

SYMFONY

framework PHP

Services

Anthony PARIS - Formateur

L'UTILISATION DES SERVICES

Vous l'avez vu jusqu'ici, une application PHP, qu'elle soit faite avec Symfony ou non, utilise beaucoup d'objets PHP. Un objet remplit une fonction comme envoyer un e-mail, enregistrer des informations dans une base de données, récupérer le contenu d'un template, etc. Vous pouvez créer vos propres objets qui auront les fonctions que vous leur donnez. Bref, une application est en réalité un moyen de faire travailler tous ces objets ensemble, et de profiter du meilleur de chacun d'entre eux.

Dans bien des cas, un objet a besoin d'un ou plusieurs autres objets pour réaliser sa fonction. Se pose alors la question de savoir comment organiser linstanciation de tous ces objets. Si chaque objet a besoin d'autres objets, par lequel commencer ?

L'objectif de ce chapitre est de vous présenter le conteneur de services. Chaque objet est défini en tant que service, et le conteneur de services permet d'instancier, d'organiser et de récupérer les nombreux services de votre application. Étant donné que tous les objets fondamentaux de Symfony utilisent le conteneur de services, nous allons apprendre à nous en servir. C'est une des fonctionnalités incontournables de Symfony, et c'est ce qui fait sa très grande flexibilité.

QU'EST CE Q'UN SERVICE?

Un service est simplement un objet PHP qui remplit une fonction, associé à une configuration.

Cette fonction peut être simple : envoyer des e-mails, vérifier qu'un texte n'est pas un spam, etc. Mais elle peut aussi être bien plus complexe : gérer une base de données (le service Doctrine !), etc.

Un service est donc un objet PHP qui a pour vocation d'être accessible depuis n'importe où dans votre code. Pour chaque fonctionnalité dont vous aurez besoin dans toute votre application, vous pourrez créer un ou plusieurs services (et donc une ou plusieurs classes et leur configuration). Il faut vraiment bien comprendre cela : un service est avant tout une simple classe.

L'AVANTAGE

L'avantage de réfléchir sur les services est que cela force à bien séparer chaque fonctionnalité de l'application. Comme chaque service ne remplit qu'une seule et unique fonction, ils sont facilement réutilisables. Et vous pouvez surtout facilement les développer, les tester et les configurer puisqu'ils sont assez indépendants.

Cette façon de programmer est connue sous le nom d'architecture orientée services, et n'est pas spécifique à Symfony ni au PHP

