




Symfony #1

| | |
|-------------------------|-------------------------|
| ▼ Version | V1 |
| ≡ Type | Technique |
| 📅 Date de création | @14 décembre 2023 15:31 |
| 📅 Dernière modification | @14 décembre 2023 15:41 |


Pré-requis

Installer le CLI de Symfony

 [documentation installer Symfony CLI](#)

```
curl -sLf 'https://dl.cloudsmith.io/public/symfony/stable/setup.deb.sh'  
sudo -E bash  
$ sudo apt install symfony-cli
```

Installer Composer, le gestionnaire de librairies pour PHP


 [documentation installer Composer](#)

```
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
php -r "if (hash_file('sha384', 'composer-setup.php') === '55ce33d7678c5a611085589f1f3ddf8b3c52d662cd01d4ba75c0ee04599766')
```

```
php composer-setup.php
php -r "unlink('composer-setup.php');"
```

Rendez composer accessible depuis le terminal.

```
sudo mv composer.phar /usr/local/bin/composer
```

 Pour vérifier que votre machine est prêt à installer Symfony utiliser la commande Symfony CLI suivante :

```
symfony check:requirements
```

Installations

 [Documentation installation Symfony](#).

▼ I - Initialiser son projet Symfony

Créer votre projet Symfony grâce à Composer.

```
composer create-project symfony/skeleton:"6.2.*" <le_nom_de_votre_projet>
cd <le_nom_de_votre_projet>
composer require webapp
```

En production, vous devez installer un serveur Web comme Nginx ou Apache et le configurer pour exécuter Symfony. Cependant, pour le développement local, le moyen le plus pratique d'exécuter Symfony consiste à utiliser le serveur Web local fourni par le binaire Symfony. Ce serveur local fournit entre autres le

support de HTTP/2, les requêtes simultanées, TLS/SSL et la génération automatique de certificats de sécurité.

Lancer le serveur local.

```
symfony server:start
```

Ouvrez votre navigateur et accédez à <http://localhost:8000/>. Si tout fonctionne, vous verrez une page de bienvenue. Plus tard, arrêtez le serveur en appuyant sur `Ctrl+C` depuis votre terminal.

▼ II - Installer Webpack et Sass

 [Documentation installation Webpack et SASS](#)

Pour utiliser le SCSS dans l'application Symfony nous avons besoin d'un compilateur, nous utiliserons Webpack. Pour l'installer nous profiterons de Symfony Flex qui se chargera du plus gros de la configuration pour nous.

```
symfony composer req encore
```

Puis nous installerons la librairie Webpack à l'aide de NPM.

```
npm install node-sass sass-loader --save-dev
```

Vous devriez maintenant avoir un dossier `assets` à la racine de votre projet, celui-ci contient :

- un dossier `controllers` qui est utile la librairie Stimulus que nous n'utiliserons pas,
- un dossier `styles` où l'on peut stocker nos fichier CSS / SCSS
- un fichier `app.js` où l'on importera nos fichier SCSS pour que ceux-ci puissent être compilés grâce à Webpack

En plus du dossier `assets` vous devriez retrouver un fichier `webpack.config.js` à la racine de votre projet, celui-ci contient la configuration de Webpack.

Dans le fichier `webpack.config.js` pensez à dé-commenter la ligne de code suivante pour activer la compilation du SCSS : `.enableSassLoader()`

Pour exécuter webpack et ainsi compiler notre code Javascript et SCSS nous devons exécuter la commande suivante, l'argument `--watch` permet que la compilation se fasse automatiquement.

```
npm run watch
```

▼ III - Configurer Webpack et Sass

Vous pouvez d'ores et déjà créer un dossier `js` dans le dossier `assets`, c'est dans celui-ci que nous stockerons nos fichiers JS. Pour que vos fichiers JS soient eux aussi compilés il est nécessaire que vous ajoutiez de la configuration à Webpack, rendez-vous dans le fichier `webpack.config.js`

Vous devriez voir à minima un `.addEntry('app', './assets/app.js')`, c'est le point d'entrée qui fait le lien vers notre fichier `assets/app.js`, nous devons faire de même pour tous les fichiers JS que nous créerons. Par exemple, j'ai ajouté un fichier `assets/js/hello.js`, pour qu'il soit compilé j'ajoute le point d'entrée `.addEntry('hello', './assets/js/hello.js')`.

Le premier argument de `addEntry` est le nom que vous souhaitez donner à votre fichier JS pour l'appeler plus tard, le second le chemin menant au fichier depuis la racine de votre projet.

```
const Encore = require('@symfony/webpack-encore');

...

/*
 * ENTRY CONFIG
 *
 * Each entry will result in one JavaScript file (e.g. app.js)
 * and one CSS file (e.g. app.css) if your JavaScript imports CSS.
 */
.addEntry('app', './assets/app.js')
```

```

        .addEntry('hello', './assets/js/hello.js')

        ...

module.exports = Encore.getWebpackConfig();

```

▼ IV - Exploiter les fichiers JS et SCSS

Rendez-vous dans l'un de vos templates Twig, ici nous utiliserons le `base.html.twig` pour y lier notre fichier `assets/app.js`.

