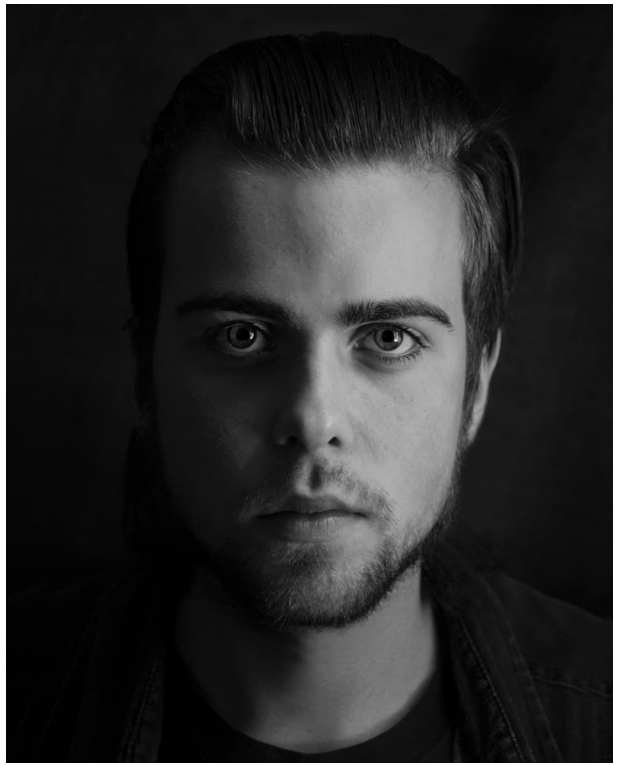
Веб разработка на PHP Symfony

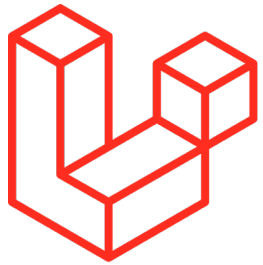
Сервис контейнер

Introduction



Демьян Костельный
Middle PHP Developer

demian-kostelny-613b90151



Тема урока

Сервис контейнер

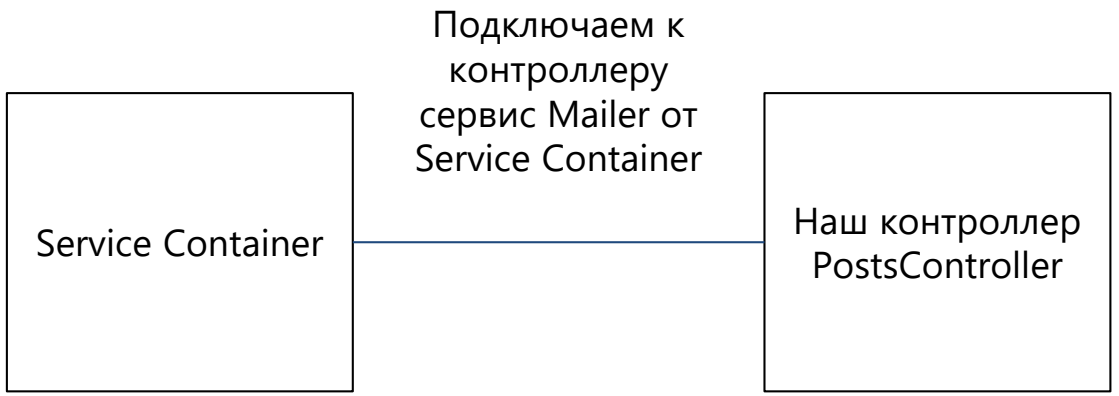
План урока

1. Service container, DI, DIP, IoC
2. Autowiring
3. Биндинг параметров
4. Тегирование сервисов
5. Compiler pass

Service container, DI, DIP, IoC

Service container, DI, DIP, IoC

В любом Symfony приложении есть такая вещь, как Service Container. Это то, что может предоставить возможность подключить какое-то дополнительное расширение функционала - например класс Дебагера, который можно использовать в нашем контроллере.



