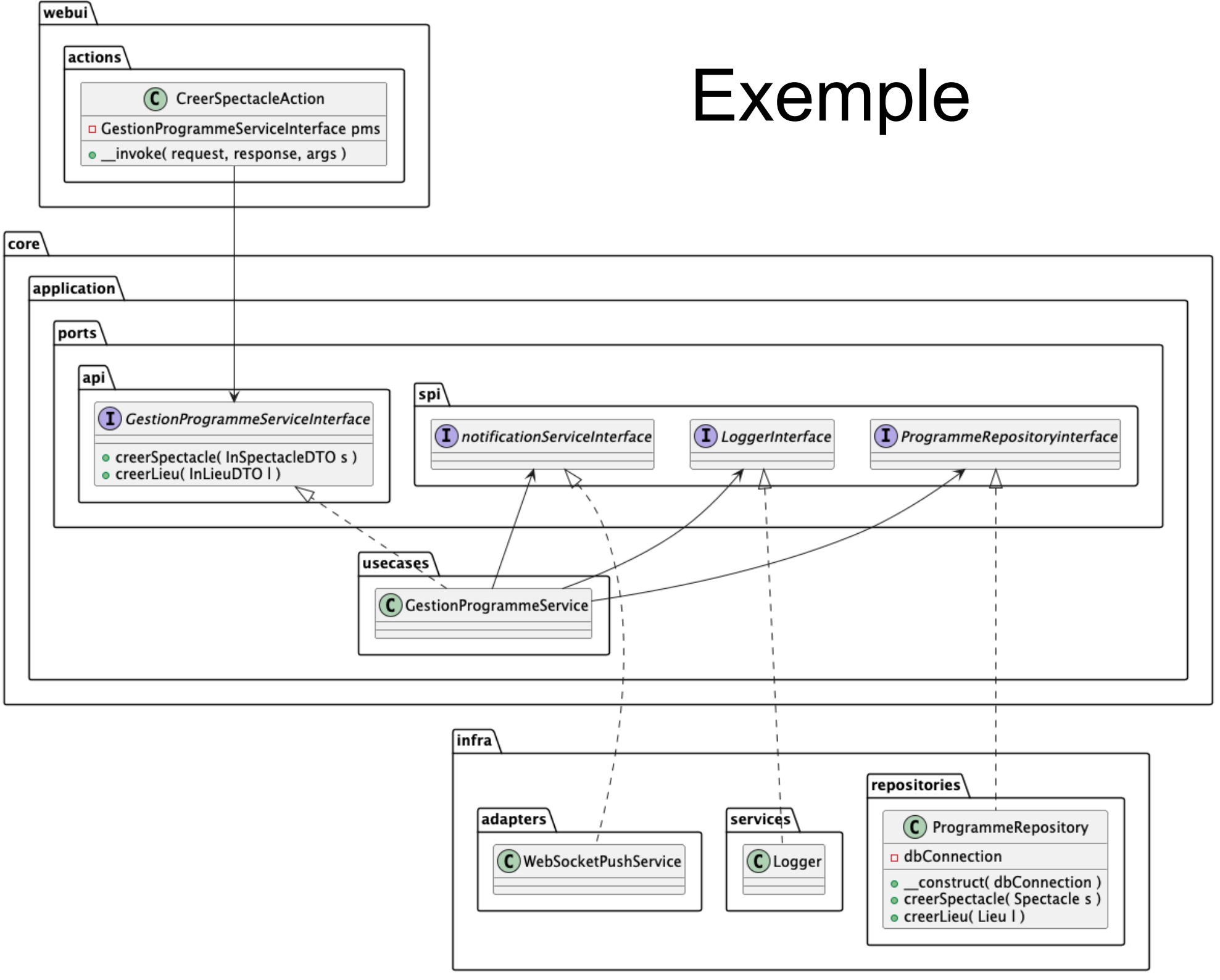
Gestion des dépendances à l'exécution
Injection et conteneur de dépendances

