Course - Laravel Framework

Service Container

Service Container là một công cụ quản lý các thư viện class và thực hiện kỹ thuật Dependency Injection.

Tags: Service Container

Service Container là một công cụ quản lý các thư viện class và thực hiện kỹ thuật *Dependency Injection*. *Dependency Injection* là cụm từ kỹ thuật có nghĩa là các thư viện class sẽ được "tiêm" vào trong một class qua constructor của class đó hoặc trong một vài trường hợp có thể là phương thức "setter".

Hãy xem qua ví dụ sau đây.

```php
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Repositories\UserRepository;
use App\Models\User;

class UserController extends Controller
{
    /**
     * The user repository implementation.
     *
     * @var UserRepository
     */
    protected $users;

    /**
     * Create a new controller instance.
     *
     * @param  UserRepository  $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        $this->users = $users;
    }

    /**
     * Show the profile for the given user.
     *
```

```
     * @param  int  $id
     * @return Response
     */
    public function show($id)
    {
        $user = $this->users->find($id);

        return view('user.profile', ['user' => $user]);
    }
}
```

Trong ví dụ này `UserController` cần nguồn dữ liệu người dùng, vì thế chúng ta sẽ tiêm một service mà có thể nhận được thông tin người dùng. Trong đó `UserRepository` gần như sử dụng *Eloquent* để nhận thông tin người dùng từ cơ sở dữ liệu. Nhưng với phương thức tiêm này chúng ta sẽ dễ dàng tiếp cận các hoạt động khác, chúng ta có thể "mock" hay tạo ra nguồn dữ liệu nhái để test ứng dụng của chúng ta.

Hiểu về service container trong Laravel là điều cần thiết trong việc build một ứng dụng lớn, cũng như đóng góp rất nhiều vào trong chính lõi của framework Laravel.

Có nhiều class khi viết ứng dụng Laravel hỗ trợ việc tự động nhận các thư viện qua các container, như controller, middleware, eventlisterner, vân vân.

Khi nào dùng container

Nhiệm vụ chính của container là quản lý các thư viện class được tiêm vào ứng dụng.

Ví dụ bạn cần tiêm `Illuminate\Http\Request` vào route để dễ dàng truy cập vào request hiện tại, chúng ta chưa bao giờ động vào container này khi viết code, nhưng ở đằng sau nó đã tiêm các thư viện này.

```
use Illuminate\Http\Request;

Route::get('/', function (Request $request) {
    // ...
});
```

Trước hết, nếu bạn viết ra một class để thực thi một interface nào đó, và bạn muốn type-hint

(gợi ý kiểu) interface đó trên route hay constructor của class, bạn phải báo container cách để giải quyết interface đó. Thứ hai, là nếu bạn đang viết một package Laravel với ý định chia sẽ cho các developer Laravel khác, thì bạn có thể sẽ cần bind các service của package vào container.

Ràng buộc đơn giản

Hầu hết các ràng buộc service container đều được đăng ký với serivce provider. Với một service provider, chúng ta sẽ luôn luôn phải truy cập vào container qua thuộc tính `$this->app`. Chúng ta có thể đăng ký một ràng buộc bằng cách sử dụng phương thức `bind`, truyền tên class hay interface mà chúng ta muốn đăng ký vào block mà sẽ trả lại instance của class.

```
use App\Services\Transistor;
use App\Services\PodcastParser;

$this->app->bind(Transistor::class, function ($app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

Chúng ta nhận container từ resolver, và lại dùng container để resolve các thư viện con của đối tượng mà chúng ta đang build.

Như đã đề cập, chúng ta thường tương tác container với service provider, nhưng nếu bạn muốn tương tác container bên ngoài service provider thì bạn có thể thực hiện qua facade `App`.

```
use App\Services\Transistor;
use Illuminate\Support\Facades\App;

App::bind(Transistor::class, function ($app) {
    // ...
});
```

Không cần phải bind các class vào trong container, nếu như chúng không có bất kỳ interface nào, Container không cần phải được chỉ thị về cách build các đối tượng này, vì nó có thể tự động resolve các đối tượng này bằng reflection.

Ràng buộc một Singleton

Phương thức `singleton` sẽ gắn một class hay interface vào container mà chỉ resolve một lần. Một khi ràng buộc được resolve đối tượng được trả lại sẽ được gọi vào container.

