Learning Symfony 7 : Fundamentals part 1

Symfonycasts : « Cosmic Coding with Symfony 7 »

*Documentation faite par Annaig en avril 2024*

20 vidéos en accès libre : <https://symfonycasts.com/screencast/symfony/setup>

Résumé, notes et commandes principales à retenir

Il est recommandé de regarder cette série de vidéos et de faire les manipulations montrées en même temps.

Table des matières

[01 : Installation d’une app Symfony 2](#_Toc165370853)

[02 : Découverte du squelette de Symfony 2](#_Toc165370854)

[03 : Routes, Controllers & Responses 2](#_Toc165370855)

[04 : Magical Flex Recipes 3](#_Toc165370856)

[05 : Twig & Templates 4](#_Toc165370857)

[06 : Héritage de template Twig 4](#_Toc165370858)

[07 : Outils de débugage 5](#_Toc165370859)

[08 : Création de endpoints pour API JSON 5](#_Toc165370860)

[09 : Les services, colonne vertébrale de Symfony 6](#_Toc165370861)

[10 : Créer son propre service 8](#_Toc165370862)

[11 : Amélioration de routes : paramètres, méthodes, 404, etc. 8](#_Toc165370863)

[12 : Générer des URL 10](#_Toc165370864)

[13 : CSS et JavaScript avec AssetMapper 11](#_Toc165370865)

[14 : Tailwind CSS 12](#_Toc165370866)

[15 : Twig partials & for-loop 13](#_Toc165370867)

[16 : PHP Enums 13](#_Toc165370868)

[17 : Méthodes de modèle personnalisées & templates dynamiques 14](#_Toc165370869)

[18 : Stimulus : writing pro JavaScript 15](#_Toc165370870)

[19 : Turbo : Your Single Page App 15](#_Toc165370871)

[20 : Maker Bundle 15](#_Toc165370872)

# 01 : Installation d’une app Symfony

Install the standalone Symfony binary : <https://symfony.com/download>

Checkings in terminal : symfony --help

symfony check :req

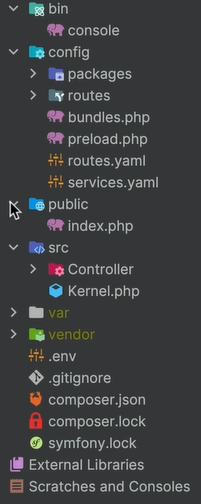
**Install a basic Sf app : symfony new my-project-name (add --webapp for more pre-installed features)**

Check what’s installed :

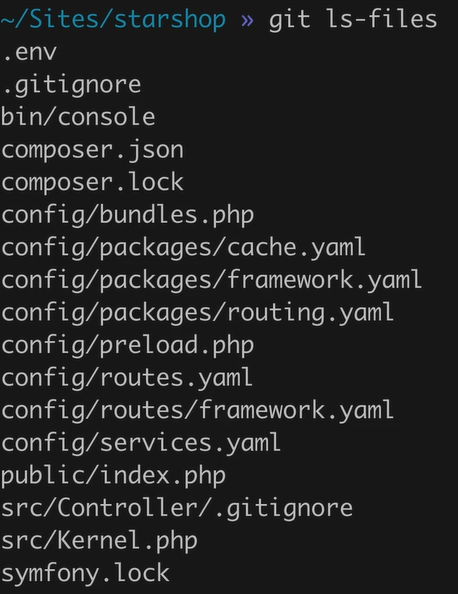
Start the server : symfony :serve 🡪 go to localhost to see app

symfony serve est idéal pour le développement local et les prototypes, tandis que symfony server:start est plus adapté pour les environnements de production ou les applications nécessitant des performances optimisées.

# 02 : Découverte du squelette de Symfony

****

**To stop the server :**

****ctrl+C

or

symfony server :stop

git log

**Important files :**

composer.json

vendor/symfony/

.gitignore

**Test :**

delete vendor/

& run composer install

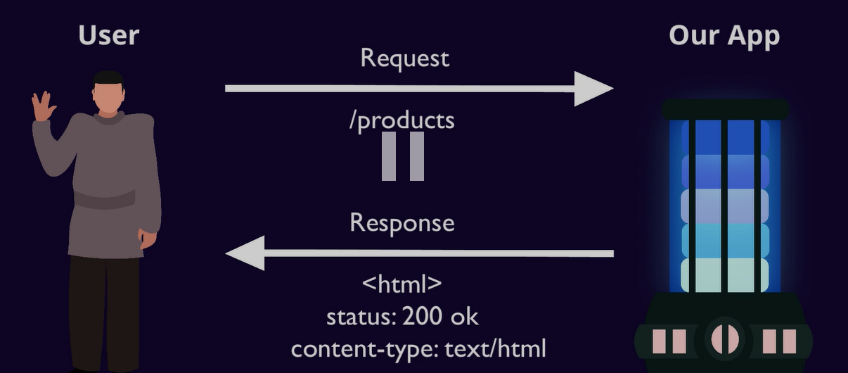
**Where you mostly work :**

config/

src/

# 03 : Routes, Controllers & Responses

**Controller = method in a PHP class that builds a page.**

****Rules :

* Every class must have a **namespace** matching the directory structure
* Your **class name** must match your file name – or it will be an error (not finding the class)

Route : **attribute** + add « use »

# 04 : Magical Flex Recipes

Symfony **skeleton** = 1 file !

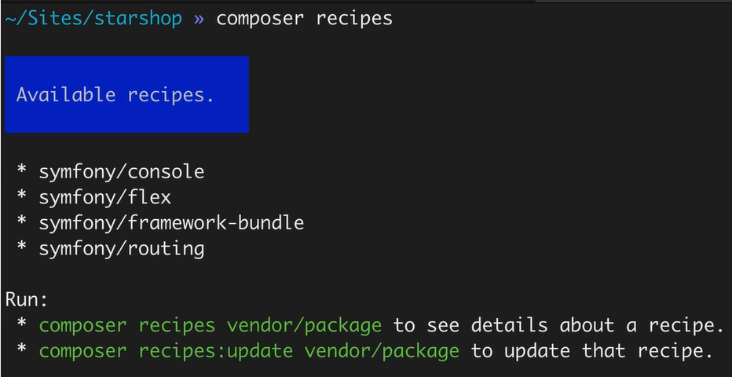
symfony/**flex** est un plugin Composer qui ajoute les aliases et les recipes

Alias

= shortcut quand on veut installer un package composer (avec la commande composer require suivie du nom du package ou d’un alias, plus facile)