Service container, DI, DIP, IoC

Пример вызова Service Container для использования класса логера для контроллера ProductController, взятый из официальной документации:

```php
namespace App\Controller;

use Psr\Log\LoggerInterface; // Подключаем класс из Service Container - логер
use Symfony\Component\HttpFoundation\Response;

class ProductController
{
    /**
     * @Route("/products")
     */
    public function list(LoggerInterface $logger): Response
    {
        // Для того чтобы мы могли использовать логгер от Service Container
        // мы создали новый объект класса логгера в переменной $logger

        $logger->info('Мы воспользовались сервисом от Service Container');

    }

}
```

Service container, DI, DIP, IoC

Здесь есть список того, какие сервисы существуют в Service Container, которые можно подключать в свои контейнеры.

ID сервиса от Service Container	**Имя класса**
doctrine	Doctine\Bundle\DoctrineBundle\Registry
filesystem	Symfony\Component\Filesystem\Filesystem
form.factory	Symfony\Component\Form\FormFactory
logger	Symfony\Bridge\Monolog\Logger
request_stack	Symfony\Component\HttpFoundation\RequestStack
router	Symfony\Bundle\FrameworkBundle\Routing\Router
session	Symfony\Component\HttpFoundation\Session\Session
translator	Symfony\Component\Translation\DataCollectorTranslator

Service container, DI, DIP, IoC

Можно также и создать новый сервис в своем приложении. Простой пример. Для этого создадим новый файл RandGenerator.php по адресу src/Service/. Данный сервис с помощью своей функции new_number() будет выдавать случайное число:

```php
namespace App\Service;

class RandGenerator
{
    public function new_number(int $from, int $to): string
    {
        return rand($from, $to);
    }
}
```

Service container, DI, DIP, IoC

После создания нового сервиса, можем им воспользоваться в нашем контроллере NumberController:

```php
use App\Service\RandGenerator;
use Symfony\Component\HttpFoundation\Response;

class NumberController
{

    /**
     * @Route("/generate/number")
     */
    public function newNumber(RandGenerator $rand_generator): Response
    {
        $result = $rand_generator->new_number(1, 15);
        echo $result;
    }

}
```

Service container, DI, DIP, IoC

DI - Dependency Injection, продолжаем дальше говорить об внедрении зависимостей, и следующий компонент, который нам может помочь с этим, является DI, который можно внедрить в своё Symfony приложение.

Установить данный компонент можно с помощью следующей команды через Composer:

```
composer require symfony/dependency-injection
```

Service container, DI, DIP, IoC

Чтобы понять как работает DI давайте разберём небольшой пример. Представим что у нас будет класс для отправки почты, который будет называться MyMailer:

```php
namespace App\Service;

class MyMailer
{

    private $transport;

    public function __construct()
    {
        $this->transport = 'sendmail';
    }

}
```

Service container, DI, DIP, IoC

Вместо того, чтобы заново как-то создавать класс и подключать его к нашему контроллеру, мы просто можем воспользоваться ContainerBuilder:

```php
use Symfony\Component\DependencyInjection\ContainerBuilder;
// Выше подключаем ContainerBuilder чтобы воспользоваться им
// внутри функций нашего контроллера



$containerBuilder = new ContainerBuilder();
// Дальше внутри любой функции класса можем создать новый объект
// для класса ContainerBuilder чтобы в дальнейшем зарегистрировать
// новый сервис на основе класса MyMailer

$containerBuilder->register('mailer', 'MyMailer');
// С помощью функции register регистрируем новый сервис
// который будем называть как и наш класс MyMailer.
// Первый аргумент функции это ID сервиса, второй - название уже самого сервиса
```

Service container, DI, DIP, IoC

В коде нашего класса MyMailer был параметр transport, поэтому давайте и его добавим в наш сервис созданный через ContainerBuilder:

```
$containerBuilder = new ContainerBuilder();

$containerBuilder
    ->register('mailer', 'MyMailer')
    ->addArgument('sendmail');
// Добавили аргумент мы с помощью функции addArgument() - внутри неё достаточно было
// просто указать название параметра в самом классе
```

Service container, DI, DIP, IoC

Если нам нужно будет достать наш сервис, добавленный через ContainerBuilder, то это можно будет сделать следующим способом:

```php
$containerBuilder = new ContainerBuilder();

$containerBuilder
    ->register('mailer', 'MyMailer')
    ->addArgument('sendmail');



$MyMailer = $containerBuilder->get('MyMailer');
// Чтобы достать созданный нами сервис нужно обратиться к ContainerBuilder
// и воспользоваться функцией get() с названием нашего сервиса
```

DIP (Dependency Inversion Principle)

IoC (Inversion of Control)

Autowiring

Autowiring

Autowriting или Автоматическое определение зависимостей сервиса - представляет из себя возможность управлять сервисами, которые есть в контейнере, с минимальными затратами на саму конфигурацию сервисов. В официальной документации есть хороший пример с двумя классами для Твиттер модуля.