- Pour lier une feuille de style nous utiliserons `{{ encore_entry_link_tags('app') }}` enveloppé dans un bloque `{% block stylesheets %} ... {% endblock %}`.
L'argument 'app' passé à la fonction `encore_entry_link_tags('app')` correspond au nom d'entrée que l'on a défini pour notre fichier dans le `webpack.config.js` précédent.
- Pour lier une feuille de script nous utiliserons `{{ encore_entry_link_tags('app') }}` enveloppé dans un bloque `{% block javascript %} ... {% endblock %}`.
L'argument 'app' passé à la fonction `encore_entry_link_tags('app')` correspond au nom d'entrée que l'on a défini pour notre fichier dans le `webpack.config.js` précédent.

Effectivement nous appelons un fichier Javascript `app` dans notre balise de style car c'est bien celui-ci qui contient tous les imports de nos feuilles de styles SCSS. Ces feuilles de styles SCSS doivent être compilés par Webpack et sont donc rendu accessible par l'intermédiaire du fichier `app`.

 Il sera peut être nécessaire de redémarrer le watcher de Webpack pour que les changements soient pris en compte.

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>{% block title %}Welcome!{% endblock %}</title>

    {% block stylesheets %}
      {{ encore_entry_link_tags('app') }}

```

```

        {% endblock %}

        {% block javascripts %}
            {{ encore_entry_script_tags('app') }}
        {% endblock %}

    </head>
    <body>
        {% block body %}{% endblock %}
    </body>
</html>

```

▼ V - Configurer sa base de données

Les informations de connexion à la base de données sont stockées dans une variable d'environnement appelée `DATABASE_URL`. Pour le développement, vous pouvez trouver et personnaliser ceci dans le fichier `.env`

Dé-commentez la ligne correspondant à votre logiciel de gestion de base de données, dans notre cas MySQL

```

...

###> doctrine/doctrine-bundle ###
# Format described at https://www.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/configuration.html#
# IMPORTANT: You MUST configure your server version, either here or in config/packages/doctrine.yaml
#
# DATABASE_URL="sqlite:///kernel.project_dir%/var/data.db"
DATABASE_URL="mysql://db_username:db_password@127.0.0.1:3306/db_name?serverVersion=8.0.32&charset=utf8mb4"
# DATABASE_URL="mysql://app:!ChangeMe!@127.0.0.1:3306/app?serverVersion=10.11.2-MariaDB&charset=utf8mb4"
# DATABASE_URL="postgresql://app:!ChangeMe!@127.0.0.1:5432/app?serverVersion=15&charset=utf8"
###< doctrine/doctrine-bundle ###

```

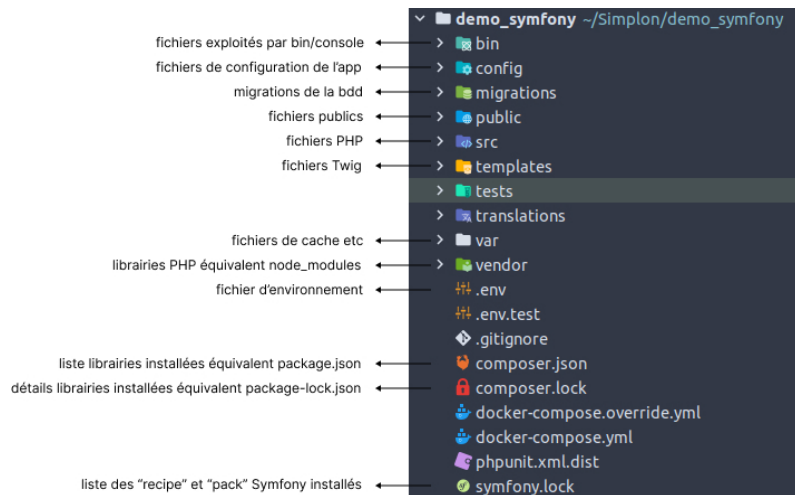
...

Pour vérifier que la configuration est bonne vous pouvez exécuter la commande suivante pour créer votre base de données. Si votre base de données est créée ou si elle ne peut pas l'être car déjà existante cela signifie que la connexion entre Doctrine et votre base de données est réussie.

