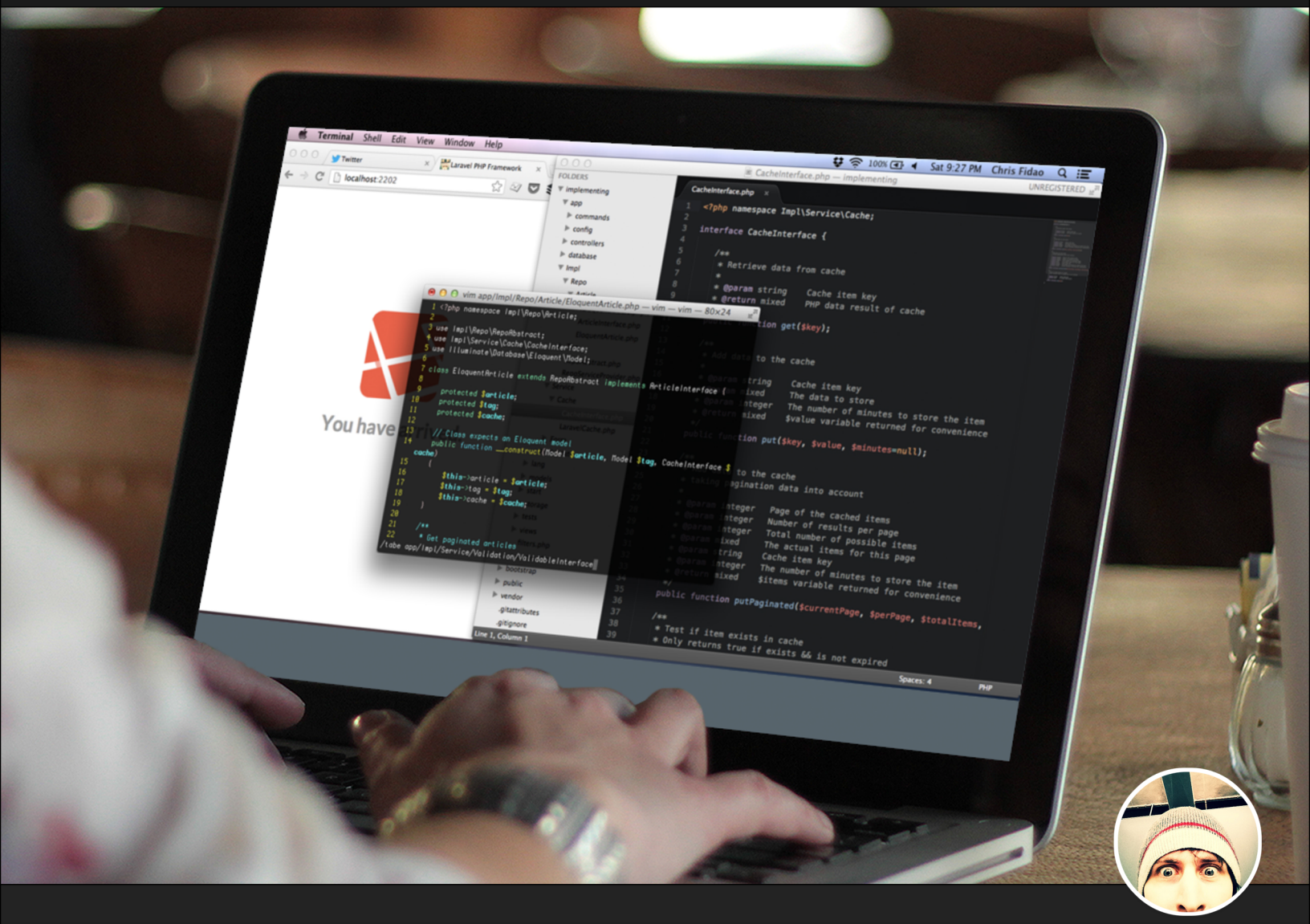

Implementando Laravel

POR
CHRIS FIDAO

TRADUÇÃO
ANTONIO CARLOS RIBEIRO

Implementando Laravel

Chris Fidao and Antonio Carlos Ribeiro

This book is for sale at http://leanpub.com/implementinglaravel-pt

This version was published on 2013-09-28

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Conteúdo

Conceitos Básicos **1**

O Contêiner **2**
Uso Básico 2
Indo Para Algo Mais Avançado 3
Inversão de Controle 3
Uso no Mundo Real 5

Injeção de Dependência **7**
O que é Injeção de Dependência? 7
Criando Novas Dependências no Controlador 8
Interfaces como Dependências 9
Por que Injeção de Dependência? 11
Conclusão 14

Conceitos Básicos

Ao longo deste livro nós vamos fazer uso de alguns dos recursos mais poderosos do Laravel.

