

# Symfony

L'aide en ligne sur ce framework, excellente, est la référence incontournable et à jour.

## Installation et configuration de Symfony

Pour créer une nouvelle application Symfony, s'assurer que les étapes précédentes ont été respectées. Ensuite, exécuter la commande suivante:

```
composer create-project symfony/skeleton:"6.2.*" monProjet
```

```

mihaela@earth:~/Bureau$ composer create-project symfony/skeleton:"6.2.*" monProjet
Creating a "symfony/skeleton:6.2.*" project at "./monProjet"
Deprecation Notice: Using ${var} in strings is deprecated, use {$var} instead in
Deprecation Notice: Using ${var} in strings is deprecated, use {$var} instead in
Installing symfony/skeleton (v6.2.99)
  - Downloading symfony/skeleton (v6.2.99)
  - Installing symfony/skeleton (v6.2.99): Extracting archive
Created project in /home/mihaela/Bureau/monProjet
Loading composer repositories with package information
Updating dependencies
Lock file operations: 1 install, 0 updates, 0 removals
  - Locking symfony/flex (v2.2.5)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
  - Installing symfony/flex (v2.2.5): Extracting archive
Generating autoload files
1 package you are using is looking for funding.
Use the `composer fund` command to find out more!

Run composer recipes at any time to see the status of your Symfony recipes.

Loading composer repositories with package information
Restricting packages listed in "symfony/symfony" to "6.2.*"
Updating dependencies
Lock file operations: 29 installs, 0 updates, 0 removals
  - Locking psr/cache (3.0.0)
  - Locking psr/container (2.0.2)
  - Locking psr/event-dispatcher (1.0.0)
  - Locking psr/log (3.0.0)
  - Locking symfony/cache (v6.2.8)
  - Locking symfony/cache-contracts (v3.2.1)
  - Locking symfony/config (v6.2.7)
  - Locking symfony/console (v6.2.8)
  - Locking symfony/dependency-injection (v6.2.8)
  - Locking symfony/deprecation-contracts (v3.2.1)
  - Locking symfony/dotenv (v6.2.8)
  - Locking symfony/error-handler (v6.2.7)
  - Locking symfony/event-dispatcher (v6.2.8)
  - Locking symfony/event-dispatcher-contracts (v3.2.1)
  - Locking symfony/filesystem (v6.2.7)
  - Locking symfony/finder (v6.2.7)
  - Locking symfony/framework-bundle (v6.2.8)
  - Locking symfony/http-foundation (v6.2.8)
  - Locking symfony/http-kernel (v6.2.8)
  - Locking symfony/polyfill-intl-grapheme (v1.27.0)
  - Locking symfony/polyfill-intl-normalizer (v1.27.0)
  - Locking symfony/polyfill-mbstring (v1.27.0)

```

Cette commande va créer un nouveau répertoire monProjet, y télécharger des dépendances et même générer les répertoires et fichiers de base nécessaires pour commencer.

Si on ne précise pas la version (ici v6.2.\*), Composer prendra automatiquement la dernière version de Symfony.

Ensuite, positionnez-vous dans le repertoire du projet et entrez la commande

```
composer require webapp
```

```

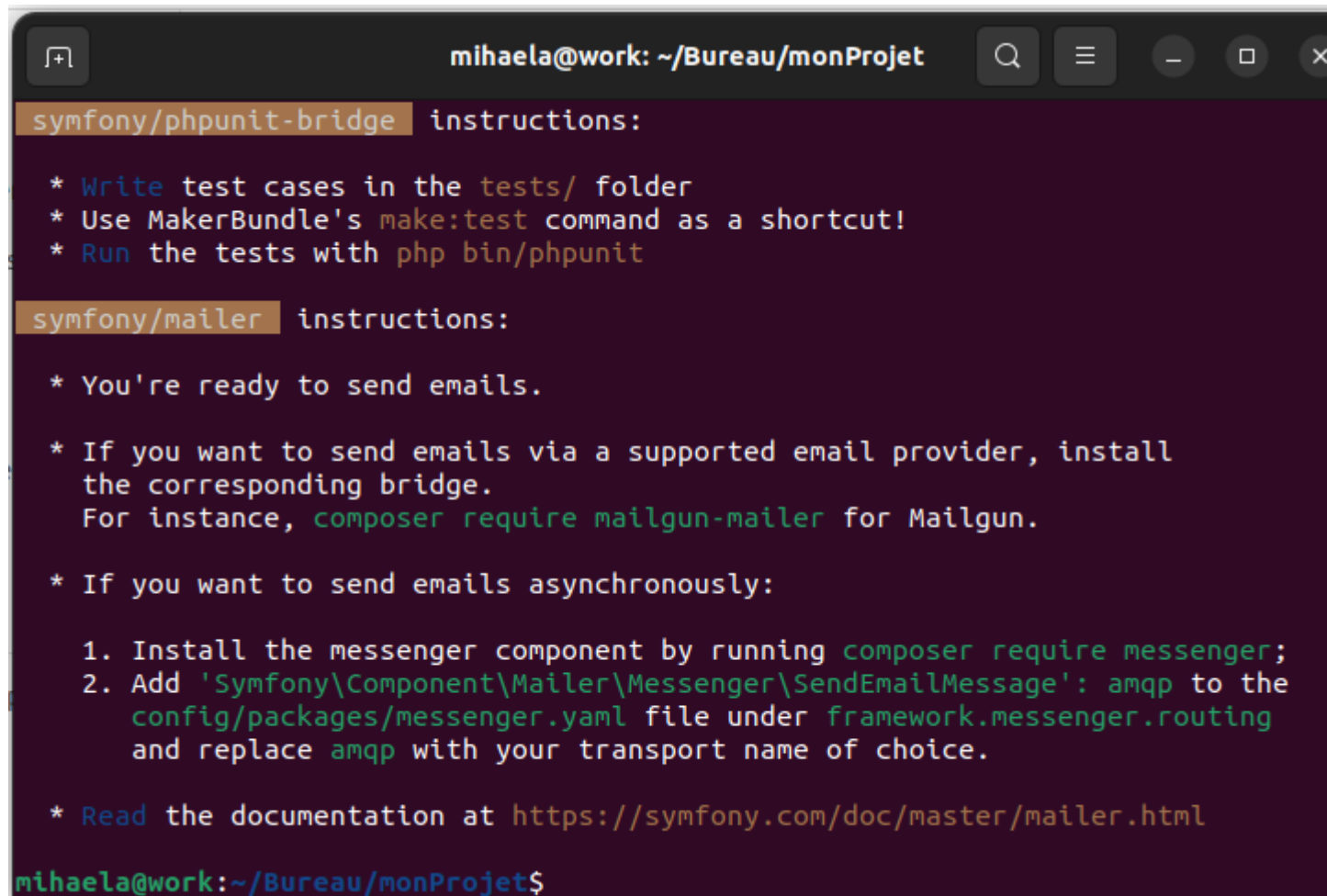
mihaela@earth:~/Bureau/monProjet$ composer require webapp
Deprecation Notice: Using ${var} in strings is deprecated, use {$var} instead in
Deprecation Notice: Using ${var} in strings is deprecated, use {$var} instead in
Using version ^1.1 for symfony/webapp-pack
./composer.json has been updated
Running composer update symfony/webapp-pack
Loading composer repositories with package information
Restricting packages listed in "symfony/symfony" to "6.2.*"
Updating dependencies
Lock file operations: 102 installs, 0 updates, 0 removals
- Locking doctrine/annotations (2.0.1)
- Locking doctrine/cache (2.2.0)
- Locking doctrine/collections (2.1.2)
- Locking doctrine/common (3.4.3)
- Locking doctrine/dbal (3.6.1)
- Locking doctrine/deprecations (v1.0.0)
- Locking doctrine/doctrine-bundle (2.9.0)
- Locking doctrine/doctrine-migrations-bundle (3.2.2)
- Locking doctrine/event-manager (2.0.0)
- Locking doctrine/inflector (2.0.6)
- Locking doctrine/instantiator (1.5.0)
- Locking doctrine/lexer (2.1.0)
- Locking doctrine/migrations (3.6.0)
- Locking doctrine/orm (2.14.1)
- Locking doctrine/persistence (3.1.4)
- Locking doctrine/sql-formatter (1.1.3)
- Locking egulias/email-validator (4.0.1)
- Locking masterminds/html5 (2.7.6)
- Locking monolog/monolog (3.3.1)
- Locking myclabs/deep-copy (1.11.1)
- Locking nikic/php-parser (v4.15.4)
- Locking phar-io/manifest (2.0.3)
- Locking phar-io/version (3.2.1)
- Locking phpdocumentor/reflection-common (2.2.0)
- Locking phpdocumentor/reflection-docblock (5.3.0)
- Locking phpdocumentor/type-resolver (1.7.1)
- Locking phpstan/phpdoc-parser (1.16.1)
- Locking phpunit/php-code-coverage (9.2.26)
- Locking phpunit/php-file-iterator (3.0.6)
- Locking phpunit/php-invoker (3.1.1)
- Locking phpunit/php-text-template (2.0.4)
- Locking phpunit/php-timer (5.0.3)
- Locking phpunit/phpunit (9.6.6)
- Locking psr/link (2.0.1)
- Locking sebastian/cli-parser (1.0.1)
- Locking sebastian/code-unit (1.0.8)
- Locking sebastian/code-unit-reverse-lookup (2.0.3)