- Dans une application basée sur les principes d'architecture hexagonale, de nombreux composants dépendent d'autres composants au travers d'une interface
- Notamment :
  - les actions utilisent les interfaces exposées par un ou plusieurs services métier
  - un service métier utilise des services de l'infrastructure au travers d'interfaces exposées par le métier et implantées par des adaptateurs/repository

Exemple


webui
actions
CreerSpectacleAction
GestionProgrammeServiceInterface pms
__invoke(request, response, args)
core
application
ports
api
GestionProgrammeServiceInterface
creerSpectacle(InSpectacleDTO s)
creerLieu(InLieuDTO l)
spi
notificationServiceInterface
LoggerInterface
ProgrammeRepositoryinterface
usecases
GestionProgrammeService
infra
adapters
WebSocketPushService
services
Logger
repositories
ProgrammeRepository
dbConnection
__construct(dbConnection)
creerSpectacle(Spectacle s)
creerLieu(Lieu l)

A l'exécution

- L'action utilise le service de gestion du programme au travers de son interface/API *GestionProgrammeServiceInterface*
- Le service métier utilise 3 interfaces SPI implantés par des adapter fournis par l'infra : un repository, un service de log et un service de notification
- **Au moment de l'exécution, il faut créer les objets concrets !**
- En évitant de créer des dépendances supplémentaires

Besoins et objectifs

- **Créer** et **configurer** les composants
  - *Créer et configurer le logger adapté selon le service métier*
- **Choisir** l'implantation adaptée au contexte
  - Plusieurs composants concrets peuvent implanter la même interface
  - *le service de vente de ticket utilise des SMS pour notifier les confirmations aux clients et des mails pour envoyer les tickets*
- Créer uniquement les composants **nécessaires** à l'exécution de la requête courante

Exemple

```
class GestionProgrammeService
      implements GestionProgrammeServiceInterface  {

  private ProgrammeRepositoryInferface $repo;
  private Logger $logger;

  public function __construct()  {

    $this->repo = new ProgrammeRepository();
    $this->logger =  new \Monolog\Logger('njp.program.log');
    $this->logger->pushHandler(
       new \Monolog\Handler\StreamHandler(
                __DIR__ . '/log/njp.program.error.log',
                \Monolog\Level::Debug ));;
    }
 ...
}
```

- Quels problèmes avec ce code ?

Problèmes

- Il recrée une dépendance métier → infrastructure
  - en instanciant la classe ProgrammeRepository
- Le logger n'est pas configurable
  - difficile de changer l'emplacement du fichier de log ou le niveau de log
- Le logger n'est pas remplaçable
  - impossible d'utiliser autre chose que le logger Monolog

- **Que peut-on faire pour résoudre ces difficultés ?**

Injection des dépendances

1) Injecter les dépendances au travers du constructeur

2) Toujours utiliser des interfaces pour inverser les dépendances

```php
class GestionProgrammeService
      implements GestionProgrammeServiceInterface
{
  private ProgrammeRepositoryInferface $repo;

  private LoggerInterface $logger;

  public function __construct(
              ProgrammeRepositoryInferface $repo,
              LoggerInterface $logger
  ) {
    $this->repo = $repo;
    $this->logger = $logger;
  }
```

Pourquoi c’est mieux ?

- Grace aux interfaces, on a conservé les dépendances inversées
  - le service métier ne dépend pas de l’infrastructure
  - le repository et le logger sont remplaçables tant qu’ils sont conformes aux interfaces
- Le repository et le logger sont créés et configurés indépendamment du service, puis injectés dans le service
  - on peut donc créer ce service avec une configuration adaptée aux besoins

Question

- Où doit-on créer les composants injectés dans le service métier ?
- Et qui crée ce service métier ?