```
php bin/console doctrine:database:create
```

Pour aller plus loin dans la gestion de votre base de données rendez-vous dans le sous-chapitre "Doctrine et la base de données" du chapitre "Symfony"

Architecture d'un projet



Architecture d'un projet Symfony vierge

Symfony est un framework PHP open-source largement utilisé pour le développement d'applications web. Il offre une structure solide et modulaire pour construire des projets PHP de manière efficace. **Symfony suit le modèle de conception MVC (Modèle-Vue-Contrôleur)** et favorise les bonnes pratiques de développement.

Le modèle MVC

Le modèle MVC (Modèle-Vue-Contrôleur) est un modèle architectural couramment utilisé dans le développement d'applications web et logicielles. **Il permet de séparer la logique métier, la présentation et la gestion des interactions utilisateur** au sein d'une application.

Vue

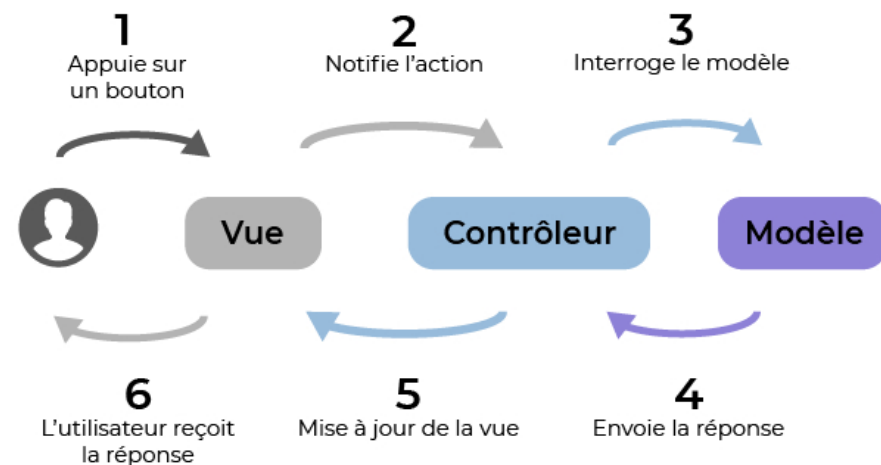
La vue est responsable de l'affichage des données au sein de l'interface utilisateur. Elle peut être utilisée pour l'affichage, la saisie ou la visualisation des données. La vue est généralement passive et n'effectue aucune logique métier significative.

Contrôleur

Le contrôleur gère les interactions utilisateur et les événements. Il reçoit les requêtes de l'utilisateur, interagit avec le modèle en récupérant les données nécessaires et choisit la vue appropriée pour afficher les résultats. Le contrôleur agit comme un médiateur entre la vue et le modèle, coordonnant les actions et garantissant la cohérence de l'application.

Modèle

Le modèle représente la logique métier et les données de l'application. Il gère la manipulation des données, les règles métier et les opérations de persistance.



Symfony Flex



Symfony Flex est un outil puissant pour simplifier l'installation et la configuration de packages dans les applications Symfony. Il utilise des "recipes" pour automatiser les tâches courantes et permet également l'utilisation de "packs" pour installer plusieurs dépendances en une seule fois.

 [Documentation Symfony Flex](#)

Lors du développement d'applications Symfony, il est courant d'installer des packages (appelés bundles) qui fournissent des fonctionnalités prêtes à l'emploi.

Cependant, l'installation et la configuration de ces packages peuvent être fastidieuses. Symfony propose une solution appelée **Symfony Flex**, qui est un outil permettant de simplifier l'installation et la suppression de packages dans les applications Symfony. **Symfony Flex est un plugin de Composer** qui est installé par défaut lors de la création d'une nouvelle application Symfony et qui **automatise les tâches les plus courantes des applications Symfony**.

Symfony Flex modifie le comportement des commandes require, update et remove de Composer pour fournir des fonctionnalités avancées. Par exemple, si vous exécutez la commande `composer require logger` dans une application Symfony sans Flex, vous obtiendrez une erreur de Composer indiquant que "logger" n'est pas un nom de package valide. Cependant, si l'application a Symfony Flex installé, cette commande **installe et active tous les packages nécessaires pour utiliser le logger officiel de Symfony**. Cela est possible grâce aux "recipes" de Symfony Flex. **Les "recipes" sont un ensemble d'instructions automatisées permettant d'installer et d'activer des packages** dans les applications Symfony. Symfony Flex conserve une trace des "recipes" qu'il a installés dans un fichier `symfony.lock`, qui doit être ajouté à votre dépôt de code.

Symfony propose également des "packs", qui sont des **méta-packages Composer comprenant plusieurs dépendances**. Par exemple, pour ajouter des fonctionnalités de débogage à votre application, vous pouvez exécuter la commande `composer require --dev debug`. Cela installe le "pack" symfony/debug-pack, qui à son tour installe plusieurs packages tels que symfony/debug-bundle, symfony/monolog-bundle, symfony/var-dumper, etc. Vous ne verrez pas la dépendance symfony/debug-pack dans votre fichier composer.json, car Flex décompresse automatiquement le "pack". Cela signifie qu'il n'ajoute que les packages réels en tant que dépendances (par exemple, vous verrez un nouveau symfony/var-dumper dans require-dev).



Symfony Maker

 [Documentation Symfony Maker](#)

Le Maker Bundle de Symfony est un composant additionnel du framework Symfony qui facilite et accélère le processus de création et de génération de code dans une application Symfony. Il fournit des commandes en ligne pour **générer automatiquement du code boilerplate** et éviter d'avoir à le taper manuellement. Il permet de **créer rapidement des éléments tels que des entités, des contrôleurs, des formulaires**, des tests unitaires, des gabarits de vue et d'autres artefacts couramment utilisés dans le développement d'applications web.

Liste des maker

```
php bin/console list make
```

Les contrôleurs

 [Document Symfony contrôleur](#)

Un contrôleur est une fonction PHP qui lit les informations de l'objet `Request` et crée et renvoie un objet `Response`. La réponse peut être une page HTML, JSON, XML, un téléchargement de fichier, une redirection, une erreur 404 ou autre.

Le contrôleur exécute la logique arbitraire dont notre application a besoin pour restituer le contenu d'une page.

Pour cet exemple nous créerons un contrôleur retournant une page HTML (à vrai dire, un template Twig mais cela revient au même). Pour cela nous utiliserons le maker bundle de Symfony pour nous générer le contrôleur ainsi que le template Twig associé.

Nous souhaitons créer une page `homepage`, pour respecter l'architecture MVC nous allons donc avoir besoin d'une vue (template Twig) et d'un contrôleur.

Pour nous faciliter la tâche nous utiliserons le Maker bundle

```
php bin/console make:controller
```

Le maker-bundle nous a généré un contrôleur situé dans `src/controller` et un template Twig situé dans `templates/homepage`. La page est accessible à l'URL suivante : <http://127.0.0.1:8000/homepage>.

Le contrôleur `HomepageController` a bien été créé et celui-ci a reçu l'attribut `Route()` pour spécifier la route qui appellera ce contrôleur.

La fonction `index` utilise la méthode `render` mise à disposition par la classe `AbstractController` que notre contrôleur étend. La méthode `render` indique le template (la "vue" dans notre système MVC) à utiliser.