```

**Remarque** Nous avons créé un projet de type website-skeleton, il est optimisé pour les applications web traditionnelles. En revanche, pour construire des microservices, des applications console ou des API, il est préférable d'utiliser un projet de type skeleton. Il faudra alors importer manuellement toutes les dépendances dont on pourrait avoir besoin.

## Instructions pour les tests et l'envoi de mails

Notez qu'à la fin de l'installation, vous devriez avoir ces lignes:



```
mihaela@work: ~/Bureau/monProjet
symfony/phpunit-bridge instructions:

* Write test cases in the tests/ folder
* Use MakerBundle's make:test command as a shortcut!
* Run the tests with php bin/phpunit

symfony/mailer instructions:

* You're ready to send emails.

* If you want to send emails via a supported email provider, install
  the corresponding bridge.
  For instance, composer require mailgun-mailer for Mailgun.

* If you want to send emails asynchronously:

  1. Install the messenger component by running composer require messenger;
  2. Add 'Symfony\Component\Mailer\Messenger\SendEmailMessage': amqp to the
     config/packages/messenger.yaml file under framework.messenger.routing
     and replace amqp with your transport name of choice.

* Read the documentation at https://symfony.com/doc/master/mailer.html

mihaela@work:~/Bureau/monProjet$
```

Ce sont des instructions pour utiliser PHPUnit (un framework PHP pour tester son code) et le Mailer (composant intégré à Symfony que vous utiliserez pour envoyer des mails)

## Exécution d'une application Symfony

En environnement de production, il est courant d'utiliser un serveur Web comme Apache ou Nginx (voir la configuration d'un serveur Web pour exécuter symfony).

Mais pour le développement, il est recommandé d'utiliser le serveur web Symfony.

### Utilisation du serveur web Symfony

Pour cela, ouvrir un terminal et utilisez les commandes suivantes :

Lancement du serveur local :

```
symfony serve
```

The screenshot shows an IDE interface with a menu bar (File, Edit, View, Navigate, Code, Refactor, Run, Tools, VCS, Window, Help) and a project sidebar on the left labeled 'monProjet'. The main terminal window has a title bar 'Terminal: Local x +'. It displays the following output:

```
INFO A new Symfony CLI version is available (5.5.2, currently running 5.4.13).

If you installed the Symfony CLI via a package manager, updates are going to be
If not, upgrade by downloading the new version at https://github.com/symfony-cl
And replace the current binary (symfony) by the new one.

Following Web Server log file (/home/mihaela/.symfony5/log/d27956f81b136c40a6ddad7a50938aab7b)
Following PHP log file (/home/mihaela/.symfony5/log/d27956f81b136c40a6ddad7a50938aab7b)

[WARNING] The local web server is optimized for local development and MUST never be used in production.

[OK] Web server listening
The Web server is using PHP CLI 8.2.4
https://127.0.0.1:8000

[Web Server ] Apr  1 01:15:01 |DEBUG | PHP      Reloading PHP versions
[Web Server ] Apr  1 01:15:01 |DEBUG | PHP      Using PHP version 8.2.4 (from default version)
[Application] Mar 31 22:27:04 |INFO   | DEPREC  User Deprecated: The "Monolog\Logger" class is deprecated. Use "Symfony\Component\Logging\Monolog\Logger" instead.
```

Le serveur local est démarré, il ne reste plus qu'à ouvrir un navigateur et accéder à <http://localhost:8000/> (ou 8001, le port peut varier).

Si tout fonctionne, une page d'accueil s'affiche et confirme que l'environnement de développement est opérationnel.



Welcome to Symfony!



https://127.0.0.1:8001



You're seeing this page because



Your applica



Documentation

[Guides](#), [components](#), [references](#)

404

10 ms

4.0 MiB



1



n/a



1 ms

Pour arrêter le processus du serveur, appuyer sur **Ctrl C** à partir du terminal.

