Learning Symfony 7 : Upgrade & news

Symfonycasts

*Documentation faite par Annaig en mai 2024*

13 vidéos en accès libre : <https://symfonycasts.com/screencast/symfony7-upgrade>

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.

Il faut payer pour télécharger le code traité dans le cours (mais on peut sans doute le retrouver gratuitement dans le cours sur Symfony 6 ?), et voir les vidéos à partir de la 4e.

Autres liens intéressants :

* <https://symfony.com/doc/current/setup/upgrade_major.html> (EN)
* <https://symfony.com/7> (EN)
* <https://github.com/symfony/symfony/blob/7.0/UPGRADE-7.0.md> (EN)
* <https://github.com/symfony/symfony/blob/6.4/CHANGELOG-6.4.md> (EN)

Table des matières

[I-Migration vers Symfony 7 3](#_Toc168578154)

[01 : Migrer vers Symfony 6.4 3](#_Toc168578155)

[02 : Mise à jour des Flex Recipes 3](#_Toc168578156)

[03 : Màj des recettes liées à Encore & Stimulus 4](#_Toc168578157)

[04 : Goodbye SensioFrameworkExtraBundle 4](#_Toc168578158)

[05 : Préparer Symfony 7 : trouver et éliminer les dépréciations 5](#_Toc168578159)

[06 : Mise à niveau vers Symfony 7 6](#_Toc168578160)

[07 : Migration de Encore 🡪 AssetMapper 1/2 7](#_Toc168578161)

[08 : Migration de Encore 🡪 AssetMapper 2/2 8](#_Toc168578162)

[09 : Moderniser et simplifier JS 10](#_Toc168578163)

[II-Quelques nouveautés de Symfony 7 10](#_Toc168578164)

[10 : Nouveaux attributs d’autowiring 10](#_Toc168578165)

[11 : MapQueryParameter & Request Payload 11](#_Toc168578166)

[12 : Profiler les commandes de la console 12](#_Toc168578167)

[13 : Nouveau composant :Scheduler 13](#_Toc168578168)

# I-Migration vers Symfony 7

## 01 : Migrer vers Symfony 6.4

Versions 6.4 et 7.0 sont identiques et sorties le même jour (novembre 2023)

On a simplement retiré les dépréciations dans la version 7.

Il faut donc d’abord mettre à jour de 6.1 vers 6.4 (fait avec l’app Snowtricks) :

1. **Lancer l’application**

* Lancer un serveur de développement avec symfony. serve –d

1. **Mettre à jour la version de PHP**

* Vérifier ma version de php en local avec php –v (ici 8.2.10)
* Dans compser.json, ajouter le code suivant (ou modifier php 8.1 en 8.2)

"config": {

En résumé :

Dans ***composer.json***, remplacer les versions de PHP & Symfony

Puis **composer up**

…

"platform": {

"php": "~~8.1.0~~ 8.2"

}

},

* Lancer la commande composer up
* Il est possible de devoir ré installer les dépendences nodes : yarn install - -force ou npm install - -force puis yarn watch ou npm run watch

1. **Mettre à jour la version de Symfony**

* Dans composer.json, trouver 6.1.\* et les remplacer tous par 6.4.\* (Ctrl+maj+H dans VSCode)
* Ignorer pour l’instant les ^6.1 (qui ne viennent pas du répertoire principal, et suivent leur propre calendrier de mises à jours… mais qu’on mettra à jour quand même plus tard)
* Vérifier en particulier le changement dans extra > symfony > require (en bas du fichier)
* Lancer la commande composer up

## 02 : Mise à jour des Flex Recipes

Pour corriger les dépréciations par un raccourci.

Quand on installe des packages, beaucoup d'entre eux ont des recettes Flex qui ajoutent de nouveaux fichiers et modifient parfois des fichiers existants : tout le nécessaire pour que le package fonctionne immédiatement.

Pour voir les recettes : composer recipes

Pour les mettre à jour : composer recipes : update 🡪 on a la liste, il faut les faire une par une (et il faut que ce soit un repo git). A chaque fois, vérifier les changements et voir si on les garde ou non.

