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

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

Responsabilité unique (Single Responsability)

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. Faite une recherche sur ces concepts. Pour le premier principe de « Single Responsability » 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 ou 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 Responsability permet d'avoir un code réutilisable et facilement maintenable.

Le conteneur de services

Un service est une classe associée 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 à 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

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 tout.

L'injection de dépendance est un patron de conception qui a vu le jour afin de palier à un problème très récurent : 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.

Accéder à un service

Plusieurs services sont déjà accessibles dés 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. Dans le guide des bonnes pratiques de Symfony, cette méthode est à éviter au maximum.

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

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

Création d'un premier service

Créer un dossier service.

Créer une classe PremierService

Introduirez y une méthode getRandomString($nb) qui prend en paramètre le nombre de caractère et retourne une chaine de taille ce nombre la. Ensuite, appelez ce service dans une de vos actions et afficher la valeur de retour. .

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

```php
use AppBundle\Service\PremierService;

public function
testAction(PremierService $ps){
    dump($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.

```yaml
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.
```

php bin/console debug:autowiring : permet d'afficher les services autowiré

Configurer un groupe de service

La seconde partie de la configuration, comme l'indique les commentaires, permet de définir un ensemble de services en une seule ligne. Ici nous indiquons à Symfony que toutes les classes du dossier src 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.

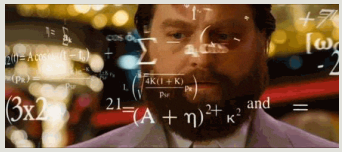
```
# makes classes in src/ available to be used as services
# this creates a service per class whose id is the fully-qualified class name
App\:
    resource: '../src/*'
    exclude: '../src/{DependencyInjection,Entity,Migrations,Tests,Kernel.php}'
```

Configurer les contrôleurs

Cette partie permet de faire en sorte que les contrôleurs puissent injecter les services directement dans les actions mêmes sans étendre le AbstractController

```
# controllers are imported separately to make sure services can be injected
# as action arguments even if you don't extend any base controller class
App\Controller\:
    resource: '../src/Controller'
    tags: ['controller.service_arguments']
```

Exercice



- Le logger est un service très important. Il permet de logger des informations que vous pouvez récupérer en mode dev mais surtout e mode prod ou votre débugger n'est pas fonctionnel.
- Reprenez l'action de modification et d'ajout d'une formation et logger les informations sur ces actions.

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.

Si vous avez besoin d'injecter la variable uniquement au niveau d'un ou plusieurs services particuliers, vous devez le faire de la manière suivante :

```
App\Controller\PersonneController:
    arguments:
        - $defaultImagePath: '%kernel.project_dir%/public/images/default.png'
```

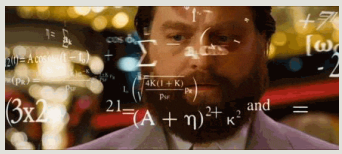
Si vous avez besoin d'injecter la variable partout, vous devez le faire de la manière suivante :

```
_defaults:
    autowire: true
    autoconfigure: true
    bind:
        $defaultImagePath: '%kernel.project_dir%/public/images/default.png'
```

Ajouter des paramètres

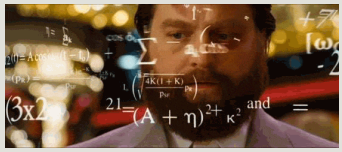
```yaml
parameters:
    locale: 'en'
    imagePath: 'images/default.png'

services:
_defaults:
        autowire: true
        autoconfigure: true
        bind:
            $defaultImagePath: '%imagePath%'
```

Exercice



- L'upload d'image est une fonctionnalité qui se répète souvent. D'un autre coté ceci est un traitement métier qui ne doit pas figurer dans vos contrôleurs. N'oublier pas «thin controller ».
- Créer un service qui permet de gérer l'upload et utiliser le.

Exercice



- Dans la plupart de vos contrôleurs, vous utilisez l'entity manager et le repository. Cependant vous êtes entrain de les récupérer à travers des helpers.
- Maintenait que nous savons faire de l'injection de dépendance. Et que ce sont des services, injectons les pour avoir plus de découplage au niveau de notre projet.

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 contructeur!');
    }
}
```

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

Génération de PDF avec KnpSnappyBundle

Télécharger et installer l'outil wkhtmltopdf http://wkhtmltopdf.org/

Installer le bundle KnpSnappyBundle permettant la génération de PDF.

https://github.com/KnpLabs/KnpSnappyBundle/tree/322a3b33ea9d71c4395f600ad81f299b9a526d36

Créer un service permettant de manipuler ce bundle.

N'utiliser pas le get de la documentation injecter vos services en utilisant le type Hint.

Envoi d'email

Symfony utilise le bundle SwiftMailer pour l'envoi d'email.

Si vous n'avez pas le bundle installer le : composer require symfony/swiftmailer-bundle

La configuration se fait au niveau de votre fichier .env dans la variable MAILER_URL. Sachant que vous allez utiliser des informations critiques telques le mot de passe du compte email à utiliser, utiliser le fichier .env.local.

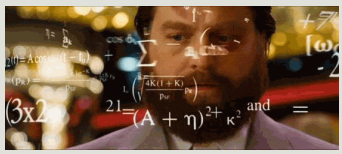
Si vous utiliser Gmail voici la configuration requise :

MAILER_URL=gmail://username:password@localhost

Envoi d'un email

```
$message = (new \Swift_Message('Hello Email'))
    ->setFrom('aymn.noreply@gmail.com')
    ->setTo($to)
    ->setBody(
        $template,
        $contentType
    );
if ($attachement) {
    $attachementObject = new \Swift_Attachment($attachement,
        'attachement.pdf',
        'application/pdf' );
    $message->attach($attachementObject);
}

$this->mailer->send($message);
```

Exercice



- Créer un service permettant la manipulation des emails.
- Essayer de rendre votre service générique.