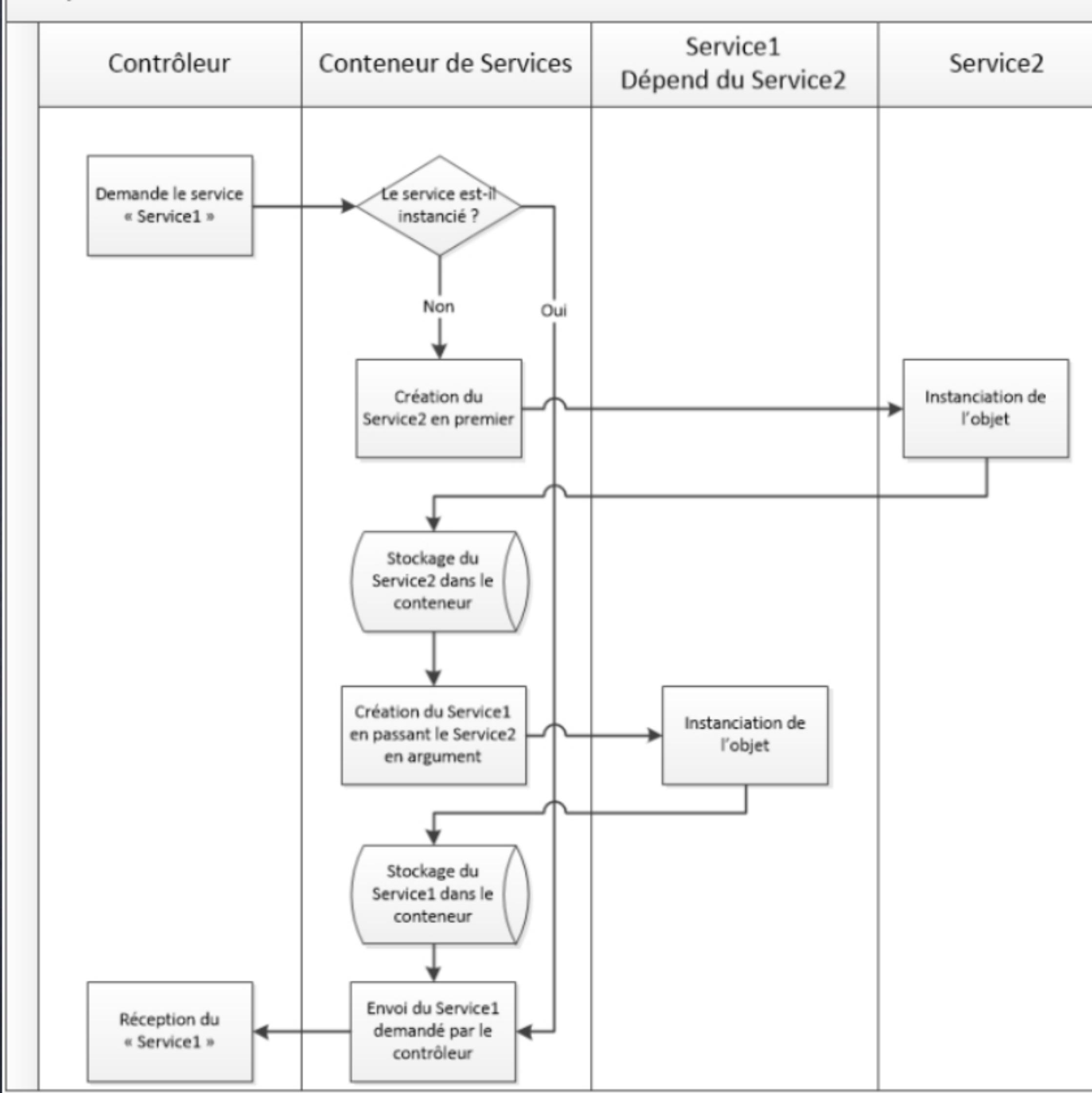
CONTENEUR DE SERVICES

L'intérêt réel des services réside dans leur association avec le conteneur de services. Ce conteneur de services (services container en anglais) est une sorte de super-objet qui gère tous les services. Ainsi, pour accéder à un service, il faut passer par le conteneur.

L'intérêt principal du conteneur est d'organiser et d'instancier (créer) vos services très facilement. L'objectif est de simplifier au maximum la récupération des services depuis votre code à vous (depuis le contrôleur ou autre). Vous demandez au conteneur un certain service en l'appelant par son nom, et le conteneur s'occupe de tout pour vous retourner le service demandé

CONTENEUR DE SERVICES

Comportement du conteneur de service



```

<?php

class Container
{
    protected $service1 = null;
    protected $service2 = null;

    public function getService1()
    {
        if (null !== $this->service1) {
            return $this->service1;
        }

        $service2 = $this->getService2();
        $this->service1 = new Service1($service2);

        return $this->service1;
    }

    public function getService2()
    {
        if (null !== $this->service2) {
            return $this->service2;
        }

        $this->service2 = new Service2();

        return $this->service2;
    }
}
  
```

CONTENEUR DE SERVICES

Au moment où vous démarrez une application Symfony, votre conteneur contient déjà de nombreux services. Ce sont comme des outils : attendre que vous en profitiez. Dans votre contrôleur, vous pouvez «demander» un service à partir du conteneur en tapant un argument avec le nom de classe ou d'interface du service. Vous voulez enregistrer quelque chose? Aucun problème:

```
// src/Controller/ProductController.php
namespace App\Controller;

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

class ProductController extends AbstractController
{
    #[Route('/products')]
    public function list(LoggerInterface $logger): Response
    {
        $logger->info('Look, I just used a service!');

        // ...
    }
}
```

