

The background is a light gray gradient. It is decorated with numerous realistic water droplets of various sizes, some with highlights and shadows, scattered across the surface. In the upper center, there is a faint, circular logo or watermark that appears to contain a stylized 'S' or a similar emblem.

SYMFONY : LES BASES

C'EST QUOI SYMFONY



C'est un ensemble de composants PHP, ainsi qu'un Framework **MVC PHP**.

Il est fourni plusieurs fonctionnalités modulables permettant de faciliter la réalisation d'un site web, et aussi de l'accélérer,

Symfony est apparu en 2005 par SensioLabs, une entreprise française.

Le fondateur de Symfony est **Fabien Potencier**.

PREREQUIS

Download Symfony :

<https://symfony.com/download>

PHP version minimum : 7.4 (**nous allons utiliser du 8.0 minimum**)

Download Composer :

<https://getcomposer.org/download/>

Yarn :

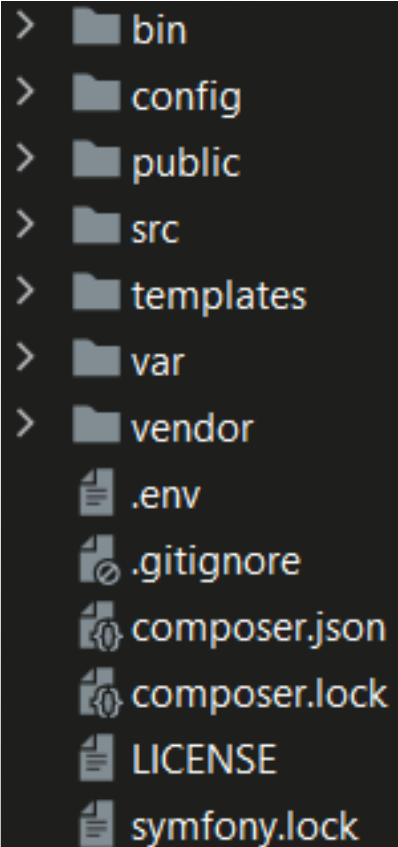
<https://classic.yarnpkg.com/lang/en/docs/install/#windows-stable>



SYMFONY : COMMANDES DE BASE

Commande	Effet
<code>symfony new testProject --version=5.4 --full</code>	Créer un projet Symfony avec la version spécifique
<code>composer create-project symfony/website-skeleton my_project "5.4.*"</code>	Créer un projet Symfony avec la version spécifique, via composer (plus complet !)
<code>composer install</code>	Met à jour les dépendances du projet du composer.json

SYMFONY : STRUCTURE D'UN PROJET



- > bin
- > config
- > public
- > src
- > templates
- > var
- > vendor
- .env
- .gitignore
- composer.json
- composer.lock
- LICENSE
- symfony.lock

bin : grâce à lui que l'on peut utiliser les commandes Symfony

config: config Symfony => parfois les bundles peuvent demander des modifications

node_modules : dossier des dépendances yarn (si ajouté au projet)

public : l'index.php de Symfony

src : là où l'on va écrire la majorité de notre code php

templates : dossier où seront les vues de l'application (**html.twig**)

var : cache, entre autres

vendor : dossier des bundles Symfony, et des sources de Symfony

composer.json : dépendances du projet

Symfony utilise twig comme moteur de template pour ses pages html, il a l'avantage de nous aider à simplifier le code, à l'intérieur des pages html, notamment pour tout ce qui affichage de variables et les conditions ou boucles.

SYMFONY : INSTALLATION DE WEBPACK

Commande	Effet
<code>composer require symfony/webpack-encore-bundle</code>	Ajoute Webpack encore au projet
<code>yarn install</code>	Ajoute Yarn au projet
<code>yarn add @symfony/webpack-encore --dev</code>	Ajoute Webpack à Yarn
<code>yarn add sass-loader@^12.0.0 sass --dev</code>	Ajoute un loader sass/scss au projet
<code>yarn add typescript ts-loader@^9.0.0 --dev</code>	Ajoute un compiler Typescript
<code>yarn add bootstrap</code>	Ajout Bootstrap au projet

SYMFONY : CONFIGURATION DE WEBPACK

```
* 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(name: 'styles', src: './assets/styles/main.scss')
.addEntry(name: 'scripts', src: './assets/scripts/main.ts')
```

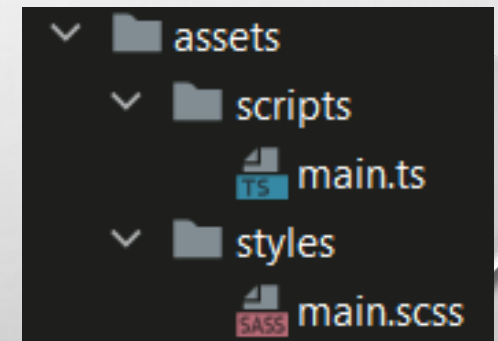
```
// enables Sass/SCSS support
.enableSassLoader()

// uncomment if you use TypeScript
.enableTypeScriptLoader()
```

- il faut décommenter ces deux lignes
(Il faut aussi un **tsconfig.json** pour Typescript !)

