



Kiến trúc trong Laravel

Giảng viên: Bùi Quang Đăng



Request LifeCycle

Service Container

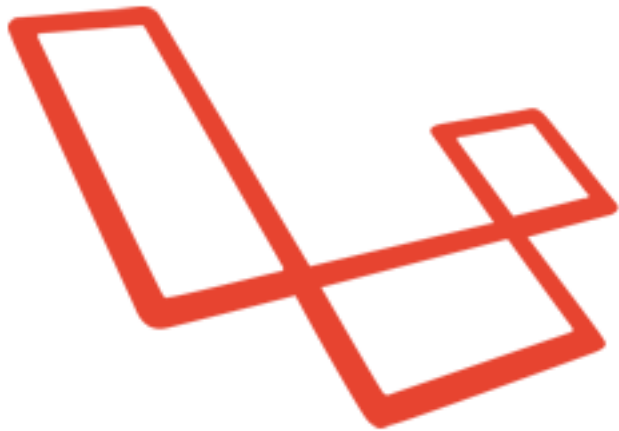
Service Provider

Facade, Contract



Laravel Framework

www.stanford.com.vn

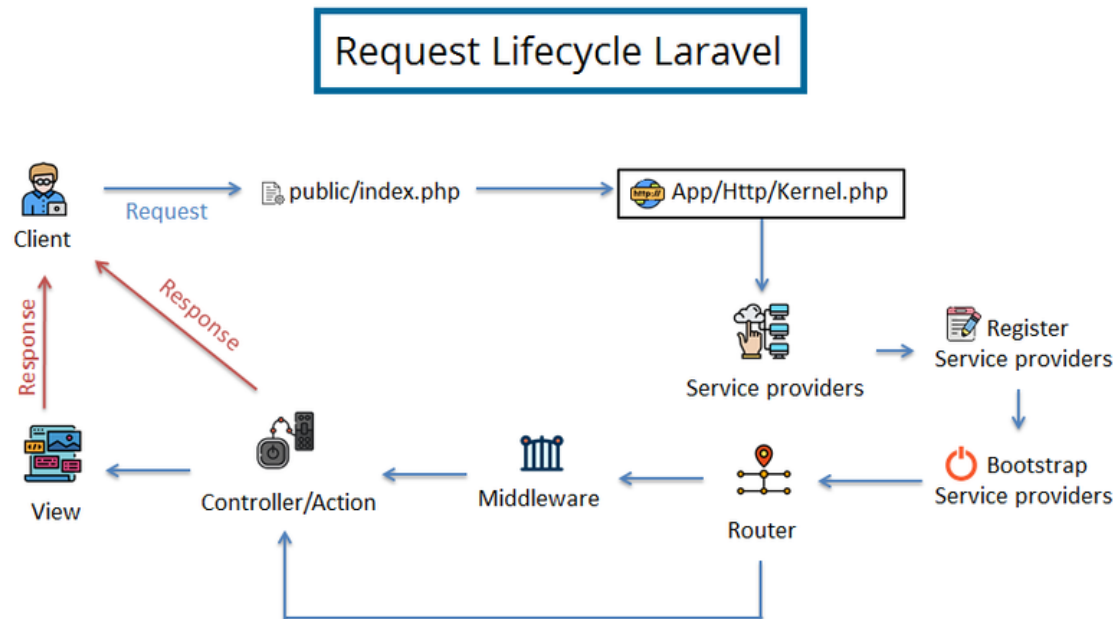


Laravel

Request LifeCycle

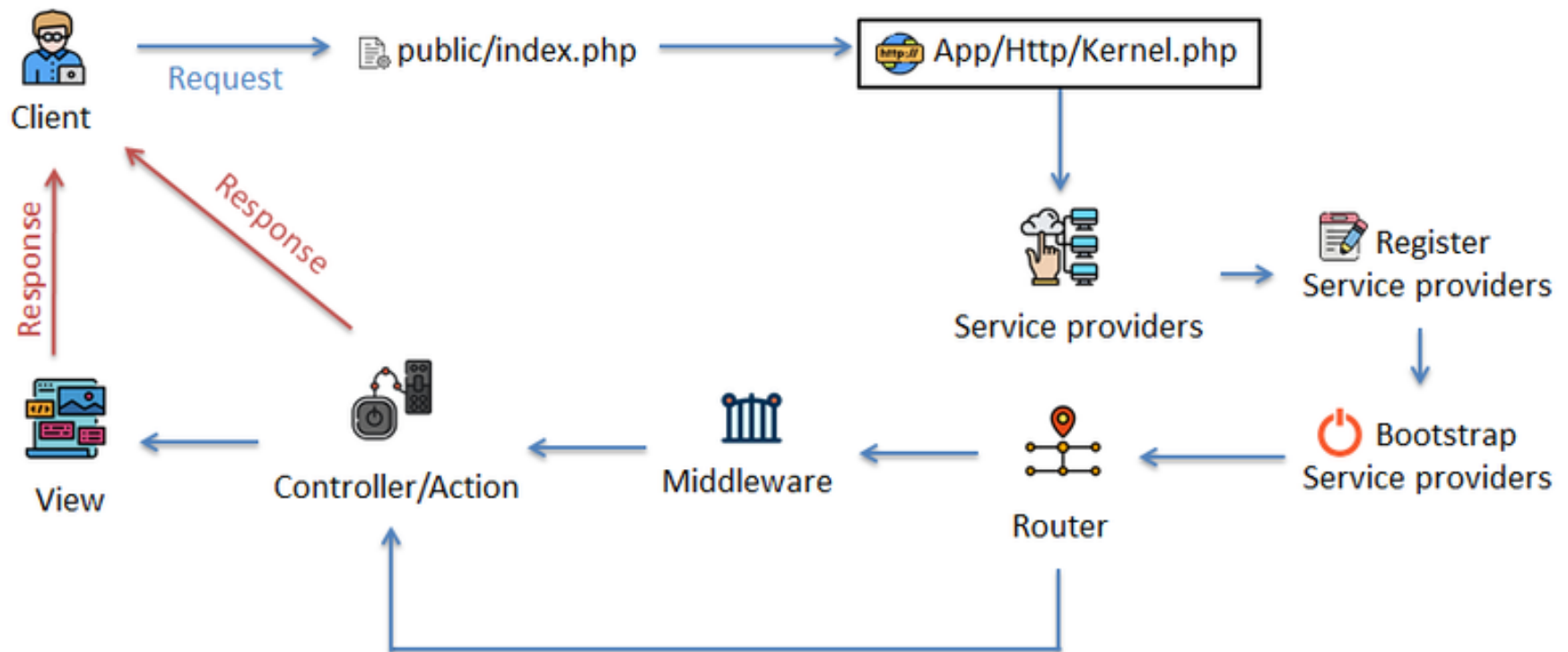
❖ Request Life Cycle

- Khi người dùng thực hiện một thao tác gửi yêu cầu (request) đến server cho đến khi nhận lại được phản hồi (response) từ server.



❖ Request Life Cycle

Request Lifecycle Laravel



❖ Request Life Cycle

■ Public/Index.php:

- Tất cả các request được gửi từ client đến ứng dụng Laravel hay server (tùy theo cách gọi của mọi người) đều phải chạy qua một file nằm trong folder public, đó là file index.php.
- File index.php này không chứa nhiều code mà chỉ nạp một số dịch vụ cần thiết cho framework.

❖ Request Life Cycle

■ Public/Index.php:

```
require __DIR__.'../vendor/autoload.php';  
  
$app = require_once __DIR__.'../bootstrap/app.php';
```

- Load tất cả khai báo của các thư viện trong file autoload.php (File này xuất hiện khi ta chạy lệnh composer dump autoload hoặc khi cài mới hoặc update một thư viện nào đó trong file composer.json).
- Lấy ra tất cả các instance được tạo ra ở trong file bootstrap/app.php. Các instance này được tạo ra từ việc resolve các binding từ trong Service Container.

