



Simon GAUTIER



# Services Symfony

# Injection de dépendances

Injection de dépendances en Symfony

Le conteneur de services

Comprendre les mécanismes autowire et autoconfigure

Les tags

# Travailler sans injection de dépendance

- Quels problèmes avec cette manière de faire ?

```
class MonLoggerSansInjectionDeDependance
{
    public function log($message, $level = 'INFO')
    {
        $formatter = new XmlFormatter();
        $log = $formatter->format($message, $level);
        $this->writeLog($log);
    }
}
```

```
$logger = new Logger();
$logger->log('My message');
```

# Travailler sans injection de dépendance

- Quels problèmes avec cette manière de faire ?

```
class MonLoggerSansInjectionDeDependance
{
    public function log($message, $level = 'INFO')
    {
        $formatter = new XmlFormatter();
        $log = $formatter->format($message, $level);
        $this->writeLog($log);
    }
}
```

```
$logger = new Logger();
$logger->log('My message');
```

- Le logger est limité car il ne sait pas formater autrement qu'en XML
- Le logger n'est pas flexible car il ne sait utiliser que le formatter XmlFormatter et ne sait pas utiliser un autre formatter XML
- Le logger sera difficile à tester car il crée sa dépendance à un formatter → comment savoir si c'est le logger ou le formatter qui ne fonctionne pas ? Dans la pratique, éviter d'utiliser le new

# Retour sur l'exemple

## Supprimer la dépendance à XmlFormatter

```
class MonLoggerSansInjectionDeDependance
{
    private $formatter;

    // La dépendance est désormais injectée via le constructeur
    function __construct(XmlFormatter $formatter)
    {
        $this->formatter = $formatter;
    }

    // La méthode log() s'appuie sur le XmlFormatter injecté lors de la construction de l'objet
    public function log($message, $level = 'INFO')
    {
        $log = $this->formatter->format($message, $level);
        $this->writeLog($log);
    }
}
```

```
$logger = new Logger(new XmlFormatter());
$logger->log('My message');
```

# Retour sur l'exemple

## Abstraction du formatter utilisé

```
class MonLoggerSansInjectionDeDependance
{
    private $formatter;

    // Plutôt que de typer fortement le formatter, on utilise un type générique
    // Notre classe est donc plus souple, elle peut s'appuyer sur tout type de formatter
    // Bonne pratique : Formatter doit être une classe abstraite ou une interface
    function __construct(Formatter $formatter)
    {
        $this->formatter = $formatter;
    }

    public function log($message, $level = 'INFO')
    {
        $log = $this->formatter->format($message, $level);
        $this->writeLog($log);
    }
}
```

```
// Il est possible d'utiliser notre classe avec un formatter XML ...
```

```
$logger = new Logger(new XmlFormatter());
$logger->log('My message');
```

```
// ... ou un formatter Json...
```

```
$logger = new Logger(new JsonFormatter());
$logger->log('My message');
```

```
// ... ou plus généralement à toute classe qui hérite de Formatter (ou implémente Formatter si Formatter est une interface)
```

# Qu'est-ce que l'injection de dépendances ?

- Définition Wikipedia :
  - L'injection de dépendances (dependency injection en anglais) est un mécanisme qui permet d'implémenter le principe de l'inversion de contrôle.
  - Il consiste à créer dynamiquement (injecter) les dépendances entre les différentes classes en s'appuyant sur une description (fichier de configuration ou métadonnées) ou de manière programmatique. Ainsi les dépendances entre composants logiciels ne sont plus exprimées dans le code de manière statique mais déterminées dynamiquement à l'exécution.

