Implementing Laravel

Chris Fidao

Implementing Laravel 日本語版

堅牢原則を応用し、十分な機能を持つアプリケーション構築の基礎となる実装方法

Chris Fidao and Hirohisa Kawase

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

This version was published on 2014-02-06

Contents

Laravel コアのコンセプト

この本を通じ、Laravel のとてもパワフルな特徴を使用していきます。

実際に取り掛かる前に、最低でも Laravel のコンテナと、それが依存注入にとても簡単に使用できることを理解することが重要です。

この章では制御の逆転 (Inversion of Control) と依存注入 (Dependency Injection) に使用する、Laravel のコンテナを取り上げます。

コンテナ

`Illuminate\Foundation\Application` クラスは、Laravel の全てを一つにまとめています。このクラスは**コンテナ** *(container)* で、データー、オブジェクト、クラス、そしてクロージャーでさえ含む (conatain) ことができます。

基本的な使い方

コンテナがどのように動作するのかを確認するため、ルートファイル[1]の中で、実践してみてください。

Laravel のコンテナは、`ArrayAccess` を実装しており、そのため配列のようにアクセスできます。連想配列のようにアクセスする方法を確認してください。

File: app/routes.php

```
1  Route::get('/container', function()
2  {
3          // アプリケーションインスタンスを取得
4          $app = App::getFacadeRoot();
5
6          $app['some_array'] = array('foo' => 'bar');
7
8          var_dump($app['some_array']);
9  });
```

`/conatainer` ルートにアクセスすれば、結果が表示されます。

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

この「アプリケーション」で確認したように、属性とメソッドを持つクラスであっても、配列のようにアクセスできるのです！

ファサード (Facade)

`App::getFacadeRoot()` は何を行なっているか、混乱していませんか?`App` クラスはファサードです。ファサードにより、どこであろうと静的メソッドのようにアクセスできるのです。しかしながら、実際には静的なクラスではありません。この例では、配列のようにアクセスできるのを示すため、`getFacadeRoot` で、クラスの実際のインスタンスを取得しています。

App とその他のファサードを確認するには、`Illuminate\Support\Facades` 名前空間を調べてください。

[1] Laravel では、app/routes.php にルーティングを定義しますが、コントローラークラスを使用せず、クロージャーで定義することも可能です。学習段階や小さなアプリケーションでは、この機能を手軽に利用できます。

より高度な例

それでは、更にコンテナの愛用者になれるように、より高度な、クロージャーによる結合を取り扱いましょう。

File: app/routes.php

```
Route::get('/container', function()
{
        // アプリケーションインスタンスを取得
        $app = App::getFacadeRoot();

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

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

再度、`/container` ルートへアクセスしてください。以下のように表示されます。

```
Hello, World!
```

一見シンプルですが、本当はとてもパワフルです。実際、Laravel のフレームワークを構成している、それぞれ互いに関連した Illuminate パッケージを分離している方法の基礎になっています。

この後、サービスプロバイダーでアイテムをコンテナに結合する手法を確認します。いろいろな Illuminate パッケージ間で、接着剤のように動作しています。

制御の逆転

Laravel の `Container` クラスは、ただの配列のように見せかける以上の働きをします。制御の逆転 (IoC) コンテナとしても機能します。

制御の逆転とはテクニックの一つで、あるクラスやインターフェイスをアプリケーションでどう実装するかを定義することです。例えば、アプリケーションが `FooInterface` へ依存しているとして、`ConcreteFoo` クラスをその実装として使用したい場合、IoC コンテナへ定義します。

動作を示すため、もう一度 `/container` ルートを使用した、基本的な例を紹介しましょう。

最初に、クラスを用意します。インターフェイスと、実装クラスです。分かりやすいように、2つを `app/routes.php` ファイルで定義しましょう。

File: app/routes.php

```php
interface GreetableInterface {

    public function greet();

}

class HelloWorld implements GreetableInterface {

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

では、何ができるのかを確認しましょう。両クラスをコンテナで使用してみましょう。先ず、「結合」のコンセプトを紹介します。

File: app/routes.php

```php
Route::get('/container', function()
{
    // アプリケーションインスタンスを取得
    $app = App::getFacadeRoot();

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

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

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

`$app['GreetableInterface']` のように、配列形式のアクセスを使用する代わりに、`bind()` メソッドを使用しました。

これで `GreetableInterface` で参照すれば、いつでも `HelloWorld` クラスが IoC コンテナからリターンされます。

この方法を使えば、実装を「交換」できます！例えば、`HelloWorld` インスタンスの代わりに、`GoodbyeCruelWorld` 実装を作成し、`GreetableInterface` で参照されたら、いつでもそのインスタンスをコンテナから返させることができます。

これにより、アプリケーションの保守性が向上します。コンテナを使用すれば、アプリケーションの他の領域に影響を与えることなく、一箇所で実装を（理想的に）交換できます。

使用の実例

この様な結合をアプリケーションのどこにまとめて全部設置すれば良いのでしょうか? `start.php` や `filters.php`、`routes.php`、その他のブートストラップファイルを結合の定義で取っ散らかしたくなければ、サービスプロバイダークラスを使用しましょう。

サービスプロバイダーは、特に Laravel のコンテナへ結合を登録するため、作成されることが多いのです。実際、ほとんどの Illuminate パッケージは、このためにサービスプロバイダーを使用しています。

`Illuminate` パッケージの中で、サービスプロバイダーがどのように使用されているか、例を見てみましょう。Pegination パッケージを調べてみましょう。

最初に、Pagination サービスプロバイダーの `register` メソッドです。

IlluminatePaginationPaginationServiceProvider.php