```
use App\Services\Transistor;
use App\Services\PodcastParser;

$this->app->singleton(Transistor::class, function ($app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

Phương thức `scoped` sẽ gắn một class hay interface vào container mà chỉ resolve một lần với vòng đời của một request hay job nào đó, các instance được đăng ký bằng phương thức `scoped` sẽ được rút đi mỗi khi ứng dụng Laravel bắt đầu một vòng đời mới, chẳng hạn như lúc *Laravel Octane* xử lý request mới hoặc khi một *queue workder* xử lý một job mới.

```
use App\Services\Transistor;
use App\Services\PodcastParser;

$this->app->scoped(Transistor::class, function ($app) {
    return new Transistor($app->make(PodcastParser::class));
});
```

Ràng buộc một instance

Bạn có thể ràng buộc một instance đã tồn tại vào container bằng cách sử dụng phương thức instance. Instance đó sẽ luôn được gọi vào trong container.

```
use App\Services\Transistor;
use App\Services\PodcastParser;

$service = new Transistor(new PodcastParser);

$this->app->instance(Transistor::class, $service);
```

Ràng buộc interface

Một tính năng rất ưu việt của service container đó chính là ràng buộc một interface vào một bản thực thi nào đó. Ví dụ, ta có một interface là EventPusher và một thứ đại loại class RedisEventPusher. Một khi chúng ta đã tạo RedisEventPusher từ event EventPusher, thì chúng ta có thể đăng ký nó vào service container như thế này:

```
use App\Contracts\EventPusher;
use App\Services\RedisEventPusher;

$this->app->bind(EventPusher::class, RedisEventPusher::class);
```

Phát biểu này nói với container sẽ tiêm `RedisEventPusher` khi một class cần một bản thực thi cho interface `EventPusher`. Bây giờ chúng ta dùng các gợi ý (type-hint) về interface `EventPusher` trong constructor của một class được resolve bởi container. Hãy ghi nhớ, controller, event listener, middleware và các kiểu class khác trong một ứng dụng Laravel luôn được resolve bởi container.

```
use App\Contracts\EventPusher;

/**
 * Create a new class instance.
 *
 * @param  \App\Contracts\EventPusher  $pusher
 * @return void
 */
public function __construct(EventPusher $pusher)
{
    $this->pusher = $pusher;
}
```

Ràng buộc theo ngữ cảnh

Thi thoảng bạn sẽ có hai class cùng thực thi một interface, nhưng lại muốn tiêm sự thực thi khác nhau cho từng class tùy theo hoàn cảnh của mỗi bản thực thi. Ví dụ,

```php
use App\Http\Controllers\PhotoController;
use App\Http\Controllers\UploadController;
use App\Http\Controllers\VideoController;
use Illuminate\Contracts\Filesystem\Filesystem;
use Illuminate\Support\Facades\Storage;

$this->app->when(PhotoController::class)
->needs(Filesystem::class)
->give(function () {
  return Storage::disk('local');
});

$this->app->when([VideoController::class, UploadController::class])
->needs(Filesystem::class)
->give(function () {
  return Storage::disk('s3');
});
```

Trong ví dụ trên ta có bản thực thi `PhotoController`, và `VideoController` đều thực thi interface `Filesystem`, nhưng `PhotoController` sẽ thực thi việc lưu trữ khác và `VideoController` thực thi việc lưu trữ khác.

Ràng buộc các kiểu dữ liệu thuần

Thi thoảng bạn có thể có một class mà nhận vài class đã tiêm, nhưng nó vẫn cần được tiêm thêm các giá trị kiểu thuần như một số nguyên. Bạn có thể dễ dàng ứng dụng ràng buộc theo ngữ cảnh để tiêm bất kỳ giá trị nào mà class của bạn cần.

```php
$this->app->when('App\Http\Controllers\UserController')
->needs('$variableName')
->give($value);
```

Thỉnh thoảng một class có thể sử dụng một nhóm các giá trị tag. Bằng việc sử dụng phương thức `giveTagged`, bạn có thể dễ dàng tiêm tất cả các container đã ràng buộc với giá trị tag đó.

```php
$this->app->when(ReportAggregator::class)
->needs('$reports')
->giveTagged('reports');
```

