

Symfony Les Services

AYMEN SELLAOUTI

Définition d'un Service

- Lorsque vous travaillez avec Symfony, vous aurez besoin d'une même fonctionnalité à plusieurs endroits différents. Vous aurez par exemple besoin d'accéder à votre base de données dans plusieurs contrôleurs.
- Un Service est **une classe simple** qui fournit un « Service ». Vous avez déjà utilisé plusieurs services tel que Doctrine ou TWIG. Elle doit être accessible partout dans votre code.
- Dans Symfony tous les services sont géré par le **conteneur de service « Symfony container »**.

Service = Classe PHP accessible partout + une configuration

http://symfony.com/doc/3.4/service_container.html#service-psr4-loader

Responsabilité unique (Single Responsibility)

L'un des principes de base sur lequel se base la notion de Service est le principe de Responsabilité unique. Ce principe fait parti des principes **SOLID** de l'orienté objet. Faites une recherche sur ces concepts. Pour le premier principe de « **Single Responsibility** » il implique que votre code au sein d'une classe **ne doit avoir qu'une seule responsabilité**.

Prenons l'exemple du service mailer qui se charge de l'envoi de mail, donc un seul type de tâche à effectuer.

Dans le cas où votre service se charge de 2 tâches différentes, pensez à diviser votre classe en deux services différents.

Les services de symfony suivent l'**architecture Orientées Service**

L'utilisation des services et du principe de **Single Responsibility** permet d'avoir un **code réutilisable** et facilement **maintenable**.

Le conteneur de services

Un service est une classe associé à une configuration. Si on utilise plusieurs services et qu'on doit tout gérer, l'utilisation des services deviendrait un peu pénible. C'est pour cela que Symfony nous fournit un conteneur de service.

Un **conteneur de service** est un **objet** qui a pour rôle de **gérer l'ensemble des services** de votre application. C'est lui qui vous permettra d'accéder vos services. **Si vous avez besoin d'utiliser un service vous devez passer par le conteneur.**

Cependant le **rôle du conteneur** n'est pas simplement de fournir les services mais aussi de les **préparer**. En effet, c'est lui qui va **préparer l'objet en l'instanciant et en instanciant toutes les classes dont dépend votre service.**

Tous les services (objets) ne sont pas instanciés à chaque requête. En effet, le conteneur est **paresseux il utilise le lazy loading afin d'assurer une plus grande vitesse d'exécution** : il n'instancie pas un service avant que vous le demandiez. Par exemple, si vous n'utilisez jamais le service de mailing lors d'une requête, le conteneur ne l'instanciera jamais.

Workflow d'un conteneur de service

Lors de la demande d'un service S1, le conteneur suit le workflow suivant :

Vérifie si la classe S1 est déjà instancié.

1. Si oui la fournir
2. Sinon
 1. Vérifier si le service dépend d'autres classes.
 1. Si oui les instancier ou les récupérer si elles sont instanciées
 2. Instancier le service

Accéder à un service

Plusieurs services sont déjà accessibles dès l'installation de Symfony. Comme nous l'avons déjà mentionné, vous avez déjà utilisé des services comme TWIG et Doctrine.

Deux façon permettent d'accéder à un service :

1- En passant par le **conteneur de service** et sa méthode **get**. Cette méthode prend en paramètre l'identifiant unique du service.

Exemple : pour récupérer doctrine on utilise la syntaxe suivante :

```
$doctrine = $this->container->get('doctrine');
```

Lorsque nous avons utilisé doctrine, nous avons utilisé un helper et la méthode `getDoctrine`, cette méthode n'est rien qu'un raccourcie qui appelle le service de doctrine.

2- En utilisant la nouvelle façon utilisable à partir de la version 3.3 et qui est le type hinting.

Exemple :

```
use Psr\Log\LoggerInterface;

public function
listAction(LoggerInterface $logger)
{
    $logger->info('J utilise le service
de logging');
}
```

Lister les services disponibles

Afin de lister les services disponibles dans votre projet, utiliser la commande suivante

`php bin/console debug:container`

Les services les plus connues sont :

Service ID	Class name
doctrine	Doctrine\Bundle\DoctrineBundle\Registry
logger	Symfony\Bridge\Monolog\Logger
router	Symfony\Bundle\FrameworkBundle\Routing\Router
session	Symfony\Component\HttpFoundation\Session\Session
twig	Twig_Environment
validator	Symfony\Component\Validator\Validator\ValidatorInterface

http://symfony.com/doc/3.4/service_container.html

Création d'un premier service

Créer un dossier service sous votre bundle.

Je suppose que vous avez créé ce dossier sous AppBundle. Créer une classe PremierService

Introduisez y une méthode getRandomString(\$nb) qui prend en paramètre le nombre de caractère et retourne une chaîne de taille ce nombre. Ensuite appelez ce service dans une de vos actions et affichez la valeur de retour. .

```
public function
getRandomString($nb) {
    $char =
    'abcdefghijklmnopqrstuvwxyz01234
    56789';
    $chaîne = str_shuffle($char);
    return substr($nom, 0, $nb);
}
```

```
use AppBundle\Service\PremierService;

public function
testAction(PremierService $ps) {
    die($ps->getRandomString($nb));
}
```


Configuration des services

Le premier service que vous venez de créer a fonctionné automatiquement sans aucune intervention de votre part et sans aucune configuration. Ceci est le cas à partir de la version 3.3 de Symfony qui a permis d'avoir le principe [d'autowiring](#).