Dans le fichier **webpack.config.js** :

- Modifier le fichier afin d'avoir un dossier **scripts** et un dossier **styles** dans assets, chacun ayant un main.ts et l'un main.scss



SYMFONY : LANCER NOTRE PROJET

Il nous reste une dernière chose à faire : importer nos fichiers générés par Webpack dans toutes les pages Web de notre projet, pour cela on va aller dans le **base.html.twig**, à la racine du dossier **templates**, et on va ajouter ces deux lignes :

```
{% block stylesheets %}
    {{ encore_entry_link_tags('styles') }}
{% endblock %}
{% block javascripts %}
    {{ encore_entry_script_tags('scripts') }}
{% endblock %}
```

- Ajouter Bootstrap au fichier main.scss : `@import '~bootstrap';`
- Lancer dans un terminal, faire un **yarn watch**, afin de compiler nos fichiers typescript et scss.
- Dans un autre terminal faire un : **symfony server:start**

SYMFONY : LES CONTROLLER 1 / 3

Lorsque vous lancez votre projet, vous devez avoir une erreur, indiquant :



You're seeing this page because you haven't configured any homepage URL and [debug_mode](#) is enabled.

Cela est dû au fait que l'on n'a pas défini d'URL « **homepage** », c'est-à-dire que l'on n'a pas de route par défaut lorsque l'on lance notre site.

Les routes dans Symfony sont gérées par les **controller** (dû au MVC), à la différence d'Angular, une route correspond à une fonction d'un controller.

On va donc créer un controller, avec la commande : **php bin/console make:controller**

```
Choose a name for your controller class (e.g. OrangeKangarooController):
```

```
> Home
```

```
created: src/Controller/HomeController.php
```

```
created: templates/home/index.html.twig
```

```
Success!
```

```
controller avec la commande : php bin/console make:controller
```

SYMFONY : LES CONTROLLER 2/3

Si l'on ouvre le fichier **HomeController.php**, il y a déjà une fonction qui a été créée :

- Le commentaire est une **annotation** php, c'est-à-dire qu'elle est interprétée par le code.
Ici il nous a créé par défaut une **route**, par **@Route**, le paramètre sans nom est le **path** qui sera appelé dans l'URL afin d'accéder à cette fonction, **name** est le nom de la route, utilisée pour communiquer entre les pages Symfony.
- La fonction se contente de renvoyer une **Response**, sous la forme d'une page html
- Si l'on remplace **"/home"** par **"/"** et que l'on remplace la page ?

```
#[Route('/', name: 'app_home')]
public function index(): Response
{
    return $this->render(view: 'front/home/index.html.twig', [
        'controller_name' => 'HomeController'
    ]);
}
```

SYMFONY : LES CONTROLLER 3/3

La fonction **render** est une fonction de l'**AbstractController** de Symfony, elle prend en paramètre un nom de template html et un tableau associatif qui correspondent aux paramètres que l'on souhaite passer à notre page html.

Ici on a deux paramètres que l'on souhaite faire passer à la vue : **controller_name** et **message**.

```
#[Route('/', name: 'app_home')]
public function index(): Response
{
    return $this->render(view: 'front/home/index.html.twig', [
        'controller_name' => 'HomeController',
        'message' => 'Coucou toi',
    ]);
}
```

SYMFONY : TWIG, LES BASES 1 / 3

On ouvre maintenant le fichier contenu dans le dossier home que Symfony nous a créé : **index.html.twig**

J'ai ajouté la ligne du if, mais concrètement twig nous permet d'afficher les variables que l'on lui passe via le tableau associatif, par des **{{...}}** comme en Angular !

Le if s'utilise par le biais de balise **{%...%}**, et il faut bien penser à la fermer avec un endif.

Voilà, nous avons fait notre première page Symfony !

```
<h1>Hello {{ controller_name }}! ✓</h1>

{% if message is defined %}
    {{ message }}<br>
{% endif %}
```

SYMFONY : TWIG, LES BASES 2/3

Auparavant, je vous ai dit qu'en ajouter deux lignes dans le base.html.twig on pouvait récupérer nos fichiers de css et typescript partout dans l'application, mais comment cela est possible ?

Si l'on regarde bien le fichier index.html.twig généré par Symfony pour notre controller, il y a ces lignes en début de fichier :

- **Extends** : prends en paramètre un nom de template, cela signifie que le contenu du template en question sera actif sur la page actuelle, comme un système d'héritage.
- **Block body** : ce bloc est aussi présent dans le base.html.twig, mais il est vide, ici il est « redéfini », ainsi s'il est redéfini twig affichera les éléments du dernier template.

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

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

{% block body %}
```


SYMFONY : TWIG, LES BASES 3/3

Il y a une notion importante dans Symfony et Twig, c'est celle des routes, car c'est quand même plus intéressant lorsque l'on peut naviguer d'une page HTML à une autre, non ?

Pour cela il y a une **fonction twig** qui permet de « naviguer » d'une page à une autre, en utilisant les **routes**, on va utiliser la fonction **path** :

```
<a href="{{ path('student_index') }}">
    Student
</a>
```

Cela implique que l'on ait une route ayant la valeur « **student_index** » pour l'**attribut** « **name** »

SYMFONY : LES ENTITES 1 / 5

En Symfony, les Entités sont ce que l'on appelle les classes « métier », autrement dit les classes correspondant aux différents objets nécessaires au fonctionnement de notre site web.

Tout comme les controllers, Symfony nous offre une commande permettant de créer les différentes entités de notre site web : **php bin/console make:entity**

Un point intéressant ici, c'est que l'on voit les classes que Symfony nous a créées : Student.php, soit l'entité que l'on a demandé de créer.

Et un StudentRepository.php, il s'agit de la classe d'accès à la base de données pour permettre la récupération de nos Student, en plus de nous créer directement les objets !

