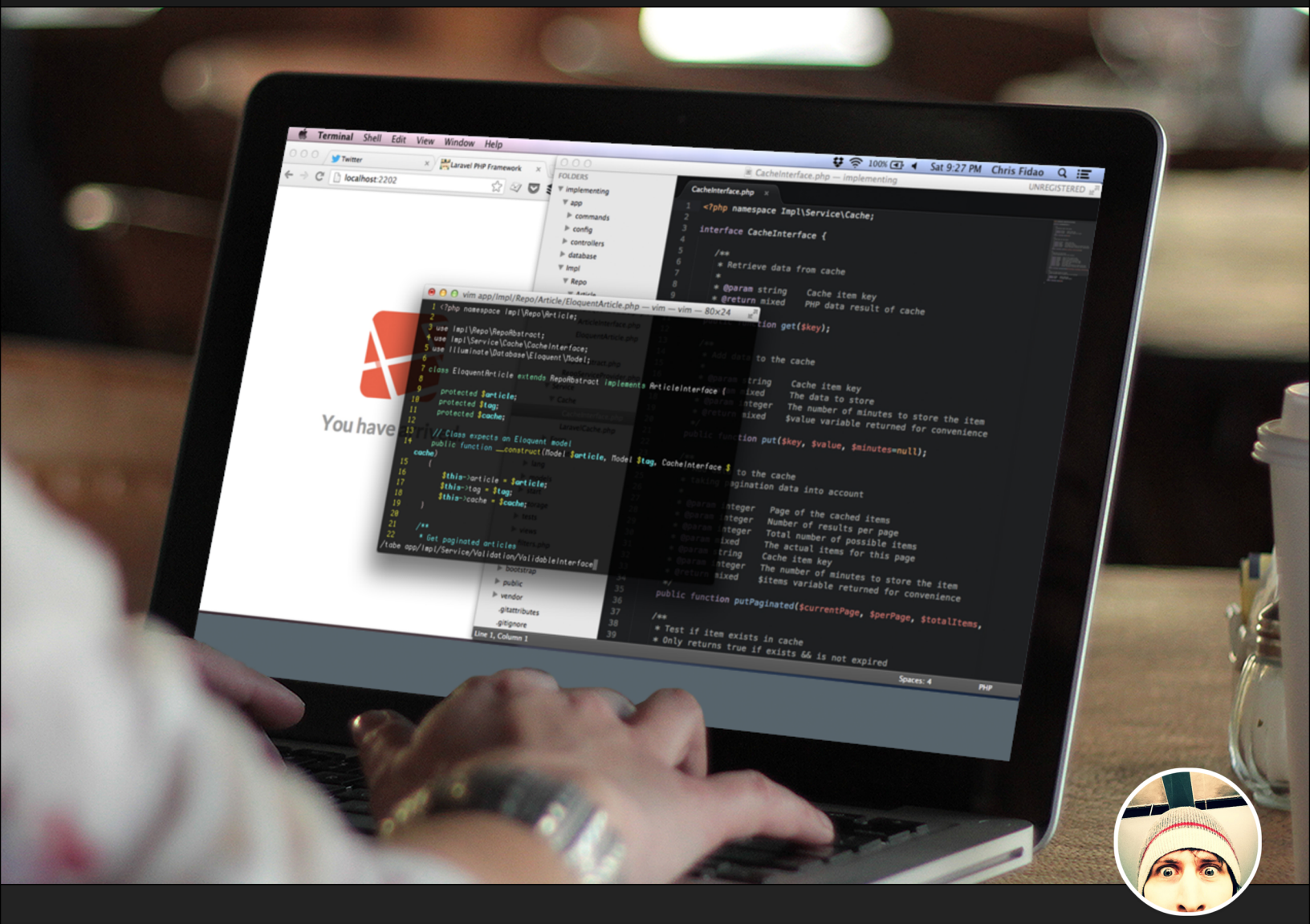

Implementing Laravel

Chris Fidao

Implementing Laravel

Chris Fidao

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

This version was published on 2013-10-21

Contents

Core Concepts **1**

The Container **2**
- Basic Usage 2
- Getting More Advanced 3
- Inversion of Control 3
- Real-World Usage 5

Dependency Injection **7**
- What is Dependency Injection? 7
- Adding Controller Dependencies 7
- Interfaces as Dependencies 9
- Why Dependency Injection? 11
- Wrapping Up 14

Core Concepts

Throughout this book, we'll be making use of some of Laravel's more powerful features.

Before jumping in, it's important to at least know about Laravel's container and how it allows us to more easily use Dependency Injection.

This chapter will cover Laravel's container, its use of Inversion of Control, and Dependency Injection.

The Container

The `Illuminate\Foundation\Application` class ties all of Laravel together. This class is a *container* - it can "contain" data, objects, classes and even closures.

Basic Usage

To see how the container works, let's run through an exercise in our routes file.

Laravel's container implements `ArrayAccess`, and so we know we can access it like an array. Here we'll see how we can access it like an associative array.