**Remarque** Si vous souhaitez passer votre version locale du site en **https**, vous devez entrer cette commande:

```
symfony server:ca:install
```

```
mihaela@work:~/Bureau/monProjet$ symfony server:ca:install
WARNING "certutil" is not available, so the CA can't be automatically installed in Firefox
Install "certutil" with "apt install libnss3-tools" and re-run the command

[OK] The local Certificate Authority is installed and trusted

mihaela@work:~/Bureau/monProjet$
```

Vous pouvez constater sur la capture d'écran qu'il y a un avertissement (Warning) qui nous dit qu'il nous manque un util pour installer le certificat - le **certutil**. Vous avez aussi la commande à saisir dans le terminal pour installer cet outil:

```
sudo apt install libnss3-tools
```

Une fois **certutil** installé, vous retapez la commande

```
symfony server:ca:install
```

et vous verrez **https** à la place du **http**



Symfony

Installation et configuration de Symfony

Exécution d'une application Symfony

Utilisation du serveur web Symfony

{}

⑩ MS FULL STACK 22486

⑩ MS FULL STACK 22197

⑩ MS DEV APPLI 22258

⑩ Déconnexion

## MS DEV APPLI - 22258

séance prévue du 12/12/2023 au 18/12/2023 Séance terminée le 24/01/2024

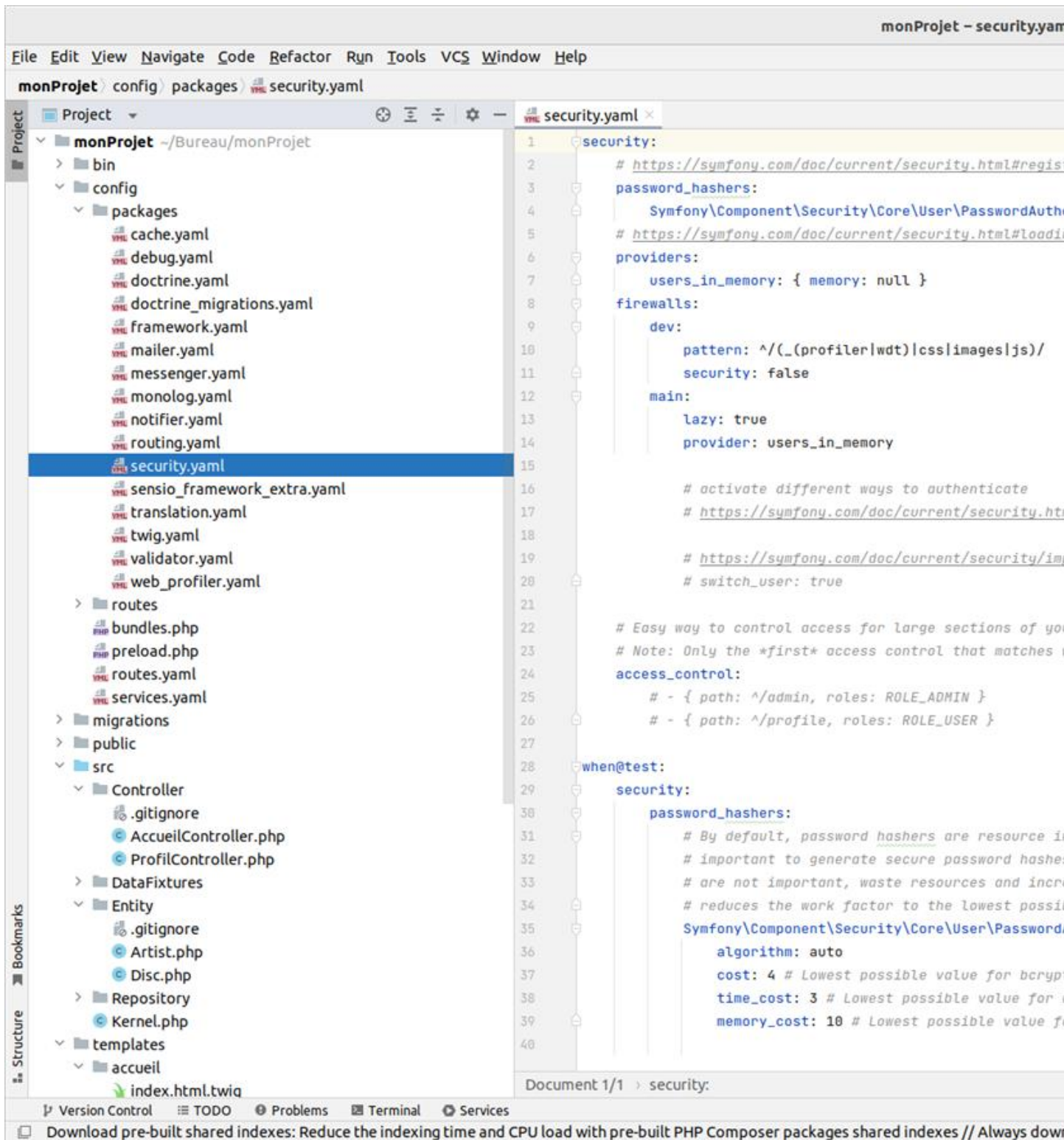


## **Création du projet**

Une fois les outils installés, nous pouvons enfin initialiser notre projet Symfony.

Suivez les différentes étapes de ce tuto pour initialiser votre premier projet Symfony

# Structure d'un projet Symfony :



**[bin]** - ce dossier contient les fichiers de commandes permettant, par exemple, de vider le cache Symfony ou de mettre à jour la base de données

**[config]** - ce dossier contient toute la configuration des packages, services et routes

**[src]** - ce dossier constitue le coeur de votre projet (ce sont les parties M&C du pattern MVC)

**[templates]** - ce dossier contient les vues (c'est la partie V du pattern MVC)

**[public]** - c'est le point d'entrée de l'application, il contient le contrôleur frontal "index.php" (chaque requête passe par là)

[!NOTE] Les noms des fichiers de configuration finissent en **.yaml**.

**YAML** est un format de données très simple, basé sur des données au format **clé : valeur**.

Une fois votre projet Symfony installé, nous allons enfin pouvoir créer une première vue !!

## Afficher une page avec Symfony

Le contrôleur est une classe PHP qui a en charge de construire la page.

Un contrôleur possède plusieurs méthodes, chaque méthode est associée à une route. Une méthode exploite les informations de la **Request** entrante et les utilise pour créer un objet Symfony de type **Response** pouvant contenir du HTML, une chaîne JSON ou même un fichier binaire, tel qu'une image ou un PDF.

Une route associe l'URL (par exemple `/apropos`) à une méthode d'un contrôleur.

Si la page doit afficher du code html, la méthode utilise un template. Ce template est la vue que le contrôleur affichera à l'utilisateur.

### Le contrôleur

Symfony Documentation : <https://symfony.com/doc/current/controller.html>

Les Contrôleurs ont pour rôle de gérer chaque requête adressée à l'application Symfony, et dans la plupart des cas, de rendre un template (vue) qui va élaborer le contenu de la réponse au client.

Un contrôleur est une méthode PHP que vous créez, qui lit les informations de l'objet **Request** (requêtes du client envoyées vers le site), crée et retourne un objet **Response**. La réponse peut être une page HTML, JSON, XML, un téléchargement de fichier, une redirection, une erreur 404, ... Le contrôleur exécute le traitement métier dont l'application a besoin pour restituer le contenu d'une page.

### Le routage

Une route est une correspondance entre une URL et une méthode. Cette route peut être configurée de plusieurs manières :

- ⑩ avec un fichier XML
- ⑩ avec une classe PHP
- ⑩ avec un fichier de configuration .yaml
- ⑩ avec des annotations

Tout dépend comment vous comptez gérer votre projet, mais de manière générale, les annotations restent plus pratiques: la route est affichée directement au-dessus de la méthode, ce qui permet une meilleure visibilité des informations dont vous pourriez avoir besoin.

Commençons par créer notre contrôleur en suivant cette ressource :Création du contrôleur

La façon la plus efficace de créer un contrôleur est d'utiliser le **maker**

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

Cette commande vous permet de créer un contrôleur **Accueil**. En fait deux fichiers viennent d'être créés.