```
<?php

namespace App\Controller;

use Symfony\Bridge\Twig\Attribute\Template;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class HomepageController extends AbstractController
{
    #[Route('/homepage', name: 'app_homepage')]
    public function index(): Response
    {
        # Il est possible de passer un tableau de variables à votre template en second argument de la fonction render
        return $this->render('homepage/index.html.twig', [
            'controller_name' => 'HomepageController',
        ]);
    }

    # Alternative possible, écrire le chemin vers le template en attribut de la route
```

```

        #[Route('/homepage', name: 'app_homepage')]
        #[Template('/homepage/index.html.twig')]
        public function index(): Response
        {
            # Puisque nous avons indiqué la route via un attribut nous n'avons pas retourner un tableau de variable à nous
            return [
                'controller_name' => 'HomepageController',
            ]
        }
    }
}

```

Le template `homepage/index.html.twig` a bien été créé (la majorité du code peut être supprimé) et se charge de l'affichage lorsqu'un utilisateur se rend sur la route [homepage](#).


La variable `controller_name` qui a été passé à mon template est bien utilisé dans le `<h1>`, voir

```

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

{% block title %}Hello HomepageController!{% endblock %}

{% block body %}
<style>
    .example-wrapper { margin: 1em auto; max-width: 800px; width: 95%; font: 18px/1.5 sans-serif; }
    .example-wrapper code { background: #F5F5F5; padding: 2px 6px; }
</style>

<div class="example-wrapper">
    <h1>Hello {{ controller_name }}! </h1>

```

```

    This friendly message is coming from:
    <ul>
        <li>Your controller at <code><a href="{{ '/home/kevin/Simplon/demo_symfony/src/Controller/HomepageController.php' | file_exists }}">{{ '/home/kevin/Simplon/demo_symfony/src/Controller/HomepageController.php' | file_exists }}</a>
        <li>Your template at <code><a href="{{ '/home/kevin/Simplon/demo_symfony/templates/homepage/index.html.twig' | file_exists }}">{{ '/home/kevin/Simplon/demo_symfony/templates/homepage/index.html.twig' | file_exists }}</a>
    </ul>
</div>
{% endblock %}

```

Si votre contrôleur est utile à plusieurs routes vous pouvez définir un attribut `Route` au contrôleur pour ne pas avoir à le répéter pour chaque une des routes.

```

<?php

namespace App\Controller;

use Symfony\Bridge\Twig\Attribute\Template;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

# La base de toutes les routes du contrôleur sera "/library"
#[Route('/library')]
class LibraryController extends AbstractController
{
    # Cette route correspond à "/library/news" sans avoir à re-indiquer "/library"
    #[Route('/news', name: 'app_library_news')]
    public function news(): Response
    {
        ...
    }
}

```


```

    # Cette route correspond à "/library/upcomming" sans avoir à re-indiquer "/library"
    #[Route('/upcomming', name: 'app_library_upcomming')]
    #[Template('/homepage/index.html.twig')]
    public function upcomming(): Response
    {
        ...
    }
}

```

Doctrine et la base de données

Pour la gestion de bases de données Symfony utilise **Doctrine** qui est un **ensemble de librairie PHP facilitant la gestion de bases de données**. Doctrine est un ORM (Object-Relational Mapping) qui s'intègre parfaitement avec Symfony. Il facilite la gestion de la persistance des données en offrant une couche d'abstraction entre la base de données et l'application. **Doctrine permet de travailler avec la base de données en utilisant des objets PHP plutôt que des requêtes SQL** directes, ce qui simplifie le développement et améliore la portabilité de l'application. Ces outils prennent en charge les bases de données relationnelles telles que MySQL et PostgreSQL, ainsi que les bases de données NoSQL telles que MongoDB.

 Pensez à vérifier le fichier `.env` et assurez-vous que la `DATABASE_URL` est bien configuré :

`app` vert → le nom de votre utilisateur SQL

`!ChangeMe!` bleu → le mot de passe de votre utilisateur SQL

`app` marron → le nom souhaité pour la base de données

```

DATABASE_URL="mysql://app:!ChangeMe!@127.0.0.1:3306/app?serverVersion=8.0.32&charset=utf8mb4"
DATABASE_URL="mysql://root:root@127.0.0.1:3306/demo_bdd?serverVersion=8.0.32&charset=utf8mb4"

```

▼ Liste des commandes Doctrine

| Commande | Raccourcis | Description |
|----------|------------|-------------|
|----------|------------|-------------|

| | | |
|---|--|--|
| <code>php bin/console list doctrine</code> | | Liste toutes les commandes disponibles pour Doctrine |
| <code>php bin/console doctrine:database:drop --force</code> | <code>php bin/console d:d:d --force</code> | Supprime la base de données |
| <code>php bin/console doctrine:database:create</code> | <code>php bin/console d:d:c</code> | Créer la base de données |
| <code>php bin/console make:entity</code> | | Créer une entité |
| <code>php bin/console make:migration</code> | <code>php bin/console make:mig</code> | Créer une migration |
| <code>php bin/console doctrine:migrations:migrate</code> | <code>php bin/console d:m:m</code> | Exécute une migration |

▼ I - Créer une entité

i pré-requis : avoir créer votre base de données, voir Installations chapitre “Installations”.

Créer une entité à l'aide de la commande suivante permet de générer un objet PHP représentant l'entité, avec ses champs, getter et setter. Le fichier généré se trouvera dans le dossier `src/Entity` et l'on retrouvera aussi son “repository” dans le dossier `src/Repository`.