```
Class name of the entity to create or update (e.g. AgreeablePuppy):  
> Student  
Student  
  
created: src/Entity/Student.php  
created: src/Repository/StudentRepository.php  
  
Entity generated! Now let's add some fields!  
You can always add more fields later manually or by re-running this command.
```

SYMFONY : LES ENTITES 2/5

Reprenons notre terminal, Symfony nous demande maintenant d'ajouter des propriétés à notre entité, ici j'ai ajouté « **name** », qui est un **string de taille 255 et non-null**.

Symfony nous indique ensuite « **updated src/Entity/Student.php** », allons ouvrir ce fichier et voir ce qu'il a changé à l'intérieur.

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

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/Student.php
```

SYMFONY : LES ENTITES 3/5

Indiquons à Symfony que l'on ne veut pas ajouter d'autres propriétés à notre classe.
Il nous indique une nouvelle commande à passer :

Success!

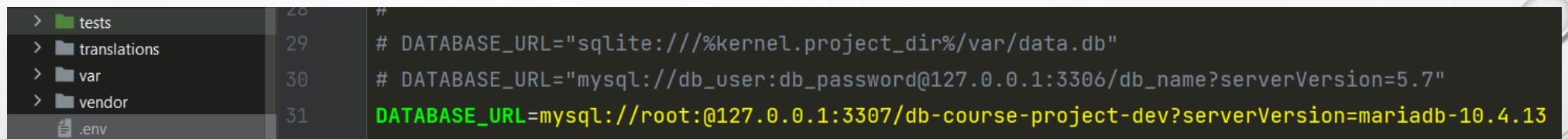
Next: When you're ready, create a migration with `php bin/console make:migration`

php bin/console make:migration : il s'agit de créer une migration Symfony, mais qu'est-ce qu'une migration ?

Symfony nous permet de générer des migrations, qui correspondent aux changements entre nos entités et la base de données, qui nous permette de générer la base de données directement en fonction de nos entités !

SYMFONY : LES ENTITES 4/5

Avant de passer cette commande, il faut indiquer à Symfony dans quelle base de données créer la table représentant notre entité, pour cela il faut configurer le **.env** à la racine du projet (copier/coller => renommer en env.local) :



The image shows a file explorer on the left with a tree view containing folders 'tests', 'translations', 'var', and 'vendor', and a file '.env'. To the right, a code editor displays the contents of the .env file. Line 28 is a comment '#'. Line 29 is a comment '# DATABASE_URL="sqlite:///kernel.project_dir%/var/data.db"'. Line 30 is a comment '# DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=5.7"'. Line 31 is the active configuration: 'DATABASE_URL=mysql://root:@127.0.0.1:3307/db-course-project-dev?serverVersion=mariadb-10.4.13'.

```
28 #  
29 # DATABASE_URL="sqlite:///kernel.project_dir%/var/data.db"  
30 # DATABASE_URL="mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=5.7"  
31 DATABASE_URL=mysql://root:@127.0.0.1:3307/db-course-project-dev?serverVersion=mariadb-10.4.13
```

Il faut remplacer la valeur par défaut de Symfony pour la clé DATABASE_URL, par l'URL de notre base de données ici, j'ai indiqué que mon projet aura la base de données de nom « **db-course-project-dev** »

Il reste encore une chose à faire avant de lancer la commande que Symfony nous a donné auparavant, c'est la création de la base de données :

php bin/console doctrine:database:create

SYMFONY : LES ENTITES 5/5

Une fois les commandes lancées, Symfony nous affiche ceci :

Success!

Il faut remplacer la valeur par défaut de Symfony pour la clé DATABASE_URL, par l'URL de « select-dev »
Next: Review the new migration `"migrations/Version20211229142456.php"`
Then: Run the migration with `php bin/console doctrine:migrations:migrate`
See <https://symfony.com/doc/current/bundles/DoctrineMigrationsBundle/index.html>

Il a créé un fichier de migration dans le dossier migrations et nous propose maintenant une nouvelle commande : **php bin/console doctrine:migrations:migrate -n**

Cette commande exécute les migrations non-passées en base de données !

SYMFONY : DOCTRINE

Symfony utilise Doctrine comme ORM (Object Relational Mapping), il s'agit de la couche relationnelle permettant de faire le lien (ou mapping) entre les objets et éléments de la base de données.

Ainsi, lorsque l'on fera nos appels à la base de données, c'est Doctrine qui créera directement nos objets en fonction de ce qu'il y a en base de données, et de faire la relation entre un champ d'une table à une propriété d'une classe.

Doctrine propose par défaut des requêtes simplifiées pour récupérer nos objets, dans le **xxxxRepository.php**, mais on peut en créer nous même, pour cela Doctrine utilise le DQL (Doctrine Query Language) afin de simplifier la rédaction de nos requêtes SQL en PHP.

SYMFONY : REPOSITORY

On a vu que les repository sont liés à une Entité et c'est Symfony qui les gère pour nous, mais maintenant comment peut-on les réutiliser et récupérer nos objets depuis la base de données ?

A chaque fois que vous allez vouloir récupérer des objets depuis de la base de données, vous devrez utiliser un Repository, il s'appelle par l'injection de dépendance, là où vous en avez besoin :

```
#[Route('/users', name: 'app_user_index')]
public function studentIndex(UserRepository $userRepository): Response
{
    return $this->render( view: 'front/home/index.html.twig', [
        'users' => $userRepository->findAll()
    ]);
}
```

SYMFONY : ENTITY RELATION 1 / 2

Il s'agit des relations entre les entités de l'application, c'est-à-dire qu'une région a plusieurs départements, et qu'un département appartient à une région, **on parle de relation**.