❖ Request Life Cycle

■ Public/Index.php:

- Sau đó, request được đẩy tiếp theo đến file Kernel.php (HTTP hoặc Console) thông qua đoạn mã:

```
$kernel = $app->make(Kernel::class);  
  
$response = tap($kernel->handle(  
    $request = Request::capture()  
))->send();  
  
$kernel->terminate($request, $response);
```

❖ Request Life Cycle

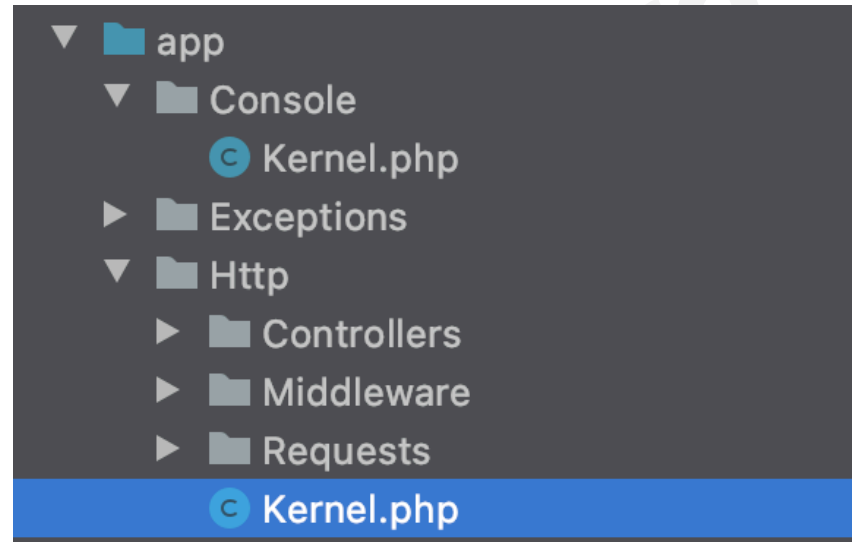
■ HTTP KERNEL & CONSOLE KERNEL

- Trong file **Kernel.php** chứa rất nhiều biến liên quan đến middleware – bộ lọc xử lý các yêu cầu trước khi xử lý logic.
- Http Kernel còn định nghĩa một mảng các bootstrappers cần phải chạy trước khi các request được xử lý bao gồm: cấu hình xử lý lỗi (error handling), cấu hình ghi log (logging), cấu hình ứng dụng các biến env và thực hiện các tác vụ khác cần được hoàn thành trước khi request thực sự được xử lý.

❖ Request Life Cycle

■ HTTP KERNEL & CONSOLE KERNEL

- Http Kernel nhận vào một request và trả lại một response để tiếp tục vòng đời của một request.



❖ Request Life Cycle

■ CONSOLE KERNEL

- Artisan Command cung cấp một số lệnh để hỗ trợ việc xây dựng ứng dụng, giảm thiểu thời gian code cũng như tự động hóa một số thao tác. Ngoài những lệnh mà framework cung cấp sẵn thì các bạn có thể viết thêm cũng như custom các lệnh này.

❖ Request Life Cycle

■ CONSOLE KERNEL

- Chúng ta có thể thêm những lệnh mới phục vụ các mục đích khác nhau thì cần khai báo cho Laravel biết, lệnh đó nằm ở vị trí nào trong project. Khi đó, có 2 cách để khai báo:
 - Khai báo cụ thể lệnh (command) trong biến `$commands`.
 - Khai báo thư mục chứa lệnh (command) trong hàm `commands()`.

❖ Request Life Cycle

■ CONSOLE KERNEL

- Mỗi khi gọi một lệnh Artisan command, request của bạn sẽ chạy qua file Console Kernel.php này và nó sẽ xác định xem lệnh đó đã được khai báo hay chưa?
 - Nếu rồi sẽ trả về response để chạy đúng vào file chứa command đó.
 - Nếu chưa sẽ báo lỗi (exception).

❖ Request Life Cycle

■ SERVICE PROVIDERS

- Sau khi request đã chạy qua file kernel, nó đi đến trái tim của framework – Service Providers.
- Service providers có vai trò rất quan trọng, thực hiện tải trước các bootstraper (các thư viện mà mình sử dụng trong dự án). Dựa vào file **config/app.php** mà Service provider sẽ biết để gọi và tải những bootstraper nào cần dùng.