```
php bin/console make:entity # Puis suivre le prompt de la commande pour indiquer les champs de notre entités
```

L'objet PHP qui a été généré par la commande, on y retrouve les champs définis ainsi que leurs getter et setter.

```
<?php

namespace App\Entity;

use App\Repository\FamilyRepository;
use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity(repositoryClass: FamilyRepository::class)]
class Family
{
    #[ORM\Id]
```

```
#[ORM\GeneratedValue]
#[ORM\Column]
private ?int $id = null;

#[ORM\Column(length: 50)]
private ?string $name = null;

#[ORM\Column(length: 255)]
private ?string $description = null;

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

public function getName(): ?string
{
    return $this->name;
}

public function setName(string $name): self
{
    $this->name = $name;

    return $this;
}

public function getDescription(): ?string
{
    return $this->description;
}
```



```
public function setDescription(string $description): self
{
    $this->description = $description;

    return $this;
}
```

La prochain étape vise à générer une migration Doctrine pour créer une table correspondante à notre entité en base de données.

▼ II - Modifier une entité

Utiliser la même commande que pour créer une entité, lors du premier prompt de la commande indiquer le nom d'une entité déjà existante pour la modifier (seul l'ajout de champ est possible, si vous souhaitez supprimer ou modifier un champ existant vous devrez éditer le fichier de l'entité depuis votre IDE puis créer une nouvelle migration)

```
php bin/console make:entity # Puis suivre le prompt de la commande pour indiquer les champs de notre entités
```

▼ III - Persister une entité en base de données

La commande suivante va générer un script SQL pour créer une nouvelle table dans notre base de données, cette table représentera notre entité `Family` fraîchement créée.

```
php bin/console make:migration
```

Le fichier généré se trouvera dans `migrations` et contiendra les requêtes SQL nécessaire à créer la nouvelle table. La commande suivante va persister cette modification de notre base de données.

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

Après l'exécution de la migration je retrouve bien une table `Family` reprenant la même structure que mon object PHP.

| | # | Nom | Type | Interclassement | Attributs | Null | Valeur par défaut | Commentaires | Extra |
|--------------------------|---|--|--------------|--------------------|-----------|------|-------------------|--------------|----------------|
| <input type="checkbox"/> | 1 | id  | int | | | Non | Aucun(e) | | AUTO_INCREMENT |
| <input type="checkbox"/> | 2 | name | varchar(50) | utf8mb4_unicode_ci | | Non | Aucun(e) | | |
| <input type="checkbox"/> | 3 | description | varchar(255) | utf8mb4_unicode_ci | | Non | Aucun(e) | | |

▼ IV - À propos des migrations Doctrine

Les migrations représentent une modification dans la base de données, elles contiennent le code SQL nécessaire à l'application de ses modifications dans la base de données.

Ces migrations représentent en quelque sorte **l'historique de l'évolution la base de données, elles ne doivent en aucun cas être modifiés** au risque de rendre la base de données inexploitable.

Cela ne vous empêche pas d'administrer vos migrations pour assurer la qualité de votre projet ! Imaginons le cas suivant :

Vous travaillez sur une branche Git et créez une nouvelle entité, générez une migration et l'exécutez avant de vous rendre compte que vous avez oublié un champ ... Vous ne désirez pas créer une nouvelle migration uniquement pour ce champ par soucis de qualité / propreté. Il est possible de procéder comme suit :

1. Supprimer le fichier de migration fraîchement créé
2. Supprimer votre base de données (be careful !)
3. Créer votre base de données et exécuter les migrations, vous retrouvez donc une base de données "propre" ne contenant pas votre entité bogué
4. Modifier votre entité

5. Créer une nouvelle migration et l'exécuter

L'héritage PHP

▼ Extend

En PHP, le mot-clé "extends" est utilisé pour indiquer l'héritage d'une classe à partir d'une autre classe, c'est-à-dire l'extension d'une classe existante. Lorsqu'une classe hérite d'une autre classe à l'aide de "extends", elle hérite de toutes les propriétés et méthodes de la classe parente, et peut également ajouter ses propres propriétés et méthodes ou les modifier.

```
phpCopy code
class ClasseParente {
    // Propriétés et méthodes de la classe parente
}

class ClasseEnfant extends ClasseParente {
    // Propriétés et méthodes de la classe enfant
}
```

▼ Implement

En PHP, le mot-clé "implements" est utilisé pour indiquer qu'une classe implémente une ou plusieurs interfaces. Une interface est une sorte de contrat qui spécifie un ensemble de méthodes qu'une classe qui l'implémente doit obligatoirement définir. Une classe peut implémenter plusieurs interfaces en les séparant par des virgules.