Symfony a un système de relation intégrée lorsque l'on crée ou met à jour une entité. Il permet de nous guider pour choisir la bonne relation entre nos entités.

Par exemple, ici j'ai créé une classe Grade, et je veux dire qu'un Student a plusieurs Grade et qu'un Grade a un Student, je devrais choisir laquelle ?

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

```
> relation
relation
```

```
What class should this entity be related to?:
```

```
> Grade
Grade
```

```
What type of relationship is this?
```

Type	Description
ManyToOne	Each Student relates to (has) one Grade . Each Grade can relate to (can have) many Student objects
OneToMany	Each Student can relate to (can have) many Grade objects. Each Grade relates to (has) one Student
ManyToMany	Each Student can relate to (can have) many Grade objects. Each Grade can also relate to (can also have) many Student objects
OneToOne	Each Student relates to (has) exactly one Grade . Each Grade also relates to (has) exactly one Student .

SYMFONY : ENTITY RELATION 2/2

J'avais créé la relation en provenance de **Account**, il faut un **OneToMany** : un **Account** a plusieurs **Comment**, un **Comment** appartient à un **Account**. Du coup dans l'entité **Comment**, ça sera l'inverse :

Account

```
#[ORM\OneToMany(mappedBy: 'account', targetEntity: Comment::class)]  
private Collection $comments;
```

Comment

```
#[ORM\ManyToOne(targetEntity: Account::class, inversedBy: 'comments')]  
#[ORM\JoinColumn(nullable: false)]  
private Account $account;
```


SYMFONY : SERVICE

Il est possible de créer des Service en Symfony, au même titre qu'Angular, ils sont là pour nous rendre un service, ou éviter de dupliquer un morceau de code, car il pourra être appelé à plusieurs endroits.

Il n'y a pas de commandes pour le faire, je vous recommande de faire un dossier **Service** dans le **src** et de mettre vos Service à l'intérieur.

Et c'est tout ce qu'il y a à savoir pour les Service... Juste qu'ils appellent par injection de dépendances dans les controllers, ou autre service où vous en avez besoin !

SYMFONY

DOCTRINE AVANCÉ

On a vu que doctrine nous propose des fonctions par défaut, cependant il peut-être parfois judicieux de faire nos propres requêtes, pour cela nous allons utiliser le **QueryBuilder** :

Dans un repository, on peut utiliser la fonction **createQueryBuilder**, elle prend en paramètre un alias pour la table sur laquelle on fait notre requête (celle de notre repository donc), le **andWhere** permet d'effectuer des conditions where.

Vous noterez qu'il y a un **:val** dans ce where, il s'agit d'un paramètre à notre requête, c'est-à-dire que l'on peut dynamiser notre requête comme on le souhaite.

Par contre, du moment où l'on a un paramètre dans notre requête, Doctrine attendra un « **setParameter** » pour lui préciser sa valeur.

```
public function findByExampleField($value)
{
    return $this->createQueryBuilder('s')
        ->andWhere('s.exampleField = :val')
        ->setParameter('val', $value)
        ->orderBy('s.id', 'ASC')
        ->setMaxResults(10)
        ->getQuery()
        ->getResult()
    ;
}
```

SYMFONY

DOCTRINE AVANCÉ : EXEMPLE

- Il existe aussi la méthode « **getOneOrNullResult** » (à la place du « **getResult** ») qui vous renvoie directement l'objet (sans passer par un tableau) ou null s'il n'a pas été trouvé.
(Il faut donc penser à gérer le null par la suite...)

```
// FROM table => ici game car je suis dans le repository de game
return $this->createQueryBuilder('game')
    // Différents SELECT avec un .* => game.*, genres.* etc
    ->select('game', 'genres', 'publisher', 'countries', 'comments')
    // Les différents join de table => penser à utiliser le LEFT JOIN
    // Si la relation PEUT être null
    ->join('game.genres', 'genres')
    ->join('game.countries', 'countries')
    ->leftJoin('game.comments', 'comments')
    ->leftJoin('game.publisher', 'publisher')
    // WHERE game.slug = $slug (penser au setParameter car il y a un :XXX)
    // :XXX => c'est un alias de paramètre dans votre DQL
    // Pour autant de :XXX dans vos where, vous avez autant de setParameter
    ->where('game.slug = :slug')
    ->setParameter('slug', $slug)
    ->getQuery()
    ->getResult()
```

SYMFONY : PARAMETRE DE ROUTE 1 / 3

Il existe plusieurs manières de récupérer les paramètres d'une route, la première se fait en utilisant l'objet **Request** de Symfony (Du package **HttpFoundation** !) :

Il suffit de d'appeler la fonction « **get** », depuis l'objet request avec en paramètre de la fonction, le nom du paramètre dans la route (celui entre accolades).
Ici il s'agit de « email »

```
#[Route('/account/{email}', name: 'app_account_show')]  
public function show(Request $request): Response {  
    return $this->render(view: 'front/account/show.html.twig', [  
        'show' => 'show',  
        'email' => $request->get(key: 'email'),  
    ]);  
}
```

SYMFONY : PARAMETRE DE ROUTE 2/3

Cette fois-ci, en ajoutant en paramètre de la fonction de la route une variable du même nom que le paramètre de la route : ici « **name** »

Symfony effectue implicitement le
« **\$request->get('name')** »

```
#[Route('/account/{name}', name: 'app_account_show')]  
public function show(string $name): Response {  
    return $this->render(view: 'front/account/show.html.twig', [  
        'show' => 'show',  
        'name' => $name,  
    ]);  
}
```


SYMFONY : PARAMETRE DE ROUTE 3/3

Admettons que « **name** » soit un attribut unique d'une entité, ici c'est l'entité **Account**, si Symfony trouve l'attribut « **name** » et qu'il existe bien un champ en base de données « **name** » dans la table **Account**, il va récupérer directement l'objet !