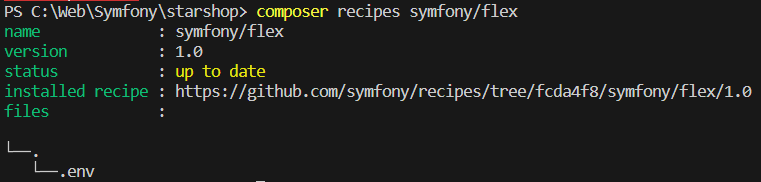
Voir tous les alias de Sf : <https://github.com/symfony/recipes/blob/flex/main/RECIPES.md>

Recipe

= set of files added to your project

Voir toutes les recettes du projet : composer recipes

Voir le détail d’une recette : composer recipes nomDeLaRecette



*le terme "recipe" en anglais, lorsqu'il s'agit de développement web ou de programmation, peut être traduit par "recette" en français. Ce terme est utilisé pour désigner un ensemble d'instructions ou de directives pour réaliser une tâche spécifique, comme la configuration d'un environnement de développement ou l'installation de dépendances via des gestionnaires de paquets comme Composer. Dans le contexte de la programmation, une "recette" peut inclure des commandes spécifiques à exécuter, des fichiers de configuration à modifier, ou des étapes à suivre pour configurer correctement un projet ou un environnement.*

*Par exemple, dans le domaine de la cuisine, une "recette" est une suite d'instructions pour réaliser un plat, et dans le domaine de la programmation, une "recette" peut être une suite d'instructions pour configurer un environnement de développement ou pour résoudre un problème technique spécifique.*

Exemple : installer CS-fixer

composer require cs-fixer-shim

🡪 modif de composer.json et composer.lock = comportement normal de composer

🡪 les autres modifs viennent de la recette

Voir toutes les commandes de cs fixer : ./vendor/bin/php-cs-fixer

Corriger les erreurs : ./vendor/bin/php-cs-fixer fix

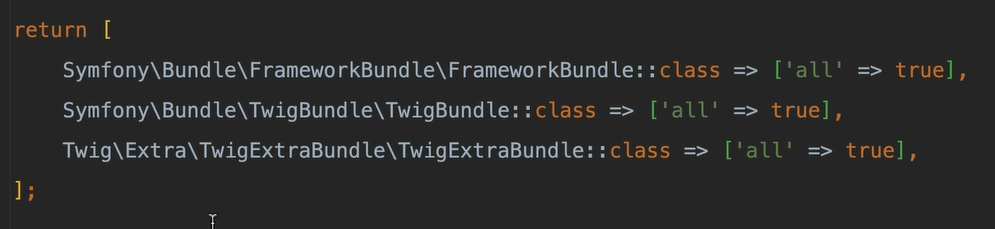
Corriger les erreurs dans src : ./vendor/bin/php-cs-fixer fix src

*CS fixer corrige conformément aux standards de codage PSR (indentation, espacements, guillements simples ou doubles… mais pas les erreurs de syntaxe).*

*On peut en exclure certains fichiers, configurer pour qu’il s’exécute automatiquement lors de commit ou de sauvegarde, configurer ses propres règles.*

# 05 : Twig & Templates

composer require twig (alias pour symfony/twig-pack)

Twig est une librairie de templates. En Sf, c’est un « pack » càd une sorte de faux package qui installe plusieurs packages d’un coup (cf. dans composer.json, il y a 3 nouveaux packages, aucun ne s’appelle twig-pack) .

* Il a été ajouté à config/bundles :
* Ça a créé un config/packages/twig.yaml, qui contrôle le comportement du bundle Twig (l’extension \*.twig par exemple)
* Et templates/base.html.twig

**Rendre le premier template :**

Controller :

* extends AbstractController
* $this->render(‘filename’, [variable, par exemple tableau]) ;
* La fonction « render » va chercher le html et le met dans l’objet Response

templates/ :

* créer un dossier du nom de la classe + un fichier du nom de la route (main/home.html.twig)
* ajouter du html
* pour ajouter une varible envoyée par Controller ::render() 🡪 {{ nomDeLaVariable }}
* on peut ajouter de la logique, par exemple if avec {% if %}

**Syntaxe :**

* {{ variable }}
* {% tag %}
* {# commentaire #}
* Rendre un tableau associatif : {{ tableau.clé }}

**tags, functions, filters, tests**

Cf. la doc

# 06 : Héritage de template Twig

Dans le fichier qui utilisera le template de base (par exemple le fichier de la page d’accueil), ajouter :

* En haut de la page {% extends ‘base.html.twig’ %}
* Entourant TOUT le contenu {% block body %} <tout le contenu html>{% endblock %}
* On peut override par exemple le titre dans le fichier enfant :
  + Dans le fichier base on a :

<title>{% block title %}Welcome!{% endblock %}</title>

* + Dans le fichier enfant on écrit :

{% block title %}New titel!{% endblock %}

* On peut garder le parent :

{% block title %}{{ parent() }} New titel!{% endblock %}

# 07 : Outils de débugage

composer require debug

pour installer le pack de dépendances pour le debugging développement :

* Monolog (librairie de log : var/log/)
* Profiler

**Le profiler**

Web debug toolbar : en bas de la page (passer la souris & cliquer dessus pour voir toutes les infos)

**La console**

php bin/console 🡪 toutes les commandes disponibles, par exemple :

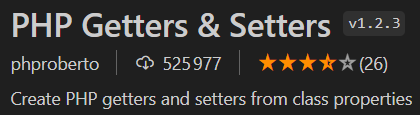
php bin/console debug  🡪 toutes les commandes de debug, par exemple :

php bin/console debug :router 🡪 toutes les routes de l’app

php bin/console debug :twig 🡪 toutes les fonctions, filtres etc. de twig… comme dans la doc + celles installées en plus par nos bundles spécifiques à l’app !