```
phpCopy code
interface InterfaceExemple {
    // Méthodes de l'interface
}

class MaClasse implements InterfaceExemple {
```

```
// Implémentation des méthodes de l'interface  
}
```

L'injection de dépendance

L'injection de dépendance consiste à fournir les dépendances requises à une classe à partir de l'extérieur, permettant ainsi une meilleure séparation des responsabilités, une plus grande modularité et une réutilisation du code.

Imaginons que nous avons créé un service nous mettant à disposition plusieurs méthodes pour réaliser une quelconque logique, ici un service `src/Service/MonSuperService` qui nous met à disposition des méthodes pour faire des calculs :

```
<?php  
  
namespace App\Service;  
  
class MonSuperService  
{  
  
    public function addition()  
    {  
        ...  
    }  
  
    public function subtraction()  
    {  
        ...  
    }  
  
    public function division()
```

```

    {
        ...
    }
}

```

Nous avons mis toutes nos méthodes de calcul dans ce fichier pour “centraliser” notre code et ensuite utiliser ce service où l’on souhaite dans l’application.

Dans notre exemple nous souhaitons utiliser le service `MonSuperService` dans notre contrôleur `MonSuperController`, pour que le service `MonSuperService` soit disponible dans le contrôleur nous allons devoir y injecter notre service. Cela revient à indiquer à Symfony que notre contrôleur `MonSuperController` est dépendant du service `MonSuperService` car il l’utilise, et par conséquent Symfony doit injecter le service dans le contrôleur lorsque celui-ci est utilisé.

```

<?php

namespace App\Controller;

use App\Service\MonSuperService;
...

class MonSuperController extends AbstractController
{
    $private MonSuperService $monSuperService;

    public function __construct(MonSuperService $monSuperService)
    {
        $this->monSuperService = $monSuperService;
    }

    #[Route('/super/route', name: 'app_super_route')]
    public function index()

```

```

{
    $this->monSuperService->addition();
    ...
}

```

```

<?php

namespace App\Controller;

use App\Service\MonSuperService; ← Importer le service
...

class MonSuperController extends AbstractController
{
    $private MonSuperService $monSuperService; ← Déclarer une propriété privée pour contenir le service

    public function __construct(MonSuperService $monSuperService) ← Indiquer le service comme argument de la méthode
    {                                     construct pour que Symfony injecte le service
        $this->monSuperService = $monSuperService; ← Assigner le service injecté par Symfony dans la propriété prévue
    }

    #[Route('/super/route', name: 'app_super_route')]
    public function index()
    {
        $this->monSuperService->addition(); ← Le service est accessible dans tous le contrôleur
        ...
    }
}

```

Les fixtures

 [Documentation librairie Faker](#)

Les fixtures sont des données de test pré-définies qui sont utilisées pour peupler la base de données lors du développement, du test ou de la démonstration de votre application.

▼ Installer DoctrineFixturesBundle

 [Documentation](#)

L'installation est très simple grâce à Symfony Flex faisant le travail pour nous. Un dossier `src/DataFixtures` sera généré automatiquement.

```
composer require --dev orm-fixtures
```

Pour produire des fixtures réalistes sans y passer des heures nous exploiterons la librairie Faker.

```
composer require fakerphp/faker
```

▼ Créer des fixtures sans relation(s)

Puisque nous aurons besoin de la librairie Faker dans chacun de nos fichier de fixtures nous commencerons par créer une classe abstraite contenant mettant à disposition la librairie. Par la suite nous étendrons cette classe abstraite pour bénéficier de la librairie Faker sans avoir à l'importer de nouveau.

```
# src/DataFixtures/AbstractFixture.php

<?php
```

```

namespace App\DataFixtures;

use Doctrine\Bundle\FixturesBundle\Fixture;
use Faker\Factory;
use Faker\Generator;
use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;

abstract class AbstractFixtures extends Fixture
{
    protected Generator $faker;
    protected UserPasswordHasherInterface $passwordHasher;

    public function __construct(UserPasswordHasherInterface $passwordHasher)
    {
        $this->faker = Factory::create('fr_FR');
        $this->passwordHasher = $passwordHasher;
    }
}

```

Dès lors, lorsque nous créerons un nouveau fichier destiné à créer des fixtures nous étendrons la classe abstraite que nous avons créer pour bénéficier de la librairie Faker.

```

<?php

namespace App\DataFixtures;

use App\Entity\Product;
use Doctrine\Persistence\ObjectManager;

```



```

class ProductFixtures extends AbstractFixture
{
    // La méthode "load" est imposé par la classe Fixture que la classe AbstractFixture étend
    // C'est cette méthode qui permet de créer des fixtures
    public function load(ObjectManager $manager)
    {
        // Une boucle de 10 pour générer 10 produits
        for ($i = 0; $i < 10; $i++) {

            // Instancie un objet Product avec un nom
            $product = new Product();
            $product->setName($this->faker->word());

            // Enregistre le produit fraîchement créé, à faire à chaque tour de boucle
            $manager->persist($product);
        }

        // Une fois la boucle terminée je persiste les produits fraîchement créés
        $manager->flush();
    }
}

```

▼ Créer des fixtures avec relation(s)

Pour créer des fixtures (fausses entités) faisant référence à une/plusieurs fixture(s) (autre fausses entités) il est nécessaire que votre classe implémente la classe `DependentFixtureInterface`.