En accédant au fichier [service.yaml](#) vous allez trouvé cette configuration générique avec un explication de chaque ligne de code. Le fait d'avoir un [autowire à true](#) a permis à Symfony d'utiliser le [Type-Hint](#) que vous avez déjà vu et qui a chargé automatiquement votre service.

services:

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

_defaults:

```
# automatically injects dependencies in your services
```

```
autowire: true
```

```
# automatically registers your services as commands, event subscribers, etc.
```

```
autoconfigure: true
```

```
# this means you cannot fetch services directly from the container via $container-
```

```
>get()
```

```
# if you need to do this, you can override this setting on individual services
```

```
public: false
```

Configurer un groupe de service

La seconde partie de la configuration, comme l'indiquent les commentaires, permet de définir un [ensemble de services](#) en une seule ligne. Ici nous indiquons à Symfony que [toutes les classes du Bundle AppBundle](#) peuvent être utilisées comme un [service](#). L'attribut `exclude` permet de spécifier les classes à exclure de cette liste. Ici c'est les classes des dossiers Entity, Repository et Tests qui le sont.

```
# makes classes in src/AppBundle available to be used as services  
# this creates a service per class whose id is the fully-qualified class  
name
```

AppBundle\:

```
resource: ' ../ ../src/AppBundle/* '
```

```
# you can exclude directories or files
```

```
# but if a service is unused, it's removed anyway
```

```
exclude: ' ../ ../src/AppBundle/{Entity,Repository,Tests} '
```

Configurer les contrôleurs

Cette partie permet de faire en sorte que les contrôleurs soient directement [accessible via le get](#). En effet, l'attribut `public` est à [true](#).

```
# controllers are imported separately to make sure they're public  
# and have a tag that allows actions to type-hint services  
AppBundle\Controller\:  
    resource: '../src/AppBundle/Controller'  
    public: true  
    tags: ['controller.service_arguments']
```

Injection de dépendances

L'injection des services dans les différentes classes de votre projet se fait à travers [l'injection de dépendances](#). En effet, vous êtes entrain de travailler avec un principe très important et qui permet de minimiser les dépendances entre les classes. Ici l'instanciation n'est plus votre soucis. C'est le conteneur de service qui s'occupe de to

L'injection de dépendance est un patron de conception qui a vu le jour afin de palier à un problème très récurrent : Avoir dans son code des classes qui [dépendent les unes des autres](#). L'injection de dépendance palie à ce problème en découplant les classes.

Avec le conteneur de service de Symfony on peut injecter des services ou des paramètres.

Injection manuelle des arguments

Lorsque vous voulez ajouter des variables à vos services, l'autowiring ne marche plus. Dans ce cas, vous devez injecter ces paramètres manuellement.

Au niveau de votre fichier service.yml, vous devez configurer votre service et définir vos arguments.

```
AppBundle\Updates\SiteUpdateManager:  
  arguments:  
    $maVariable: 'contenu de ma variable'
```

Syntaxe Injection manuelle des arguments

Avant la version 3.3 de Symfony l'injection se faisait toujours manuellement de la façon suivante :

```
transfer.common.service:
    class:
AppBundle\Service\Front\Transfer\CommonTransferService
    arguments:
        - "@mailer"
        - "%mailer_user%"
```

A partir de la version 3.3 les services sont injecter avec le type Hinting comme nous l'avons vu. Pour les paramètres nous utilisons directement le nom de la variable que nous récupérons directement au niveau du constructeur.

```
AppBundle\Updates\SiteUpdateManager:
    arguments:
        $maVariable: 'contenu var'
```

```
class MaClasse
{
    private $adminEmail;
    public function __construct($adminEmail)
    {
        $this->adminEmail = $adminEmail;
    }
}
```

Sauvegarder ses paramètres avant de les utiliser

Au lieu d'écrire directement les paramètres lors de l'injection manuelle, ce qui nous oblige à le faire chaque injection, nous devons [définir nos paramètres dans un des fichiers de configuration](#) et les invoquer ensuite.

La définition des paramètres peut se faire dans le fichier [config.yml](#), [parameters.yml](#) ou même [services.yml](#). Quel que soit le fichier ces paramètres sont sauvegardé sous la clé [parameters](#).

```
AppBundle\Updates\SiteUpdateManager:  
  arguments:  
    $maVariable: '%ma_var%'
```

```
# parameters.yml  
parameters:  
  database_host: 127.0.0.1
```

Injecter un service ou des paramètres de configuration dans un autre service

Afin d'injecter des services dans un autre service nous devons passer par le constructeur du service cible et du type Hinting. Le fait d'avoir l'option d'autowiring qui cible le service en question permet de gérer ça.

```
class MaClasse
{
    private $logger;
    public function __construct(LoggerInterface $logger)
    {
        $this->logger = $logger;
    }
    public function LoggerMessage()
    {
        $this->logger->info('J'utilise un service injecté grâce à l'autowiring et le Type Hinting appelé au niveau de mon constructeur!');
    }
}
```

Pour les paramètres, le principe reste le même et les paramètres sont injectés manuellement.