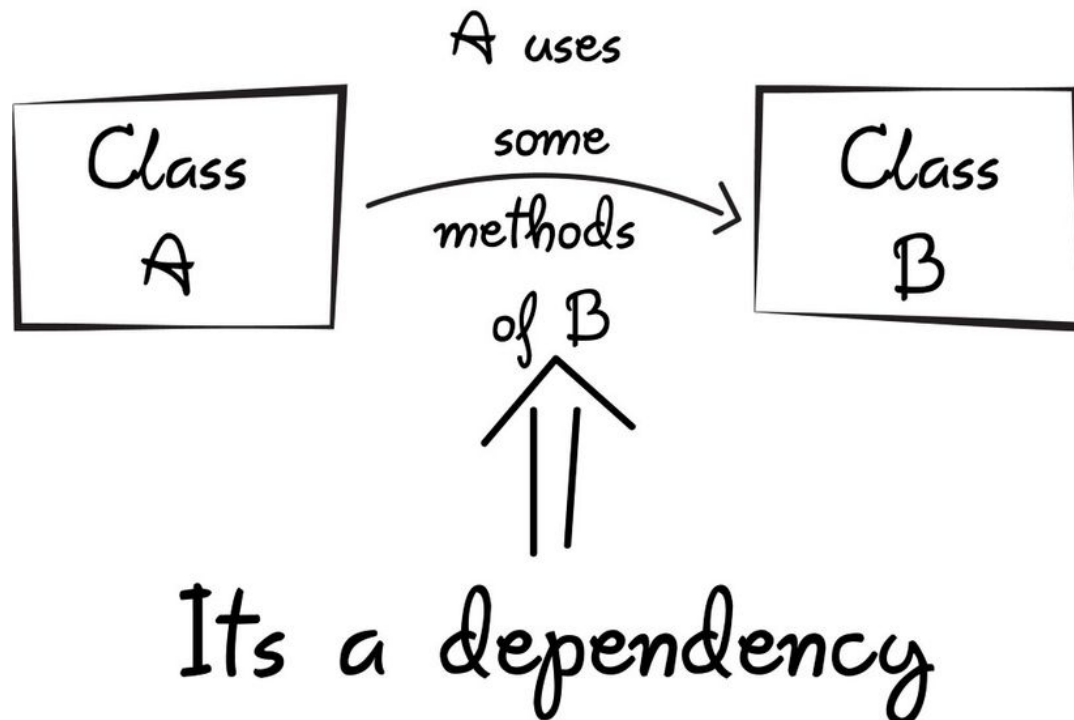
Service Container

❖ Dependency Inversion

- Nguyên lý cuối cùng trong SOLID chính là **Dependency Inversion**:
 - *Các module cấp cao không nên phụ thuộc vào các modules cấp thấp. Cả 2 nên phụ thuộc vào abstraction. Interface (abstraction) không nên phụ thuộc vào chi tiết, mà ngược lại. (Các class giao tiếp với nhau thông qua interface, không phải thông qua implementation.)*

❖ Dependency Inversion

- Nguyên lý này có thể được hiểu là “Các module phụ thuộc (dependency) sẽ được inject vào module cấp cao”. Nó giúp các module hoặc các class sẽ không quá phụ thuộc lẫn nhau, dễ dàng bảo trì code.



❖ Service Container

- Là công cụ rất quan trọng trong việc quản lý các lớp phụ thuộc (**dependency**) và thực hiện tiêm (**inject**) hay khởi tạo các đối tượng thuộc lớp đó để sử dụng thông qua hàm khởi tạo hoặc phương thức dạng setter.

❖ Service Container

■ Ví dụ:

```
class UserController extends Controller {  
  
    protected $users;  
    public function __construct(UserRepository $users)  
    {  
        $this->users = $users;  
    }  
  
    public function show($id)  
    {  
        $user = $this->users->find($id);  
        return view('user.profile', ['user' => $user]);  
    }  
}
```

❖ Service Container

- **Ví dụ:** Trong ví dụ trên trong lớp UserController để lấy được thông tin người dùng chúng ta cần inject (tiêm hay khởi tạo) một đối tượng từ lớp UserRepository để phục vụ lấy dữ liệu người dùng từ database.