Quels autres services sont disponibles?
Découvrez-le en exécutant:

```
> php bin/console debug:autowiring

# this is just a *small* sample of the output...

Describes a logger instance.
Psr\Log\LoggerInterface (monolog.logger)

Request stack that controls the lifecycle of requests.
Symfony\Component\HttpFoundation\RequestStack (request_stack)

Interface for the session.
Symfony\Component\HttpFoundation\Session\SessionInterface (session)

RouterInterface is the interface that all Router classes must implement.
Symfony\Component\Routing\RouterInterface (router.default)

[...]
```

CRÉATION DE VOTRE SERVICE

Vous pouvez également organiser votre propre code en services. Par exemple, supposons que vous deviez montrer à vos utilisateurs un message sympa et aléatoire. Si vous mettez ce code dans votre contrôleur, il ne peut pas être réutilisé. Au lieu de cela, vous décidez de créer une nouvelle classe:

Dans src/Service(à créer)/Proverbe.php

```
<?php
namespace App\Service;

You, 15 minutes ago | 1 author (You)
class Proverbe
{
    public function getProverbe()
    {
        $messages = [
            'Quand tout va bien on peut compter sur les autres, quand tout
            "Le bonheur ne s'acquiert pas, il ne réside pas dans les appar
            'Les larmes qui coulent sont amères mais plus amères encore so
        ];
        $index = array_rand($messages);
        return $messages[$index];
    }
}
You, 16 minutes ago • ok
```

CRÉATION DE VOTRE SERVICE

Une fois créé votre service doit être utilisé, ouvrez BlogController pour y ajouter :

```
public function proverbe(Proverbe $proverbe) {
    return $this->render("parts/proverbe.html.twig", ['proverbe' => $proverbe->getProverbe()]);
}
```

You, 4 days ago • ok

On peut utiliser Proverbe directement car l'autowiring permet l'utilisation des services sans configuration ni instantiation, on crée sa vue parts/proverbe.html.twig

```
<footer class="page-footer font-small blue">
    <div class="footer-copyright text-center py-3 white ">© 2020 Copyright:
        {{proverbe}}
    </div>
</footer>
```

CRÉATION DE VOTRE SERVICE

Notre but étant que cette vue soit présente partout et utilise un service il est nécessaire d'utiliser l'inclusion de controller dans le template base.html.twig

```
<!-->
<body>
{{ render(controller("App\\Controller\\BlogController::menu")) }}

{% block content %}

{% endblock %}

{% block footer %}
{{ render(controller("App\\Controller\\BlogController::proverbe")) }}
{% endblock %}      You, il y a 1 seconde • Uncommitted changes

{% block scripts %}
<script src="https://cdn.jsdelivr.net/npm/@popperjs/core@2.11.5/dist/umd/popper.min.js" integrity="sha384-Xe+8c
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0-beta1/dist/js/bootstrap.min.js" integrity="sha384-kjU
{{ encore_entry_script_tags('app') }}
{% endblock %}
</body>
```

CRÉATION DE VOTRE SERVICE

Nous avons besoin d'ajouter un peu de CSS, il suffit d'ouvrir assets/styles/app.css et ajouter :

```
.blue {  
    background-color: #2196f3 !important;  
}  
.white {  
    color: white;  
}
```

Vous ne voyez pas votre css, c'est normal il faut créer le build css avec la commande de encore Webpack :

```
# compile assets once  
> yarn encore dev  
  
# or, recompile assets automatically when files change  
> yarn encore dev --watch  
  
# on deploy, create a production build  
> yarn encore production
```

CRÉATION DE VOTRE SERVICE

C'est parfait, vous pouvez actualiser la page et voir un proverbe différent à chaque fois.