Si vous regardez le profiler Symfony, vous verrez qu'il y a une requête SQL qui est passée :

```
SELECT *  
FROM account  
WHERE account.name = {name} }
```

```
#[Route('/account/{name}', name: 'app_account_show')]  
public function show(Account $account): Response {  
    return $this->render(view: 'front/account/show.html.twig', [  
        'show' => 'show',  
        'account' => $account,  
    ]);  
}
```

Où {name} est la valeur du paramètre de la route.

SYMFONY

TWIG EXTENSION 1 / 5

Twig permet d'utiliser des filtres prédéfinis, mais il est possible d'en faire nous même !

Pour cela on utilise la commande : `symfony console make:twig-extension`

Un prompt s'ouvre et vous pouvez renseigner le nom de l'extension twig à créer.

```
λ symfony console make:twig-extension
```

```
The name of the Twig extension class (e.g. AppExtension):
```

```
> Excerpt
```

```
created: src/Twig/ExcerptExtension.php
```

```
Success!
```

SYMFONY

TWIG EXTENSION 2/5

A l'intérieur du fichier créé, vous avez une fonction **getFilters** :

- Cette fonction permet de créer des filtres.
 - Le premier paramètre de la classe **TwigFilter** est le nom du filtre à utiliser dans twig.
 - Le deuxième paramètre est la fonction à appeler et où elle se situe.
-
- Le paramètre de la fonction « **excerpt** » est du même type que celui sur lequel vous placez le filtre.
(ici chaînes de caractères)

```
public function getFilters(): array
{
    return [
        // Premier param : nom dans le twig
        // Deuxième param : l'action à effectuer, on indique
        new TwigFilter('excerpt', [$this, 'excerpt']),
    ];
}

/**
 * @param string $value => le type du paramètre est celui sur lequel on place le filtre
 * @return string
 */
public function excerpt(string $value): string
{
    return $this->excerptService->excerpt($value);
}
```

SYMFONY

TWIG EXTENSION 3/5

Dans l'html, vous pouvez maintenant appeler le filtre récemment créé (penser au « | ») :

```
<p>  
    {{ comment.content|excerpt }}  
</p>
```

Et ainsi « **comment.content** » sera le paramètre de la fonction « **excerpt** » dans la classe du file Twig.

SYMFONY

TWIG EXTENSION 4/5

Dans le cadre d'une fonction, on la définit dans le **getFunctions** :

- Les paramètres sont identiques qu'au filtre
- Ici ma fonction renvoie le résultat d'un findAll.
(très utile pour les navbar notamment)

```
public function getFunctions(): array
{
    return [
        new TwigFunction('functionTwigGetGenres', [$this, 'getGenres']),
    ];
}

public function getGenres(): array
{
    return $this->genreRepository->findAll();
}
```


SYMFONY

TWIG EXTENSION 5/5

Dans l'html il suffit de reprendre le nom de la fonction avec les « () » :

```
{% for genre in functionTwigGetGenres() %}  
  <li class="nav-item">  
    <a class="nav-link active" aria-cur  
      {{ genre.name }}  
    </a>  
  </li>  
{% endfor %}
```

SYMFONY : TRANSLATION 1 / 2

Il existe dans Symfony, les « **translations** », à quoi cela sert ?

Il s'agit de clés de traduction, c'est-à-dire qu'au lieu d'écrire un contenu, par exemple pour un bouton, « Annuler », on va préférer lui donner la valeur d'une clé de traduction, comme : « **button.cancel** », et en fonction de la langue en cours, Symfony ira chercher la valeur de cette clé. C'est aussi utile car on centralise les différents labels ou texte de notre site à un seul endroit.

Pour configurer les translations, il faut modifier le fichier **config/packages/translation.yaml** :

```
framework:
    default_locale: fr
    translator:
        default_path: '%kernel.project_dir%/translations'
        fallbacks:
            - fr
```

SYMFONY : TRANSLATION 2/2

Il faut ensuite aller dans le dossier « **translations** », normalement créé par défaut par Symfony, qui doit être actuellement vide.

On va créer un fichier « **message.fr.yaml** », vous noterez la présence de « **fr** », il s'agit du repère de Symfony pour aller chercher les clés de traduction dans la langue demandée :

```
student:  
    name: 'Nom'
```

Ainsi, lorsque l'on utilisera la clé : « **student.name** », « **Nom** » sera affiché !