Nếu bạn đang cần tiêm một giá trị từ các tập tin cấu hình của ứng dụng, bạn có thể sử dụng phương thức `giveConfig` như sau,

```php
$this->app->when(ReportAggregator::class)
->needs('$timezone')
->giveConfig('app.timezone');
```

Ràng buộc đa hình

Đôi khi, một class sẽ nhận một mảng các đối tượng bằng tham số constructor của class, ví dụ

```php
<?php

use App\Models\Filter;
use App\Services\Logger;

class Firewall
{
    /**
     * The logger instance.
     *
     * @var \App\Services\Logger
     */
    protected $logger;

    /**
     * The filter instances.
     *
     * @var array
     */
```

```
    protected $filters;

    /**
     * Create a new class instance.
     *
     * @param  \App\Services\Logger  $logger
     * @param  array  $filters
     * @return void
     */
    public function __construct(Logger $logger, Filter ...$filters)
    {
        $this->logger = $logger;
        $this->filters = $filters;
    }
}
```

Theo phương pháp ràng buộc theo ngữ cảnh, bạn có thể resolve các thư viện bằng việc cung cấp cho phương thức `give` một function có trả lại một mảng các đối tượng `Filter` đã resolve. Như sau,

```
$this->app->when(Firewall::class)
->needs(Filter::class)
->give(function ($app) {
  return [
    $app->make(NullFilter::class),
    $app->make(ProfanityFilter::class),
    $app->make(TooLongFilter::class),
  ];
});
```

Có một phương pháp khác tiện lợi hơn để resolve các thư viện này, đó là bạn có thể cung cấp một mảng chứa tên của các class mà sẽ được resolve bởi container mỗi khi `Firewall` cần các instance `Filter`.

```
$this->app->when(Firewall::class)
->needs(Filter::class)
->give([
```

```
    NullFilter::class,
    ProfanityFilter::class,
    TooLongFilter::class,
]);
```

Trong trường hợp, một class sử dụng nhiều thư viện nhưng được gợi ý qua một class nhất định `(Report ...$report)`. Với phương thức `need` và phương thức `giveTagged`, bạn có thể dễ dàng tiêm tất cả các ràng buộc container đã được tag.

```
$this->app->when(ReportAggregator::class)
->needs(Report::class)
->giveTagged('reports');
```

Tạo Tag

Đôi khi bạn có thể sẽ muốn resolve các ràng buộc mà đã phân loại trong một category nào đó. Ví dụ, bạn đang build một chương trình phân tích báo cáo, trong đó có nhiều bản thực thi interface `Report` khác nhau. Sau khi đăng ký các bản thực thi này, bạn có thể đưa chúng vào trong một tag để sau này tiêm vào.

```
$this->app->bind(CpuReport::class, function () {
    //
});

$this->app->bind(MemoryReport::class, function () {
    //
});

$this->app->tag([CpuReport::class, MemoryReport::class], 'reports');
```

Bạn có thể tiêm các service trên bằng phương thức `tagged` của container.

```
$this->app->bind(ReportAnalyzer::class, function ($app) {
    return new ReportAnalyzer($app->tagged('reports'));
});
```

Mở rộng các ràng buộc

Đối với một service đã resolve rồi, nhưng bạn vẫn muốn thay đổi cấu hình hay chạy thêm một đoạn code nào đó, thì hãy dùng phương thức `extend`, ví dụ

```
$this->app->extend(Service::class, function ($service, $app) {
    return new DecoratedService($service);
});
```

Đoạn code trên, tham số thứ nhất chính là cái service sẽ được sửa lại, tham số thứ hai là hàm dùng để chỉnh sửa service, hàm này sẽ nhận service đang cần được sửa chữa `$service`, và container `$app`.

Tìm hiểu Resolve

Phương thức `make`

Bạn dùng phương thức `make` để nhận lấy một đối tượng từ container thông qua tên class của đối tượng đó.

```
use App\Services\Transistor;

$transistor = $this->app->make(Transistor::class);
```

Trong trường hợp, thư viện cần tham số để resolve thì dùng phương thức `makeWith`, ví dụ

```
use App\Services\Transistor;

$transistor = $this->app->makeWith(Transistor::class, ['id' => 1]);
```

Trong ví dụ trên, service `Transistor` cần tham số `id`, bạn truyền tham số bằng tay với một mảng `['id' => 1]`.