```

<?php

namespace App\DataFixtures;

use App\Entity\Product;

```

```

use Doctrine\Persistence\ObjectManager;
use Doctrine\Common\DataFixtures\DependentFixtureInterface;

class carFixtures extends AbstractFixture implements DependentFixtureInterface
{
    public function load(ObjectManager $manager)
    {
        ...
    }

    // La méthode "getDependencies" est imposé par la classe DependentFixtureInterface que l'on implémente
    // C'est cette méthode qui indique les fixtures dont aura besoin notre fichiers
    public function getDependencies()
    {
        ...
    }
}

```

Nous reviendrons à ce fichier plus tard, il nous faut d'abord implémenter du code dans le fichier de fixtures dont nous sommes dépendant.

La classe `Fixture` que notre classe `AbstractFixture` étend nous met à disposition la méthode `setReference()` qui va nous permettre de "référencer" nos fixtures pour les exploiter dans d'autres fichiers de fixtures, comme nous souhaitons le faire pour nos produits.

```

<?php

namespace App\DataFixtures;

use App\Entity\Category;
use Doctrine\Persistence\ObjectManager;

```

```

class CategoryFixtures extends AbstractFixture
{
    public function load(ObjectManager $manager)
    {
        for ($i = 0; $i < 10; $i++) {
            $category = new Category();
            $category->setName($this->faker->word());

            $manager->persist($category);

            // Le premier argument est le nom que l'on souhaite donner comme référence à la fixture créée
            // Ici j'utilise une string concaténée avec l'index de ma boucle pour avoir un nom unique.
            // Le second argument est l'entité à laquelle on fait référence
            $this->setReference('category_' . $i, $category);
        }

        $manager->flush();
    }
}

```

Dorénavant je pourrais faire référence à 10 catégories différentes (10 = le nombre de tour de ma boucle).

La première d'entre elle est référencée comme `category_0`, la seconde `category_1`, et ainsi de suite jusqu'à `category_9`, donc un total de 10 catégories.

Nous avons créé une référence pour chacune des fixtures, nous pouvons maintenant les exploiter dans un fichier de fixtures différent, ici les fixtures de nos produits.

```

<?php

namespace App\DataFixtures;

```

```

use App\Entity\Product;
use Doctrine\Persistence\ObjectManager;
use Doctrine\Common\DataFixtures\DependentFixtureInterface;

class ProductFixtures extends AbstractFixture implements DependentFixtureInterface
{
    public function load(ObjectManager $manager)
    {
        ...
    }

    // J'indique à la méthode getDependencies une dépendance à la classe de fixture CategoryFixtures
    public function getDependencies()
    {
        return [
            CategoryFixtures::class,
        ];
    }
}

```

Pour définir la relation entre mon produit et sa catégorie j'utilise la méthode `getReference()` elle aussi mise à disposition par la classe `Fixture`.

La méthode `getReference()` prend comme unique paramètre le nom de référence d'une fixture, ici nous avons à disposition nos `category_0`, `category_1`, etc. Nous souhaitons que le produit soit en relation avec l'une de nos 10 catégories, pour cela nous allons devoir modifier dynamiquement le chiffre après le mot `category_`.

Pour cela nous allons prendre avantage de la librairie Faker qui nous met à disposition une méthode `numberBetween()` à laquelle on indique les nombres entre lesquels piocher, ici `numberBetween(0, 9)` car nous avons 10 catégories référencées. En concaténant le résultat de la méthode avec le mot `category_` nous obtiendrons un catégorie pioché au hasard parmi nos 10 possibilités `$product->setCategory($this->getReference("category_" . $this->faker->numberBetween(0, 9)))`

```

<?php

namespace App\DataFixtures;

use App\Entity\Product;
use Doctrine\Persistence\ObjectManager;
use Doctrine\Common\DataFixtures\DependentFixtureInterface;

class ProductFixtures extends AbstractFixture implements DependentFixtureInterface
{
    public function load(ObjectManager $manager)
    {
        for ($i = 0; $i < 10; $i++) {

            $product = new Product();
            $product->setName($this->faker->word());

            // Pour définir la catégorie en relation avec mon produit j'utilise la méthode getReference
            $product->setCategory($this->getReference("category_" . $this->faker->numberBetween(0, 9)))

            $manager->persist($product);
        }

        $manager->flush();
    }

    public function getDependencies()
    {
        return [
            CategoryFixtures::class,
        ];
    }
}

```

```

    ];
}
}

```

▼ Exécuter ses fixtures

Un prompt vous demandera de confirmer l'action, en effet exécuter vos fixtures supprimera toutes les données existantes dans votre base de données si vous n'utilisez pas l'attribut `--append`.

```

php bin/console doctrine:fixtures:load # supprimera toutes les données enregistrées
php bin/console doctrine:fixtures:load -- append # ne supprimera pas les données enregistrées

```

🔍 CRUD des entités

i **Pré-requis** : avoir créer une base de données avec au minima une entité.

▼ **i** Dans la presentation suivante nous travaillerons dans un contrôleur random pour l'exemple, **déplier pour voir le code**.

```

<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class ProductsController extends AbstractController
{
    #[Route('/products', name: 'app_products')]
    public function index(): Response
    {

```

```

        return $this->render('products/index.html.twig');
    }
}

```

Pour administrer une entité (CRUD : Create Read Update Delete) Symfony nous met à disposition la classe `EntityManager`, elle est une composante clé de Symfony et de Doctrine. **Son rôle principal est de gérer le cycle de vie des entités, de les persister, de les récupérer depuis la base de données et de les synchroniser avec cette dernière.**

▼ Exploiter l'entity manager

Grâce à l'injection de dépendance de Symfony il suffit d'indiquer la classe `EntityManager` comme paramètre de notre fonction pour avoir accès à cette classe dans notre contrôleur.