Ex. : modification de la version du server postgres en prod = inutile ; changement proposé = version 13 🡪 16, or nous sommes en version 15. (si on modifie la version utilisée en revanche, il faudra recréer / remigrer la base de données).

Dans ce chapitre, il y a les indications spécifiques (modifications mineures en général) pour :

* **doctrine/doctrine-bundle**
* **doctrine/doctrine-migrations-bundle**
* **symfony/framework-bundle (cf. chapitre 4 pour les dépendances)**
* **symfony/monolog-bundle**
* **symfony/routing**
* **symfony/translation**

Pour voir les changements : git diff –cached

Pas de changements majeurs non plus sur **security bundle, phpunit-bridge, mailer, messenger, notifier**.

## 03 : Màj des recettes liées à Encore & Stimulus

**symfony/twig-bundle**

Conflit dans un template : garder notre contenu personnalisé plutôt que le contenu par défaut/standard proposé.

S’il y avait des « encore\_entry\_link\_tags() » et « encore\_entry\_script\_tags() », ils ont été supprimés : ces lignes faisaient partie de la recette de TwigBundle, mais elles ont été déplacées vers la recette de WebpackEncoreBundle. Ainsi, lorsque nous mettons à jour la recette TwigBundle, il semble que ces lignes devraient être supprimées, alors que nous en avons toujours besoin, mais acceptons ce changement temporairement. Nous les verrons ajoutés plus tard lorsque nous mettrons à niveau la recette WebpackEncoreBundle.

**symfony/webpack-encore-bundle**

Le passage de la version 3 à la version 4 de Encore peut nécessiter des changements, y compris dans package.json ce qui suppose de relancer yarn ou npm.

**Passer à la version 2 de webpack-encore-bundle**

Dans composer.json, faire la modification suivante puis composer up :

        "symfony/webpack-encore-bundle": "^~~1.16~~ 2.0",

Suite à cela, peuvent apparaître 2 nouvelles updates :

**🡪 symfony/stimulus-bundle**

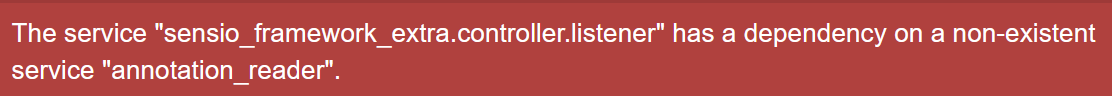
possible si certaines fonctionnalités utilisées ont été transféré au bundle Stimulus

🡪 **symfony/webpack-encore-bundle**

Cela peut créer des conflits car il y a eu beaucoup de changements avec l’introduction de Stimulus :

* assets/app.js : ne pas accepter les changements
* package.json : garder les lignes qu’il veut effacer !
* ne pas accepter la suppression des fichiers
* accepter seulement les changements dans symfony.lock (et dans les templates s’il rajoute ce qui avait été effacé : « encore\_entry\_link\_tags() » et « encore\_entry\_script\_tags() »)

## 04 : Goodbye SensioFrameworkExtraBundle

Suite à la mise à jour du symfony framework bundle, on a l’exception : « The service "sensio\_framework\_extra.controller.listener" has a dependency on a non-existent service "annotation\_reader".« 

En effet, dans config/packages/framework.yaml, il y a :

 annotations: false

En effet, le système d'annotation a été remplacé par les attributs PHP de base : or SensioFrameworkExtraBundle offrait l'annotation @Route, l'annotation de sécurité et le convertisseur de paramètres (param converter).

**Désinstaller**

Il suffit de le désinstaller avec composer remove sensio/framework-extra-bundle

**Remplacer les annotations par des attributs**

Vérifier ensuite si on l’utilisait avec git grep FrameworkExtra. Si c’est le cas, il s'agit simplement de déterminer quel nouvel attribut de Symfony remplace cette fonctionnalité : cf. [liste des attributs PHP de Symfony (EN)](https://symfony.com/doc/current/reference/attributes.html). Par exemple, SensioFrameworkExtraBundle avait une annotation de « Security ». Symfony dispose désormais d'un attribut « IsGranted » à la place.