Antes de começar é importante, pelo menos, saber algo sobre o contêiner do Laravel e como ele nos permite usar mais facilmente a Injeção de Dependência (Dependency Injection).

Este capítulo vai falar sobre o contêiner do Laravel, seu uso da Inversão de Controle (Inversion of Control) e da Injeção de Dependência.

O Contêiner

A classe `Illuminate\Foundation\Application` é quem faz a amarração de tudo o que há no Laravel. Esta classe é um *contêiner* - ela pode "conter" dados, objetos, classes e até "closures" (funções anônimas).

Uso Básico

Para ver como o contêiner funciona, vamos ver fazer um exercício no nosso arquivo de rotas (routes).

O contêiner do Laravel implementa a interface `ArrayAccess` e assim sendo é possível acessá-lo como se fosse um array. Vamos usá-lo como um array associativo:

```
1  Route::get('/container', function()
2  {
3      // Retorna a instância da Aplicação
4      $app = App::getFacadeRoot();
5
6      $app['some_array'] = array('foo' => 'bar');
7
8      var_dump($app['some_array']);
9  });
```

Apontando o browser para a rota `/container`, teremos este resultado:

```
1  array (size=1)
2      'foo' => string 'bar' (length=3)
```

Assim vemos que a `Application` (classe `App` acima), mesmo sendo uma classe com atributos e métodos está acessível como se fosse um array.

Facades

Não entendeu o que `App::getFacadeRoot()` faz? A classe `App` é uma fachada (Facade). Isso permite seu uso em qualquer lugar, acessando-a de maneira estática. Entretanto ela não é uma classe estática. O método `getFacadeRoot` retorna a instância real da classe, que nós precisávamos a fim de usá-la como um array no exemplo.

Veja esta e outras `Facades` no namespace `Illuminate\Support\Facades`.

Indo Para Algo Mais Avançado

Agora um uso mais sofisticado do contêiner, vamos atribuir a ele uma closure.

```
Route::get('/container', function()
{
    // Retorna a instância da Aplicação
    $app = App::getFacadeRoot();

    $app['say_hi'] = function()
    {
        return "Hello, World!";
    };

    return $app['say_hi'];
});
```

Novamente, apontando para a rota `/container` vamos ver:

```
Hello, World!
```

Embora aparentemente simples, isso é realmente muito poderoso. Isso é, de fato, a base de como os vários pacotes Illuminate interagem entre si, formando o `Laravel framework`.

Mais adiante vamos ver como os Provedores de Serviço (Service Providers) vinculam itens ao contêiner, funcionando como elo de ligação entre os vários pacotes Illuminate.

Inversão de Controle

A classe `Container` do Laravel possui muito mais cartas na manga do que apenas parecer ser um array. Ela também pode funcionar como contêiner de Inversão de Controle (IoC - do inglês Inversion of Control).

Inversão de Controle é uma técnica que nos permite definir como a nossa aplicação deve implementar uma classe ou uma interface.

Por exemplo, se a nossa aplicação possui como dependência `FooInterface` e nós queremos usar uma implementação dela, cuja classe vamos chamar de `ConcreteFoo`, o contêiner IoC é aonde definimos esta implementação.

Vamos ver um exemplo básico de como isso funciona usando nossa rota `/container`, novamente.

Primeiro vamos definir algumas classes - uma interface e uma classe que a implementa, estas podem ser colocadas diretamente no arquivo `app/routes.php`:

```php
interface GreetableInterface {

    public function greet();

}

class HelloWorld implements GreetableInterface {

    public function greet()
    {
        return 'Hello, World!';
    }
}
```

Agora vamos usar essas classes como nosso contêiner, para ver o que dá pra fazer. Primeiro vou introduzir o conceito de "ligação" (binding).

```php
Route::get('/container', function()
{
    // Retorna a instância da Aplicação
    $app = App::getFacadeRoot();

    $app->bind('GreetableInterface', function()
    {
        return new HelloWorld;
    });

    $greeter = $app->make('GreetableInterface');

    return $greeter->greet();
});
```

Ao invés de usar o (acessível como array) `$app['GreetableInterface']`, usamos o método `bind()`.