```

<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class ProductsController extends AbstractController
{
    #[Route('/products', name: 'app_products')]

    // Ma fonction index prend comme paramètre le EntityManagerInterface
    public function index(EntityManagerInterface $entityManager): Response
    {
        return $this->render('products/index.html.twig');
    }
}

```

```
}  
}
```

Cette dernière va nous permettre d'accéder au `repository` de la classe que l'on souhaite administrer

La classe `EntityManager` permet de récupérer des entités à partir de la base de données en utilisant des méthodes telles que `find()`, `findOneBy()`, `findAll()` et `findBy()`. Elle nous permettra aussi d'enregistrer de nouvelles entités ou celles qui ont été modifiées. Ces entités sont ensuite stockées dans la base de données lors de l'appelle de la méthode `flush()`.

```
<?php  
  
namespace App\Controller;  
  
use App\Entity\Product;  
use Doctrine\ORM\EntityManagerInterface;  
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;  
use Symfony\Component\HttpFoundation\Response;  
use Symfony\Component\Routing\Annotation\Route;  
  
class ProductsController extends AbstractController  
{  
    #[Route('/products', name: 'app_products')]  
    public function index(EntityManagerInterface $entityManager): Response  
    {  
        // J'accède au repertoire de la classe Product grâce au EntityManagerInterface  
        $productRepository = $entityManager->getRepository(Product::class);  
  
        return $this->render('products/index.html.twig');  
    }  
}
```


▼ CREATE

```
<?php

namespace App\Controller;

...

class ProductsController extends AbstractController
{
    #[Route('/product/create', name: 'app_product_create')]
    public function index(EntityManagerInterface $entityManager): Response
    {
        // Créer une instance d'un produit et insert les données nécessaires
        $product = new Product();
        $product->setName('Coca-Cola');
        $product->setPrice(1.99);
        $product->setDescription('Bien frais chacal');

        // Indique à Doctrine que vous souhaitez potentiellement sauvegarder ce produit
        $entityManager->persist($product);

        // Exécute la query SQL pour insérer les objets persistés en base de données
        $entityManager->flush();

        return $this->render('product_create/index.html.twig');
    }
}
```

▼ READ

```

<?php

namespace App\Controller;

...

class ProductsController extends AbstractController
{
    #[Route('/products', name: 'app_products')]

    // Injecter le EntityManagerInterface comme paramètre de la fonction
    public function index(EntityManagerInterface $entityManager): Response
    {
        // Récupérer le répertoire de l'entité Product
        $repository = $entityManager->getRepository(Product::class);

        // Cherche un produit grâce à sa primary key ( couramment "id" )
        $product = $repository->find($id);

        // Cherche un produit grâce à son nom
        $product = $repository->findOneBy(['name' => 'Keyboard']);

        // Cherche un produit grâce à son nom et son prix
        $product = $repository->findOneBy([
            'name' => 'Keyboard',
            'price' => 1999,
        ]);

        // Cherche plusieurs produits avec un nom correspondant, trier par prix
        $products = $repository->findBy(

```

```

        ['name' => 'Keyboard'],
        ['price' => 'ASC']
    );

    // Cherche tous les produits
    $products = $repository->findAll();

    return $this->render('products/index.html.twig');
}
}

```

▼ UPDATE

```

<?php

namespace App\Controller;

...

class ProductsController extends AbstractController
{
    #[Route('/product/update/{id}', name: 'app_product_update')]
    public function index(EntityManagerInterface $entityManager, int $id): Response
    {
        // Récupérer le répertoire de l'entité Product
        $repository = $entityManager->getRepository(Product::class);

        // Cherche un produit grâce à sa primary key
        // La variable $id est issue du paramètre de l'url, voir l'attribut Route de la fonction
        $product = $repository->find($id);
    }
}

```

```

        // On vérifie que l'on a bien récupéré un produit en base de données,
        // si ce n'est pas le cas il n'y a pas de produit à modifier et l'on retourne une erreur à l'util
        if (!$product) {
            throw $this->createNotFoundException(
                'No product found for id '.$id
            );
        }

        // Donne un nouveau nom à notre produit
        $product->setName('New product name!');

        // Enregistre et persiste les changements
        $entityManager->persist($product)
        $entityManager->flush();

        return $this->render('product_update/index.html.twig');
    }
}

```

▼ DELETE

```

<?php

namespace App\Controller;

...

class ProductsController extends AbstractController
{
    #[Route('/product/delete/{id}', name: 'app_product_delete')]
    public function index(EntityManagerInterface $entityManager, int $id): Response
    {
    }
}

```

```

{
    // Récupérer le répertoire de l'entité Product
    $repository = $entityManager->getRepository(Product::class);

    // Cherche un produit grâce à sa primary key
    // La variable $id est issue du paramètre de l'url, voir l'attribut Route de la fonction
    $product = $repository->find($id);

    // On vérifie que l'on a bien récupéré un produit en base de données,
    // si ce n'est pas le cas il n'y a pas de produit à modifier et l'on retourne une erreur à l'util
    if (!$product) {
        throw $this->createNotFoundException(
            'No product found for id '.$id
        );
    }

    // Supprime le produit et persiste les changements
    $entityManager->remove($product);
    $entityManager->flush();

    return $this->render('product_delete/index.html.twig');
}
}

```