**Le ParamConverter**

Il existe une fonctionnalité de SensioFrameworkExtraBundle qui ne nécessitait pas d'annotation... donc peut être utilisé sans s’en rendre compte : le Param Converter (par exemple pour trouver l’équivalent objet Doctrine d’un argument {slug} dans l’URL). Cela fonctionne toujours ! La fonctionnalité réside désormais dans le noyau (core). Et dans la plupart des cas, il continuera à faire son travail en silence, comme avant.

Si on ajoute une lettre supplémentaire à la fin du slug pour obtenir un 404, on voit que le système derrière cela est EntityValueResolver. Si besoin d'un contrôle supplémentaire, on peut le configurer avec l'attribut #[MapEntity].

## 05 : Préparer Symfony 7 : trouver et éliminer les dépréciations

**Qu’est-ce qu’une dépréciation ?**

Supposons que Symfony change le nom d'une méthode : simplement la renommer casserait le code des apps. Au lieu de cela, l’ancien nom de la méthode est conservé, le nouveau nom de la méthode est ajouté, ainsi qu’une petite fonction de code de dépréciation dans cette ancienne méthode. Ceci est publié sur une version mineure, comme la 6.3 ou la 6.4. Ensuite, lorsqu’on effectue une mise à niveau vers cette version et, puisque le code appelle l'ancienne méthode, il atteint cette dépréciation, ce qui déclenche un avertissement de dépréciation. Il faut alors mettre à jour le code pour appeler le nouveau nom de méthode. Une fois toutes les dépréciations disparues, on peut passer en toute sécurité à Symfony 7.0. (Rappel : la seule différence entre Symfony 6.4 et Symfony 7.0 est que les chemins de code obsolètes ont disparu. Dans notre exemple, cela signifie que l'ancien nom de méthode est finalement supprimé dans la version 7.0.)

**Comment procéder ?**

Commencer par vider le cache php bin/console cache :clear (& le refaire à chaque correction de dépréciation)

Dans le Profiler, étudier les déprécations.

* Bien regarder pour chaque page
* Pour chercher les infos sur ces dépréciations : composer why <nom de composant>

Si une nouvelle version majeure d’un bundle a été publiée, se renseigner précisément sur les nouveautés / dépendences avec la doc. Changer le numéro de version (par exemple ^2.0) dans composer.json & composer up. Exemple :

* message de dépréciation de symfony/templating
* composer why symfony/templating montre qu’il est utilisé par knp-time-bundle, qui est en v1 alors qu’il y a désormais une v2
* on modifier composer.json & on lance composer up : knp-time est màj, et symfony/templating a disparu.

**Types de retour PHP natifs (cf.** [**documentation en anglais**](https://symfony.com/doc/current/setup/upgrade_major.html#upgrading-to-symfony-6-add-native-return-types)**)**

Symfony 6 et Symfony 7 ont ajouté des types de retour PHP natifs à (presque toutes) les méthodes.

En PHP, si le parent a une déclaration de type de retour, toute classe implémentant ou remplaçant la méthode doit également avoir le type de retour. Cependant, vous pouvez ajouter un type de retour avant que le parent n'en ajoute un. Cela signifie qu'il est important d'ajouter les types de retour PHP natifs à vos classes avant de passer à Symfony 6.0 ou 7.0. Sinon, vous obtiendrez des erreurs de déclaration incompatibles.

**Exemple 1 : la méthode UserInterface::getRoles() aura un type de retour array dans Symfony 6**. Dans Symfony 5.4, vous recevrez un avis de dépréciation à ce sujet et vous devrez ajouter la déclaration du type de retour à votre méthode getRoles().

**Exemple 2 : la méthode UserInterface::eraseCredentials() aura un type de retour void dans Symfony 7.** Dans Symfony 6.4, il faut ajouter la déclaration du type de retour void à cette méthode.

Pour aider, Symfony fournit un script qui peut ajouter automatiquement ces types de retour avec le composant symfony/error-handler.

**Disparition de la méthode renderForm() et remplacement par render()**

<https://symfony.com/doc/current/forms.html#rendering-forms>

**Dépréciations à ignorer :**

* Il y a une fausse dépréciation de DoctrineFixturesBundle (se renseigner si elle apparaît).
* Parfois, l'avertissement peut provenir d'une bibliothèque ou d'un ensemble tiers qui n’est pas de Symfony : dans ce cas, il y a de fortes chances que ces dépréciations aient déjà été mises à jour. On peut mettre à niveau la bibliothèque pour les corriger, et cela peut être fait plus tard (elles n’empêcheront pas de passer en Symfony 7). Par exemple, si cela concerne doctrine/orm, ce sera à faire lors du passage à une version majeure de ce package, non de Symfony.