# Types d'injection de dépendance

- Injection via le constructeur
  - Recommandé pour la plupart des applications Symfony
- Injection via un setter
  - Principalement utilisé lorsque les dépendances sont optionnelles
- Injection via une propriété de l'objet
  - Approche risquée

# Injection de dépendance via le constructeur

- Présenté dans l'exemple précédent
- Recommandé dans les applications Symfony
- Avantages :
  - Les dépendances injectées via le constructeur sont obligatoires → on est certain d'y accéder dans l'objet
  - Les dépendances injectées restent inchangées durant la vie de l'objet
  - Il est possible de rendre générique la classe injectée
- Inconvénients :
  - Une fois l'objet construit, il n'est plus possible de modifier la dépendance injectée
  - Dispersion du code (classes et sous-classes)

```
class Logger
{
    private $formatter;

    function __construct(Formatter $formatter)
    {
        $this->formatter = $formatter;
    }
}
```



# Injection de dépendance via un setter

- Avantages :
  - Permet facilement d'avoir des dépendances optionnelles
  - Il est possible d'appeler plusieurs fois la méthode si par exemple on veut avoir plusieurs dépendances injectées
- Inconvénients :
  - Les dépendances peuvent évoluer au cours de la vie de l'objet, ce qui rend son utilisation plus complexe
  - Rien ne garantit à un instant que le setter a été appelé

```
class Logger
{
    private $formatter;

    function setFormatter(Formatter $formatter)
    {
        $this->formatter = $formatter;
    }
}
```

# Injection de dépendance via une propriété de l'objet

- Pour fonctionner, la propriété doit être publique, ce qui permet d'injecter la dépendance depuis partout dans l'application
- Avantages :
  - Permet facilement d'avoir des dépendances optionnelles
- Inconvénients :
  - Les mêmes que l'injection de dépendance via un setter
  - Il n'y a aucun typage et il est donc possible d'injecter n'importe quoi

```
class Logger
{
    public $formatter;
}
```

# Le conteneur de services Symfony

- Problématiques à traiter lorsqu'on souhaite faire de l'injection de dépendances :
  - Lors de la construction d'un objet, il est nécessaire de connaître chaque objet injecté (dans l'exemple ci-dessous : XmlFormatter)
  - Il faut gérer les dépendances circulaires  
➔ dans l'exemple ci-dessous, comment ferions-nous si XmlFormatter avait besoin pour fonctionner du Logger ?

```
$logger = new Logger(new XmlFormatter());  
$logger->log('My message');
```

- Le conteneur de services Symfony répond à ces problématiques
- Dans cette présentation, les services sont les classes manipulées (en l'occurrence : Logger et XmlFormatter)

# Définition générale d'un service

- Un service est une simple classe PHP qui remplit une tâche définie (ex : logger, envoyer un mail, crypter / décrypter une valeur / ...)
- Un service est accessible de partout dans Symfony
- La notion de service permet de parler de programmation orientée services
  - Découper / séparer chaque fonctionnalité
  - Créer des fonctionnalités unitaires ➔ permet de les rendre réutilisables

# Le conteneur de services Symfony

- Définition générale : Un conteneur de services est un objet PHP dont le rôle est de gérer l'instanciation des services
  - Joue le rôle de chef d'orchestre des services dans Symfony
  - Si un service est déjà instancié et qu'on le demande de nouveau, le conteneur le retourne sans créer de nouvel objet
  - Un service peut dépendre d'autres services pour fonctionner
    - Le conteneur de services instancie les dépendances avant d'instancier le service demandé et de le retourner
    - Les dépendances de services sont déclarées dans la configuration des services (à suivre)
- Symfony fournit de base plusieurs services, il est possible d'en créer d'autres pour ses propres besoins
- Le conteneur de services Symfony est central dans le framework :
  - Toutes les classes du noyau l'utilisent
  - Il fournit une architecture qui favorise un code réutilisable et découplé
  - Doc : [symfony.com/doc/current/service\\_container.html](https://symfony.com/doc/current/service_container.html)

# Lister les services via la console

**php bin/console debug:container**

- Chaque service est une classe qui a son propre ID
- On constate que certaines classes que nous avons créées sont déjà déclarées comme des services