La classe `Accueil` contenant la méthode `index`

```
<?php

namespace App\Controller;

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

class AccueilController extends AbstractController
{
    #[Route('/', name: 'app_accueil')]
    public function index(): Response
    {
        return $this->render('accueil/index.html.twig', [
            'controller_name' => 'AccueilController',
        ]);
    }
}
```

Remarquez que l'annotation spécifiant la route est au format php8

Et une vue

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

{% block title %}Hello AccueilController!{% 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/mihaela/Bureau/monProjet/src/Controller/AccueilController.php' | file_link(0) }}">
src/Controller/AccueilController.php</a></code></li>
        <li>Your template at <code><a
href="{{ ' /home/mihaela/Bureau/monProjet/templates/accueil/index.html.twig' | file_link(0) }}">tem
plates/accueil/index.html.twig</a></code></li>
    </ul>
</div>
{% endblock %}
```

Vous pouvez vérifier le bon fonctionnement dans votre navigateur à l'adresse <https://localhost:8000>.

# Hello AccueilController!

This friendly message is coming from:

- Your controller at [src/Controller/AccueilController.php](#)
- Your template at [templates/accueil/index.html.twig](#)



## Création du contrôleur

### Le template

Pour afficher des informations à l'utilisateur, Symfony utilise un moteur de template appelé Twig. Il a la particularité de récupérer facilement les données transmises par le contrôleur, et intègre la notion d'héritage sur les templates.

Sa structure est très semblable au HTML, les fichiers templates auront une extension `.html.twig`.

Pour créer notre première vue avec Symfony, suivez ce tuto : [Les templates](#)

Dans un projet symfony, les templates vont se trouver dans le répertoire 'templates'.



Un fichier 'base.html.twig' est automatiquement créé par Symfony. Voyons son contenu :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="UTF-8">
    <title>{% block title %}Welcome!{% endblock %}</title>
    {% block stylesheets %}{% endblock %}
  </head>
  <body>
    {% block body %}{% endblock %}
    {% block javascripts %}{% endblock %}
  </body>
</html>
```

On retrouve une structure HTML assez classique (présence du '`<!DOCTYPE html>`', des balises `html` et `body`, ...). Toutefois, on retrouve une nouvelle syntaxe :

```
{% block body %} {% endblock %}
```

C'est grâce à cette syntaxe que l'on va retrouver la notion d'héritage dans nos templates.

Pour faire simple, ce fichier est la base (comme son nom l'indique) de notre site. Nous mettrons dedans tous les éléments récurrents de notre application :

- ⑩ la navbar
- ⑩ le footer
- ⑩ les imports CSS et Js
- ⑩ le 'head' des pages
- ⑩ etc.

Chacun de ces 'block' pourront être appelés dans un autre template, pour y introduire du code.

Le chemin de notre vue appelé dans la méthode que l'on vient de créer est 'accueil/index.html.twig'. Nous allons reprendre le fichier 'index.html.twig'.

La première ligne à écrire est celle-ci :

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

Cela permet "d'appeler" le fichier 'base.html.twig', de dire que nous aurons besoin de son contenu. Nous avons ici notre **héritage** des templates.

Pour afficher du texte dans le corps de notre page, nous utiliserons le `{% block body %}`, et nous y inscrirons du code (remplacez le contenu de votre vue `index.html.twig` par les lignes suivantes):

```
{% extends 'base.html.twig' %}
{% block body %}
    <h1>Première page web avec Symfony</h1>
    <p>Hello World !!!</p>
{% endblock %}
```

Allons regarder le résultat dans le navigateur. En utilisant la même url que précédemment (localhost:8000 ou localhost/symfony/monProjet/public/index.php) nous obtenons l'affichage suivant :

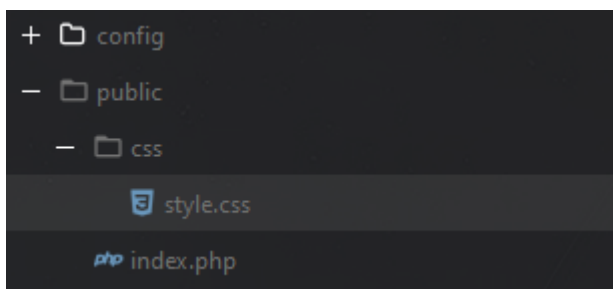


Bravo !!! Vous avez créé votre première page avec Symfony !!!

## Ajout de CSS et Js

Maintenant que nous avons un affichage, nous allons voir comment ajouter une feuille de style et, pourquoi pas, en même temps utiliser un framework CSS. Dans notre cas on utilisera Bootstrap.

Créons d'abord notre feuille de style : pour cela, nous nous placerons dans le dossier 'public', et nous créerons un dossier 'CSS', dans lequel nous placerons notre fichier 'style.css'.



Dans le fichier 'base.html.twig', on va insérer dans le block 'head' notre feuille de style de manière conventionnelle, seule la notation du lien sera un peu différente :

```
<head>
```

```

    <meta charset="UTF-8">
    <title>{% block title %}Welcome!{% endblock %}</title>
    <link rel="stylesheet" href="{{ asset('css/style.css') }}">
    {% block stylesheets %}{% endblock %}
</head>

```

Afin de vérifier que cela est fonctionnel, changeons la couleur du fond et de la police.

```

body {
    background-color: #222222;
}
h1, p {
    color: #C9C9C9;
}

```

Le rendu :



'asset' permet de renseigner que nous faisons appel à une ressource de l'application. On utiliserait la même syntaxe pour importer un script Js, ou pour afficher une image.

De manière générale, toutes ces ressources doivent se trouver dans le dossier 'public'.

## Ajout d'un framework CSS

De la même manière, nous pouvons insérer un framework CSS à notre application Symfony. Prenons par exemple Bootstrap et incluons une navbar sur notre page. Nous prendrons pour l'exemple directement le CDN de Bootstrap, donc inutile d'utiliser la méthode 'asset()' :

```

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

```



```

<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css"
rel="stylesheet" integrity="sha384-
1BmE4kWBq78iYhFIdvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3" crossorigin="anonymous">
<link rel="stylesheet" href="/css/style.css">
{% block stylesheets %} {% endblock %}
</head>
<body>
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <a class="navbar-brand" href="#">Navbar</a>
  <button class="navbar-toggler" type="button" data-toggle="collapse" data-
target="#navbarSupportedContent" aria-controls="navbarSupportedContent" aria-expanded="false"
aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>

  <div class="collapse navbar-collapse" id="navbarSupportedContent">
    <ul class="navbar-nav mr-auto">
      <li class="nav-item active">
        <a class="nav-link" href="#">Home <span class="sr-only">(current)</span></a>
      </li>
      <li class="nav-item">
        <a class="nav-link" href="#">Link</a>
      </li>
      <li class="nav-item dropdown">
        <a class="nav-link dropdown-toggle" href="#" id="navbarDropdown"
role="button" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
          Dropdown
        </a>
        <div class="dropdown-menu" aria-labelledby="navbarDropdown">
          <a class="dropdown-item" href="#">Action</a>
          <a class="dropdown-item" href="#">Another action</a>
          <div class="dropdown-divider"></div>
          <a class="dropdown-item" href="#">Something else here</a>
        </div>
      </li>
      <li class="nav-item">
        <a class="nav-link disabled" href="#" tabindex="-1" aria-
disabled="true">Disabled</a>
      </li>
    </ul>
    <form class="form-inline my-2 my-lg-0">
      <input class="form-control mr-sm-2" type="search" placeholder="Search" aria-
label="Search">
      <button class="btn btn-outline-success my-2 my-sm-0"
type="submit">Search</button>
    </form>
  </div>
</nav>
{% block body %} {% endblock %}

<script src="https://code.jquery.com/jquery-3.4.1.slim.min.js" integrity="sha384-
J6qa4849bIE2+poT4WnyKhv5vZF5SrPo0iEjwBvKU7imGFAV0wwj1yYfoRSJoZ+n"
crossorigin="anonymous"></script>
<script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"
integrity="sha384-Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfooAo"
crossorigin="anonymous"></script>