**Vérifier qu’on a vu toutes les dépréciaitons**

Il faut vérifier dans le log et aussi en production.

* Corriger toutes les dépréciations qu’on peut trouver,
* déployer en production,
* attendre un jour ou deux,
* puis vérifier les logs pour voir s'il y a des dépréciations.
* Une fois qu’il n’y en a plus, on peut passer en toute sécurité à Symfony 7.0

## 06 : Mise à niveau vers Symfony 7

1. **Mise à niveau de composer.json**

* Dans ce fichier, emplacer 6.4.\* par 7.0.\*
* composer up 🡪 ça bloque

1. **Trouver les packages qui bloquent**

* *babdev/pagerfanta-bundle* : Dans composer.json, recherche pagerfanta et emplacer tout cela par ^4.0 pour obtenir la nouvelle version majeure ; puis composer up
* *symfony/proxy-manager-bridge* : il faisait partie du pack doctrine-orm, mais Symfony a ajouté ses propres versions de proxy appelées "ghost objects".
  + On peut le supprimer : composer remove symfony/proxy-manager-bridge
  + Si cela pose problème : remettre dans composer.json à version 6.4 pour ce bundle
* Relancer composer up

1. **Mettre à jour les autres packages (packages tiers)**

* composer outdated
* vérifier toutes les dépendances avec composer why-not <package> <dernière version> , par exemple :
  + doctrine/orm est en 2.19 et apparaît outdated car la dernière version est 3.2 :
  + composer why-not doctrine/orm 3.2 montre que cette version nécessite doctrine/annotations 2.0
  + composer why-not doctrine/annotations 2.0 montre que dans composer.json, ce package est à la version 1.0. Il faut donc commencer par changer cela, puis composer update
  + vérifier que tout est en ordre, puis modifier la version de doctrine/orm de la même manière.
* On peut aussi tenter, même si c’est plus risqué, composer update doctrine/orm –with-all-dependencies

1. **Mettre à niveau les recettes**

Cf. plus haut

1. **Modifier le namespace de #[Route]**

Dans les contrôleurs, remplacer les ‘use’ utilisant le terme « Annotation » par ceux qui utilisent « Attribute ».

1. **Autres correctifs**

Par exemple, la constante LAST\_USERNAME

Voir les fichiers CHANGELOG.md, par exemple : <https://github.com/symfony/symfony/blob/6.4/src/Symfony/Bundle/SecurityBundle/CHANGELOG.md>

## 07 : Migration de Encore 🡪 AssetMapper 1/2

AssetMapper : composant apparu Symfony 6.3, permet de faire du JS et du CSS modernes sans build.

Il remplace Encore, qui reste utilisable. Faut-il faire le remplacement maintenant ?

* Pas obligatoire
* Bien si le processus de build de Encore nous semble lent
* Non si on utilise React ou Vue, qui nécessitent l’étape de build

1. **Supprimer le Webpack Encore**

* Sous Windows : tasklist | findstr node pour trouver l’ID du processus associé à npm run watch
* taskkill /F /PID <PID> pour arrêter le processus de build
* composer remove symfony/webpack-encore-bundle pour supprimer l’ancien bundle (cela va également désinstaller sa recette)
* cela a supprimé:
  + package.json,
  + webpack.config.js,
  + *config/packages/webpack\_encore.yaml*
  + encore\_entry\_ functions dans base.html.twig
  + app.js
  + app.css
* On a besoin de ces deux derniers fichiers 🡪 git checkout assets/
* Supprimer
  + node\_modules,
  + public/build/
  + fichier yarn error
  + yarn.lock ou package-lock.json

1. **Installer AssetMapper**

La commande composer require symfony/asset-mapper installe toute la recette (cf. <https://github.com/symfony/recipes/tree/main/symfony/asset-mapper> pour les détails)