On peut l'utiliser dans un template twig en utilisant le filtre « **trans** » :

```
{{ 'student.name'|trans }}
```

SYMFONY : PAGINATOR INSTALLATION

La pagination avec Symfony se fait avec le bundle KnpPaginator,
l'installation se fait avec la commande : `composer require knplabs/knp-paginator-bundle`

Vérifier qu'il a bien été ajouté dans le fichier « config/bundles.php » :

```
return [  
    Knp\Bundle\PaginatorBundle\KnpPaginatorBundle::class => ['all' => true],  
];
```

Lien vers la doc du bundle :

<https://github.com/KnpLabs/KnpPaginatorBundle>

SYMFONY : PAGINATOR UTILISATION

Dans la fonction ou le constructeur du contrôleur dont vous souhaitez ajouter la pagination :

- Importer `PaginatorInterface $paginator` `Request $request`

- A l'intérieur de la fonction renvoyant sur un template nécessitant une pagination :

```
$comments = $paginator->paginate(  
    $commentRepository->getQueryBuilderByGame($slug),  
    $request->query->getInt('page', 1),  
    6  
);
```

- On appelle la fonction « **paginate** » de l'objet `PaginatorInterface`, le premier paramètre est un **queryBuilder**, récupérer du repository souhaité (Donc **PAS** de **getQuery** et **getResult**)
- En deuxième paramètre à « **paginate** » on récupère la requête http, précisément la « **page** »

- `Knnpaginator` redéfinit les requêtes http et ajoute le paramètre « **page** » à celles-ci.
- « **10** » représente la limit, soit le nombre d'entités par page

SYMFONY : PAGINATOR TWIG

Si vous souhaitez ajouter les filtres « Order by » sur vos objets, il faut utiliser :

- « **orders** » est le résultat du « **paginate** » côté php
- « **label** » est le nom affiché, qui permettra de faire les filtres « order by »
- « **o.id** » provient de l'alias du queryBuilder, en effet si vous avez déclaré dans votre queryBuilder un alias de nom « toto », vous devrez utiliser cet alias dans votre twig.
- Il ne faut pas oublier le « **|raw** » à la fin ! Car Knp Paginator nous rajoute du code Html, avec le filtre twig « **raw** » cela permet de l'interpréter directement.
- Pour appeler le template de la pagination de Knp, il faut utiliser « **knp_pagination_render(NOM_PARAM_ENTITE_A_PAGINER)** »

```
return $this->createQueryBuilder( alias: 'account')
    ->select( select: 'account', 'libraries')
    ->leftJoin( join: 'account.libraries', alias: 'libraries')
    ;
```

```
{{
    knp_pagination_sortable(
        accounts,
        'Nom',
        'account name'
    )|raw
}}
```

```
<div class="navigation d-flex justify-content-center mt-2">
    {{ knp_pagination_render(comments) }}
</div>
```

SYMFONY : ENTITYMANAGER

On le récupère via un **EntityManagerInterface**, mais à quoi sert-il ?

Il s'agit de l'entité capable d'indiquer à Doctrine que l'on souhaite insérer des objets en base de données !

Pour cela on utilise deux fonctions :

```
public function __construct(  
    EntityManagerInterface $entityManager  
)  
{  
    $this->entityManager = $entityManager;  
}
```

```
$this->entityManager->persist($move);  
$this->entityManager->flush();
```

SYMFONY : LES FORMULAIRES 1 / 4

Afin de réaliser les formulaires, Symfony nous propose encore... une commande pour nous faciliter la tâche :
php bin/console make:form

Le terminal vous demande ensuite le nom du formulaire et sur quelle entité il doit se baser pour le faire :

```
The name of the form class (e.g. VictoriousChefType):  
> Student  
  
The name of Entity or fully qualified model class name that the new form will be bound to (empty for none):  
> Student  
Student  
  
created: src/Form/StudentType.php  
  
Success!
```

SYMFONY : LES FORMULAIRES 2/4

Dans un controller, si vous voulez créer un formulaire et le passer au template html, vous devez d'abord le créer, via la méthode « **createForm** ».

Tout comme Angular, si vous lui passer un objet vide, les champs du formulaire seront vides, si l'objet a des valeurs, alors elles seront affichées.

Enfin, pour passer le formulaire au template, il faut utiliser la fonction **createView**

```
public function createAccount(Request $request): Response {  
    $form = $this->createForm(type: AccountType::class, new Account());  
    $form->handleRequest($request);  
    if ($form->isSubmitted() && $form->isValid()) {  
        dump($form->getData());  
    }  
    return $this->render(view: 'account/new.html.twig', [  
        'form' => $form->createView(),  
    ]);  
}
```

SYMFONY : LES FORMULAIRES 3/4

Enfin, pour afficher notre formulaire dans l'HTML, il faut appeler « form_start » et « form_end ».
Si l'on ne précise pas de « form_widget », Symfony va directement afficher tous les champs de notre formulaire à la suite, sinon il affichera ceux que l'on a demandé puis les autres.

```
{% block body %}  
    {{ form_start(form) }}  
        {{ form_widget(form.name) }}  
    {{ form_end(form) }}  
{% endblock %}
```

(PS : pour ne pas afficher un label, il faut dans l'AbstractType, déclarer le label => false)

```
'label' => false,
```


SYMFONY : LES FORMULAIRES 4/4

Dans le cadre de relation « ManyToOne » on doit utiliser un « EntityType » afin de renseigner directement l'entité en relation avec notre entité initiale.