# 08 : Création de endpoints pour API JSON

1. **Créer un contrôleur** 
   * avec une route de type /api/starships
   * une variable sous forme de tableau associatif
   * une réponse en json : return $this->json($lavariable) ;
   * vérifier dans le navigateur que le tableau est bien rendu en json
2. **Créer un objet pour le rendre en json**
   * Créer une classe Model/Starship.php
   * Avec une fonction construct ayant en argument des propriétés = les clés du tableau associatif
   * Générer les getters automatiquement dans VSCode :



*Cette extension ajoute trois commandes à la palette de commandes de VSCode : "Insert PHP getter", "Insert PHP setter", et "Insert PHP getter and setter". Vous pouvez également accéder à ces commandes depuis le menu contextuel en* ***cliquant sur une propriété de classe***

* + Dans le contrôleur, remplacer le tableau par l’instanciation d’un nouvel objet : new Starship(<les valeurs des propriétés en arguments>)

1. **Utiliser le serializer**
   * La fonction json() utilise json\_encode(), insuffisant pour des propriétés privées : il faut le pack Serializer (désérialiser du json en objets ou … l’inverse) : composer require serializer
   * Json() utilise automatiquement le serializer s’il est enabled, pas besoin de config !

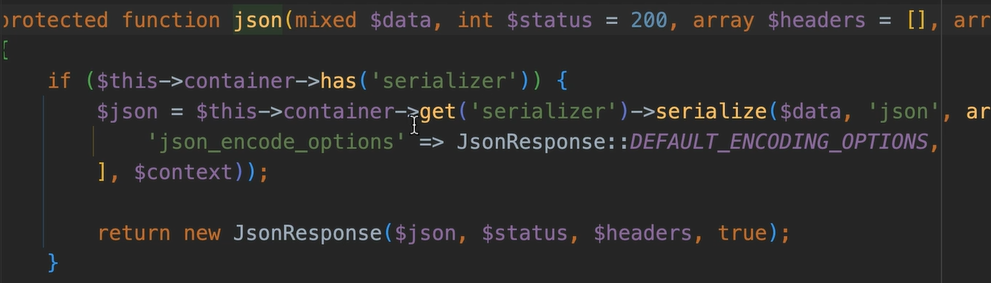
# 09 : Les services, colonne vertébrale de Symfony

Un sevice est un « objet qui marche » : un logger, un service de connection à la bdd… contrairement à une entité par exemple, qui est un objet, mais ne « fait » rien. Les contrôleurs sont des sevices, tout comme :

* Le routeur
* Le service qui écrit les messages de log dans le bon fichier
* Le service qui rend les vues twig
* …

Tous ces services sont organisés dans l’objet « Service container » : cf. bin/console debug :container

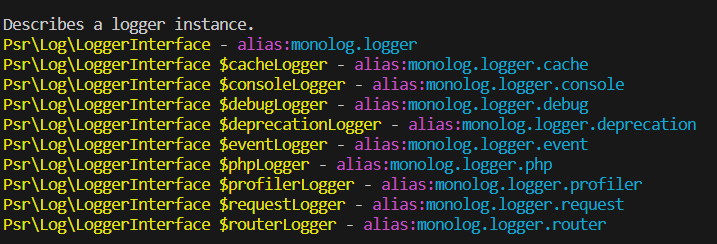
Il est par exemple utilisé dans la fonction json() :



Les bundles nous donnent des services (par ex. TwigBundle donne le service twig).  
Les services sont des outils

php bin/console debut :autowiring 🡪 liste des services autowireable (auto-wiring = technique pour les récupérer) = les services les plus courants.

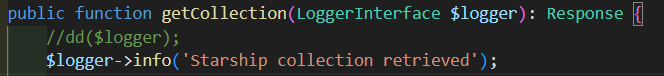
php bin/console debut :autowiring log 🡪 pour trouver un service de log ; résultat :



Ce service vient du MonologBundle, on peut le configurer dans **config\packages\**monolog.yaml.

Pour l’utiliser (le tout premier / principal) :





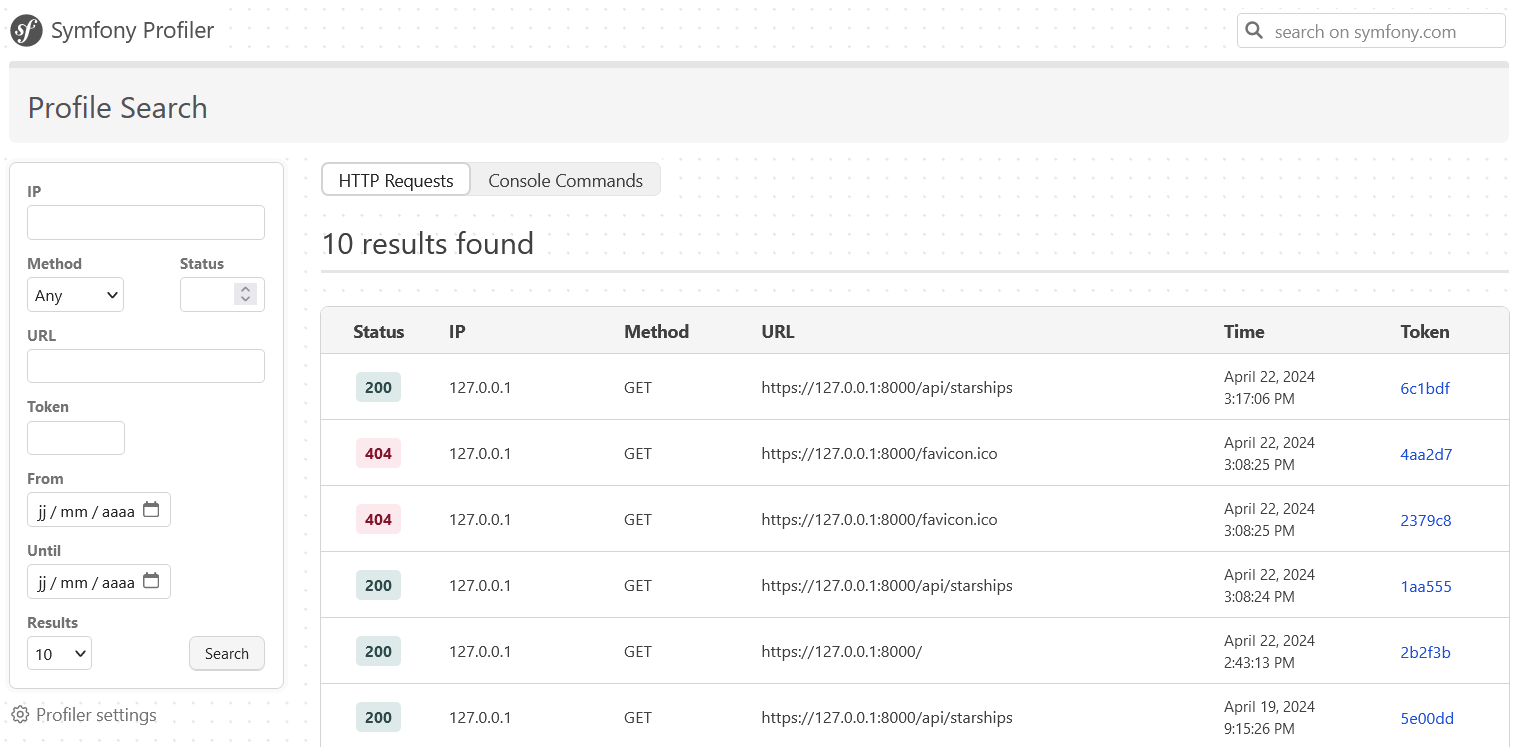
Cette « astuce » d’utilisation - appelée autowiring - fonctionne exactement à deux endroits :