Ele configura o contêiner IoC do Laravel, para que este devolva uma instância da classe `HelloWorld` toda vez que for solicitado pela interface `GreetableInterface`.

Desta forma podemos mudar implementações! Por exemplo, ao invés de `HelloWorld`, eu poderia criar uma nova implementação, chamá-la de `GoodbyeCruelWorld` e decidir que o contêiner a devolva sempre que alguém solicitar `GreetableInterface`.

Isso nos leva ao encontro da manutenibilidade da nossa aplicação. Usando o contêiner nós podemos (de forma ideal) mudar implementações em determinados pontos sem afetar outras áreas do código da nossa aplicação.

Uso no Mundo Real

Onde na sua aplicação você vai colocar todas essas ligações? Se você não quer abarrotar os arquivos `start.php`, `filters.php`, `routes.php` e outros arquivos envolvidos na inicialização com ligações, então você pode usar classes Provedoras de Serviços (Service Providers).

Provedores de Serviço são criados especificamente para registrar as ligações feitas ao contêiner do Laravel. De fato praticamente todos os pacotes Illuminate usam um Provedor de Serviço para fazer apenas isso.

Vamos ver um exemplo de como Provedores de Serviços são usados em um pacote `Illuminate`. Examinemos o pacote Pagination.

Primeiro, aqui está o método `register()` do Provedor de Serviço Pagination:

```
1  public function register()
2  {
3      $this->app['paginator'] = $this->app->share(function($app)
4      {
5          $paginator = new Environment(
6              $app['request'],
7              $app['view'],
8              $app['translator']
9          );
10
11         $paginator->setViewName(
12             $app['config']['view.pagination']
13         );
14
15         return $paginator;
16     });
17 }
```

O método `register()` é executado automaticamente em cada Provedor de Serviço especificado no arquivo `app/config/app.php`.

E o que está acontecendo nesse método `register()`? Em primeiro lugar, e acima de tudo, ele registra a instância do "paginator" no contêiner. Isso disponibilizará `$app['paginator']` e `App::make('paginator')` para uso em outras áreas da aplicação.

Em seguida ele define a instância do "paginator" como retorno (resultado) de uma "closure", exatamente como fizemos no exemplo `say_hi`.

O método Share em `$this->app->share()` apenas provê um caminho para que a "closure" seja utilizada como um "singleton" (garantia de apenas uma instância).

Esta closure cria um novo objeto `Pagination\Environment`, altera nele um parâmetro de configuração e o devolve.

Você provavelmente notou que o Provedor de Serviço usa outras ligações com a aplicação. A classe `PaginationEnvironment` claramente possui algumas dependências no seu método construtor (__construct): um objeto request (`$app['request']`), um objeto view (`$app['view']`) e um objeto translator (`$app['translator']`). Felizmente estas ligações são criadas em outros pacotes do Illuminate, definidas em diversos Provedores de Serviços.

Podemos, assim, ver como os diversos pacotes Illuminate interagem entre si. Devido ao fato deles estarem "ligados" ao contêiner da aplicação, nós podemos usá-los em outros pacotes (ou em nosso próprio código!), sem amarar nosso código a uma classe específica.

Injeção de Dependência

Agora que sabemos como o contêiner funciona, vamos ver como nós podemos usá-lo para implementar Injeção de Dependência no Laravel.

O que é Injeção de Dependência?

Injeção de Dependência (Dependency Injection) é o ato de injetar (introduzir) qualquer dependência no momento da instanciação de uma classe, ao invés de invés de instanciar as dependências em algum lugar dentro próprio código da classe. Frequentemente dependências são definidas como indução de tipo (type-hinting), que é forçar que os parâmetros de um método construtor sejam objetos de uma determinada classe.

Veja este método construtor, por exemplo:

```
1  public function __construct(HelloWorld $greeter)
2  {
3      $this->greeter = $greeter;
4  }
```

Ao forçar (induzir) que o parâmetro `$greeter` seja do tipo `HelloWorld`, estamos afirmando explicitamente que a classe depende de uma instância de `HelloWorld`.