Autowiring

Настроить автоматическое определение зависимостей, можно через конфигурацию файла config/services.yaml

```yaml
services:
  _defaults:
    autowire: true
    autoconfigure: true

  App\Service\TwitterClient:
    autowire: true

  App\Util\Rot13Transformer:
    autowire: true
```

Autowiring

После того как включили автоматическое определение зависимостей, можно спокойно воспользоваться нашим сервисом в контроллере.

```php
/**
 * @Route("/tweet", methods={"POST"})
 */
public function tweet(TwitterClient $twitterClient, Request $request): Response
{

    // В данном случае для класса TwitterClient все параметры будут определяться автоматически
    $twitterClient->tweet($user, $key, $status);

}
```

Биндинг параметров

Биндинг параметров

Можно добавлять в сервисы параметры. Вместо того, чтобы постоянно объявлять их - можно создать параметры, которые будут иметь постоянное значение. Пример создания параметров в файле config/services.yaml

```yaml
parameters:
    $superMail: 'cool@mail' # Здесь мы создали параметр с названием $superMail

    Psr\Log\LoggerInterface: '@monolog.logger.request' # А здесь уже создан параметр только для сервиса LoggerInterface

    string $myMail: 'string@email.com' # Кроме этого можно задавать тип данных для созданных параметров
```

Биндинг параметров

Чтобы проверить были ли все аргументы внедрены в наши сервисы, нужно воспользоваться командой lint:container

```
C:\Dev\my_project_name>php bin/console lint:container

 [OK] The container was lint successfully: all services are injected with
      values that are compatible with their type declarations.
```

Биндинг параметров

Сервисы можно делать публичными либо приватными, делается это через файл конфигурации config/services.yaml:

```yaml
services:

  App\Services\PublicServer:
    public: true # Таким образом сервис становится  публичным если оставить значение true
```

Биндинг параметров

Дальше можно увидеть функции, которые позволяют работать с параметрами в PHP коде:

```
// Функция hasParameter проверяет есть ли параметр с указанным внутри функции ID
$container->hasParameter('mailer.transport');

// Функцию getParameter как ясно из названия    нужно использовать для того чтобы
получить значение какого-то параметра по его ID
$container->getParameter('mailer.transport');

// Функция setParameter добавляет новый параметр либо просто изменяет значение
уже существующего с таким же ID который указан внутри функции
$container->setParameter('mailer.transport', 'sendmail');
```

Биндинг параметров

Кроме того, что параметры могут быть строками, также они могут быть и массивами, пример в YAML:

```yaml
parameters:

  my_mailer.mails: # Можем увидеть что параметр my_mailer.mails мы превратили в массив
    - mail1
    - mail2
    - mail3
```

Биндинг параметров

Если понадобится чтобы параметры имели постоянное значение, которое никогда не меняется, - в таком случае нужно создавать параметр в качестве константы:

```
// В примере показанном ниже, мы создаём полностью голобальную константу с ID global.constant.value
// и значение от GLOBAL_CONSTANT - важно также заметить что перед этим нужно создать саму константу
// с помощью функции define()
$container->setParameter('global.constant.value', GLOBAL_CONSTANT);

// Во втором же примере от класса MyClass создаётся константа со значением MyClass::CONSTANT_NAME
$container->setParameter('myclass.constant.value', MyClass::CONSTANT_NAME);
```

Тегирование сервисов

Тегирование сервисов

Тегирование сервисов позволяет давать инструкции по сервисам для бандлов, и не только.
Пример тегирования сервиса в файле config/services.yaml:

```yaml
services:
  App\Twig\AppExtension: # Класс AppExtension является примером класса-расширения  для Twig
    tags: ['twig.extension']
```