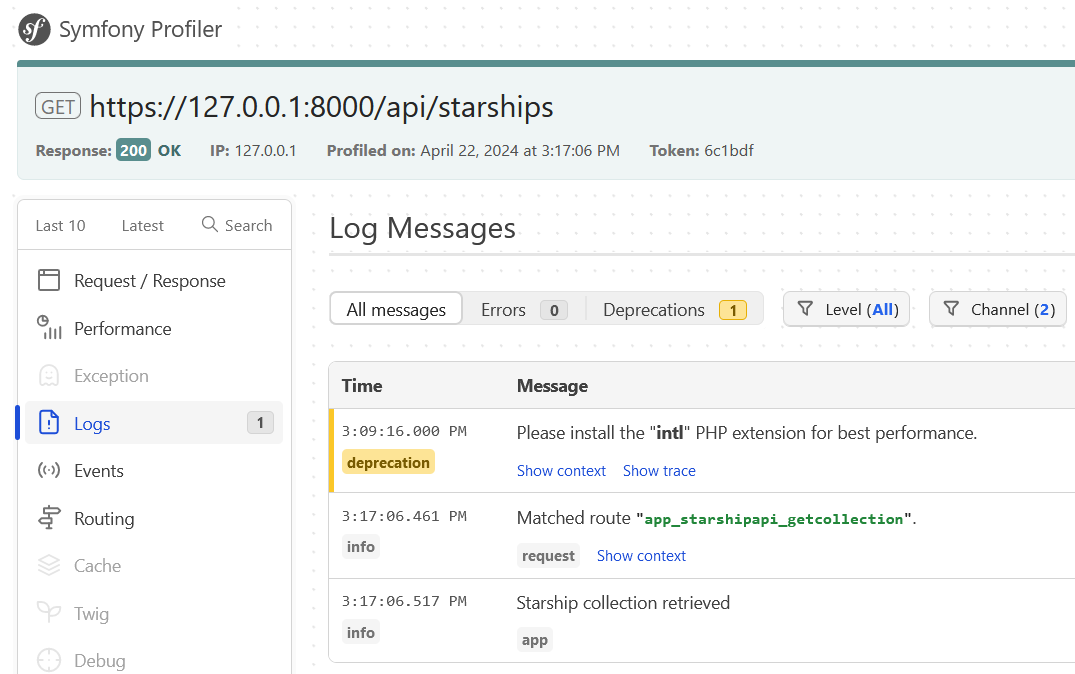
* nos méthodes de contrôleurs
* la méthode \_\_construct() de n'importe quel service (cf. chapitre suivant)

Résultats :

🡪 dans le var/log/dev.log

🡪 ou dans le profiler – pour une api, la debug bar n’apparaît pas, donc utiliser l’url **/\_profiler** :





Résumé : Nous savons que les services fonctionnent et nous savons que Symfony regorge de services que nous pouvons utiliser. Si vous exécutez : débogage php bin/console :autowiring, vous obtenez le **menu des services** du dîner, où vous pouvez **commander n'importe lequel** d'entre eux **en ajoutant un argument** avec un indice de type avec la classe ou l'interface correspondante (« adding an argument type-hinted with the matching class or interface ») : **'utilisation de l'autowiring pour injecter automatiquement les dépendances dans vos services**. L'autowiring est une fonctionnalité de Symfony qui permet de gérer les services dans le conteneur avec une configuration minimale. Il lit les **indices de type (type-hints)** sur le **constructeur d’une classe (ou des méthodes des classes contrôleurs)** de vos services et injecte automatiquement les services corrects pour chaque méthode.

# 10 : Créer son propre service

Quand on veut réutiliser du code qui « fait » quelque chose à plusieurs endroits de l’application, on crée un service (un contrôleur qui exécute ce service et sera réutilisable par plusieurs autres contrôleurs).

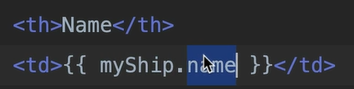
**Exemple 1 : créer une query complexe vers la database = service de Repository**

* créer src/Repository/StarshipRepository.php
* y mettre par exemple un fonction findAll() qui renvoie un tableau avec les données
* l’utiliser dans le contrôleur avec l’autowiring
* Sf crée automatiquement une instance de cet objet

**Exemple 2 :**

Utiliser le service de log dans le repo… possible, à condition de le passer en argument de la méthode \_construct() obligatoirement. C’est la façon « normale » d’utiliser l’autowiring, et non pas en passant le service en argument d’une autre méthode, ce qui ne marche que pour les contrôleurs.

**Utiliser le service (Repository) sur une autre page :**

* Sur la page d’accueil, construire un nouvel objet starship
* Transformer le tableau associatif en objet 🡪 Twig = pas besoin de changer la syntaxe, c’est la même pour accéder à la propriété d’un objet (par le getter) ou à l’élément d’un tableau
* On peut même faire le compte directement dans Twig avec le filtre length

# 11 : Amélioration de routes : paramètres, méthodes, 404, etc.

**Ajouter un paramètre à une route (wildcard)**

* Le paramètre id est entre {} et en argument de la méthode :

    #[Route('/api/starships/{id}')]

    public function get($id) : Response {…}

* on ajoute une expression régulière au paramètre et un type à l’argument :

    #[Route('/api/starships/{id<\d+>}')]

