



Les Services

Lotfi KHEDIRI



Un service?

- Une application symfony regorge d'**objets utiles**
- Par exemple un objet "Mailer" peut vous aider à envoyer des e-mails
- Presque *tout* ce que votre application «fait» est en fait réalisé par l'un de ces objets.
- En installant un nouveau bundle, on obtient plus de ces objets utiles.
- Dans Symfony, ces objets utiles sont appelés **services**
- Chaque service vit à l'intérieur d'un objet très spécial appelé **conteneur de service (Service container)**.
- Le conteneur vous permet de **centraliser** la façon dont ces objets (services) sont construits.
- Ils facilitent la vie du développeur, favorise une **architecture solide** et une meilleure **maintenabilité** de l'application.



Récupération et utilisation des services

- Dans une application Symfony vierge, le conteneur contient *déjà* de nombreux services.
- Dans un contrôleur, vous pouvez "**demander**" un service à partir du conteneur en tapant **un argument avec le nom de classe ou d'interface du service**.
- Exemple : Enregistrer un message dans les fichiers Log

Récupération et utilisation des services

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

use Psr\Log\LoggerInterface;

class ProductController
{
    /**
     * @Route("/products")
     */
    public function list(LoggerInterface $logger)
    {
        $logger->info('Look! I just used a service');

        // ...
    }
}
```

➤ En php lorsque un argument d'une méthode est typé : c'est le **Type Hinting**

➤ **Injection** du service (\$logger) de type "LoggerInterface" dans l'action contrôleur "list" ==> c'est récupération du service

➤ **Utilisation** du service \$logger en appelant la méthode info

Quels autres services sont disponibles?

```
$ 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)
```

```
[...]
```

➤ Lorsqu'on utilise le **type-hints** dans les méthodes du contrôleur, symfony (container) nous pass l'objet service ayant le type indiqué.

➤ Cette commande ne donne pas la liste complète de tous les services, pour l'avoir utilisé plutôt :
bin/console debug:container

Création / configuration de services dans le conteneur

- **Exemple** : créer un service qui permet d'afficher à vos utilisateurs un message heureux et aléatoire.
- Il faut créer une **nouvelle classe**
- On peut utiliser ce service immédiatement à l'intérieur d'un contrôleur
- Par défaut cette classe est un service (grâce à la configuration dans services.yml)

```
// src/Service/MessageGenerator.php
namespace App\Service;

class MessageGenerator
{
    public function getHappyMessage()
    {
        $messages = [
            'You did it! You updated the system! Amazing!',
            'That was one of the coolest updates I\'ve seen all day!',
            'Great work! Keep going!',
        ];

        $index = array_rand($messages);

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


Utilisation du service

- On demande le service **MessageGenerator** en utilisant le "Type Hinting"
- le **container** construit et renvoie **automatiquement** un nouvel objet de type `MessageGenerator`.
- Ceci est possible grâce au **composant "DependencyInjection"** de symfony
- Ce dernier assure **l'injection du service**
- Ce service représente une **dépendance** par rapport à la classe contrôleur

- On peut **utiliser** ce service **immédiatement** à l'intérieur d'un contrôleur

```
use App\Service\MessageGenerator;  
  
public function new MessageGenerator $messageGenerator)  
{  
    // thanks to the type-hint, the container will instantiate a  
    // new MessageGenerator and pass it to you!  
    // ...  
  
    $message = $messageGenerator->getHappyMessage();  
    $this->addFlash('success', $message);  
    // ...  
}
```

➡ c'est le concept d'**autowiring**.



Remarques

- Un service ***jamais demandé***, n'est ***jamais construit***.
- Ceci permet une économie de mémoire et de vitesse.
- Un service n'est **créé** (instancié) *qu'une seule fois*.
- La **même instance** est retournée par le container à chaque fois que le service est demandé

Configurations des services (services.yml)

- La configuration de service suivante est la configuration **par défaut** pour un **nouveau projet**:

```
# config/services.yml
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, e
                               e

    # 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}'

    # ...
```

Configurations des services (services.yml)

Grace à cette configuration

- Toutes les classes du répertoire "src/" peuvent être utilisées en tant que service
- Pas de configuration manuellement supplémentaire.
- La configuration explicite et manuelle (avant la version 3 de symfony) service par service est possible.
- L'option **resource** permet d'importer plusieurs services simultanément.
- L'option **exclude** permet d'exclure certains chemins
- Tous les services sont privés par défaut : "**public : false**" cela signifie que le service ne peut être récupéré directement à partir du conteneur (avec `$container->get("idservice")`)

L'option autowire

- La section "**_defaults**" permet de définir des propriétés qui s'appliquent à tous les services définis dans ce fichier.
- "**autowire: true**" : Active **l'Autowiring** c'est-à-dire que les services peuvent être injecter dans les constructeurs d'autres services ou dans les actions des controleurs.

```
# config/services.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, a
```

L'option de configuration automatique

- **"autoconfigure: true"** : Avec ce paramètre, le conteneur appliquera automatiquement une certaine configuration à vos services, en fonction de la classe de votre service.

```
# config/services.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, a
```

Public Versus Private Services

```
class MessageGenerator
{
    private $container;

    public function __construct(ContainerInterface $container)
    {
        $this->container = $container;
    }

    public function generate(string $message, string $template = null, array $context = [])
    {
        if ($template && $this->container->has('twig')) {
            // there IS a public "twig" service in the container
            $twig = $this->container->get('twig');

            return $twig->render($template, $context + ['message' => $message]);
        }
    }
}
```