- Obtenir les détails d'un service en particulier :

**php bin/console debug:container ID-DU-SERVICE**, exemple :

**php bin/console debug:container twig**

# Configuration / déclaration des services

- Avant Symfony 3.3, chaque service devait être explicitement déclaré
- Depuis, le mécanisme d'« autowiring » simplifie le travail
- Pour illustrer le fonctionnement, on reprend l'exemple du Logger :
  - Le Logger s'appuie sur un formatter Json
  - Le Logger est utilisé par une classe Mailer qui devra loguer tout envoi de mail
  - Pour déclencher l'envoi d'un mail, on utilise un contrôleur
  - Injections à prévoir :
    - Formatter Json dans Logger / Logger dans Mailer / Mailer dans contrôleur

# Classe Logger

```
<?php
namespace App\TestServices;

use Monolog\Formatter\FormatterInterface;
use Psr\Log\LoggerInterface;

// Implémentation d'une interface générique de logger
class Logger implements LoggerInterface
{
    private $formatter;

    // On injecte un formatter sous forme d'une interface générique
    // ==> Notre logger est générique
    public function __construct(FormatterInterface $formatter)
    {
        $this->formatter = $formatter;
    }

    public function info($message, array $context = array())
    {
        $log = $this->formatter->format(array($message));

        // On se contente de dumper le log dans notre exemple
        dump($log);
    }

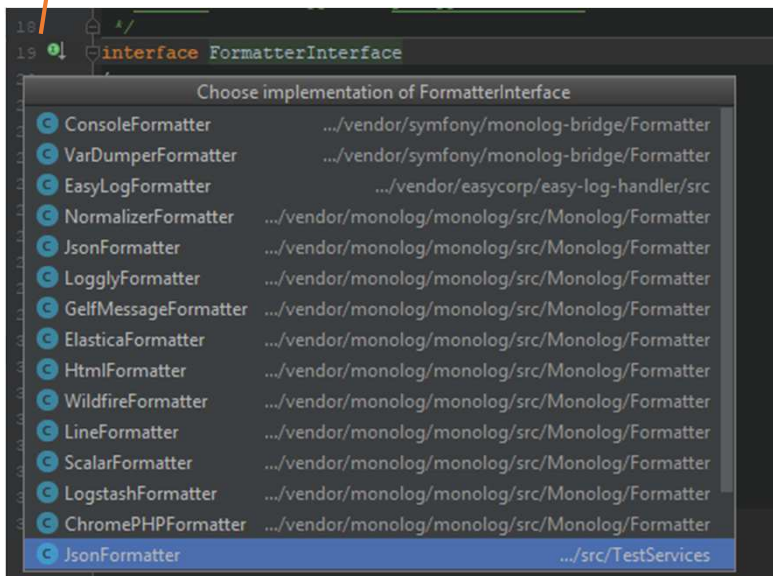
    // Implémentation des autres méthodes de l'interface
}
```



Zoom sur l'interface

## **\Monolog\Formatter\FormatterInterface**

- On constate que de nombreuses classes implémentent cette interface  
→ Problématique : comment faire pour que Symfony injecte notre classe ?



Nous allons voir que la réponse se situe dans le fichier de configuration des services : **config/services.yaml**

# Classe JsonFormatter

```
<?php
namespace App\TestServices;

use Monolog\Formatter\FormatterInterface;

class JsonFormatter implements FormatterInterface
{
    public function format(array $record)
    {
        // On se contente de faire simplement un json_encode
        return json_encode($record);
    }

    public function formatBatch(array $records)
    {
        // On se contente de faire simplement un json_encode
        return json_encode($records);
    }
}
```