File: app/routes.php

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

Going to the `/container` route, We'll get this result:

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

So, we can see that the `Application`, while still a class with attributes and methods, is also accessible like an array!

Facades

Confused as to what `App::getFacadeRoot()` is doing? The `App` class is a Facade. This allows us to use it anywhere, accessing it in a static manner. However, it's actually not a static class. `getFacadeRoot` will get the real instance of the class, which we needed to do in order to use it like an array in this example.

See this and other Facades in the `Illuminate\Support\Facades` namespace.

Getting More Advanced

Now, let's get a little fancier with the container and assign a closure:

File: app/routes.php

```
Route::get('/container', function()
{
        // Get Application instance
        $app = App::getFacadeRoot();

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

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

Once again, run the `/container` route and we'll see:

```
Hello, World!
```

While seemingly simple, this is actually quite powerful. This is, in fact, the basis for how the separate Illuminate packages interact with each other in order to make up the Laravel framework.

Later we'll see how Service Providers bind items to the container, acting as the glue between the various Illuminate packages.

Inversion of Control

Laravel's `Container` class has more up its sleeve than simply masquerading as an array. It also can function as an Inversion of Control (IoC) container.

Inversion of Control is a technique which let's us define how our application should implement a class or interface. For instance, if our application has a dependency `FooInterface`, and we want to use an implementing class `ConcreteFoo`, the IoC container is where we define that implementation.

Let's see a basic example of how that works using our `/container` route once again.

First, we'll setup some classes - an interface and an implementing class. For simplicity, these can go right into the `app/routes.php` file:

File: app/routes.php

```
interface GreetableInterface {

    public function greet();

}

class HelloWorld implements GreetableInterface {

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

Now, let's use these classes with our container to see what we can do. First, I'll introduce the concept of "binding".

File: app/routes.php

```
Route::get('/container', function()
{
    // Get Application instance
    $app = App::getFacadeRoot();

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

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

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

Instead of using the array-accessible `$app['GreetableInterface']`, we used the `bind()` method.

This is using Laravel's IoC container to return the class `HelloWorld` anytime it's asked for `GreetableInterface`.

In this way, we can "swap out" implementations! For example, instead of `HelloWorld`, I could make a `GoodbyeCruelWorld` implementation and decide to have the container return it whenever `GreetableInterface` was asked for.

This goes towards maintainability in our applications. Using the container, we can (ideally) swap out implementations in one location without affecting other areas of our application code.

Real-World Usage

Where do you put all these bindings in your applications? If you don't want to litter your `start.php`, `filters.php`, `routes.php` and other bootstrapping files with bindings, then you can use Service Provider classes.

Service Providers are created specifically to register bindings to Laravel's container. In fact, nearly all Illuminate packages use a Service Provider to do just that.

Let's see an example of how Service Providers are used within an `Illuminate` package. We'll examine the Pagination package.

First, here is the Pagination Service Provider's `register()` method:

Illuminate\Pagination\PaginationServiceProvider.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 }
```

The `register()` method is automatically called on each Service Provider specified within the `app/config/app.php` file.

So, what's going on in this `register()` method? First and foremost, it registers the "paginator" instance to the container. This will make `$app['paginator']` and `App::make('paginator')` available for use by other areas of the application.

Next, it's defining the 'paginator' instance as the returned result of a closure, just as we did in the 'say_hi' example.

Don't be confused by the use of `$this->app->share()`. The Share method simply provides a way for the closure to be used as a singleton, similar to calling `$this->app->instance('paginator', new Environment)`.

This closure creates a new `Pagination\Environment` object, sets a configuration value on it and returns it.

You likely noticed that the Service Provider uses other application bindings! The `PaginationEnvironment` class clearly takes some dependencies in its constructor method - a request object `$app['request']`, a view object `$app['view']`, and a translator `$app['translator']`. Luckily, those bindings are created in other packages of Illuminate, defined in various Service Providers.

We can see, then, how the various Illuminate packages interact with each other. Because they are bound to the application container, we can use them in other packages (or our own code!), without actually tying our code to a specific class.

Dependency Injection

Now that we see how the container works, let's see how we can use it to implement Dependency Injection in Laravel.

What is Dependency Injection?

Dependency Injection is the act of adding (injecting) any dependencies into a class, rather than instantiating them somewhere within the class code itself. Often, dependencies are defined as type-hinted parameters of a constructor method.

Take this constructor method, for example:

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

By type-hinting `HelloWorld` as a parameter, we're explicitly stating that an instance of `HelloWorld` is a class dependency.

This is opposite of direct instantiation:

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

If you find yourself asking *why* Dependency Injection is used, this Stack Overflow answer[1] is a great place to start. I'll cover some benefits of it in the following examples.

Next, we'll see an example of Dependency Injection in action, using Laravel's IoC container.

Adding Controller Dependencies

This is a very common use case within Laravel.