Nếu bạn không thể truy cập vào biến `$app` được vì đang bên ngoài service provider, thì dùng facade `App`.

```php
use App\Services\Transistor;
use Illuminate\Support\Facades\App;

$transistor = App::make(Transistor::class);
```

Trong ví dụ trên, bạn nhận lấy service `Transistor` từ container khi đang ở bên ngoài service provider.

Nếu bạn muốn có container bên trong class đang được resolve thì dùng class `Illuminate\Container\Container` trong tham số constructor của class đó.

```php
use Illuminate\Container\Container;

/**
 * Create a new class instance.
 *
 * @param  \Illuminate\Container\Container  $container
 * @return void
 */
public function __construct(Container $container)
{
    $this->container = $container;
}
```

Tiêm tự động

Trong thực tế đây là cách thường dùng để nhận lấy các service từ container, bạn có thể dùng gợi ý kiểu trên các parameter constructor, hay trên phương thức `handle` của các queued jobs.

```php
<?php

namespace App\Http\Controllers;

use App\Repositories\UserRepository;
```

```php
class UserController extends Controller
{
    /**
     * The user repository instance.
     *
     * @var \App\Repositories\UserRepository
     */
    protected $users;

    /**
     * Create a new controller instance.
     *
     * @param  \App\Repositories\UserRepository  $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        $this->users = $users;
    }

    /**
     * Show the user with the given ID.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function show($id)
    {
        //
    }
}
```

Trong ví dụ trên, `UserRepository` sẽ được tự động resolve và tiêm vào trong class `UserController`.

Gọi phương thức và tiêm

Trong trường hợp bạn muốn tiêm một service vào một phương thức của một class nào đó, thì dùng phương thức `call` của facade `App`. Ví dụ, bạn có class `UserReport` chứa phương thức `generate` muốn tiêm `UserRepository` vào trong phương thức đó.

```php
<?php

namespace App;

use App\Repositories\UserRepository;

class UserReport
{
    /**
     * Generate a new user report.
     *
     * @param  \App\Repositories\UserRepository  $repository
     * @return array
     */
    public function generate(UserRepository $repository)
    {
        // ...
    }
}
```

Và bây giờ, ta sẽ gọi phương thức đó như sau,

```php
use App\UserReport;
use Illuminate\Support\Facades\App;

$report = App::call([new UserReport, 'generate']);
```

Ngoài ra, phương thức `call` cũng có thể gọi một hàm mà muốn tiêm service vào bên trong nó như sau,

```php
use App\Repositories\UserRepository;
use Illuminate\Support\Facades\App;
```

```php
$result = App::call(function (UserRepository $repository) {
    // ...
});
```

Event của Container

Khi container của bạn resolve một đối tượng nào đó, thì nó sẽ phát đi một event, để bắt event này bạn dùng phương thức `resolving`, như trong ví dụ sau,

```php
use App\Services\Transistor;

$this->app->resolving(Transistor::class, function ($transistor, $app) {
  // Called when container resolves objects of type "Transistor"...
});

$this->app->resolving(function ($object, $app) {
  // Called when container resolves object of any type...
});
```

Trong ví dụ trên, đối tượng được resolve sẽ được đưa vào callback, ở đây bạn có thể thêm được vào bất kỳ thuộc tính nào vào đối tượng trước khi nó được đem đi tiêu thụ.

PSR-11

Service container trong Laravel thực thi interface PSR-11, do đó bạn có thể khai báo kiểu interface của container PSR-11 để nhận đối tượng container.

```php
use App\Services\Transistor;
use Psr\Container\ContainerInterface;

Route::get('/', function (ContainerInterface $container) {
    $service = $container->get(Transistor::class);

    //
});
```

Nếu không thấy đối tượng phù hợp thì nó sẽ quăng ra exception `Psr\Container\NotFoundExceptionInterface`. trong khi đó, nếu tìm thấy mà không resolve được đối tượng thì exception `Psr\Container\ContainerExceptionInterface` sẽ được quăng ra.