```

```

<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/bootstrap.bundle.min.js"
integrity="sha384-ka7Sk0Gln4gmtz2MlQnikT1wXgYs0g+OMhuP+I lRH9sENB00LRn5q+8nbTov4+1p"
crossorigin="anonymous"></script>
    {% block javascripts %} {% endblock %}
</body>
</html>

```

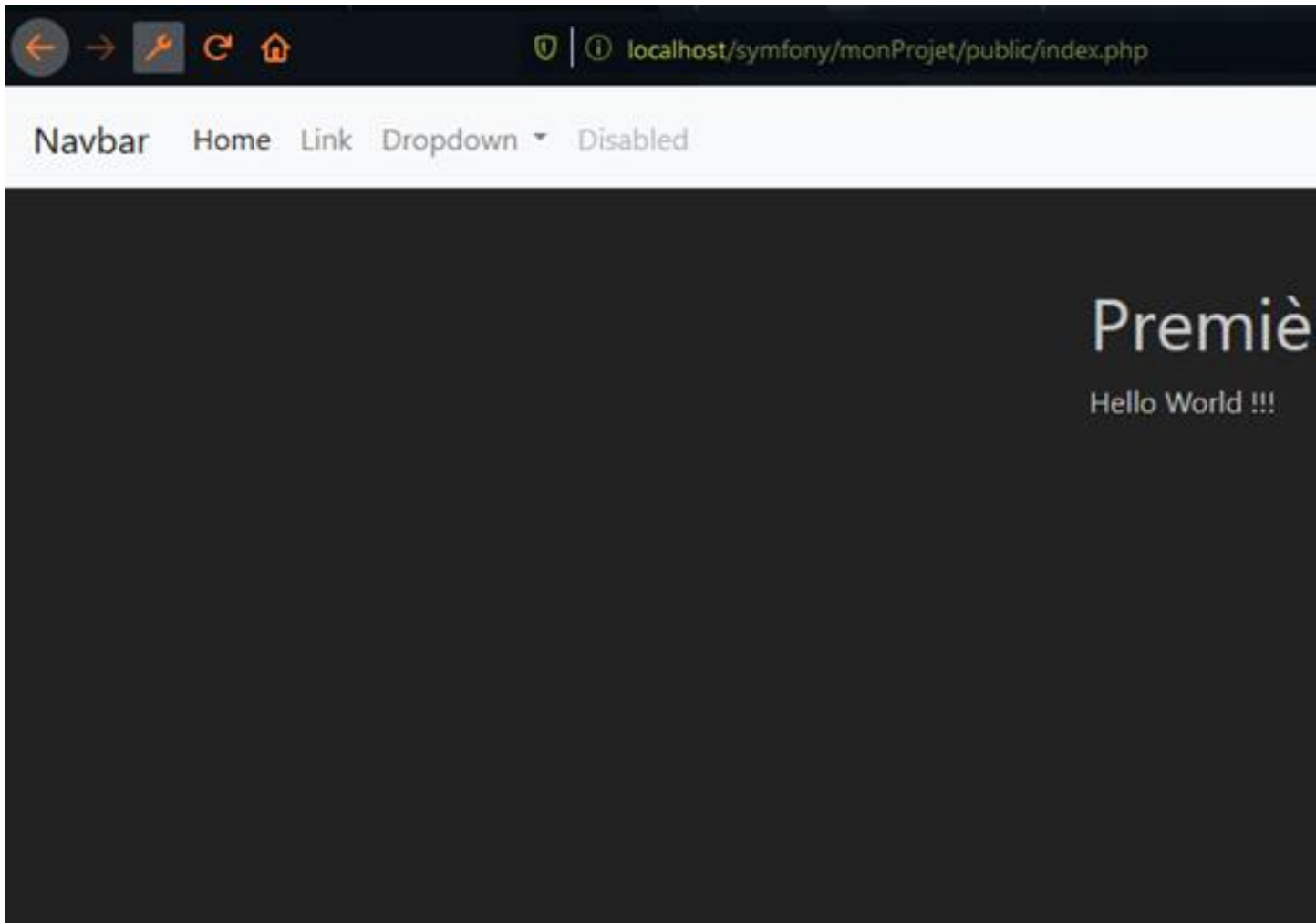
Avec un peu de mis en page Bootstrap sur notre 'index.html.twig' :

```

{% extends 'base.html.twig' %}
{% block body %}
    <div class="container">
        <div class="row mt-5">
            <div class="col-sm-8 offset-sm-2">
                <h1>Première page web avec Symfony</h1>
                <p>Hello World !!!</p>
            </div>
        </div>
    </div>
{% endblock %}

```

Nous obtenons le résultat suivant :



Les templates

Ajout de CSS et Js

Ajout d'un framework CSS

En suivant cette ressource :

## Création d'une seconde page

### Nouvelle vue, nouveau contrôleur

De la même manière que pour l'affichage de notre première page, nous allons construire une seconde page.

Imaginons que nous voulons construire une page profil.

Nous allons donc construire un nouveau contrôleur, avec une méthode permettant l'affichage de cet vue, ainsi qu'un nouveau template :

#### ⑩ Contrôler (/Controller/ProfilController.php)

```
class ProfilController extends AbstractController
{
    #[Route('/profil', name: 'app_profil')]
    public function index(): Response
    {
        return $this->render( view: 'profil/index.html.twig', [
            'controller_name' => 'ProfilController',
        ]);
    }
}
```

#### ⑩ Template (/Template/profil/index.html.twig)

```
{% extends 'base.html.twig' %}
{% block body %}
    <div class="container">
        <div class="row mt-5">
            <div class="col-sm-8 offset-sm-2">
                <h1>Vos informations</h1>
            </div>
        </div>
    </div>
{% endblock %}
```

La structure du template et du contrôleur est identique à ce qu'on a vu précédemment, les seuls changements majeurs sont dans le contrôleur :

- ⑩ le nom du contrôleur et de la classe sont évidemment différents.
- ⑩ nous avons ajouté une route au contrôleur pour le différencier d'AccueilController'.
- ⑩ le nom et le chemin de la méthode sont également différents, afin de les repérer facilement.

Il ne nous reste plus qu'à mettre un lien qui nous permettra de naviguer entre les 2 pages.