- L’action qui utilise ce service ?

```php
class creerSpectacleAction
{
    private GestionProgrammeServiceInterface $programmeService;

    public function __invoke( /*...*/): ResponseInterface
    {

        $data = $rq->getParsedBody() ?? null;
        $spectacleDTO = new InSpectacleDTO($data);

        $logger =  new \Monolog\Logger('njp.program.log');
        $logger->pushHandler(
            new \Monolog\Handler\StreamHandler(
                __DIR__ . '/log/njp.program.error.log',
                \Monolog\Level::Debug ));

        $this->programmeService =
                    new GestionProgrammeService(new ProgrammeRepository(), $logger);
        $this->programManagementService->createSpectacle($spectacleDTO);

        return $rs;

}
```

- On a simplement déplacé et recréé le problème
  - dépendance vers un service métier concret
  - ce service n’est pas configurable et adaptable
  - création d’une dépendance application → infrastructure

- Il faut appliquer le principe d'injection de dépendances à l'action :

```php
class creerSpectacleAction
{
    private GestionProgrammeServiceInterface $programmeService;

     public function __construct(
           GestionProgrammeServiceInterface $service)
    {
        $this->programmeService = $service;
    }

    public function __invoke( /*...*/): ResponseInterface
    {

        $data = $rq->getParsedBody() ?? null;
        $spectacleDTO = new InSpectacleDTO($data);
        $this->programmeService->creerSpectacle($spectacleDTO);

        return $rs;

}
```

- On injecte le service métier adéquat, configuré et adapté aux besoins dans l'action

- Il faut cependant créer ce service de façon adaptée
- Et créer l'action en lui injectant le service nécessaire

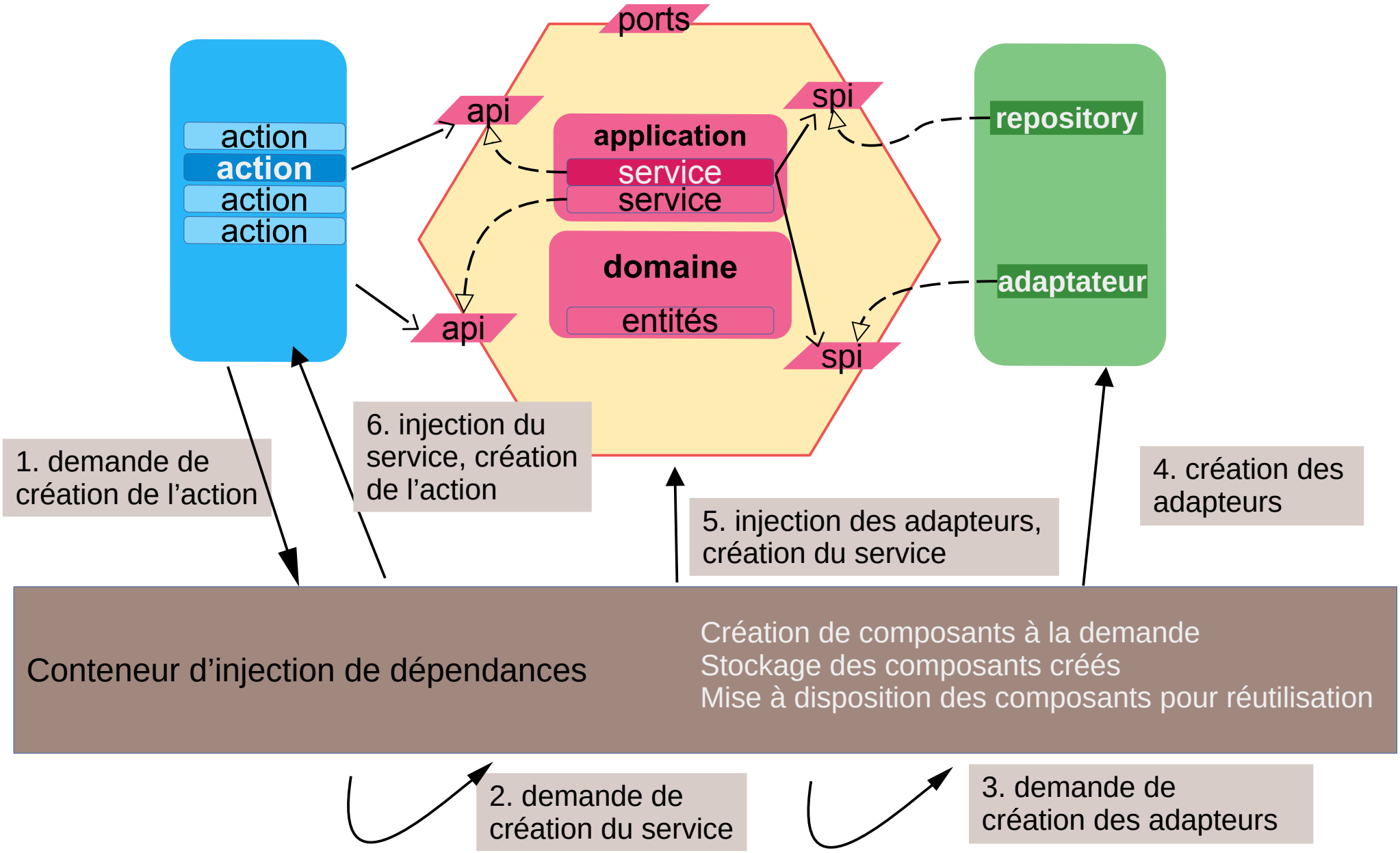
Il faut créer et configurer les composants au démarrage de l'application