* Dans .***gitignore*** : ignore l'emplacement final des assets construits (/public/assets/) et l'endroit où se trouvent les fichiers du vendor (/assets/vendor/).
* dans ***templates/base.html.twig*** : a ajouté une fonction importmap() qui affichera CSS et JavaScript – si ce n’est pas fait (droits admin), ajouter à la main dans <head> :

{% block stylesheets %}

{% endblock %}

{% block javascripts %}

{% block importmap %} {{ importmap(‘app’) }} {% endblock %}

{% endblock %}

* création de ***importmap.php*** = le nouveau package.json, fichier de configuration pour les packages tiers… on y voit Stimulus et Turbo – normalement, sinon ajouter à la main :

return [

'app' => [

'path' => './assets/app.js',

'entrypoint' => true,

],

'@hotwired/stimulus' => [

'version' => '3.2.2',

],

'@symfony/stimulus-bundle' => [

'path' => './vendor/symfony/stimulus-bundle/assets/dist/loader.js',

],

'@hotwired/turbo' => [

'version' => '7.3.0',

],

];

* Création de ***assets/app.js*** s’il n’existe pas (principal fichier JavaScript);
* Création de ***assets/styles/app.css*** s’il n’existe pas (principal fichier CSS);
* Création de ***config/packages/asset\_mapper.yaml*** (ou on définit les asset "paths");
* Création de ***manifest.json*** ? pas nécessaire, mais on peut l’ajouter à la main (cf. recette lien)

## 08 : Migration de Encore 🡪 AssetMapper 2/2

1. **Installer Bootstrap CSS**

Faire fonctionner les fichiers CSS tiers est l'une des choses les plus délicates à faire dans AssetMapper. On installe des packages tiers avec bin/console importmap:require <le nom du package> .

* donc ici php bin/console importmap:require bootstrap 🡪 ajout à ***importmap.php***. : le package JavaScript, une dépendance du package JavaScript, un fichier CSS contenu par ce package
* copier le chemin du fichier CSS et l’ajouter à ***app.css***, et supprimer la ligne du haut
* dans ***app.js***, ajouter : import 'bootstrap/dist/css/bootstrap.min.css';

1. **Ajouter FontAwesome**

Une grande différence entre Encore et AssetMapper est que si on veut importer un fichier spécifique à partir d'un package, il faut importermap:require <ce fichier>, pas seulement le package en général.

* php bin/console importmap:require @fortawesome/fontawesome-free/css/all.css 🡪 les fichiers sont dans assets/vendor
* Ajouter « import '@fortawesome/fontawesome-free/css/all.css'; » dans ***app.js***

N.B. : Il est désormais recommandé d’utiliser plutôt des kits FontAwesome. Ou, mieux, de restituer un SVG en ligne.

1. **Ajouter CSS Fonts**

Le dernier élément de ***app.css*** est une police. C'est plus délicat.

Si vous exécutez importmap:require packageName, vous obtiendrez le JavaScript pour ce package. Or ici nous avons besoin d’un fichier CSS. Avec AssetMapper : nous devons déterminer quel est le chemin d'accès au fichier CSS, puis l'exiger.

On peut faire cela sur <jsDelivr.com>. Il s'agit du CDN qu'AssetMapper utilise en arrière-plan pour récupérer les packages : on cherche le package et on copie le nom du chemin, ici :

php bin/console importmap:require @fontsource-variable/roboto-condensed/index.min.css

Comme toujours, supprimer l'importation de ***app.css*** et l’importer depuis ***app.js***.

1. **Ajouter l’extension .js**

Dans app.js : import "./bootstrap.js";

(Webpack Encore fonctionne dans un environnement Node, où, si le fichier importé se termine par .js, on n’a pas besoin d'inclure le .js. Mais dans un environnement JavaScript réel, comme le navigateur, l’extension est nécessaire.)

1. **stimulus-bridge 🡪 stimulus-bundle**

Il existe deux types d'importations :

* si elle commence par ./ ​​ou ../, il s'agit d'une **importation** **relative**. C'est simple : on importe un fichier à côté de ce fichier. Le deuxième type est appelé
* **importation** **nue** : importer un package ou un fichier dans un package. Pour ceux-ci, la chaîne à l’intérieur de l’importation doit exister exactement dans importmap.php. Si ce n'est pas le cas, on voit une erreur. C’est le cas ici. Pour la corriger :
  + dans bootstrap.js, transformer @symfony/stimulus-bridge en @symfony/stimulus-bundle :

import { startStimulusApp } from ‘@symfony/stimulus-bundle' ;

export const app = startStimulusApp() ;

1. **Installer les packages manquants**

Si la même erreur apparaît avec d’autres packages (axios…) : même solution.

