : {{ user.firstName }}</p>
<p>Nom : {{ user.lastName }}</p>
<p>Né(e) le : {{ user.birthday|date('d/m/Y') }}</p>
<p>Description : {{ user.description }}</p>
</div>
{% endblock %}

```

Modification du Controller

Le Controller **UserController.php** doit également être modifié pour retourner le template Twig :

```

<?php
...
public function getOneUser(UserManager $userManager, int $id)
{
    $user = $userManager->findOneWithDescription($id);
    return $this->render('users/user.html.twig', ['user' => $user]);
}
...

```

Résultat

La fiche utilisateur devrait ressembler à cela (<http://localhost:8000/users/1>) :

Tableau utilisateurs

Prénom	Nom	Date anniversaire	Description
Fidel	Sapes	26/10/1983	Je suis Fidel SAPES, mon anniversaire est le 26/10/1983
Myrtis	Lubowitz	16/02/1999	Je suis Myrtis LUBOWITZ, mon anniversaire est le 16/02/1999
Shawn	Nienow	25/02/1985	Je suis Shawn NIENOW, mon anniversaire est le 25/02/1985
Vernie	Brown	17/02/1981	Je suis Vernie BROWN, mon anniversaire est le 17/02/1981
Destiney	Haley	27/03/1976	Je suis Destiney HALEY, mon anniversaire est le 27/03/1976
Adele	Nitzsche	29/10/2010	Je suis Adele NITZSCHE, mon anniversaire est le 29/10/2010
Nathanael	Borer	23/09/1997	Je suis Nathanael BORER, mon anniversaire est le 23/09/1997
Claudine	Marks	27/09/1970	Je suis Claudine MARKS, mon anniversaire est le 27/09/1970
Shany	Parisian	23/06/1986	Je suis Shany PARISIAN, mon anniversaire est le 23/06/1986
Dortha	Witting	09/09/1981	Je suis Dortha WITTING, mon anniversaire est le 09/09/1981
Maritza	Hirthe	16/08/2009	Je suis Maritza HIRTHE, mon anniversaire est le 16/08/2009
Alvah	Kreiger	14/04/2013	Je suis Alvah KREIGER, mon anniversaire est le 14/04/2013
Sage	Emmerich	25/05/1990	Je suis Sage EMMERICH, mon anniversaire est le 25/05/1990

Lier les 2 templates

Modification du template users.html.twig

Nous allons rajouter un bouton à chaque utilisateur pour accéder à sa fiche :

```
...  
<table class="table table-responsive">  
<tr>  
<th>Prénom</th>  
<th>Nom</th>  
<th>Date anniversaire</th>  
<th>Description</th>  
  
<!-- Ajout d'une entête vide -->  
<th></th>  
</tr>  
{% for user in users %}  
<tr>
```

```

<td>{{ user.firstName }}</td>
<td>{{ user.lastName }}</td>
<td>{{ user.birthday|date('d/m/Y') }}</td>
<td>{{ user.description }}</td>

<!-- Ajout du bouton pour accéder à la fiche d'un user -->
<td>
<a href="{{ path('get-user', {id: user.id}) }}" class="btn btn-primary">Voir</a>
</td>
</tr>
{% endfor %}
</table>

...

```

Résultat

Un nouveau bouton est disponible pour accéder à la fiche utilisateur :

Prénom	Nom	Date anniversaire	Description	
Fidel	Sapes	26/10/1983	Je suis Fidel SAPE5, mon anniversaire est le 26/10/1983	Voir
Myrtis	Lubowitz	16/02/1999	Je suis Myrtis LUBOWITZ, mon anniversaire est le 16/02/1999	Voir
Shawn	Nienow	25/02/1985	Je suis Shawn NIENOW, mon anniversaire est le 25/02/1985	Voir
Vernie	Brown	17/02/1981	Je suis Vernie BROWN, mon anniversaire est le 17/02/1981	Voir
Destiney	Haley	27/03/1976	Je suis Destiney HALEY, mon anniversaire est le 27/03/1976	Voir
Adele	Nitzsche	29/10/2010	Je suis Adele NITZSCHE, mon anniversaire est le 29/10/2010	Voir
Nathanael	Borer	23/09/1997	Je suis Nathanael BORER, mon anniversaire est le 23/09/1997	Voir
Claudine	Marks	27/09/1970	Je suis Claudine MARKS, mon anniversaire est le 27/09/1970	Voir
Shany	Parisian	23/06/1986	Je suis Shany PARISIAN, mon anniversaire est le 23/06/1986	Voir
Dortha	Witting	09/09/1981	Je suis Dortha WITTING, mon anniversaire est le 09/09/1981	Voir

Utilisation des Forms Symfony

Symfony nous permet, à travers l'utilisation de **Forms**, de mettre très facilement en place des formulaires de création / mise à jour.

Nous verrons ici comment **modifier un utilisateur** à travers un Form que nous créerons.

Installation des Forms

Il nous faudra tout d'abord ajouter le **package form** :