- « **choice_label** » : le nom de la propriété à présente dans l'entité en relation, à afficher dans la liste déroulante
- « **class** » : l'entité en relation
- « **query_builder** » : la requête à passer pour trier les valeurs à afficher de l'entité en relation

```
->add('country', EntityType::class, [  
    'choice_label' => 'nationality',  
    'class' => Country::class,  
    'query_builder' => function (EntityManager $em) {  
        return $em->createQueryBuilder('p')  
            ->orderBy('p.nationality', 'ASC')  
    };  
])
```

SYMFONY : STYLISER UN FORMULAIRE 1 / 2

La solution de facilité est de le faire via Bootstrap, car Symfony inclue des templates par défaut pour nos formulaires, il suffit d'indiquer à Symfony et à Twig de les utiliser. Pour cela il faut modifier le fichier **config/packages/twig.yaml** comme ceci :

```
twig:  
    default_path: '%kernel.project_dir%/templates'  
    form_themes: ['bootstrap_5_layout.html.twig']
```

Et dans un template ayant un formulaire, il faut faire cette modification :

```
{% form_theme form 'bootstrap_5_horizontal_layout.html.twig' %}
```

Où « **form** » est le nom du formulaire passé au template twig.

SYMFONY : STYLISER UN FORMULAIRE 2/2

Une solution plus complexe, mais forcément plus modulable, est de le faire dans le code :

Il existe plusieurs options, notamment masquer le label, modifier le nom du label (car par défaut Symfony va mettre le même que le nom du champ), etc...

```
$builder  
    ->add( child: 'name', type: TextType::class, [  
        'attr' => [  
            'class' => 'col-6'  
        ]  
    ]  
);
```

SYMFONY

LEXIK FILTER 1 / 3

Ajouter le bundle au projet : **symfony composer require lexik/form-filter-bundle**

Le but de ce bundle est d'ajouter des champs de recherche dans les tables des CRUD, entre autres, afin de faciliter les recherches.

Lien vers la documentation : <https://github.com/lexik/LexikFormFilterBundle>

Injecter le
« **FilterBuilderInterface** »
dans une fonction où vous
souhaitez ajouter des filtres.

```
use Lexik\Bundle\FormFilterBundle\Filter\FilterBuilderInterface;

#[Route('/admin/account')]

class AccountController extends AbstractController
{
    #[Route('/', name: 'app_admin_account_index')]
    public function index(
        AccountRepository $accountRepository,
        PaginatorInterface $paginator,
        FilterBuilderInterface $builderUpdater,
```

SYMFONY

LEXIK FILTER 2/3

Dans la fonction en question :

- Se créer un form « xxxFilterType », **ne passez pas par la commande !**
- Il faut ensuite vérifier si la query a un paramètre du formFilter en cours.
- Si c'est le cas, alors on l'ajoute dans le queryBuilder

De la même manière que **Knnpaginator**, il va venir intervenir dans le query builder en ajoutant les conditions « **where** » en fonction de ce que l'on a saisi.

Il s'intègre très bien avec Knnpaginator !

```
$qb = $accountRepository->getQbAll();

$filterForm = $this->createForm(AccountFilterType::class, null, [
    'method' => 'GET',
]);

if ($request->query->has($filterForm->getName())) {
    $filterForm->submit($request->query->get($filterForm->getName()));
    $builderUpdater->addFilterConditions($filterForm, $qb);
}

$accounts = $paginator->paginate(
    $qb,
    $request->query->getInt('page', 1),
    15
);
```


SYMFONY

LEXIK FILTER 3/3

Dans l'HTML, il s'agit d'un formulaire normal, il faut donc bien penser à déclarer un « **form_start(formFilter)** » et un « **form_end(formFilter)** ».

Souvent, on place les champs de recherche en entête de table, cette fois-ci on va utiliser le « **form_widget(nom_champ)** » pour placer les champs de recherche où l'on veut.

```
<th>
  {{
    knp_pagination_sortable(
      accounts,
      'Nom',
      'account.name'
    )|raw
  }}
  {{ form_widget(filters.name) }}
</th>
```

SYMFONY

FILE UPLOADER 1 / 2

Il faut se créer un FileUploader, permettant de récupérer le fichier dans la requête http et le déplacer sur le serveur :

Notez une chose importante :
L'injection de « **\$uploadDir** ».

Il s'agit d'un param dont la valeur a été déclarée dans le « **services.yaml** »

```
public function __construct(
    private string $uploadsDir
) {}

public function uploadFile(UploadedFile $uploadedFile, string $namespace = ''): string
{
    $destination = $this->uploadsDir.$namespace;
    $originalFilename = pathinfo($uploadedFile->getClientOriginalName(), PATHINFO_FILENAME);
    $newFilename = $originalFilename.'-'.uniqid().'.'.$uploadedFile->guessExtension();
    $uploadedFile->move($destination, $newFilename);
    return '/uploads'.$namespace.'/'.$newFilename;
}
```

services:

*# default configuration for services in *this* file*

_defaults:

autowire: true # Automatically injects dependencies in your services.

autoconfigure: true # Automatically registers your services as commands, event subscribers, etc.

bind:

\$uploadDir: '%kernel.project_dir%/public/uploads'

SYMFONY

FILE UPLOADER 2/2

Dans le FormType de l'entité, déclarer un champ de type « **FileType** » :

Il doit avoir le « **mapped => false** » et « **required => false** », afin de ne pas laisser Symfony gérer le champ du formulaire, mais nous allons le faire en code.