Тегирование сервисов

Посмотреть список доступных всех тегов можно по адресу:

https://symfony.com/doc/current/reference/dic_tags.html

Тегирование сервисов

Кроме ручного ввода тегов для сервисов, можно включить автоконфигурацию тегов в приложении. В таком случае Symfony framework автоматически будет добавлять нужные теги в сервис - распознавая по параметрам класса. Делается включение автоматической конфигурации через файл `config/services.yaml` в параметре `_instanceof:`

```yaml
services:

  _instanceof:

    App\Twig\CustomExtension: # В данном случае к сервису CustomExtesion теги будут конфигурироваться автоматически
      tags: ['app.custom_tag']
```

Тегирование сервисов

Можно добавлять также и собственные теги, например добавив тег app.mail_transport от класса `Swift_SmtpTransport` к классу `Swift_SendmailTransport`, мы сможем воспользоваться функцией класса `TransportChain - addTransport()`:

```
services:
    Swift_SmtpTransport:
      arguments: ['%mailer_host%']
      tags: ['app.mail_transport']

    Swift_SendmailTransport:
      tags: ['app.mail_transport']
```

Compiler Pass

Compiler Pass

Вспомним о custom tags (собственных тегах) и, теперь, если мы хотим полноценно манипулировать дефинициями сервисов - то мы можем воспользоваться таким компонентом как Compiler Pass.

В следующем примере мы создадим Compiler Pass для класса MailTransportPass чтобы контейнер мог спрашивать любые другие сервисы с тегом app.mail_transport.

Compiler Pass

Чтобы наш compiler pass запускался когда контейнер будет скомпилирован, нам нужно зарегистрировать его в файле конфигурации `src/Kernel.php` внутри защищённой функции `build()`:

```php
namespace App;

use App\DependencyInjection\Compiler\MailTransportPass;
use Symfony\Component\DependencyInjection\ContainerBuilder;
use Symfony\Component\HttpKernel\Kernel as BaseKernel;
// ...

class Kernel
{
    // ...

    protected function build(ContainerBuiler $container): void
    {
        $container->addCompilerPass(new MailTransportPass());
    }
}
```

Информационный видеосервис для разработчиков программного обеспечения

TestProvider Pearson CyberBionic systematics Microsoft Partner Windows Azure EBA

Проверка знаний

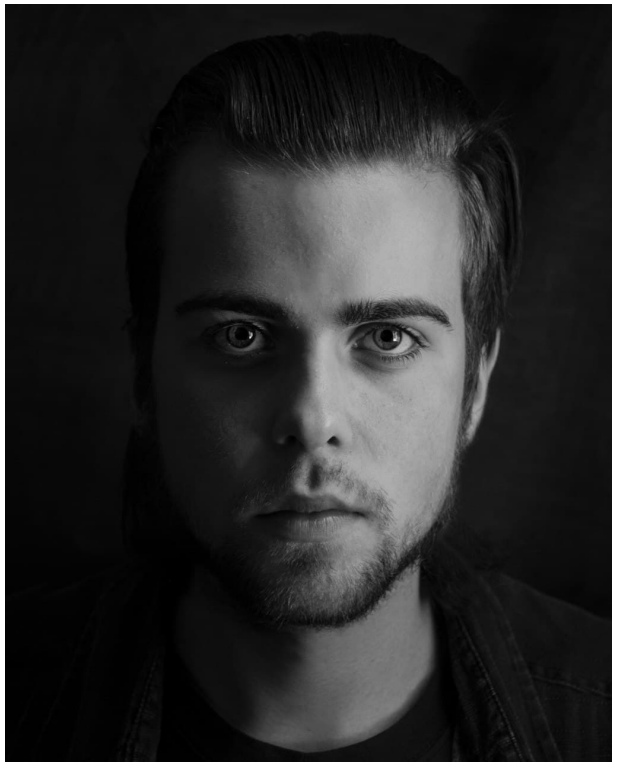
TestProvider.com

TestProvider – это online сервис проверки знаний по информационным технологиям. С его помощью Вы можете оценить Ваш уровень и выявить слабые места. Он будет полезен как в процессе изучения технологии, так и для общей оценки знаний IT специалиста.

Проверьте как Вы усвоили данный материал на TestProvider.com

Успешное прохождение финального тестирования позволит Вам получить соответствующий Сертификат.

Спасибо за внимание! До новых встреч!



Демьян Костельный
Middle PHP Developer