1. **Downgrader (rétrograder ?) une dépendance**

Par exemple, si mon projet utilisait bootstrap 5.1 : AssetMapper a installé 5.3 et cela fait des changements que je ne veux pas :

* Dans importmap.php, changer le numéro de version (2 occurrences) à « 'version' => '5.1.3', »
* Puis php bin/console importmap:install (c’est l’équivalent de composer install !)

## 09 : Moderniser et simplifier JS

Utiliser fetch() à la place de axios pour faire des appels Ajax

* php bin/console importmap:remove axios
* dans le controller.js, supprimer l’import et remplacer axios.get() par fetch()

Pour traiter une Promise, remplacer .then() par async & await.

# II-Quelques nouveautés de Symfony 7

## 10 : Nouveaux attributs d’autowiring

**Présentation rapide des nouvelles fonctionnalités**

* Le meilleur endroit pour trouver ce genre de choses... est le [blog Symfony](https://symfony.com/blog/). Voir la section « Living on the edge » du blog et en filtrant par version
* le profileur de flux de travail / workflow profiler. En utilisant le composant workflow, on peut désormais voir une visualisation du flux de travail dans le profileur.
* quelques changements dans le système de déconnexion - juste pour simplifier la vie... quelques nouvelles contraintes, comme *PasswordStrengthConstraint* et une autre qui empêche les caractères suspects, comme les caractères d'espacement de largeur nulle. Cela peut être utilisé pour empêcher quelqu'un de créer un nom d'utilisateur qui ressemble à celui de quelqu'un d'autre.
* API : commande debug:serializer pour voir toutes les métadonnées d'une classe.
* les nouveaux composants Webhook et RemoteEvent

**L’attribut Autowire**

L’effet global est qu’on n’aura probablement plus jamais besoin d’accéder à services.yaml !

Depuis quelques versions de Symfony, il existe un attribut Autowire : si on a un argument qui ne peut pas utiliser l’autowiring, on peut ajouter cet attribut avant et définir ce qu’on veut. Cela peut être un service, une expression, une variable d'environnement, un paramètre, un chaton, peu importe. Nous voulons un paramètre : kernel.debug. Exemple dans un « VinylController » :

use Symfony\Component\DependencyInjection\Attribute\Autowire ;

class VinylController extends AbstractController {

public function \_\_construct(

#[Autowire('%kernel.debug%')]

private bool $isDebug

)

}

**Autres attributs**

Répertoire « Attribute » 🡪 toute une liste d'attributs intéressants liés à l'injection de dépendances.

Exclude est un moyen d'exclure une classe de l'enregistrement automatique en tant que service. Autoconfigure et AutoconfigureTag sont deux façons de configurer les options d’un service : à placer au-dessus de la classe - ou même au-dessus d'une interface - et les options s'appliqueront au\*x service\*s qui implémente\*nt cette interface.

Il existe également AutowireIterator et AutowireLocator. Si on dispose d'un ensemble de services qui implémentent un tag, on peut utiliser AutowireIterator pour que ces services soient transmis en tant qu'itérateur, ou AutowireLocator pour les transmettre en tant que localisateur, un array associatif de services.

**Autowire Iterator**

Par exemple, supposons que, dans VinylController, nous souhaitions obtenir un itérable de chaque commande de console dans notre application :

* utiliser private iterable $commands
* Nous voulons un itérable de tous les services qui implémentent une balise spécifique. On les récupère avec #[AutowireIterator], puis la balise que nous voulons : console.command.
* pour prouver que cela fonctionne, foreach sur $this->commands as $command

class VinylController extends AbstractController {

public function \_\_construct(…

#[AutowireIterator('console.command')]

private iterable $commands,

)

{ foreach ($this->command as $command) {dump($command) ; }

}

🡪 On voit toutes les commandes de l’app !