Pour cela nous allons dans un premier temps personnaliser la navbar :

⑩ on ajoute un lien pour aller sur la page d'accueil

```
<a class="navbar-brand" href="{{ path('app_accueil') }}">Accueil</a>
```

⑩ on ajoute un lien pour aller sur la page 'profil'

```
<a class="nav-link" href="{{ path('app_profil') }}">Profil</a>
```

⑩ on en profite pour enlever ce dont nous n'avons pas besoin, pour obtenir au final :

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
<a class="navbar-brand" href="{{ path('app_accueil') }}">Accueil</a>
<button class="navbar-toggler" type="button" data-toggle="collapse" data-
target="#navbarSupportedContent" aria-controls="navbarSupportedContent" aria-expanded="false"
aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
</button>

<div class="collapse navbar-collapse" id="navbarSupportedContent">
    <ul class="navbar-nav mr-auto">
        <li class="nav-item active">
            <a class="nav-link" href="{{ path('app_profil') }}">Profil</a>
        </li>
        <li class="nav-item">
            <a class="nav-link" href="#">Link</a>
        </li>
    </ul>
</div>
</nav>
```

'path' indique le nom de la route à cibler (dans les annotations, 'name="app-profil" ').

Maintenant que nous arrivons à naviguer entre 2 pages, il serait intéressant d'afficher des données qui viennent du contrôleur sur la vue.

## Faire transiter des données du contrôleur à la vue

### Passer un tableau

Dans 'ProfilController', dans la méthode 'index', créons un tableau qui contiendra les informations à faire transiter :

```
$info = ['Loper', 'Dave', 'daveloper@code.dom', '01/01/1970'];
```

Pour afficher les informations contenues dans ce tableau, il suffit de le passer en second paramètre de la méthode 'render()', via un tableau d'options :

```
return $this->render('profil/index.html.twig', [
    'informations' => $info
]);
```

Notre contrôleur aura donc cette structure :

```
class ProfilController extends AbstractController
```

```

{
    #[Route('/profil', name: 'app_profil')]
    public function index(): Response
    {
        $info = ['Loper', 'Dave', 'daveloper@code.dom', '01/01/1970'];

        return $this->render('profil/index.html.twig', [
            'informations' => $info
        ]);
    }
}

```

Affichons ces informations dans un tableau à l'aide d'une boucle for :

```

<!-- profil/index.html.twig -->

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

{% block body %}
    <div class="container">
        <div class="row mt-5">
            <div class="col-sm-8 offset-sm-2">
                <h1>Vos informations</h1>
                <ul>
                    {% for info in informations %}
                        <li class="text-white">{{ info }}</li>
                    {% endfor %}
                </ul>
            </div>
        </div>
    </div>
{% endblock %}

```

### Passer un tableau associatif

Pour passer un tableau associatif du contrôleur vers la vue, le procédé reste le même, seul la méthode d'affichage va changer :

```

class ProfilController extends AbstractController
{
    #[Route('/profil', name: 'app_profil')]
    public function index(): Response
    {
        $info = ['lastname' => 'Loper', 'firstname' => 'Dave', 'email' => 'daveloper@code.dom',
'birthdate' => '01/01/1970'];

        return $this->render('profil/index.html.twig', [
            'informations' => $info
        ]);
    }
}

```

```

<!-- profil/index.html.twig -->
{% block body %}
    <div class="container">

```

```

<div class="row mt-5">
  <div class="col-sm-8 offset-sm-2">
    <h1>Vos informations</h1>
    <table class="table table-light table-hover">
      <thead>
        <tr>
          <th>Nom</th>
          <th>Prénom</th>
          <th>Date de naissance</th>
          <th>Email</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>{{ informations.lastname }}</td>
          <td>{{ informations.firstname }}</td>
          <td>{{ informations.birthdate }}</td>
          <td>{{ informations.email }}</td>
        </tr>
      </tbody>
    </table>
  </div>
</div>
</div>
{% endblock %}

```

On se sert ici du nom du tableau, avec la clé associative associée à la valeur que l'on veut afficher.  
Et voici le résultat:

[←](#)
[→](#)
[↻](#)
⚠ Non sécurisé | <https://127.0.0.1:8000/profil>

[Accueil](#)
[Profil](#)
[Link](#)

## Vos informations

Nom	Prénom	Date de naissance	Email
Loper	Dave	01/01/1970	daveloper@code.do



Création d'une seconde page

Nouvelle vue, nouveau contrôleur

Faire transiter des données du contrôleur à la vue

Passer un tableau

Passer un tableau associatif

, vous verrez comment faire des liens entre les différentes vues et contrôleurs de votre application Symfony, ainsi que le passage de données du contrôleur vers une vue.

Maintenant que nous savons afficher du contenu sur une vue, et naviguer d'une page vers une autre, voyons la connexion à une base de données.

## Base de données et Doctrine

Pour accéder et manipuler une base de données avec Symfony, Nous allons utiliser un ORM (Object-relational mapping), Doctrine.

Un ORM est un type de programme qui se place en interface entre une application (notre site par exemple) et une base de données relationnelle.

Ce programme définit les correspondances entre les schémas de la base de données et les classes de votre application. L'association objets-tables est appelée **mapping**. Les classes sont associées à une table et chaque propriété de la classe est associé à un champ de la table. Avec **Doctrine**, cette opération est réalisée grâce à des annotations.

Avant de vous lancer dans **doctrine**, lisez ce support :ORM

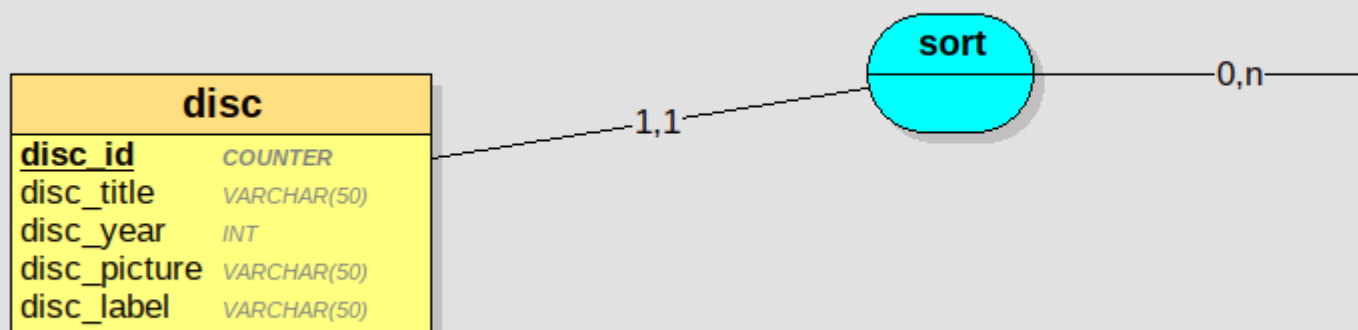
### Un peu de méthodologie

Dans les phase de conception traditionnelles, vous commencez par concevoir votre base de données avec un outils tel que **Merise**.

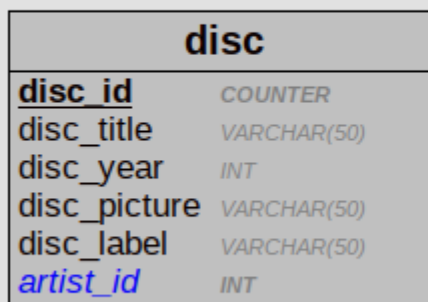
Puis, vous générez et exécutez le script de votre base de données et enfin vous alimentez vos tables avec un jeu de test.

Dans cette démarche, le script de création de votre base et le script d'alimentation sont deux fichiers **sql**.

Le modèle de référence votre MCD



soit le modèle physique de la base.



Deux remarques:

- ⑩ Le MCD vous impose des noms différents pour chaque champ.
- ⑩ Le modèle relationnel vous propose de manipuler les relations de votre base avec des clés étrangères...

Lorsque vous manipulez vos données via un ORM, l'approche est un peu différente...

Puisque vous allez manipuler des objets, un nouveau modèle de référence s'impose, le diagramme de classes UML.

Dans ce modèle plus besoin d'avoir des noms de champs uniques. Mais surtout, il n'y a plus de clés étrangères.



Les relations sont représentées par deux nouvelles propriétés:

- ⑩ `artist` dans la classe `Disc` (contient l'objet `Artist` associé au disque)
- ⑩ `discs` dans la classe `Artist` (contient une liste de `Disc` représentant les disques de l'artiste)

## Mise en oeuvre

Même si vous pouvez reprendre la structure d'une base existante, il est conseillé de reconstruire vos entités. C'est la démarche que nous vous proposons.



## Configuration de la base de données

Dans un premier temps vous devez indiquer à Doctrine quelle base de données vous voulez utiliser et où elle se trouve.

Dans un projet Symfony, vous avez par défaut deux fichiers `.env` : `.env` et `.env.test`.

Vous allez créer un autre fichier que vous appellerez `.env.local`. C'est dans ce fichier (qui a la structure du fichier `.env`) que vous stockerez des informations confidentielles de votre projet (comme les informations de connexion à votre base de données).

[!NOTE] Le fichier `.env.local` ne sera pas ajouté à git, il faudrait donc le recréer sur votre serveur.

Dans le fichier `.env.local`, modifiez la variable `DATABASE_URL` comme ci-dessous et placez-y les informations de connexion à votre base.

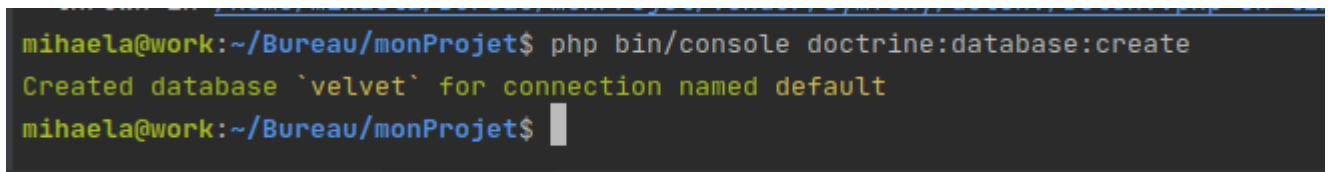
```
DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=mariadb-10.3.34&charset=utf8mb4"
```

Ici, on va appeler notre base `velvet`.

Ensuite deux commandes sont disponibles pour supprimer et créer la base de données.