# Classe Mailer

```
<?php
namespace App\TestServices;

use Swift_Message;

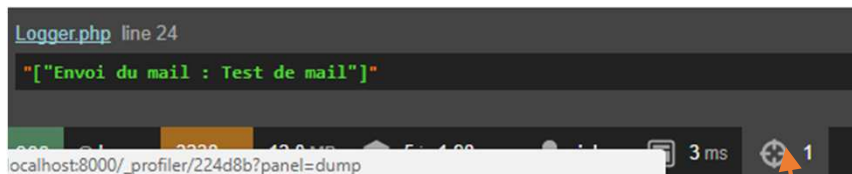
class Mailer
{
    private $logger;

    // On injecte notre logger même si on aurait pu injecter une interface générique à la place
    // ==> on fait ça pour l'exercice, nous verrons comment Symfony traite ce cas
    public function __construct(Logger $logger)
    {
        $this->logger = $logger;
    }

    // Nécessite d'avoir installé un recipe dédié :
    // $ composer req mailer
    public function send(Swift_Message $message)
    {
        // Ajouter du code pour faire l'envoi de l'email ...
        $this->logger->info('Envoi du mail : ' . $message->getSubject());
    }
}
```

# Contrôleur

Résultat :



Survoler pour voir le dump

```
<?php
namespace App\Controller;

use App\TestServices\Mailer;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

/**
 * Class DefaultController
 * @package App\Controller
 */
class DefaultController extends Controller
{
    /**
     * @Route("/", name="home")
     * @param Mailer $mailer
     * @return Response
     */
    public function indexAction(Mailer $mailer)
    {
        // Le mailer est injecté directement dans l'action

        // Construction et envoi du message
        $message = (new \Swift_Message('Test de mail'))
            ->setFrom('send@example.com')
            ->setTo('recipient@example.com')
            ->setBody(
                $this->renderView('emails/test.html.twig'),
                'text/html'
            );
        $mailer->send($message);
        return $this->render('hello_world.html.twig');
    }
}
```

# Quelle configuration pour ce résultat ?

## **config/services.yaml**

Commençons par analyser le contenu du fichier fourni de base par Symfony :

```
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. => notion vue plus tard
        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

    # makes classes in src/ available to be used as services
    # this creates a service per class whose id is the fully-qualified class name (FQCN)
    App\:
        resource: '../src/*'
        # you can exclude directories or files
        # but if a service is unused, it's removed anyway
        exclude: '../src/{Entity,Migrations,Tests}'

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

Ce mécanisme déclare automatiquement nos classes `Logger`, `JsonFormatter` et `Mailer` comme des services (car elles sont dans le namespace `App`)

Chaque service déclaré a un ID unique (le nom de classe complet, ex : **`App\TestServices\Logger`**)

# Quelle configuration pour ce résultat ?

## **config/services.yaml**

```
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. => notion vue plus tard
        autoconfigure: true
        # this means you cannot fetch services directly from
        # if you need to do this, you can override this setting
        public: false

    # 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/*'
        # you can exclude directories or files
        # but if a service is unused, it's removed anyway
        exclude: '../src/{Entity,Migrations,Tests}'

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

autowire à true signifie que Symfony injecte de manière automatique des services à d'autres services en se basant uniquement sur le type défini dans le constructeur du service.

Exemple : classe Mailer → grâce à l'autowire, Symfony injecte le service Logger grâce à sa définition dans le constructeur.

On a vu que dans certains cas le type était ambigu car il pouvait correspondre à plusieurs classes (ex : une interface implémentée par plusieurs classes) → on verra comment gérer ce cas.