```
composer require symfony/form
```

Création d'un FormType : UserType.php

Un **FormType** nous **permet d'instancier un Form** qui pourra être utilisé plusieurs fois par la suite. Cela évite la redondance de code et donc de créer plusieurs fois le même Form.

Le fichier **UserType.php** est créé dans le dossier **Type** du dossier **Form** (à créer si nécessaire) :

```
<?php
namespace App\Form\Type;

use Symfony\Component\Form\AbstractType;
```

```

use Symfony\Component\Form\Extension\Core\Type\DateType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\FormBuilderInterface;

class UserType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('firstname', TextType::class, ['label' => 'Prénom'])
            ->add('lastname', TextType::class, ['label' => 'Nom'])
            ->add('birthday', DateType::class, [
                'label' => 'Né(e) le',
                'attr' => ['placeholder' => 'dd/mm/yyyy'],
                'widget' => 'single_text',
                'format' => 'dd/MM/yyyy',
                'input' => 'datetime',
                'html5' => false
            ])
            ->add('save', SubmitType::class)
            ->getForm();
    }
}

```

Nous pouvons voir que le Form comporte :

- **firstname** : Champ de type Text pour le prénom de l'utilisateur.
- **lastname** : Champ de type Text pour le nom de l'utilisateur.
- **birthday** : Champ de type Date pour la date d'anniversaire de l'utilisateur.
- **save** : Champ de type Submit pour soumettre le formulaire.

Si vous avez besoin d'un autre type de champ (textarea, number, datetime, ...) ou pour découvrir plus en détails la configuration d'un Form, vous pouvez vous référer à la [documentation](#) Symfony.

Création et affichage du Form Symfony

Nous devons maintenant modifier notre UserController.php pour y ajouter le Form que nous venons de créer.

Le formulaire de modification sera visible dans la fiche utilisateur, donc c'est la **fonction** `getOneUser()` qui doit être modifiée en conséquence :

```
<?php
...
public function getOneUser(UserManager $userManager, int $id)
{
    $user = $userManager->findOneWithDescription($id);

    // Création de notre Form auquel on passe notre objet User.
    $form = $this->createForm(UserType::class, $user);
    // On rajoute le Form au template de la fiche utilisateur pour pouvoir l'afficher
    .
    return $this->render('users/user.html.twig', ['user' => $user, 'form' => $form->
    createView()]);
}
...
```

Côté template Twig, la **nouvelle variable** form passée au template va nous permettre d'afficher le formulaire :

```

{% extends "base.html.twig" %}

{% block title %}Fiche utilisateur{% endblock %}
{% block body %}
<div class="p-5">
<div class="row">
<div class="col">
<h2>Fiche utilisateur</h2>
</div>
<div class="col-auto">
<a class="btn btn-primary" href="{{ path('get-users') }}"> Retour</a>
</div>
</div>

<p>Prénom : {{ user.firstName }}</p>
<p>Nom : {{ user.lastName }}</p>
<p>Né(e) le : {{ user.birthday|date('d/m/Y') }}</p>
<p>Description : {{ user.description }}</p>