## 11 : MapQueryParameter & Request Payload

Quelques nouvelles façons subtiles mais puissantes de récupérer les données de request telles que les paramètres de query et la request payload.

**L’attribut MapQueryParameter**

Par exemple, on ajoute ?query=banana à l'URL. Pour récupérer cela dans notre contrôleur, nous tapions historiquement un argument avec Request, puis le récupérions à partir de là. Et même si cela fonctionne toujours, nous pouvons maintenant ajouter un argument ?string $query. Pour indiquer à Symfony que c'est quelque chose qu'il doit récupérer à partir d'un paramètre de requête, on ajoute un attribut devant : #[MapQueryParameter] :

use Symfony\Component\HttpKernel\Attribute\MapQueryParameter ;

class VinylController extends AbstractController {…

public function homepage(…

#[MapQueryParameter] string $query=’’,

) : Response

}

**Validation à partir du type-hint**

L'attribut a également quelques options. Par exemple, si le paramètre de requête est appelé différemment de l’argument, on peut le mettre ici.

Et au-delà de la simple récupération de la valeur de la demande, ce système effectue également une validation. Exemple : on ajoute un argument int $page = 1 (et un paramètre page=3 à l’url).

public function homepage(…

#[MapQueryParameter] string $query=’’,

#[MapQueryParameter] int $page =1,

) : Response

Si dans mon url j’écris page=banana, j’aurai une erreur car l’agument est défini comme int.

**La fonction filter\_var()**

Tout ce système est géré par la classe QueryParameterValueResolver, qui valide grâce à la fonction PHP filter\_var(). On peut passer à cette fonction une valeur, des filtres… et elle dit si la vlaeur satisfait ces filtres. On peut aussi lui passer des options de contrôle des filtres.

**Valider qu’un entier est dans un ensemble (range), avec une valeur par défaut de 10 :**

public function homepage(…

#[MapQueryParameter] int $page =1,

#[MapQueryParameter(options: [‘min\_range’ => 1, ‘max\_range’ => 10)] int $limit = 10,

): Response

Si dans l’url on pass ?limit=3, ça marche ; limit=13, on a une erreur 404

**Récupérer des paramètres d’un array**

public function homepage(…

#[MapQueryParameter] array $filters = [],

): Response

Dans l’url, je peux ainsi récupérer ?filters[]=banana&filters[]=apple

Ca marche aussi avec des tableaux associatifs : ?filters[foo : bar]

**Request Body**

Depuis Sf 6.3, pour récupérer le body de la request, il y a une nouvelle méthode : $request->getPayload().

Ainsi, dans une API, si le client envoie du JSON dans le body, on peut utiliser cette méthode pour le décoder et le mettre dans un tableau associatif.

Si l’utilisateur\*ice soument un formulaire HTML, elle décode $\_POST et le met en tableau.

**MapRequestPayload**

Avec JSON, il est courant d'utiliser le sérialiseur pour désérialiser la payload en objet. Cela concerne une autre nouvelle fonctionnalité : #[MapRequestPayload]. Dans ce cas, \_\_invoke est l'action du contrôleur. ça dit : prenez le JSON de la requête et désérialisez-le dans un ProductReviewDto, qui est l'exemple de classe ci-dessus. Après avoir envoyé le JSON via le sérialiseur, il effectue même une validation.

## 12 : Profiler les commandes de la console

Nouvelle fonctionnalité de Symfony 6.4

**Déclencher un profil : --profile**

* php bin/console debug:container –profile -v
* cliquer le lien, copier le token, accédez au profileur, coller le token dans l'URL

**Exploration du Profiler**

informations sur la commande, l'entrée, la sortie...

Les événements montrent les événements réels qui ont été envoyés et les listeners pour chacun d'eux. Ceux-ci sont totalement différents des événements déclenchés lors d’une requête.

la plupart des sections du profileur sont grisées, mais si on rendait un modèle Twig... ou faisait une requête HTTP ou de base de données, ceux-ci seraient activés.

Même avec cette simple commande, nous déverrouillons la section performances.