- Depuis Symfony 4.0, chaque service défini est **privé par défaut**.
- Lorsqu'un service **est** public, on peut y **accéder directement** depuis l'objet **conteneur**, qui peut également être injecté grâce au câblage automatique.

```
# config/services.yaml
```

```
services:
```

```
# ... same code as before
```

```
# explicitly configure the service
```

```
Acme\PublicService:
    public: true
```

Injection et configuration de services dans un service

- Si on a besoin d'un **Service2** à l'intérieur d'un **Service1** il faut juste créer la méthode **__construct()** dans service1 avec un argument typé (type-hint) avec Service2.
- Définir une nouvelle propriété **\$service2** et utiliser-la plus tard

```
// src/Service/MessageGenerator.php
namespace App\Service;

use Psr\Log\LoggerInterface;

class MessageGenerator
{
    private $logger;

    public function __construct(LoggerInterface $logger)
    {
        $this->logger = $logger;
    }

    public function getHappyMessage()
    {
        $this->logger->info('About to find a happy message!');
        // ...
    }
}
```


Injection de **plusieurs** services dans un service

- Un service peut dépendre de **plusieurs autres services**, dans ce cas il suffit de les injecter dans son constructeur.


```
class SiteUpdateManager
{
    private $messageGenerator;
    private $mailer;

    public function __construct(MessageGenerator $messageGenerator, MailerInterface $mailer)
    {
        $this->messageGenerator = $messageGenerator;
        $this->mailer = $mailer;
    }
}
```



Et si un service à besoin d'un argument scalaire (n'est pas une classe)

```
public function __construct(MessageGenerator $messageGenerator, \Swift_Mailer $mailer, $adminEmail)
{
    // ...
    $this->adminEmail = $adminEmail;
}
```



Argument non typé donc il ne peut être autowiré automatiquement

- Pour résoudre ce problème il faut **configurer un argument** pour le service dans le fichier **services.yml**.

Configurer un argument de service

- En modifiant le fichier de configuration on obtient :

```
# config/services.yaml
services:
    # ... same as before

    # same as before
    App\:
        resource: '../src/*'
        exclude: '../src/{DependencyInjection,Entity,Migrations,Tests,Kernel.php}'

    # explicitly configure the service
    App\Updates\SiteUpdateManager:
        arguments:
            $adminEmail: 'manager@example.com'
```

Choisir un service spécifique

```
use Psr\Log\LoggerInterface;

class MessageGenerator
{
    private $logger;

    public function __construct(LoggerInterface $logger)
    {
        $this->logger = $logger;
    }
    // ...
}
```

- Le service MessageGenerator créé précédemment nécessite un argument **LoggerInterface** (le type est une interface)
- Cependant, il y a **plusieurs services** dans le conteneur qui implementent LoggerInterface, tels que logger, monolog.logger.request, monolog.logger.php, etc.
Comment le conteneur decide lequel utiliser?
- Dans ces situations, le conteneur est généralement configuré pour choisir automatiquement l'un des services dans ce cas **logger**

Choisir un service spécifique

- Symfony permet de **choisir** une classe spécifique dans la configuration

```
# config/services.yaml
services:
    # ... same code as before

    # explicitly configure the service
    App\Service\MessageGenerator:
        arguments:
            # the '@' symbol is important: that's what tells the container
            # you want to pass the *service* whose id is 'monolog.logger.request',
            # and not just the *string* 'monolog.logger.request'
            $logger: '@monolog.logger.request'
```

Les services courants Symfony

Identifiant	Description
doctrine.orm.entity_manager	Ce service est l'instance de l'EntityManager de Doctrine ORM. Ainsi, lorsque dans un contrôleur vous faites <code>\$this->getDoctrine()->getManager()</code> , vous récupérez en réalité le service <code>doctrine.orm.entity_manager</code> .
event_dispatcher	Ce service donne accès au gestionnaire d'évènements.
kernel	Ce service vous donne accès au noyau de Symfony. Grâce à lui, vous pouvez localiser des bundles, récupérer le chemin de base du site, etc.
logger	Ce service gère les logs de votre application. Grâce à lui, vous pouvez utiliser des fichiers de logs très simplement.
mailer	Ce service vous renvoie par défaut une instance de <code>Swift_Mailer</code> , une classe permettant d'envoyer des e-mails facilement.

Les services courants de Symfony

Identifiant	Description
request_stack	Ce service est très important : Donne un objet qui permet de récupérer la requête Request courante via sa méthode <code>getCurrentRequest</code> .
router	Ce service vous donne accès au routeur (Symfony\Component\Routing\Router).
security.token_storage	Ce service permet de gérer l'authentification sur votre site internet. On l'utilise notamment pour récupérer l'utilisateur courant. Le raccourci du contrôleur <code>\$this->getUser()</code> exécute en réalité <code>\$this->container->get('security.token_storage')->getToken()->getUser()</code> !
service_container	Ce service vous renvoie le conteneur de services lui-même.
twig	Ce service représente une instance de Twig_Environment . Il permet d'afficher ou de retourner une vue.
templating	Ce service représente le moteur de templates de Symfony. Par défaut il s'agit de Twig.