Principe : on utilise un composant supplémentaire dans l'architecture, dédié à la création d'objets et de composants configurés : un conteneur d'injection de dépendances

En pratique

- un **conteneur d'injection de dépendances** est un composant registre dans lequel on peut stocker et rendre accessibles :
  - des **variables de configuration**
  - des **fabriques de composants** (services, actions ...)
- Ce conteneur est créé et rempli lors du démarrage de l'application
- Il permet de récupérer des composants configurés et de les injecter dans les constructeurs de façon paresseuse, c'est à dire uniquement en cas de besoin

Le conteneur d'injection de dépendances



conteneurs d'injection de dépendances en PHP

- En PHP, les conteneurs d'injection de dépendances sont spécifiés dans PSR-11
- Il en existe de nombreux :
  - Symfony DI, Service Container (laravel), **PHP-DI**
- Slim peut utiliser tout conteneur d'injection conforme à PSR-11
- Lorsqu'une application slim est créée avec un conteneur, ce conteneur est disponible :
  - dans l'application,
  - dans les fonctions de route
  - dans les middleware

PHP-DI, un conteneur PSR-7

- PHP-DI permet d'enregistrer des données de différentes sortes (variables, fabriques) sous la forme de **définitions PHP**
  - PHP-DI peut aussi utiliser des attributs ou des définitions implicites (autowiring)
- On crée le conteneur, on ajoute des définitions, puis on l'injecte dans l'application Slim

bootstrap.php

```php
$builder = new ContainerBuilder();
$builder->useAutowiring(false);
$builder->addDefinitions(__DIR__ . '/DI_definitions.php');

$c=$builder->build();
$app = AppFactory::createFromContainer($c);
```

Définitions PHP-DI

- On peut enregistrer des **valeurs** : strings, int … associées à une **clé**
- En général pour définir des variables de configuration de l'application

DI_definitions.php

```php
return  [
  'displayErrorDetails' => true ,
  'db.config' => __DIR__ . '/conf.db.ini',
  'logs.dir'  => __DIR__ . '/../logs'

 ] ;
```

Définitions PHP-DI

- PHP-DI permet d'enregistrer des *Factory* associées à des clés
  - une *factory* est un *callable* (fonction, closure, méthode) chargé de fabriquer une instance (d'un composant)
  - la *factory* est exécutée uniquement au moment de l'accès à la clé correspondante
  - elle reçoit le conteneur en paramètre, ce qui permet d'utiliser des clés enregistrées dans la *factory*

DI_definitions.php

```php
return [
    'db.host' => 'sql4563.db' ,
    'Foo' => function (ContainerInterface $c) {
        return new Foo($c->get('db.host'));
    }
] ;
```

Créer des composants configurés et configurables

- Enregistrer une factory pour créer et configurer un composant en utilisant des valeurs stockées dans le conteneur