## 13 : Nouveau composant :Scheduler

Gère la planification des tâches dans l’application. On peut l’utiliser pour la maintenance (nettoyage de base de données, effacement du cache, etc.), le traitement en arrière-plan (gestion des files d'attente, synchronisation des données, etc.), les mises à jour périodiques des données, les notifications programmées (emails, alertes), etc.

**Installation**

composer require symfony/scheduler symfony/messenger

Scheduler s'appuie sur Messenger : on crée une classe de message et un gestionnaire, comme normalement avec Messenger. Ensuite, on dit à Symfony : « Je veux que vous envoyiez ce message pour qu'il soit traité tous les sept jours, ou toutes les heures... ».

**Création de classe message & handler**

* composer require symfony/scheduler symfony/messenger
* php bin/console make:message (on l’appelle LogHello ici) 🡪 crée
  + src/Message/LogHello.php
  + src/MessageHandler/LogHelloHandler.php, dont la méthdoe \_\_invoke() sera appelée lorsque LogHello sera dispatché par Messenger
* ajouter les constructeurs dans ces deux fichiers
* compléter invoke() du MessageHandler

**Création du Schedule**

* créer une classe qui implémente ScheduleProviderInterface (elle peut gérer les différentes tâches, ou on peut créer une classe par tâche), avec l’attribut #[AsSchedule], qui a un attribut optionnel (name, qui par défaut est… « default »).

**Création des messages récurrents**

* implémenter la méthode getSchedule()
* elle doit renvoyer un nouveau Schedule(), auquel on ajoute des éléments en appelant ->add(). À l'intérieur, pour chaque "chose" à planifier, ajouter RecurringMessage::
* Il existe plusieurs manières de créer ces messages récurrents. Le plus simple est each(), comme tous les 7 jours ou toutes les 5 minutes. On peut également transmettre une syntaxe cron ou appeler trigger(). Dans ce cas, il faut définir sa propre logique pour savoir exactement quand on souhaite que le message étrage soit déclenché.

use App\Message\LogHello;

use Symfony\Component\Scheduler\RecurringMessage;

class MainSchedule implements ScheduleProviderInterface

{

public function getSchedule(): Schedule

{

return (new Schedule())->add(

RecurringMessage::every('4 seconds', new LogHello(4)),

RecurringMessage::every('3 seconds', new LogHello(3)),

);

}

}

**Consumer le Schedule Transport**

Le résultat de la création d’un fournisseur de planification (schedule provider) est la création d’un nouveau transport Messenger. Pour que les messages récurrents soient traités, il faut disposer d'un Worker qui exécute la commande messager:consume.

php bin/console messenger:consume -v scheduler\_default

🡪 les messages apparaissent dans la console comme planifiés !

**Comment marche Scheduler ?**

Lorsque la commande de travail démarre, elle parcourt chaque RecurringMessage, calcule la prochaine exécution de chacun et l'utilise pour créer une liste - appelée "heap" (tas) - des messages à venir. Ensuite, ça tourne en boucle pour toujours. Dès que l'heure actuelle correspond - ou est postérieure - à l'heure d'exécution prévue du prochain message dans le tas, il prend ce message et l'envoie via Messenger. Il demande ensuite ce message récurrent pour sa prochaine exécution et le place dans le tas.

**Rendre son Schedule stateful**

Problème : si nous redémarrons la commande, elle crée le planning à partir de zéro. Cela signifie qu’elle attend encore trois et quatre secondes avant d'envoyer les messages.

Dans une vraie application, ce sera un problème. Imaginez que vous ayez un message diffusé tous les sept jours. Pour une raison quelconque, après 5 jours, votre commande Messenger:consume se ferme et redémarre. De ce fait, votre message récurrent sera désormais diffusé sept jours après ce redémarrage : il sera donc diffusé au jour 12. S'il continue à être redémarré, votre message risque de ne jamais être diffusé!

Ainsi, dans le monde réel, nous définissons toujours notre planning avec état en créant une méthode \_\_construct dans le MainSchudele et en ajoutant ->stateful($this->cache); dans le retour de getSchedule().

**Workers multiples : verrouiller son Schedule**

Il faut autowire la lock factory, puis appeler ->lock() pour transmettre un nouveau verrou (lock). Cela garantira que deux workers ne récupèrent pas le même message récurrent en même temps et ne le traitent pas tous les deux.