# Quelle configuration pour ce résultat ?

## **config/services.yaml**

```
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. => notion vue plus tard
        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

    # makes classes in src/ available to be used as services
    # this creates a service per class whose id is the fully-qualified class name (FQCN)
    App\:
        resource: '../src/*'
        # you can exclude directories or files
        # but if a service is unused, it's removed anyway
        exclude: '../src/{Entity,Migrations,Tests}'

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

Ce mécanisme ajoute un comportement supplémentaire aux contrôleurs : l'injection de dépendance n'est pas seulement réservée au constructeur, elle est aussi disponible sur chaque méthode d'un contrôleur.

Exemple : `DefaultController::indexAction()` : Symfony injecte le service Mailer grâce à sa définition dans le prototype de la méthode `indexAction()`.

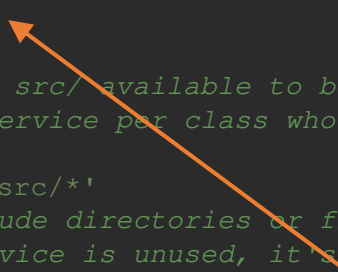
# Quelle configuration pour ce résultat ?

## **config/services.yaml**

```
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. => notion vue plus tard
        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

    # makes classes in src/ available to be used as services by this container
    # this creates a service per class whose id is the class name
    App\:
        resource: '../src/*'
        # you can exclude directories or files if you want
        # but if a service is unused, it's not loaded
        exclude: '../src/{Entity,Migrations,Tests}'

    # controllers are imported separately to make sure services use
    # the container's autowiring
    App\Controller\:
        resource: '../src/Controller'
        tags: ['controller.service_arguments']
```



Par défaut, les services sont tous privés.

Il ne s'agit que d'un comportement par défaut, il est possible de rendre un service public par exception à cette règle générale (vu plus loin).

Afficher les services y compris les services privés :

**php bin/console debug:container** **--show-hidden**

→ On constate bien que les services créés ont comme ID leur FQCN, ex :

**App\TestServices\Logger**



# Accès aux services depuis un contrôleur

- Via l'injection de dépendance comme présenté (à privilégier) :

```
public function indexAction(Mailer $mailer)
{
    // ...
}
```

- En invoquant explicitement le service dans le code (à éviter au maximum) :

```
public function indexAction()
{
    // Attention : cette méthode requiert que le service soit public ==> à faire dans config/services.yaml
    // Rappel : le service a par défaut pour code son FQCN ==> Mailer::class correspond à App\TestServices\Mailer
    $mailer = $this->get(Mailer::class);

    // Méthode équivalente :
    $mailer = $this->container->get(Mailer::class);

    // ...
}
```

➔ Pour que cette méthode puisse fonctionner, rendre le service public :

```
services:
    _defaults:
        # ...
        public: false
        # ...

    App\TestServices\Mailer:
        public: true # Exception valable uniquement pour ce service
```

# Comment définir la classe injectée quand Symfony ne peut pas faire le choix ?

- Rappel : notre classe `Logger` injecte l'interface `FormatterInterface` qui correspond potentiellement à plusieurs classes

```
services:
  # ...
  App\TestServices\Logger: # On identifie le service Logger par son ID
    bind: # bind : mapper chaque paramètre du contrôleur avec la classe réelle à instancier
      Monolog\Formatter\FormatterInterface: '@App\TestServices\JsonFormatter'
      # Monolog\Formatter\FormatterInterface : type en paramètre du constructeur
      # @App\TestServices\JsonFormatter : service à injecter ==> ici, le code identifie le service JsonFormatter
```

- Il est possible de définir la règle `bind` dans la section **`_defaults`** → tous les services auraient cette règle appliquée

# Les tags

- Les tags permettent de « marquer » les services dans Symfony
  - Exemple : dire d'un service qu'il est une extension Twig ➔ utiliser le tag « **twig.extension** »

```
services:
    # ...
    App\Twig\MonExtensionTwig:
        tags:
            # grâce au tag 'twig.extension', le service sera reconnu comme une extension Twig
            - { name: 'twig.extension' }
```

# autoconfigure

```
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. => notion vue plus tard
        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

    # makes classes in src/ available to be used as services by the container
    # this creates a service per class
    App\:
        resource: '../src/*'
        # you can exclude directories or files
        # but if a service is unused, it will not be registered
        exclude: '../src/{Entity,Migrations,Controller}'