```
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 }
```

`app/config/app.php` ファイルの中で登録されている、サービスプロバイダーの `register()` メソッドは、自動的に呼び出されます。

では、`register()` メソッドでは、何を行なっているのでしょう?最初に行われ、また一番重要なのは、"paginator" インスタンスをコンテナに登録していることです。それにより、アプリケーションの他の領域で、`$app['paginator']` と `App::make('peginator')` が使用できるようになります。

次に、HelloWorld の例でやったのと同様に、クロージャーの実行結果として、`paginator` インスタンスをリターンするように定義しています。

`$this->app->share()` を使用していますが、混乱しないでください。`share()` メソッドは `$this->app->instance('Peginator', new Environment)` の呼び出しと同様に、シングルトンとしてインスタンスを提供しますが、クロージャーを使用します。

今回のクロージャーでは、新しい `Pagination\Environment` オブジェクトを生成し、設定値をセットしてから、そのオブジェクトをリターンしています。

皆さん、多分、サービスプロバイダーで他のアプリケーション結合を使用していることに、気づいていることでしょう。`PaginationEnvironment` クラスは明らかに、コンストラクターメソッドの中で、他のパッケージに依存しています。リクエストオブジェクトの `$app['request']`、ビューオブジェクトの `$app['view']`、翻訳オブジェクトの `$app['translator]`です。これらの結合の登録も同様に、Illuminate の他のパッケージにより作成された、それぞれのサービスプロバイダーの中で定義されています。

これで、様々な Illuminate パッケージが存在していること、それぞれが相互に関わっていることが理解できたでしょう。各パッケージがアプリケーションコンテナへ結合されているので、実際に私達のコードを特定のクラスに結びつけなくても、他のパッケージの中でも（私達自身のコードの中でも！）使用することができるのです。

依存注入

前の章ではコンテナの動作を理解しました。続いて、依存注入をどのようにコンテナを利用し、Laravel で実現するかを学びましょう。

依存注入とは何か？

依存注入とは、あるクラスに必要な依存をそのクラスの中でインスタンス化するのではなく、外から追加（注入）する動作を意味します。コンストラクターメソッドの引数で、タイプヒントにより、よく依存は定義されます。

例えば、次のコンストラクターメソッドを取り上げましょう。

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

引数として `HelloWorld` をタイプヒントに指定することにより、`HelloWorld` クラスへ依存していることを明確に宣言しています。

この逆は、直接インスタンス化する方法です。

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

どうして依存注入が使用されるのか疑問に思えるのでしたら、この Stack Overflow の回答[2]を最初に読むことをおすすめします。以降の例でも、長所を説明していきます。

次に、Laravel の IoC コンテナを利用し、依存注入の実例を見て行きましょう。

コントローラーへ依存を追加する

これは Laravel でよく取り扱われるケースです。

あるクラスのコンストラクターメソッドに、期待するクラスをセットすれば、通常、そのクラスをインスタンス化する際、そうした依存を追加する必要が起きます。しかしながら、Laravel のコントローラーに依存を定義する場合、何が起きると思いますか？コンストラクターを手動でインスタンス化する必要があるならば、次のようになるでしょう。

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

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

これはこれで良いのですが、Laravel では、コントローラーを直接インスタンス化することはありません。ルーターが面倒を見てくれます。

しかしながら、それでも、Laravel の IoC コンテナを使用し、コントローラーへ依存を注入することができるのです！

以前と同じ、`GreetableInterface` と `HelloWorld` クラスを続けて使用しましょう。`/container` ルートをコントローラーへ結び付けていると想像してください。

File: app/routes.php

```
interface GreetableInterface {

    public function greet();

}

class HelloWorld implements GreetableInterface {

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

Route::get('/container', 'ContainerController@container);
```

では、新しいコントローラーで、`HelloWorld` をコンストラクターメソッドの引数としてセットしましょう。

File: app/controllers/ContainerController.php

```
<?php

class ContainerController extends BaseController {

    protected $greeter;

    // 依存クラス：HelloWorld
    public function __construct(HelloWorld $greeter)
    {
        $this->greeter = $greeter;
    }

    public function container()
```

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

}
```