Puis, dans le controller du formulaire, il faut vérifier si la valeur du champ existe, et appeler le **\$fileUploader**, déclaré page précédente.

```
->add('pathImage', FileType::class, [
    'label' => 'Image profil',
    'mapped' => false,
    'required' => false,
    'constraints' => [
        new File(
            maxSize: '2048k',
            mimeTypes: ['image/png', 'image/jpeg'],
            mimeTypesMessage: 'Ce format d\'image n\'est pas pris en compte',
        )
    ]
])
```

```
if ($form->get('pathImage')->getData() !== null) {
    $file = $fileUploader->uploadFile(
        $form->get('pathImage')->getData(),
        '/profile'
    );
    $data->setPathImage($file);
}
```

SYMFONY

FORM COLLECTION 1/2

Il faut déclarer un **CollectionType**, les propriétés importantes sont :

- 'entry_type' => représente quel type de form sera inclus dans la collection, ici un EntityType
- Il est impératif de mettre « **allow_add** » et « **allow_delete** » à **true** ! Afin de pouvoir autoriser l'ajout de form dans la collection, ainsi que la suppression
- Dans le « **attr** » on ajoute un attribut HTML sur lequel on se basera en Javascript après... (« **data-list-selector** »)
- « **Entry_options** » est comment « **entry_type** » va être utilisé, c'est-à-dire les différentes options que l'on va lui ajouter, comme si on avait déclaré un « **EntityType** » normalement, donc « **class** », « **choice_label** » et « **query_builder** ».

(Penser à prendre le JS dans « **collection_form.ts** »)

```
->add('countries', CollectionType::class, [
    'label' => 'Pays',
    'entry_type' => EntityType::class,
    'allow_add' => true,
    'allow_delete' => true,
    'attr' => [
        'data-list-selector' => 'countries'
    ],
    'entry_options' => [
        'label' => false,
        'class' => Country::class,
        'choice_label' => 'nationality',
        'query_builder' => function (EntityManager $em) {
            return $em->createQueryBuilder('e')
                ->orderBy('e.nationality', 'ASC')
        };
    ]
])
```

SYMFONY

FORM COLLECTION 2/2

- On ajoute ensuite un button, qui va permettre de supprimer le bloc
- Là encore, pour que le Javascript fonctionne, on ajoute le « **data-btn-selector** » qui doit avoir la même valeur que le « **data-list-selector** » du CollectionType

```
->add('addCountry', ButtonType::class, [  
    'label' => 'Ajouter un pays',  
    'attr' => [  
        'class' => 'btn btn-info',  
        'data-btn-selector' => 'countries',  
    ]  
)
```


SYMFONY : COMMANDS 1 / 2

Il est possible en Symfony de créer vos propres commandes, comme un **php bin/console make:controller**. L'intérêt est de pouvoir personnaliser et automatiser certains traitements, enfin en lançant la commande...

Pour cela il faut créer une classe qui étend l'objet **Command** de Symfony.

Il y a des fonctions obligatoires à implémenter :

- **configure** : qui correspond à la configuration de la commande (le nom de celle-ci), si elle a des arguments (paramètres) et définir sa description

```
protected function configure()
{
    $this
        ->setName('app:pokemon:all')
        ->addArgument('lang', InputArgument::REQUIRED, 'Language used')
        ->setDescription('Execute app:pokemon to fetch all pokemon for language');
}
```

SYMFONY : COMMANDS 2/2

Il y a aussi la fonction **execute**, qui, comme son nom l'indique, effectue le traitement attendu, comme par exemple créer des données et les insérer en base de données, ou modifier des objets, ou faire des appels vers une API pour récupérer les données et les charger dans notre base de données !

```
protected function execute(InputInterface $input, OutputInterface $output): int
{
    $output->writeln(messages: '');
    $output->writeln(messages: '<info>Fetching all pokemons...');
}

$lang = $input->getArgument(name: 'lang');
```

On peut récupérer les paramètres de notre commande via la fonction **getArgument** en reprenant la clé de l'argument attendu,

SYMFONY

KNP SNAPPY

Commande pour installer KnpSnappy : `symfony composer require knplabs/knp-snappy-bundle`

(PS : cette installation assume que vous ayez wkhtmltopdf d'installer et fonctionnel sur votre machine)

Doc : <https://github.com/KnpLabs/snappy>

Ajouter dans le dossier **config/packages**, un fichier **knp_snappy.yaml**, il doit ressembler à ça :

Il faut aussi penser à ajouter dans le config/bundle.php :

```
Knp\Bundle\SnappyBundle\KnpSnappyBundle::class => ['all' => true],
```

```
knp_snappy:

# dossier temporaire des pdfs
temporary_folder: "%kernel.cache_dir%/snappy"

# timeout de génération des pdfs
process_timeout: 60

pdf:
  enabled: true
  binary: 'Chemin vers wkhtmltopdf.exe sur votre machine'
  options: []
image:
  enabled: true
  binary: 'Chemin vers wkhtmltoimage.exe sur votre machine'
  options: []
```

SYMFONY

KNP SNAPPY

Pour utiliser KnpSnappy on doit importer la classe **Pdf** du namespace **Knp\Snappy**

On génère une page html depuis un template Twig, puis la fonction doit renvoyer une **PdfResponse**, on lui passe l'html généré et un nom de fichier pour notre pdf :

```
$html = $this->renderView('back/pdf/library.html.twig', [
    'account' => $account
]);

return new PdfResponse(
    $knpSnappyPdf->getOutputFromHtml($html),
    $account->getName().uniqid().'pdf'
);
```

```
use Knp\Snappy\Pdf;

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

class PdfController extends AbstractController
{

    #[Route('/pdf/{slug}', name: 'app_pdf_library')]
    public function pdfAction(
        Pdf $knpSnappyPdf,
        Account $account
```