Opostamente, esta é uma instanciação direta:

```
1  public function __construct()
2  {
3      $this->greeter = new HelloWorld;
4  }
```

Se você está se perguntando o *porquê* de se usar Injeção de Dependência, esta resposta no Stack Overflow[1] é um ótimo ponto de partida. Eu vou cobrir alguns dos benefícios dela nos exemplos que seguem.

A seguir vamos ver um exemplo de Injeção de Dependência em ação, usando o contêiner IoC do Laravel.

[1] http://stackoverflow.com/questions/130794/what-is-dependency-injection

Criando Novas Dependências no Controlador

Este é um caso de uso bem comum em Laravel.

Normalmente, criarmos um controlador que precisa receber uma classe instanciada em seu método construtor (__construct), nós precisaremos passar essas dependências (sob a forma de parâmetros) quando a classe for criada. Mas o que acontece quando você cria uma dependência em um controlador Laravel? Você vai precisar instanciar o controlador manualmente a fim de solucionar a dependência (enviá-la através de parâmetro). Veja um exemplo:

```
1  $crtl = new ContainerController( new HelloWorld );
```

Isso é ótimo, mas ninguém instancia controladores diretamente em Laravel - o roteador lida com isso para nós.

Nós podemos, entretanto, injetar as dependências do controlador através do uso do contêiner IoC do Laravel!

Mantendo as mesmas classes `GreetableInterface` e `HelloWorld` de antes, vamos imaginar agora que ligamos nossa rota `/container` a um controlador:

Arquivo: app/routes.php

```
1  interface GreetableInterface {
2
3      public function greet();
4
5  }
6
7  class HelloWorld implements GreetableInterface {
8
9      public function greet()
10     {
11         return 'Hello, World!';
12     }
13 }
14
15 Route::get('/container', 'ContainerController@container);
```

Agora em nosso novo controlador, podemos definir `HelloWorld` como um parâmetro no método construtor:

Arquivo: app/controllers/ContainerController.php

```php
class ContainerController extends BaseController {

    protected $greeter;

    // Dependência da classe: HelloWorld
    public function __construct(HelloWorld $greeter)
    {
        $this->greeter = $greeter;
    }

    public function container()
    {
        return $this->greeter->greet();
    }

}
```

Agora execute a rota `/container` e você deve, novamente, ver:

```
Hello, World!
```

Observe, porém, que nós não fizemos nenhuma ligação no contêiner. Ele "simplesmente funcionou" - uma instância de `HelloWorld` foi automaticamente enviada ao controlador.

Isso aconteceu porque o contêiner IoC tenta resolver qualquer dependência definida no método construtor de um controlador. E Laravel vai injetar as dependências necessárias para nós!

Interfaces como Dependências

Mas nós ainda não terminamos, finalmente chegamos ao que queríamos mostrar.

E se ao invés de especificar `HelloWorld` como dependência do controlador, nós especificássemos a interface `GreetableInterface`?

Vamos ver como fica:

```php
class ContainerController extends BaseController {

    protected $greeter;

    // Dependência da classe: GreetableInterface
    public function __construct(GreetableInterface $greeter)
    {
        $this->greeter = $greeter;
    }

    public function container()
    {
        echo $this->greeter->greet();
    }

}
```

Se tentarmos executar como está, vamos receber o seguinte erro:

```
Illuminate\Container\BindingResolutionException:
Target [GreetableInterface] is not instantiable
```

A classe `GreetableInterface`, claro, não é instanciável, porque ela é uma interface. Mas podemos ver que Laravel está tentando instanciá-la, para resolver a dependência definida na classe.

Resolvendo isso - nosso controlador depende de uma instância de `GreetableInterface`, então vamos então usar o método `bind()` do contêiner para indicar ao Laravel que ele deve enviar uma instância de `HelloWorld`:

```php
interface GreetableInterface {

    public function greet();

}

class HelloWorld implements GreetableInterface {

    public function greet()
    {
        return 'Hello, World!';
    }
}
```

```
14
15 // Ligando a classe HelloWorld quando for solicitada uma instância de
16 // GreetableInterface aqui!!
17 App::bind('GreetableInterface', 'HelloWorld');
18
19 Route::get('/container', 'ContainerController@container);
```

Agora execute a rota `/container` e você deve ver `Hello, World!` novamente!