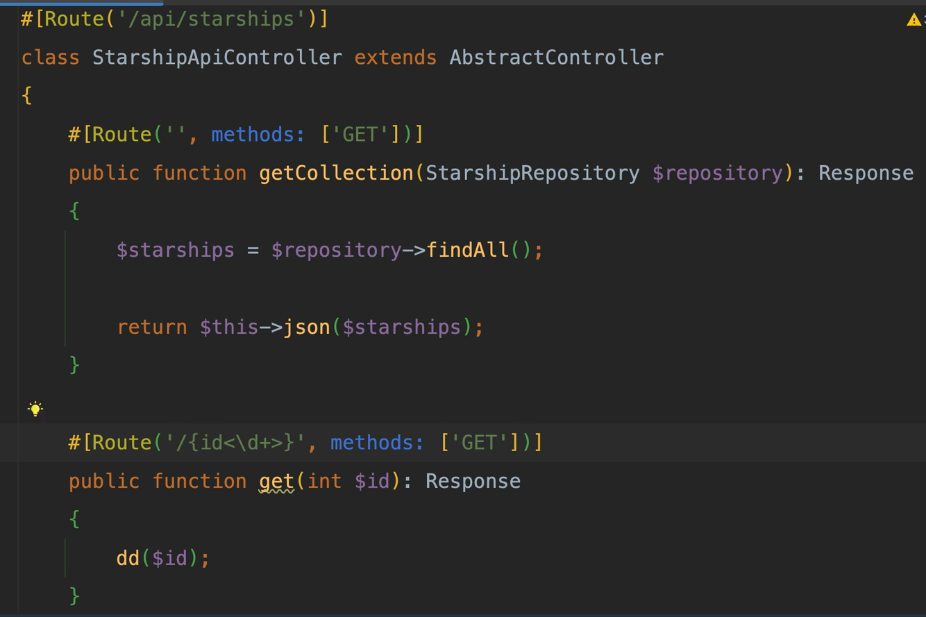
    public function get(int $id) : Response {…}

**Ajouter une méthode http**

    #[Route('/api/starships/{id<\d+>}', methods:['GET'])]

Visible avec php bin/console debug :router

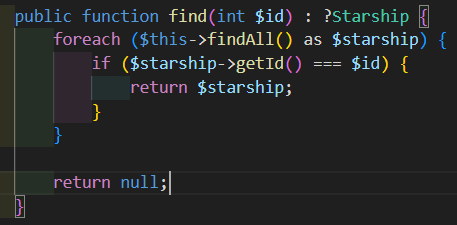


**Préfixer automatiquement toutes les routes de la classe**

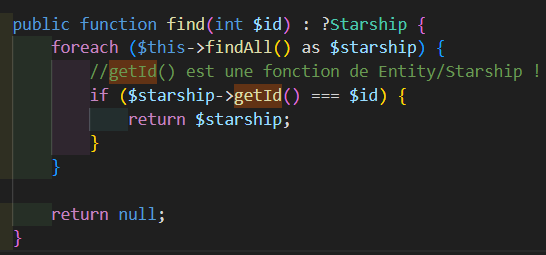
On ajoute le préfixe (ici « /api/starships/ ») en attribut de classe (et on le supprime dans les attributs de méthodes) :



**Ajouter au repository une fonction allant chercher un objet selon son id**

****

**Compléter la méthode du contrôleur en utilisant le repository**

****

*S’il existe un objet avec cet id, le récupérer ; sinon : rien*

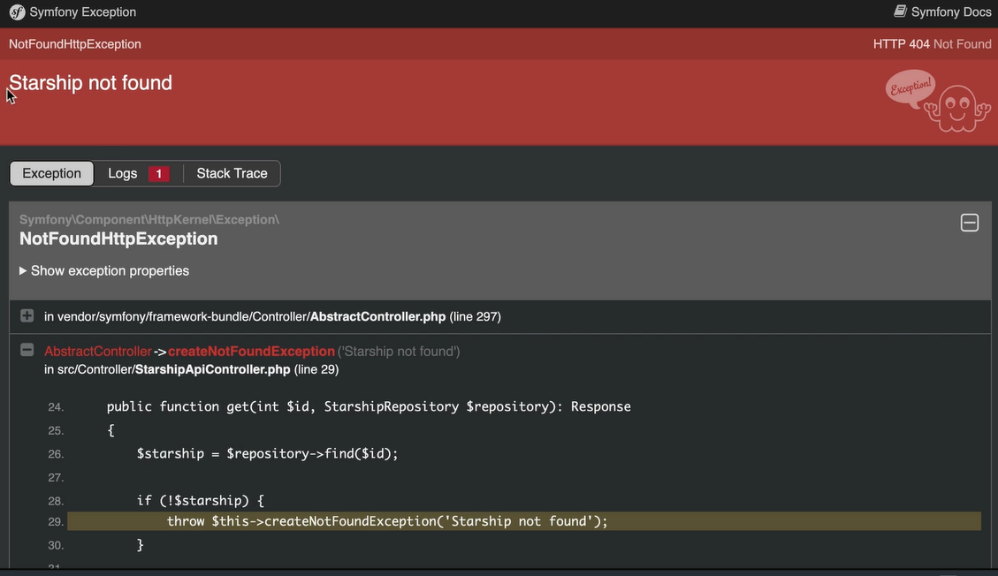
**Déclencher une page 404**

        if (!$starship) {

            throw $this->createNotFoundException('Starship not found');

        }

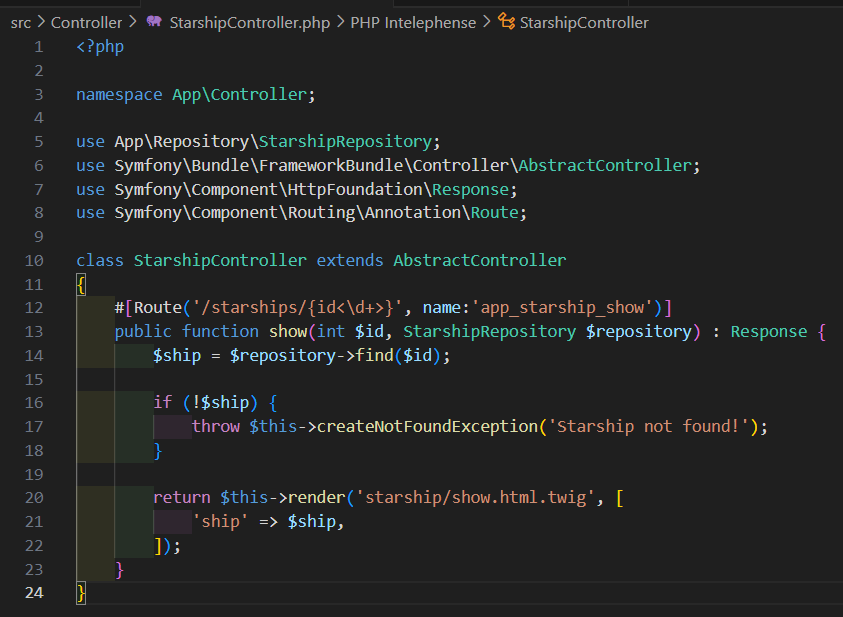
Notez le mot-clé ***throw*** : nous lançons une **exception spéciale** qui déclenche un 404. C'est bien car, dès qu'il atteint cette ligne, **rien ne sera exécuté après**.