    # controllers are imported separately to make sure services can
    # have the tag that allows actions to be dispatched
    App\Controller\:
        resource: '../src/Controller'
        tags: ['controller.service_arguments']
```

Par défaut, autoconfigure est à true pour tous les services de ce fichier  
Conséquence : Symfony n'a plus besoin que les tags soient explicitement précisés.

Dans l'exemple, pour ne pas avoir à définir le tag « **twig.extension** », il suffit que la classe du service implémente l'interface **\Twig\Extension\AbstractExtension**. Ceci suffit à Symfony pour faire le lien, si le service est défini en autoconfigure.

# autoconfigure

- Sans autoconfigure :

```
services:
    # ...
    App\Twig\MonExtensionTwig:
        tags:
            # grâce au tag 'twig.extension', le service sera reconnu comme une extension Twig
            - { name: 'twig.extension' }
```

- Avec autoconfigure :

```
services:
    # ...
```

- Symfony intègre de base de nombreux tags, cf. doc :  
[symfony.com/doc/current/reference/dic\\_tags.html](https://symfony.com/doc/current/reference/dic_tags.html)

# Injection d'arguments scalaires à un service

- Ex : dans notre service Mailer, injecter en plus du logger une chaîne de caractères

```
<?php
namespace App\TestServices;

use Swift_Message;

class Mailer
{
    private $logger;
    private $logPrefix;

    // Contrairement au paramètre Logger, le paramètre string (comme tout scalaire) ne peut pas être
    // alimenté en autowire car Symfony ne saurait pas quelle valeur injecter !
    public function __construct(Logger $logger, string $logPrefix)
    {
        $this->logger = $logger;
        $this->logPrefix = $logPrefix;
    }

    public function send(Swift_Message $message)
    {
        // Ajouter du code pour faire l'envoi de l'email ...
        $this->logger->info($this->logPrefix . $message->getSubject());
    }
}
```

```
services:
    # ...
    App\TestServices\Mailer:
        arguments:
            $logPrefix: 'Envoi du message : '
```

```
parameters:
    # ...
    logger.prefix: 'Envoi du message : '

services:
    # ...
    App\TestServices\Mailer:
        arguments: # Bonne pratique : utiliser des variables
            $logPrefix: '%logger.prefix%'
```

# Injection d'arguments scalaires à un service

- Cas où le service n'a que des arguments scalaires :
  - Possibilité de simplifier le passage des paramètres, respecter l'ordre

```
<?php
namespace App\TestServices;

class TestScalar
{
    private $myString;
    private $myInt;

    public function __construct($myString, $myInt)
    {
        $this->myString = $myString;
        $this->myInt = $myInt;
    }

    public function __toString()
    {
        return "Values : {$this->myString} - {$this->myInt}";
    }
}
```

```
services:
    # ...
    App\TestServices\TestScalar:
        arguments: ['ma chaîne de caractères', 123]
```

# Injection d'arguments scalaires à un service

- Cas où le service n'a aucun argument :

```
services:
    # ...
    App\TestServices\ServiceWithoutParams: ~
```

- Passer une constante PHP en argument :

```
App\TestServices\TestScalar:
    arguments: [!php/const App\TestServices\TestScalar::TOTO, 123]
    public: true
```

```
class TestScalar
{
    const TOTO = "la valeur de toto";
    // ...
}
```

- Passer un service en argument :

```
App\TestServices\Logger:
    arguments: ['@App\TestServices\JsonFormatter']
```


- Passer un tableau en argument :

```
App\TestServices\TestArray:
    arguments:
        -
            key1: val1
            key2: val2
            key3: val3
```

OU

```
App\TestServices\TestArray:
    arguments:
        $myArray:
            key1: val1
            key2: val2
            key3: val3
```