DI_definitions.php

```php
return [

  'log.prog.name' => 'prog.log',
  'log.prog.file' => __DIR__ . '/../log/program.log',
  'log.prog.level' => \Monolog\Logger::WARNING,

  'prog.logger' => function( ContainerInterface $c) {
     $log = new \Monolog\Logger( $c->get('log.prog.name'));
     $log->pushHandler(
         new StreamHandler($c->get('log.prog.file'),
                           $c->get('log.prog.level')));
        return $log;
    }
] ;
```

Injecter des composants dans d'autres composants

- Exemple : le service d'interrogation du programme utilise un logger

DI_definitions.php

```php
return [
    'log.prog.name' => 'njp.program.log',
    'log.prog.file' => __DIR__ . '/log/njp.program.error.log',
    'log.prog.level' => \Monolog\Level::Debug,

    'prog.logger' => function(ContainerInterface $c) {
        $logger = new \Monolog\Logger($c->get('log.prog.name'));
        $logger->pushHandler(
            new \Monolog\Handler\StreamHandler(
                $c->get('log.prog.file'),
                $c->get('log.prog.level')));
        return $logger;
    },

    'program.service' => function(ContainerInterface $c) {
        return new ProgrammeService( $c->get('prog.logger') );
    },
    'ticket.service' => function(ContainerInterface $c) {
        return new TicketeService( $c->get('prog.logger') );
    }
] ;
```

Injection du logger récupéré dans le conteneur

En pratique

- On utilise les noms des classes comme clés dans le conteneur
- Quand il n'y a pas d'ambiguïté pour implanter une interface, on utilise le nom de l'interface
- Lorsqu'on a plusieurs implantations concrètes d'une même interface, on utilise des clés spécifiques
- **Attention** : utiliser les noms qualifiés avec les namespaces

```php
return [
 'log.prog.name'     => 'njp.program.log',
 'log.prog.file'     => __DIR__ . '/log/njp.program.error.log',
 'log.prog.level'    => \Monolog\Level::Debug,


// Plusieurs Logger et plusieurs PDO : clés spécifiques

 'prog.logger' => function(ContainerInterface $c) {
    …
 },

 'program.db.pdo' => function(ContainerInterface $c) {
     $config = parse_ini_file($c->get('program.db.config'));
     return new \PDO($_ENV['dsn'], $_ENV['username'], $_ENV['password']);
 },

// Un seul ProgrammeRepository et un seul GestionProgrammeService

 ProgrammeRepositoryInterface::class => function(ContainerInterface $c) {
     return new ProgrammeRepository($c->get('program.db.pdo'));
 },

 GestionProgrammetServiceInterface::class => function(ContainerInterface $c) {
     return new GestionProgrammeService($c->get(ProgrammeRepositoryInterface::class),
                                        $c->get('prog.logger'));
 },

 creerSpectacleAction::class => function(ContainerInterface $c) {
     return new creerSpectacleAction
                 $c->get(GestionProgrammetServiceInterface::class));

}
] ;
```

Création des actions dans les routes

- Le framework interroge le conteneur pour instancier l'action dans une définition de route :

```php
return function( \Slim\App $app ):void {
 $app->post('/spectacles', creerSpectacleAction::class);
} ;
```

- Toutes les créations et les injections de composants nécessaires sont gérées dans le conteneur
  - services métiers utilisés par l'action
  - services, repository et adaptateurs de l'infra utilisés par les services métier

Slim et PHP-DI

- Installer

```
{
  "require" : {
    "slim/psr7": "^1.6",
    "php-di/php-di": "^7.0",
    "php-di/slim-bridge": "^3.4",
  },
  "autoload": { … }
}
```

```
$builder = new ContainerBuilder();
$builder->useAutowiring(false);
$builder->addDefinitions(__DIR__ . '/settings.php');
$builder->addDefinitions(__DIR__ . '/services.php');
$builder->addDefinitions(__DIR__ . '/actions.php');

$c=$builder->build();
$app = AppFactory::createFromContainer($c);
```

Séparer les définitions lorsqu'elles sont nombreuses