Le message - "Starship not found" - n'est affiché qu'aux développeurs en **mode développement**. En production, une page totalement différente – ou JSON – serait renvoyée.

# 12 : Générer des URL

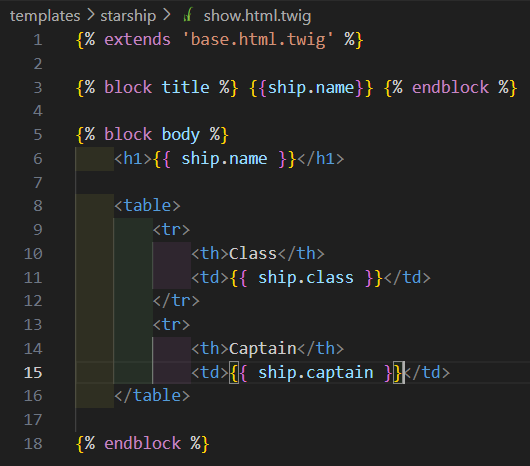
**Créer une page montrant les détails d’un objet (un vaisseau spatial !)**

Créer le StarshipController et la méthode show : code semblable à /api/starship/{id}, mais on renvoie une vue avec render() et non du json (et il n’y a pas « api » dans l’url !).

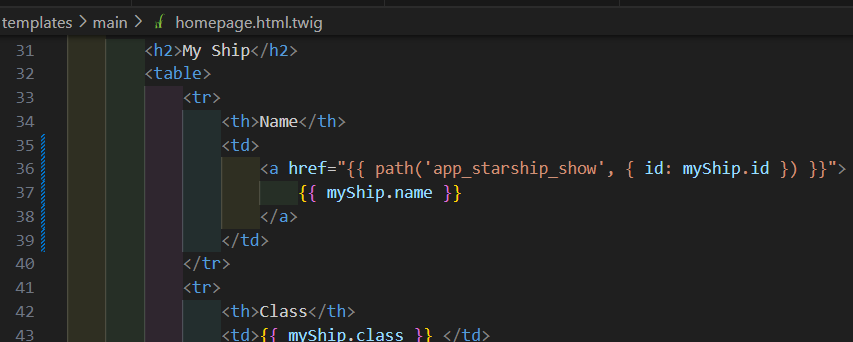


**Créer le template**

Cf. 05 et 06 ci-dessus



**Relier les pages avec le nom d’url**



# 13 : CSS et JavaScript avec AssetMapper

**Pour approfondir, voir ces 2 tutoriels :**

* celui [spécifiquement sur Asset Mapper](https://symfonycasts.com/asset-mapper)
* [LAST Stack](https://symfonycasts.com/screencast/last-stack) (« about building things with AssetMapper »)

**Installation :** composer require symfony/asset-mapper

Asset Mapper a 2 super pouvoirs :

**1/ il nous aide à charger CSS et JavaScript**

On voit le console.log qui vient de assets /app.js et le fond bleu de assets/styles/app.css

**2/ il rend public le répertoire assets/**

En configurant son chemin dans config/packages/asset\_mapper.yaml

framework:

    asset\_mapper:

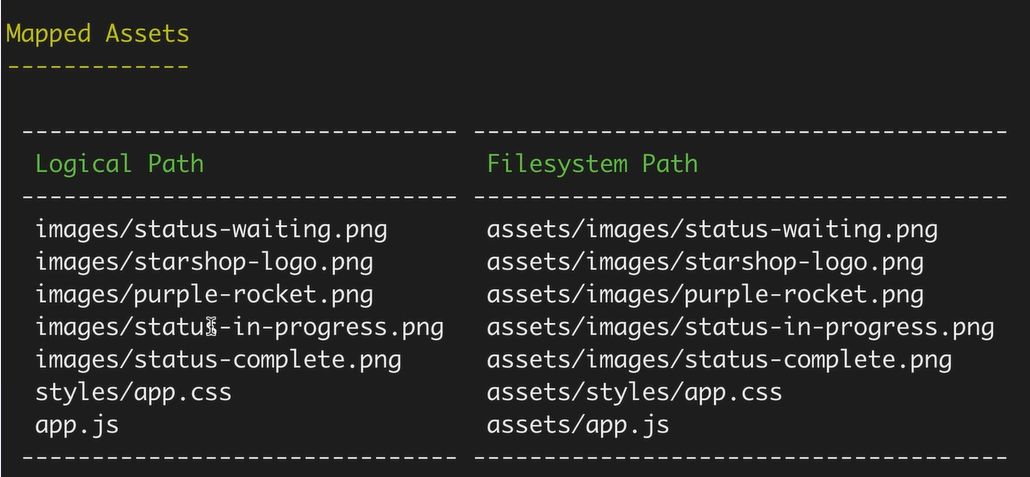
        # The paths to make available to the asset mapper.

        paths:

            - assets/

Voir ces « mapped assets » : php bin/console debug:asset

**Rendre une image stockée dans assets/ en utilisant le « logical path »**



Pour cela, il faut ajouter composer require symfony/asset

Et la fonction twig asset() dans le template souhaité :

      <img src="{{ asset('images/starshop-logo.png')}}" alt="Starshop logo"/>

**Versionning automatique des assets**

En inspectant le code source, on voit un hash ajouté par le bundle à l’url :



Cette chaîne est appelée **hachage de version** et elle est **générée en fonction du contenu du fichier**. Cela signifie que si nous mettons à jour notre logo ultérieurement, ce hachage changera automatiquement.

C'est super important. **Les navigateurs aiment mettre en cache les images, les fichiers JavaScript et CSS**, ce qui est génial : cela améliore les performances. Mais lorsque nous modifions ces fichiers, nous voulons que nos utilisateurs téléchargent la nouvelle version : ils ne continuent pas à utiliser la version obsolète mise en cache.