これで、/container ルートへアクセスすれば、以下の表示を再び目にするでしょう。

```
Hello, World!
```

コンテナで何も結合していないことに、注目してください。これでもちゃんと動作しました。HelloWorld インスタンスが、コントローラーに渡されました！

この理由は、コントローラーのコンストラクターメソッドに指定した依存を全て、IoC コンテナが自動的に解決しようとしたからです。Laravel が私達のために、指定した依存を注入してくれるのです！

依存としてのインターフェイス

今のところ、まだ何も行なっていませんが、基礎を固めているところです。

HelloWorld をコントローラーの依存として指定する代わりに、GeetableInterface インターフェイスを指定したらどうなるでしょう？

どうなるか、やってみましょう。

File: app/controllers/ContainerController.php

```
<?php

class ContainerController extends BaseController {

    protected $greeter;

    // クラス依存：GreetableInterface
    public function __construct(GreetableInterface $greeter)
    {
        $this->greeter = $greeter;
    }

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

}
```

このまま実行すれば、エラーになります。

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

GreetableInterface はインターフェイスですから、当然インスタンス化することができません。ところが、今まで学んできた通り、Laravel はクラス依存を解決しようとして、インスタンス化を試みるわけです。

修正しましょう。依存している GreetableInterface のインスタンスをコンテナが探し出せるように、コンテナの bind() メソッドで Laravel へコントローラーへ HelloWorld のインスタンスを与えるように指定しましょう。

File: app/routes.php

```
interface GreetableInterface {

    public function greet();

}

class HelloWorld implements GreetableInterface {

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

// GreetableInterface が見つかるように、
// HelloWorld をここで結合！
App::bind('GreetableInterface', 'HelloWorld');

Route::get('/container', 'ContainerController@container);
```

これで、/container ルートにアクセスすれば、再び Hello,World! が再度表示されるようになります！

HelloWorld の結合に、クロージャーを使用しなかったことに注意を払ってください。結合したい具象クラスの名前を文字列で渡すことができます。実装が、それ自身の依存をコンストラクターメソッドに渡す必要がある場合に、クロージャーは便利です。

なぜ、依存注入？

なぜ、私達は具象クラスの代わりに、インターフェイスを依存として、指定しようとしてるのでしょうか？

なぜなら、インターフェイスのサブクラスであれば、どんなクラスの依存であっても、コンストラクターに指定できるようにしたいからです。この方法を取れば、どんな実装であっても、安全に使用することができます。必要なメソッドがいつも揃っているからです。

手短に言えば、**アプリケーションコードの他の部分に影響を与えることなく、実装を希望通りに変更できるからです**。

一例を上げましょう。実際のアプリケーションで何度も実装したものです。

この例はコピー&ペーストしないでください。ポイントを明確にするために、API キーの設定変数を使用することなど、詳細を省いています。

アプリケーションで Amazon AWS を使用しメールを送るとしましょう。`Emailer` インターフェイスを定義済みで、`AwsEmailer` を実装中です。

```
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();
    }
}
```

`Emailer` に `AwsEmailer` 実装を結合します。

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

コントローラーは `Emailer` インターフェイスを依存として使用しています。

File: app/controllers/EmailController.php

```
class EmailController extends BaseController {

    protected $emailer;

    // クラス依存：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('/');
    }

}
```

話を続けましょう。将来のある日、アプリケーションはある規模以上に大きくなり、AWS が提供しているよりもっと多くの機能が必要になりました。リサーチの結果と使用できるオプションが最適なことから、SendGrid を採用することに決めました。

アプリケーションを SendGrid へ、どうやって変更するのでしょう。インターフェイスと Laravel の IoC コンテナを使用しているんですよ、SendGrid へ変更するなんて簡単です！

最初に `Emailer` の SendGrid 実装を作成しましょう！

```php
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);
    }
}
```

次に、（最後でもあります！）アプリケーションへ Aws ではなく、SendGrid を使うように設定します。IoC コンテナの `bind()` を呼び出しているので、`Emailer` の実装を `AwsEmailer` から `SendGridEmailer` へ変更します。簡単ですし、この**一箇所**のみの変更です。

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

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

アプリケーションの他の部分を一切変更することなく、行えたことに注目です。依存として `Emailer` インターフェイスを使用するように強制することで、注入されるどのクラスであっても、`send()` メソッドが使用できることが保証されています。

もう一度、今回の例を見てみましょう。実装を `AwsEmailer` から `SendGridEmailer` へ切り換えても、コントローラーは `$this->emailer->send()` を呼び出したままで、変更する必要はありません。

まとめ

依存注入と制御の反転は、Laravel の開発で、何度も繰り返して使用されるパターンです。

理解してもらった通り、コードをメンテナンスしやすくし、テスト時に楽なように、たくさんのインターフェイスを定義しましょう。Laravel の IoC コンテナは、これを簡単にしてくれています。