```
php bin/console doctrine:database:drop --force
```

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



```
mihaela@work:~/Bureau/monProjet$ php bin/console doctrine:database:create
Created database `velvet` for connection named default
mihaela@work:~/Bureau/monProjet$
```

Pour l'instant, il ne s'agit que de la base, les tables arrivent bientôt...

## Création des entités

Le principe:

- ⑩ Vous créez vos entités (des classes PHP) avec le maker ou à la main
- ⑩ Vous demandez à doctrine de construire les tables nécessaires
- ⑩ Vous vérifiez votre travail en visualisant la structure de votre base.

Même si vous pouvez construire vos classes `Entités` à la main, il est fortement recommandé d'utiliser le maker.

**ATTENTION :** Pour créer une entité **USER**, au lieu d'utiliser la commande `make:entity`, vous devriez exécuter la commande `make:user` que Symfony a créée pour cette entité!.

C'est une entité spéciale en Symfony, qui fait appel à `UserInterface`, une interface qui propose de gérer l'authentification (identifiant, mot de passe, etc.).

Cependant, si vous créez cette classe avec `make:entity`, vous devez la modifier afin de rajouter toutes les méthodes héritées de l'interface `UserInterface`!

Commençons par créer une première entité `Disc`.

```
php bin/console make:entity disc
```

Lorsque vous tapez cette commande, le maker commence par créer l'entité `Disc` dans le fichier `src/Entity/Disc.php`. Si le fichier existe déjà, il sera simplement modifier.

Notez que `Doctrine` ajoute automatiquement une propriété `id` qui deviendra la clé primaire de votre table.

Ensuite le maker vous demande le nom de la propriété, puis d'autres informations (son type, sa longueur...).

```
Terminal: Local x Local (2) x Local (3) x +
mihaela@work:~/Bureau/monProjet$ php bin/console doctrine:database:create
Created database `velvet` for connection named default
mihaela@work:~/Bureau/monProjet$ php bin/console make:entity Disc

created: src/Entity/Disc.php
created: src/Repository/DiscRepository.php

Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-running this command.

New property name (press <return> to stop adding fields):
> title

Field type (enter ? to see all types) [string]:
>

Field length [255]:
>

Can this field be null in the database (nullable) (yes/no) [no]:
>

updated: src/Entity/Disc.php

Add another property? Enter the property name (or press <return> to stop adding fields):
> picture

Field type (enter ? to see all types) [string]:
>

Field length [255]:
>

Can this field be null in the database (nullable) (yes/no) [no]:
> yes

updated: src/Entity/Disc.php
```

Quand vous avez modifié votre entité, la commande ci-dessous vous permet de générer les tables nécessaires. Doctrine examine chaque entité et génère le code sql pour créer ou modifier la structure de votre base de données.

```
php bin/console d:s:u --force
```

Vous pouvez répéter cette opération autant de fois que nécessaire pour construire vos entités et votre base.

Vérifiez que votre base est correctement configurée en visualisant le diagramme de votre base de données.

Continuez le travail pour construire les entités **D i s c** et **A r t i s t**. Ne vous souciez pas des relations, elles arrivent juste après...

### Les relations

Il existe plusieurs types de relations:

- ⑩ OneToOne
- ⑩ OneToMany
- ⑩ ManyToOne
- ⑩ ManyToMany

Dans notre cas, nous avons bien une relation entre **D i s c** et **A r t i s t**.

C'est une relation **OneToMany** quand on regarde depuis **A r t i s t**.

Mais elle devient **ManyToOne** si l'on se place coté **D i s c**.

Pour mettre en place cette relation, modifions une des deux entités.

```
php bin/console make:entity disc
```

Ajoutez la propriété **a r t i s t** (elle contiendra l'**A r t i s t** associé au **D i s c**).

Choisissez **r e l a t i o n** comme type de la propriété

Ensuite **D o c t r i n e** vous demande le type de la propriété (le nom de la classe), donc **A r t i s t**

Puis saisissez le type de relation **ManyToMany**

Validez les réponses suivantes en choisissant les réponses par défaut.