❖ Service Container

- Có 2 khái niệm trong Service Container đó là bind và resolve:
 - *Bind để chỉ việc đăng ký một class hay một interface với Container.*
 - *Resolve để lấy ra instance từ Container.*

❖ Service Container

- Có pháp đăng ký một lớp với Container như sau:

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

❖ Service Container

- Sử dụng lấy đối tượng của lớp trong Container:
 - Với mỗi lần resolve như vậy, một object khác nhau sẽ được lấy ra của lớp cần sử dụng.

```
$app->make(PodcastParser::class)
```


❖ Service Container

■ Binding A Singleton:

- Binding một đối tượng hoặc một interface chỉ một lần duy nhất.

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

❖ Service Container

■ Binding Instances:

- Có thể binding một object đã tồn tại vào Container thông qua Instance Binding. Kết quả khi resolve cũng tương tự như Singleton Binding.

```
use App\Services\Transistor;  
use App\Services\PodcastParser;  
$service = new Transistor(new PodcastParser);  
$this->app->instance(Transistor::class, $service);
```

❖ Service Container

■ Binding Interfaces To Implementations:

- Có thể binding một object từ một interface, trong ví dụ EventPusher là một interface.

```
use App\Contracts\EventPusher;  
use App\Services\RedisEventPusher;  
$this->app->bind(EventPusher::class,  
RedisEventPusher::class);
```

- Xem thêm tại đây: [Service Container](#)

Service Provider

❖ Service Provider

- **Service Provider được coi là trái tim của Laravel**, có vai trò rất quan trọng, thực hiện tải trước các bootstraper (các thư viện mà mình sử dụng trong dự án). Dựa vào file **config/app.php** mà Service provider sẽ biết để gọi và tải những bootstraper nào cần dùng.
- Cú pháp tạo Service Provider:

```
php artisan make:provider [Tên Service Provider]
```

❖ Service Provider

- Khi đó ta sẽ có một Provider với hai method là register và boot:
 - **register() method:** Trong phương thức register(), bạn chỉ nên thực hiện việc **binding class vào service container**. Bạn không bao giờ nên khai báo hay đăng ký bất cứ các event listeners, routes, hoặc các chức năng nào khác trong phương thức này.
 - **boot() method:** Method này sẽ được gọi sau khi tất cả các service provider đã được đăng ký xong, điều này có nghĩa là bạn có thể gọi đến tất cả các service provider khác đã được đăng ký với framework.

❖ Service Provider

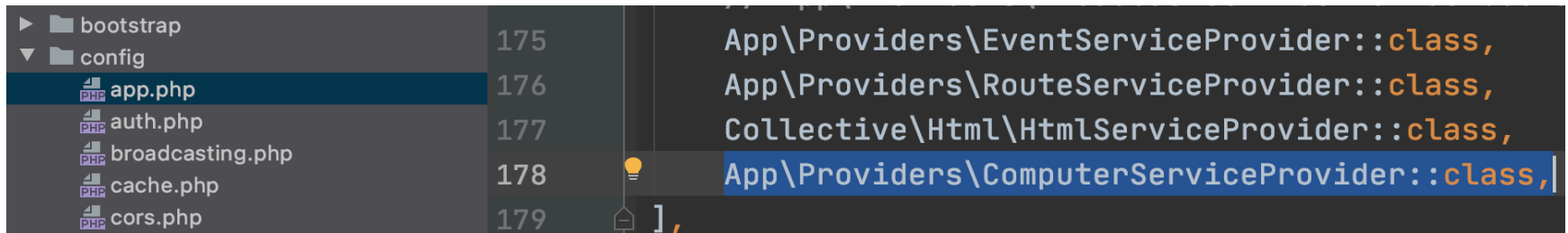
- Đăng ký một đối tượng trong hàm register như sau:

```
public function register()
{
    $this->app->bind( abstract: 'computer', function(){
        $computer = new Computer();
        $computer->name = "Macbook Pro";
        $computer->productName = "Apple";
        return $computer;
    });
}

/**
 * Bootstrap services.
 *
 * @return void
 */
public function boot(){...}
```

❖ Service Provider

- Đăng ký một đối tượng trong hàm register như sau:



```
175 App\Providers\EventServiceProvider::class,  
176 App\Providers\RouteServiceProvider::class,  
177 Collective\Html\HtmlServiceProvider::class,  
178 App\Providers\ComputerServiceProvider::class,  
179 ],
```