Normally, if we set a controller to expect a class in its constructor method, we also need to add those dependencies when the class is created. However, what happens when you define a dependency on a Laravel controller? We would need to instantiate the controller somewhere ourselves:

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

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

That's great, but we don't directly instantiate a controller within Laravel - the router handles it for us.

We can still, however, inject controller dependencies with the use of Laravel's IoC container!

Keeping the same `GreetableInterface` and `HelloWorld` classes from before, let's now imagine we bind our `/container` route to a controller:

File: app/routes.php

```
interface GreetableInterface {

    public function greet();

}

class HelloWorld implements GreetableInterface {

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

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

Now in our new controller, we can set `HelloWorld` as a parameter in the constructor method:

File: app/controllers/ContainerController.php

```
<?php

class ContainerController extends BaseController {

    protected $greeter;

    // Class dependency: HelloWorld
    public function __construct(HelloWorld $greeter)
    {
        $this->greeter = $greeter;
```

```
    }

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

}
```

Now head to your `/container` route and you should, once again, see:

```
Hello, World!
```

Note, however, that we did NOT bind anything to the container. It simply "just worked" - an instance of `HelloWorld` was passed to the controller!

This is because the IoC container will automatically attempt to resolve any dependency set in the constructor method of a controller. Laravel will inject specified dependencies for us!

Interfaces as Dependencies

We're not done, however. Here is what we're building up to!

What if, instead of specifying `HelloWorld` as the controller's dependency, we specified the interface `GreetableInterface`?

Let's see what that would look like:

File: app/controllers/ContainerController.php

```
<?php

class ContainerController extends BaseController {

    protected $greeter;

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

```

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

}
```

If we try to run this as-is, we'll get an error:

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

The class `GreetableInterface` is of course not instantiable, as it is an interface. We can see, however, that Laravel is attempting to instantiate it in order to resolve the class dependency.

Let's fix that - when the container sees that our controller depends on an instance of `GreetableInterface`, we'll use the container's `bind()` method to tell Laravel to give the controller and instance of `HelloWorld`:

File: app/routes.php

```
interface GreetableInterface {

    public function greet();

}

class HelloWorld implements GreetableInterface {

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

// Binding HelloWorld when asked for
// GreetableInterface here!!
App::bind('GreetableInterface', 'HelloWorld');

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

Now re-run your `/container` route, you'll see `Hello, World!` once again!

Note that I didn't use a closure to bind `HelloWorld` - You can simply pass the concrete class name as a string if you wish. A closure is useful when your implementation has its own dependencies that need to be passed into its constructor method.

Why Dependency Injection?

Why would we want to specify an interface as a dependency instead of a concrete class?

We want to because we need any class dependency given to the constructor to be a subclass of an interface. In this way, we can safely use any implementation - the method we need will always be available.

Put succinctly, **we can change the implementation at will, without effecting other portions of our application code**.

Here's an example. It's something I've had to do many times in real applications.

Don't copy and paste this example. I'm omitting some details, such as using configuration variables for API keys, to clarify the point.

Let's say our application sends emails using Amazon's AWS. To accomplish this, we have defined an `Emailer` interface and an implementing class `AwsEmailer`:

```
1  interface Emailer {
2
3      public function send($to, $from, $subject, $message);
4  }
5
6  class AwsEmailer implements Emailer {
7
8      protected $aws;
9
10     public function __construct(AwsSDK $aws)
11     {
12         $this->aws = $aws;
13     }
14
15     public function send($to, $from, $subject, $message)
16     {
```

```
        $this->aws->addTo($to)
            ->setFrom($from)
            ->setSubject($subject)
            ->setMessage($message);
            ->sendEmail();
    }
}
```

We bind `Emailer` to the `AwsEmailer` implementation:

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

A controller uses the `Emailer` interface as a dependency:

File: app/controllers/EmailController.php

```
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('/');
    }

}
```

Let's further pretend that someday down the line, our application grows in scope and needs some more functionality than AWS provides. After some searching and weighing of options, you decide on SendGrid.

How do you then proceed to change your application over to SendGrid? Because we used interfaces and Laravel's IoC container, switching to SendGrid is easy!

First, make an implementation of `Emailer` which uses 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);
    }
}
```

Next, (and lastly!), set the application to use SendGrid rather than Aws. Because we have our call to `bind()` in the IoC container, changing the implementation of `Emailer` from `AwsEmailer` to `SendGridEmailer` is as simple as this *one* change:

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

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

Note that we did this all without changing a line of code elsewhere in our application. Enforcing the use of the interface `Emailer` as a dependency guarantees that any class injected will have the `send()` method available.

We can see this in our example. The controller still called `$this->emailer->send()` without having to be modified when we switched from `AwsEmailer` to `SendGridEmailer` implementations.

Wrapping Up

Dependency Injection and Inversion of Control are patterns used over and over again in Laravel development.

As you'll see, we'll define a lot of interfaces in order to make our code more maintainable, and help in testing. Laravel's IoC container makes this easy for us.