Mais **comme le nom du fichier changera lorsque nous mettrons à jour l’image, le navigateur utilisera automatiquement le nouveau** ! Cela ressemble à ceci :

* L'utilisateur accède à notre site et télécharge logo-abc123.png. Leur navigateur le met en cache.
* Lors de la visite suivante, leur navigateur voit la balise img pour logo-abc123.png, trouve le fichier dans son cache et l'utilise.
* Ensuite, nous arrivons, mettons à jour ce fichier et déployons.
* La prochaine fois que l'utilisateur visitera notre site, la balise img pointera vers un nom de fichier différent : logo-def456.png. Et comme le navigateur n'a pas ce fichier dans son cache, il le télécharge à nouveau.

# 14 : Tailwind CSS

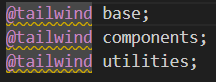
**Autres possibilités pour le front-end :**

* CSS pur : on peut en ajouter dans app/styles/app.css
* Bootstrap : voir la [documentation](https://symfony.com/doc/current/frontend/asset_mapper.html) et/ou le [symfonycasts](https://symfonycasts.com/screencast/asset-mapper/vendor-css)
* Sass : [symfonycasts/sass-bundle](https://github.com/symfonycasts/sass-bundle)

**3 commandes pour démarrer :**

Il ne s'agit pas simplement d'un gros fichier CSS que vous déposez sur votre page : il dispose d'un processus de construction qui analyse votre code pour toutes les classes Tailwind que vous utilisez. Il génère ensuite un fichier CSS final qui contient uniquement le code dont vous avez besoin.

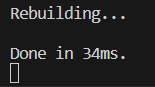
1. composer require symfonycasts/tailwind-bundle pour installer le bundle qui simplifiera l’utilisation
2. php bin/console tailwind:init

* un Tailwind binary en arrière-plan
* tailwind.config.js file à la racine = dit où Tailwind doit regarder pour trouver ses classes
* mise à jour de app.css en ajoutant trois lignes. Ce sera remplacé par le vrai code Tailwind en arrière-plan par le binary.

1. php bin/console tailwind:build –w (Tailwind has to be built)

* analyse les templates et génère le fichier CSS final en arrière-plan.
* Le -w le met en mode « surveillance » : au lieu de construire une fois et de quitter, il surveille nos modèles pour détecter les modifications. Lorsqu'il remarque des mises à jour, il reconstruit automatiquement le fichier CSS.

**Voir Tailwind en action :**

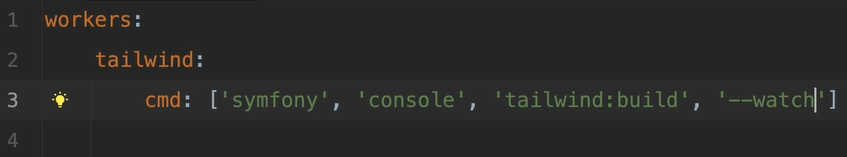
* + cf. la console si par exemple on ajoute une classe au h1
  + inspecter le code source de la page et cliquer le lien du css envoyé à l’utilisateur :
  + on voit le code construite par Tailwind ! Dans les coulisses, il existe une certaine magie qui remplace ces trois lignes Tailwind par le véritable code CSS Tailwind.

**Exécution automatique de Tailwind avec le binaire Symfony :**

Plus facile et plus automatique.

Ctrl+C dans la ligne de commande Tailwind pour l’arrêter.

A la racine du projet, créez **.symfony.local.yaml.** Il s'agit d'un fichier de configuration pour le serveur Web binaire Symfony que nous utilisons. À l'intérieur, ajoutez :



clé « workers » 🡪 au lieu de devoir exécuter la commande manuellement, lorsque nous démarrons le serveur Web symfony, il l'exécutera pour nous en arrière-plan.

On peut remplacer la commande bin/console par symfony console !

Dans votre premier onglet, appuyez sur Ctrl+C pour arrêter le serveur Web... puis réexécutez symfony serve : Il exécute la commande tailwind en arrière-plan !

Par exemple, si je rechange la classe h1 en « text-4xl », j’ai juste à rafraîchir ma page pour voir le changement.

# 15 : Twig partials & for-loop

Rendre dynamiques des parties du front-end codées en dur.

**Organiser le code avec des template partials**

Ce sont des blocs HTML qui pourront être réutilisés sur plusieurs pages, par exemple la barre latérale :

* créer un fichier **\_shipStatusAside.html.twig** (convention de nommage : partial commence par \_)
* y mettre le code du bloc <aside>
* l’inclure dans le fichier home page :

include("main/\_shipStatusAside.html.twig") }}

# 16 : PHP Enums

Ideal for sizes (small, medium, large..) or status (published, unpublished, draft…) for example.

Créer dans Model/ StarshipStatusEnum = un enum, et non pas une classe ! (cf. mot-clé n’est pas « funcion » mais « enmu »). Un enum est juste un ensemble de valeurs constantes centralisées en un seul endroit.

namespace App\Model;

// string-backed enum

enum StarshipStatusEnum: string

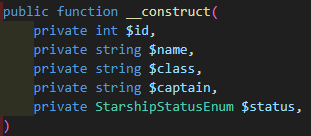
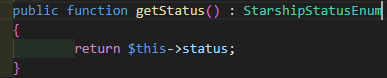
{

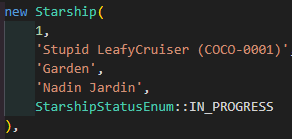
    case WAIING = 'waiting';

    case IN\_PROGRESS = 'in progress';

    case COMPLETED = 'completed';

}

Modifier la classe (l’entités, en passant l’enum) …



... le repository (création du navire) …

… et l’utilisation dans Twig :

<p class="uppercase text-xs text-nowrap">{{ ship.status.value }}</p>

Un enum, ou énumération, est un type de données qui consiste en un ensemble de valeurs constantes. Cela signifie qu'un enum représente **un ensemble fixe de valeurs** que vous pouvez utiliser dans votre code, ce qui peut aider à rendre le code plus lisible et à réduire les erreurs en limitant les valeurs possibles pour une variable à un ensemble spécifique.

**En PHP, les énumérations n'existent pas nativement** **comme elles le font dans d'autres langages** de programmation. Cependant, **PHP a introduit le support des énumérations soutenues à partir de la version 8.1**, permettant ainsi de définir des énumérations avec des valeurs associées. Ces énumérations **soutenues peuvent avoir des valeurs associées**, qui peuvent être **des littéraux ou des expressions littérales**, et elles ont une **propriété en lecture seule** value qui contient la valeur spécifiée dans la définition de l'énumération.