{# Ajout du form #}
<h2>Modifier utilisateur</h2>
{{ form(form) }}
</div>
{% endblock %}

```

Nous obtenons un formulaire pré-rempli avec le nom, prénom, date anniversaire de notre utilisateur dans sa fiche :

Fiche utilisateur

[Retour](#)

Prénom : Fidelo

Nom : Sapes

Né(e) le : 26/10/1983

Description : Je suis Fidelo SAPES, mon anniversaire est le 26/10/1983

Modifier utilisateur

Prénom Fidelo

Nom Sapes

Né(e) le 26/10/1983

Enregistrer



Ajout du thème Bootstrap au formulaire

Il serait assez long de rajouter notre style CSS sur notre formulaire mais pas de panique, il est possible d'ajouter automatiquement un thème Bootstrap à tous les formulaires.

Pour cela, il suffit de modifier le fichier **twig.yaml** dans le dossier **packages** :

```
twig:
default_path: '%kernel.project_dir%/templates'
form_themes: ['bootstrap_4_layout.html.twig'] # Ajout du thème Bootstrap
```

Le résultat est désormais un peu plus convaincant :

Fiche utilisateur

[Retour](#)

Prénom : Fidelo

Nom : Sapes

Né(e) le : 26/10/1983

Description : Je suis Fidelo SAPES, mon anniversaire est le 26/10/1983

Modifier utilisateur

Prénom

Nom

Né(e) le



Modification de notre utilisateur

Si nous essayons de cliquer sur le bouton Enregistrer de notre formulaire, nous pouvons constater que la page se recharge sans prendre en compte les modifications dans le formulaire.

Nous allons ajouter une méthode **postUser()** dans notre **UserController.php** qui se chargera de recevoir le formulaire soumis et de sauvegarder l'utilisateur :

```
<?php
/**
 * @Route("/users/{id}", name="post-user", requirements={"id"="\d+"}, methods={"POST"})
 *
 * @param UserManager $userManager
 * @param Request $request
 * @param int $id
 * @return Response
 * @throws Exception
```

```

*/
public function postUser(UserManager $userManager, Request $request, int
$id)
{
    $user = $userManager->findOneWithDescription($id);
    $form = $this->createForm(UserType::class, $user);
    $form->handleRequest($request);

    // Si le formulaire est soumis et valide
    if ($form->isSubmitted() && $form->isValid()) {
        // On récupère l'utilisateur dans le formulaire
        /** @var User $user */
        $user = $form->getData();

        // On sauvegarde l'utilisateur
        $em = $this->getDoctrine()->getManager();
        $em->persist($user);
        $em->flush();
    }

    // On redirige sur la fiche utilisateur
    return $this->redirectToRoute('get-user', ['id' => $user->getId()]);
}

```

Pour que le formulaire soit soumis dans cette fonction, il faut modifier son **action** et sa **méthode** dans le template :

```

...
{# Modification de l'action et de la méthode du form #}
<h2>Modifier utilisateur</h2>
{{ form(form, {'action': path('post-user', {'id': user.id}), 'method': 'POST'}) }}
...

```

Résultat final

Il est maintenant temps de modifier le formulaire d'une fiche utilisateur et de valider les modifications :

The image displays two side-by-side screenshots of a web application's user profile management interface, illustrating the process of updating user information.

Left Screenshot (Initial State):

- Fiche utilisateur:** Prénom : Fidelo, Nom : Sapes, Né(e) le : 26/10/1983, Description : Je suis Fidelo SAPES, mon anniversaire est le 26/10/1983.
- Modifier utilisateur:** Prénom (input: Corentin), Nom (input: Chevret), Né(e) le (input: 18/02/1997).
- Action:** A red arrow points to the "Enregistrer" button with the text "On enregistre nos modifications".

Right Screenshot (Updated State):

- Fiche utilisateur:** Prénom : Corentin, Nom : Chevret, Né(e) le : 18/02/1997, Description : Je suis Corentin CHEVRET, mon anniversaire est le 18/02/1997.
- Modifier utilisateur:** Prénom (input: Corentin), Nom (input: Chevret), Né(e) le (input: 18/02/1997).
- Action:** A red arrow points to the "Retour" button with the text "Les infos utilisateur mises à jour".

Source <https://www.axopen.com/blog/2020/12/symfony-framework-php-structure-demarrage-projet/>