Bienvenu sur mon blog

Titre 14/01/2020 ad configuration
require_once APP_ROOT . '/config.php';
if (!defined('PSI_DEBUG')) {
 require_once APP_ROOT . '/html/error_config.html';
} else {
 require_once APP_ROOT . '/includes/autoload.inc.php';
 if (!extension_loaded('pcre')) {
 die("pcre extension to php in order to work properly.");
 }
 if (!extension_loaded('pcntl')) {
 die("pcntl extension to php in order to work properly.");
 }
 if (!extension_loaded('psl')) {
 die("psl extension to php in order to work properly.");
 }
 if (!extension_loaded('pcre')) {
 die("pcre extension to php in order to work properly.");
 }
 if (!extension_loaded('pcntl')) {
 die("pcntl extension to php in order to work properly.");
 }
 if (!extension_loaded('psl')) {
 die("psl extension to php in order to work properly.");
 }
}

Titre 14/01/2020

Another Title 14/01/2019

Another Title 14/01/2019

Another Title 14/01/2019

Another Title 14/01/2019

© 2020 Copyright: Quand tout va bien on peut compter sur les autres, quand tout va mal on ne peut compter que sur sa famille.

INJECTION DE SERVICES

Et si vous avez besoin d'accéder au loggerservice de l'intérieur Proverbe? Aucun problème!

Créez une __construct() méthode avec un \$logger argument.

```
class Proverbe| You, an hour ago • ok
{
    public function __construct(LoggerInterface $logger)
    {
        $this->logger = $logger;
    }

    public function getProverbe()
    {
        $messages = [
            'Quand tout va bien on peut compter sur les autres',
            "Le bonheur ne s'acquiert pas, il ne réside pas dans l'argent",
            'Les larmes qui coulent sont amères mais plus tard elles sont douces'
        ];

        $index = array_rand($messages);

        $this->logger->info('Le proverbe a été vu!');

        return $messages[$index];
    }
}
```

Le conteneur saura automatiquement passer le loggerservice lors de l'instanciation du MessageGenerator.

Comment sait-il cela?
Grâce au cablage automatique (autowiring)

Par ailleurs, cette méthode d'ajout de dépendances à votre construct() méthode est appelée injection de dépendance .

INJECTION DE SERVICES

Logger permet de jouer avec les logs, vérifiez dans log/dev.log, vous verrez votre message apparaître. Voyons rapidement comment créer des fichiers de logs séparés pour avoir une meilleure visibilité. «config/packages/monolog.yaml»

Nous allons créer deux nouveaux channels «services» et «controller» et ajouter la configuration dans monolog pour lui dire quand je ciblerai le channel «services» tu m'ajouteras mes messages dans le fichier «services.log»
Ajouter cette configuration également dans la partie «prod»

```
monolog:  
    channels: ['services', 'controller']  
    handlers:  
        main:  
            type: stream  
            path: "%kernel.logs_dir%/%kernel.environment%.log"  
            level: debug  
            channels: ["!event"]  
        services:  
            # Log all messages (since debug is the Lowest Level)  
            level: debug  
            type: stream  
            path: '%kernel.logs_dir%/services.log'  
            channels: [services]
```

INJECTION DE SERVICES

Pour utiliser rapidement cela : (service/proverbe.php)

```
use Psr\Log\LoggerInterface;
| You, a few seconds ago • Uncommitted changes
You, a few seconds ago | 1 author (You)
class Proverbe
{
    public function __construct(LoggerInterface $servicesLogger)
    {
        $this->logger = $servicesLogger;
    }

    public function getProverbe()
    {
        $messages = [
            'Quand tout va bien on peut compter sur les autres, quand tout va mal on',
            "Le bonheur ne s'acquiert pas, il ne réside pas dans les apparences, cha",
            'Les larmes qui coulent sont amères mais plus amères encore sont celles
        ];

        $index = array_rand($messages);

        $this->logger->info('Le proverbe a été vu!');

        return $messages[$index];
    }
}
```

```
var > log > Ξ services.log
1
2 [2020-05-14 16:57:59] services.INFO: Le proverbe a été vu! []
3
```

A partir de MonologBundle 3.5 , vous pouvez lier automatiquement les différents canaux Monolog par type.

Laissant étendre vos arguments de service avec la syntaxe suivante:
Psr\Log\LoggerInterface \$<channel>Logger.

https://symfony.com/doc/current/logging/channels_handlers.html

EN PLUS

Pour plus d'informations :

https://symfony.com/doc/current/service_container.html