Avant l'introduction des énumérations soutenues, les développeurs PHP utilisaient souvent des classes ou des bibliothèques tierces pour simuler des énumérations. Par exemple, la bibliothèque myclabs/php-enum permet de créer des énumérations en PHP en étendant une classe de base et en définissant des constantes pour chaque valeur de l'énumération. Cette bibliothèque a l'avantage de permettre de mettre les constantes en privées, ce qui peut être un avantage en termes de sécurité et de conception du code.

Il est important de noter que les énumérations en PHP, qu'elles soient **soutenues ou simulées**, ne sont pas des types de données natifs comme ils le sont dans d'autres langages. Cela signifie que leur utilisation peut nécessiter une certaine connaissance des **spécificités de PHP** et peut ne pas être aussi intuitive que dans des langages qui supportent nativement les énumérations.

Les énumérations de base en PHP sont simples et ne contiennent pas de valeurs associées, tandis que les énumérations soutenues permettent de définir des énumérations avec des valeurs associées et offrent des fonctionnalités supplémentaires comme la sérialisation et la conversion de valeurs scalaires.

# 17 : Méthodes de modèle personnalisées & templates dynamiques

1/

Récupérer le statut directement en string dans le modèle/entité :

        public function getStatusString(){return $this->status->value;}

Et on appelle la méthode dans Twig, comme on appelle une variable !

<p class="uppercase text-xs text-nowrap">{{ ship.statusString }}</p>

On peut l’ajouter partout :

alt="Status: {{ ship.statusString }}

2/ pour les images, en utlisant la fonction match()

public function getStatusImageFilename(): string

{

       return match ($this->status) {

               StarshipStatusEnum::WAIING => '/images/status-waiting.png',

               StarshipStatusEnum::IN\_PROGRESS => '/images/status-in-progress.png',

               StarshipStatusEnum::COMPLETED => '/images/status-complete.png',

        };

}

Dans le template :

<img class="h-[83px] w-[84px]" src="{{ asset(ship.statusImageFilename) }}" alt="Status: {{ ship.statusString }}">

# 18 : Stimulus : écrire du JavaScript professionnel

On peut écrire dans **assets\app.js**, mais aussi utiliser la librairie JS Stimulus : <stimulus.hotwired.dev> :

Vous prenez une partie de votre HTML existant et la connectez à un petit fichier JavaScript, appelé **contrôleur**. Cela vous permet d'ajouter un comportement : lorsque vous cliquez sur tel bouton, telle méthode sur le contrôleur sera appelée.

Malgré sa simplicité, cela nous permettra de créer toutes les fonctionnalités JavaScript et d'interface utilisateur dont nous avons besoin, de manière fiable et prévisible.

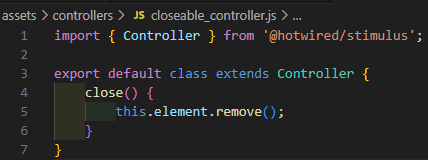
**Installation**

Symfony a un bundle qui aide à l’intégrer : composer require symfony/stimulus-bundle

* app.js : import de **bootsrap.js** qui vient d’être créé et démarre le moteur (engine) de Stimulus
* **assets\controllers\hello\_controller.js** : où il y aura nos contrôleurs personnalisés ; pour le connecter à un élément HTML, celui-ci doit avoir l’attribut « ***data-controller="hello***" »

**Exemple : ajouter sa fonction au bouton « close »**

Quand on clique le bouton, l’élément aside doit disparaître.

1-Créer un nouveau contrôleur (copier-coller hello\_controller.js) closeable\_controller.js :

* importer { Controller } de Stimulus et créer une classe qui extends Controller
* écrire la fonction close() en JS

2-Dans le template :

* attacher ce contrôleur à l’élément aside (ajouter l’attribut) :

<aside class="…" data-controller="closeable">

* attacher le bouton à la méthode close()

<button data-action="closeable#close">

**Encore mieux : Animer le bouton « close » avec une transition et une promise**

* ajouter « transition-all » à la classe de l’élément aside dans le template
* ajouter le code JS au contrôleur :

# 19 : Turbo : Your Single Page App

Pour éviter de recharger une page à chaque clic.

Turbo est une librairie JS faite par les mêmes que Stimulus, avec aussi un bundle Symfony pour faciliter :

composer require symfony/ux-turbo

* ajout à assets/controllers.json

**Comment ça marche ?**

Les rechargements en pleine page disparaissent ! C'est super rapide et tout se passe via Ajax.

Lorsque nous cliquons sur un lien, Turbo intercepte le clic et, au lieu d'un rechargement complet de la page, il effectue un **appel Ajax vers cette page**. Cet appel Ajax renvoie le HTML complet de cette page, puis Turbo le met sur cette page Cette petite chose transforme notre projet en une application d'une seule page et fait une grande différence dans la rapidité avec laquelle notre site apparaît.

**Appels AJAX et barre d'outils de débogage Web**

cf. Profiler pour voir les Request AJAX

**dans Turbo 8**, désormais disponible, votre site semblera encore plus rapide. C'est grâce à une nouvelle fonctionnalité appelée **Instant Click**. Avec cela, lorsque vous survolez un lien, Turbo effectue un appel Ajax vers cette page avant de cliquer. Ensuite, lorsque vous cliquez, il se charge instantanément... ou du moins a une longueur d'avance. Turbo a beaucoup d'autres fonctionnalités cf. notre didacticiel LAST Stack où nous construisons une interface avec des popovers, des modaux, des notifications toast, et plus encore.

**Turbo nécessite du bon JS**

Étant donné que les rechargements de pages complètes ont disparu, votre JavaScript doit être construit de manière à gérer cela. Beaucoup de JavaScript s'attend à des rechargements de pages complètes... et si du HTML est soudainement ajouté à la page sans rechargement, il se casse. La bonne nouvelle est que si vous écrivez votre JavaScript dans Stimulus, tout va bien.

# 20 : Maker Bundle

composer require symfony/maker-bundle –dev

* dans composer.json, c’est ajouté sous « require-dev » seulement (sans l’option –dev, c’est ajouté sour require et require-dev)

C’est un service avec de nouvelles commandes pour la console : cf. la liste avec

symfony console ou php bin/console

**Exemple : générer une commande de console (cf.** [**documentation**](https://symfony.com/doc/current/console/style.html)**)**

Symfony console make : command

Saisir un nom comme « app :ship-report »