- Sử dụng Service Provider đã đăng ký khi cần:

```
//Lấy đối tượng trong container để dùng  
$com = app()->make( abstract: 'computer');  
$com->inThongTin();
```




Facade, Contract

❖ Facade

- Facade chính là resolve instance được binding trong Service Container thông qua tên của instance được binding trong hàm getFacadeAccessor.
- Lợi ích của việc dùng Facade đó là dễ dùng, dễ nhớ, dễ test.
 - Ví dụ: Facade sử dụng để lấy dữ liệu từ 1 bảng trong db

```
DB::select('SELECT * FROM users');
```

- Việc bạn gọi các hàm Route::get(), hay Auth::user() như trên đều là đang sử dụng Facade.
- Sử dụng Facade để che dấu, ẩn đi việc binding và resolve của một implementation trong Service Container.

❖ Facade

- Thực hiện tạo 1 lớp kế thừa từ Facade như sau:

```
namespace App\Facade;  
use Illuminate\Support\Facades\Facade;  
  
class ComputerFacade extends Facade  
{  
    protected static function getFacadeAccessor()  
    {  
        return 'computer';  
    }  
}
```

❖ Facade

- Sử dụng Facade vừa tạo như sau:

```
use App\Facade\ComputerFacade;  
  
class NhanVienController extends Controller  
{  
    public function layDanhSach(Request $request)  
    {  
        //Lấy đối tượng trong container để dùng  
        //$com = app('computer');  
        //$com->inThongTin();  
  
        ComputerFacade::inThongTin();  
    }  
}
```

❖ Contract

- Contract (hợp đồng) là những interface định nghĩa các dịch vụ core của framework trong Laravel
- Sử dụng Contract trong khi code chính là dùng Interface. Mà dùng Interface thì lại dễ code hơn khi mở rộng, dễ đọc code, dễ bảo trì. Đó chính là ưu điểm mà Contract mang lại.

❖ Contract

- Ví dụ: Xử lý gửi mail trong Laravel

```
// Dependency Injection Style
class SendMailDemo
{
    protected $mailer;
    public function
    __constructor(Illuminate\Contracts\Mail\Mailer $mailer)
    {
        $this->mailer = $mailer;
    }
    public function sendMail()
    { // Sending mail
        $this->mail->send($view, $data);
    }
}
```

❖ Contract

- Ví dụ: Xử lý gửi mail trong Laravel
 - Như các bạn đã biết thì Laravel, với sức mạnh của Service Container sẽ có thể tự động resolve ra một instance \$mailer cho chúng ta. Nhưng vấn đề là chúng ta đang **type-hint một Interface (Contract)**, chứ không phải là **một class cụ thể**.
 - Thực tế logic của chúng ta cũng không cần biết và quan tâm đến cái class gửi mail nó là class gì, hay nó đã làm thế nào để có thể gửi mail. Cái duy nhất logic của ta quan tâm là nó **cần một instance có thể gửi mail** để hoạt động. Đó là instance của class nào không quan trọng, miễn là class đó được cài đặt từ interface Illuminate\Contracts\Mail\Mailer.

❖ Contract

- Ví dụ: Xử lý gửi mail trong Laravel
 - Hay nói cách khác, code của chúng ta không phụ thuộc vào một implementation (một class) nào cụ thể cả, mà nó phụ thuộc vào một cái "hợp đồng" Mailer thôi.
 - Ngoài ra, nếu không muốn sử dụng Service Mailer có sẵn mà Laravel cung cấp, ta có thể tự viết ra một Service của riêng mình, đương nhiên là nó sẽ phải **"thỏa mãn"** cái **"hợp đồng"** có nghĩa là phải cài đặt từ interface Illuminate\Contracts\Mail\Mailer rồi. Tiếp sau đó ta tiến hành bind cái contract Illuminate\Contracts\Mail\Mailer với cái Service của mình vừa viết vào trong Service Container là xong. Những công việc còn lại Laravel đã lo hết cho chúng ta.



Exercises



Thank You !

www.stanford.com.vn