```
array:3 ["key1" => "val1", "key2" => "val2", "key3" => "val3"]
```





# Injection d'un service facultatif

- Syntaxe : « @? »

```
App\TestServices\Logger:  
    # Mettre le "?" après le "@" permet d'éviter une exception  
    arguments: ['@?App\TestServices\ClasseQuiNExistePas']
```

```
class Logger implements LoggerInterface  
{  
    private $formatter;  
  
    // Il faut dans ce cas mettre à null la valeur par défaut du paramètre  
    // ... et bien entendu gérer le cas où le service est null dans la classe  
    public function __construct(FormatterInterface $formatter = null)  
    {  
        $this->formatter = $formatter;  
  
        // Le code ci-dessous est très sale, mais c'est pour illustrer l'exemple  
        if (is_null($formatter)) {  
            $this->formatter = new JsonFormatter();  
        }  
    }  
    // ...  
}
```

# ID des services et alias

- Rappel : par défaut, un service est codifié en fonction du nom de classe complet
- Il est possible de définir son propre code :

```
App\TestServices\Mailer:  
  arguments:  
    $logPrefix: '%logger.prefix%'  
  
# mon_mailer est un alias du service App\TestServices\Mailer  
mon_mailer:  
  alias: App\TestServices\Mailer
```

OU

```
mon_mailer: '@App\TestServices\Mailer'
```

# Exécuter des méthodes après instanciation du service

- Possibilité d'appeler des méthodes
  - Sans paramètre
  - Avec paramètres
    - Passage de service possible

```
App\TestServices\Mailer:  
  arguments:  
    $logPrefix: '%logger.prefix%'  
  public: true  
  calls:  
    - ['myMethod']  
    - ['mySecondMethod', [123, 'toto']]  
    - ['myOtherMethod', ['@App\TestServices\Logger']]
```

```
class Mailer  
{  
    // ...  
  
    public function myMethod()  
    {  
        dump('Appelle myMethod');  
    }  
  
    public function mySecondMethod($a, $b)  
    {  
        dump("$a $b");  
    }  
  
    public function myOtherMethod(Logger $aSecondLogger)  
    {  
        dump($aSecondLogger);  
    }  
}
```



Pour aller plus loin

## Autres possibilités

- Utiliser une factory pour instancier un service
  - Permet une plus grande souplesse car l'instanciation est faite au niveau PHP et on ne se limite plus qu'aux possibilités offertes par services.yaml
  - Cf. doc : [symfony.com/doc/current/service\\_container/factories.html](https://symfony.com/doc/current/service_container/factories.html)
- Hériter d'un service parent
  - Pour par exemple factoriser du code
  - Cf. doc : [symfony.com/doc/current/service\\_container/parent\\_services.html](https://symfony.com/doc/current/service_container/parent_services.html)
- ... cf. doc, cf. doc, cf. doc... 😊

# TP – Utilisation des services



- Créer un service « anti spam »
  - Vérifie qu'une chaîne est saine et qu'elle ne contient ni « aaaaa », ni « sdfsd » (prévoir un tableau de valeurs en paramètre du service). Ceci est bien entendu un exercice...
  - Le service retourne un booléen.
  - Toute erreur devra être loguée (service « **logger** »). Loguer le message (méthode **info()** du logger) en spam ainsi que l'IP de l'utilisateur.
  - Pour accéder à l'IP : service « **request\_stack** », méthode **getCurrentRequest()**  
→ donne l'objet **Request** courant. Puis, méthode **getClientIp()**
- Pour tester :
  - Pour commencer, créer une action de test dans le contrôleur qui appelle le service
  - Puis, trouver un moyen dans les actions POST d'ajout / de modification d'un article pour appeler le service de validation (valider le champ de contenu de l'article). Si erreur, alors le formulaire ne doit pas être validé et l'utilisateur doit être redirigé vers le formulaire avec un message d'erreur sur le contenu.