```
mihaela@work:~/Bureau/monProjet$ php bin/console make:entity Disc
```

Your entity already exists! So let's add some new fields!

New property name (press <return> to stop adding fields):

> artist

Field type (enter ? to see all types) [string]:

> relation

What class should this entity be related to?:

> Artist

What type of relationship is this?

Type	Description
ManyToOne	Each <b>Disc</b> relates to (has) <b>one</b> <b>Artist</b> . Each <b>Artist</b> can relate to (can have) <b>many</b> <b>Disc</b> objects.
OneToMany	Each <b>Disc</b> can relate to (can have) <b>many</b> <b>Artist</b> objects. Each <b>Artist</b> relates to (has) <b>one</b> <b>Disc</b> .
ManyToMany	Each <b>Disc</b> can relate to (can have) <b>many</b> <b>Artist</b> objects. Each <b>Artist</b> can also relate to (can also have) <b>many</b> <b>Disc</b> objects.
OneToOne	Each <b>Disc</b> relates to (has) exactly <b>one</b> <b>Artist</b> . Each <b>Artist</b> also relates to (has) exactly <b>one</b> <b>Disc</b> .

Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]:

> ManyToOne

Is the **Disc.artist** property allowed to be null (nullable)? (yes/no) [yes]:

>

Do you want to add a new property to **Artist** so that you can access/update **Disc**:

>

A new property will also be added to the **Artist** class so that you can access the

New field name inside **Artist** [discs]:

>

updated: src/Entity/Disc.php

updated: src/Entity/Artist.php

Add another property? Enter the property name (or press <return> to stop adding

>

Success!

Next: When you're ready, create a migration with `php bin/console make:migration`

Pour terminer, mettez à jour votre base

```
php bin/console d:s:u --force
```

Puis vérifiez la structure de votre base de données. Une clé étrangère est apparue dans votre base.

## Synthèse

Les clés étrangères existent bien dans la base de données, mais elles ne sont pas visibles dans vos classes PHP.

Pour manipuler vos relations, vous devez utiliser les deux attributs ajoutés par Doctrine.

⑩ `artist` dans `Disc`

⑩ `discs` dans `Artist`

Notez bien que Doctrine pluralise automatiquement les noms des propriétés. `discs` contient un tableau de `Disc` donc il est au pluriel.



## ORM

Un peu de méthodologie

Mise en oeuvre

Configuration de la base de données

Création des entités

Les relations

Synthèse

qui vous permettra de découvrir le fonctionnement et les bonnes pratiques de doctrine.

Cette partie est très importante et délicate, n'hésitez pas à vous faire aider par votre formateur en lui demandant une présentation.

Maintenant nous allons voir comment ajouter un jeu de test (`fixture` dans symfony) dans votre nouvelle base. Suivez les instructions de ce support :[Fixtures dans Symfony](#)

Les fixtures dans Symfony sont un moyen simple d'insérer des données dans vos entités.

Un maker est disponible pour créer un squelette de fixture

```
php bin/console make:fixture jeu1
```

Cette commande crée un fichier `src/DataFixtures/Jeu1.php` qui ressemble à ceci.

```
<?php
```

```
namespace App\DataFixtures;
```

```
use Doctrine\Bundle\FixturesBundle\Fixture;
```

```
use Doctrine\Persistence\ObjectManager;
```

```

class Jeu1 extends Fixture
{
    public function load(ObjectManager $manager): void
    {
        // $product = new Product();
        // $manager->persist($product);

        $manager->flush();
    }
}

```

## Remarque

Si vous rencontrez cette erreur

```

[ERROR] Missing package: to use the make:fixtures command, run:
        composer require orm-fixtures --dev

```

Il vous faudra taper la commande indiquée

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

et ensuite de nouveau la commande

```
php bin/console make:fixture jeu1
```

Quand vous exécuterez vos fixtures, la méthode `load` sera déclenchée. Remarquez que le paramètre `$manager` est une instance de la classe `ObjectManager`, il va vous permettre d'enregistrer (persister) vos entités dans la base de données.

Deux méthodes utiles du `$manager`:

`persist` qui permet de spécifier à doctrine qu'une nouvelle entité doit être persisté.

`flush` qui indique à doctrine de générer le code sql pour mettre à jour votre base.

Il ne vous reste plus qu'à compléter ce code pour insérer votre jeu de test.

Pour créer votre premier `Artist`, ajoutez le code ci-dessous:

```

$artist1 = new Artist();

$artist1->setName("Queens Of The Stone Age");
$artist1->setUrl("https://qotsa.com/");

$manager->persist($artist1);
$manager->flush();

```

Pour exécuter votre fixture

```

php bin/console doctrine:fixtures:load
// ou
php bin/console d:f:l

```

Symfony vous demande une confirmation avant de purger votre base.

```
mihaela@work:~/Bureau/monProjet$ php bin/console make:fixture jeu1

created: src/DataFixtures/Jeu1.php

Success!

Next: Open your new fixtures class and start customizing it.
Load your fixtures by running: php bin/console doctrine:fixtures:load
Docs: https://symfony.com/doc/current/bundles/DoctrineFixturesBundle/index.html
mihaela@work:~/Bureau/monProjet$ php bin/console d:f:l

Careful, database "velvet" will be purged. Do you want to continue? (yes/no) [no]:
> yes

> purging database
> loading App\DataFixtures\AppFixtures
> loading App\DataFixtures\Jeu1
mihaela@work:~/Bureau/monProjet$
```

Une fois votre fixture insérée, vérifiez dans votre base que les données sont bien présentes.

Pour l'instant vous avez ajouté un `Artist`, pour compléter notre jeu de test, nous allons ajouter un `Disc` et le relier à notre `Artist`.

```
$artist1 = new Artist();

$artist1->setName("Queens Of The Stone Age");
$artist1->setUrl("https://qotsa.com/");

$manager->persist($artist1);

$disc1 = new Disc();
$disc1->setTitle("Songs for the Deaf");
$disc1->setPicture("https://en.wikipedia.org/wiki/Songs_for_the_Deaf#/media/File:Queens_of_the_S
tone_Age_-_Songs_for_the_Deaf.png");
$disc1->setLabel("Interscope Records");

$manager->persist($disc1);

// Pour associer vos entités
$disc1->setArtist($artist1);
// ou
$artist1->addDisc($disc1);

$manager->flush();
```

Remaquez les dernières lignes qui permettent de relier les entités.



Pour indiquez l'Artist d'un Disc

```
$disc1->setArtist($artist1);
```

Pour ajouter un Disc à l'Artist

```
$artist1->addDisc($disc1);
```

Ces deux manipulations sont équivalentes, elles donnent le même résultat. Comme il s'agit d'une relation **OneToMany** vous avez le choix de la manipuler côté **One** ou côté **Many**.

Exécutez de nouveau votre **fixture** et vérifiez dans la base que la relation s'est bien faite.

Vous pouvez aussi récupérer toutes les données que nous avons dans les exercices php en suivant ce didacticiel.



Fixtures dans Symfony  
pour créer vos premières **fixtures**.