Note que eu não usei uma função anônima para ligar a `HelloWorld` - Você pode simplesmente passar o nome da classe concreta como uma string, se preferir. Uma closure é útil quando a implementação tem as suas próprias dependências que precisam ser passadas no método construtor.

Por que Injeção de Dependência?

Por que nós queremos especificar uma interface como dependência ao invés de uma classe concreta?

Porque precisamos que toda classe definida no construtor seja uma subclasse de uma interface. Desta forma podemos com segurança usar qualquer implementação, porque o método que precisamos sempre estará disponível.

Em outras palavras e de forma sucinta: **podemos mudar a implementação à vontade, sem afetar outras porções do código da nossa aplicação**.

Aqui vai um exemplo de algo que precisei fazer muitas vezes em aplicações reais.

Não copie e cole este exemplo. Estou omitindo alguns detalhes, tal como o uso de variáveis de configuração para chaves de API, para ficar mais claro.

Vamos dizer que sua aplicação envia e-mails usando o AWS da Amazon. Para tal foi definida uma interface `Emailer` implementada na classe `AwsEmailer`:

```php
interface Emailer {

    public function send($to, $from, $subject, $message);
}

class AwsEmailer implements Emailer {

    protected $aws;

    public function __construct(AwsSDK $aws)
    {
        $this->aws = $aws;
    }

    public function send($to, $from, $subject, $message)
    {
        $this->aws->addTo($to)
            ->setFrom($from)
            ->setSubject($subject)
            ->setMessage($message);
            ->sendEmail();
    }
}
```

Ligamos `Emailer` à implementação `AwsEmailer`:

```php
App::bind('Emailer', function()
{
    return new AwsEmailer( new AwsSDK );
});
```

O controlador tem a interface `Emailer` como dependência:

```
class EmailController extends BaseController {

    protected $emailer;

    // Class dependency: Emailer
    public function __construct(Emailer $emailer)
    {
        $this->emailer = $emailer;
    }

    public function email()
    {
        $this->emailer->send(
            'ex-to@example.com',
            'ex-from@example.com',
            'Peanut Butter Jelly Time!',
            "It's that time again! And so on!"
        );

        return Redirect::to('/');
    }

}
```

Vamos em seguida supor que no futuro o escopo da sua aplicação cresca e passe a precisar de funcionalidades do que a AWS não provê. Depois de alguma procura e pesar de opções, você decide pelo SendGrid.

Como você vai fazer para migrar a sua aplicação para o SendGrid? Devido ao fato de estar usando interfaces e o contêiner IoC do Laravel, migrar para o SendGrid é fácil!

Primeiro, crie uma implementação da interface `Emailer` que use SendGrid!

```
class SendGridEmailer implements Emailer {

    protected $sendgrid;

    public function __construct(SendGridSDK $sendgrid)
    {
        $this->sendgrid = $sendgrid;
    }

    public function send($to, $from, $subject, $message)
```

```
    {
        $mail = $this->sendgrid->mail->instance();

        $mail->addTo($to)
            ->setFrom($from)
            ->setSubject($subject)
            ->setText( strip_tags($message) )
            ->setHtml($message)
            ->send();

        $this->sendgrid->web->send($mail);
    }
}
```

Em seguida (e por último), configure a aplicação para usar SendGrid ao invés de AWS. Por ter nossa chamada a `bind()` no contêiner IoC, alterar a implementação do `Emailer` de `AwsEmailer` para `SendGridEmailer` é tão simples quanto *esta* mudança:

```
// De
App::bind('Emailer', function()
{
    return new AwsEmailer( new AwsSDK );
});

// Para
App::bind('Emailer', function()
{
    return new SendGridEmailer( new SendGridSDK );
});
```

Note que fizemos isso sem alterar uma linha de código em qualquer outro ponto da nossa aplicação. Impor o uso da interface `Emailer` como dependência, garante que qualquer classe injetada sempre terá o método `send()` disponível.

Podemos ver isso em nosso exemplo. O controlador continua chamando `$this->emailer->send()` e não teve que ser modificado quando mudamos a implementação de `AwsEmailer` para `SendGridEmailer`.

Conclusão

Injeção de Dependência e Inversão de Controle são padrões exaustivamente utilizados no desenvolvimento em Laravel.

Como você verá, vamos definir várias interfaces a fim de tornar nosso código manutenível e ajudar nos testes. O contêiner IoC do Laravel torna isso fácil para nós.