From Apprentice To Artisan

Advanced Application Architecture With Laravel 4

By Taylor Otwell

Laravel: From Apprentice To Artisan

Advanced Architecture With Laravel 4

Taylor Otwell

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

This version was published on 2013-09-04

Contents

Dependency Injection

The Problem

The foundation of the Laravel framework is its powerful IoC container. To truly understand the framework, a strong grasp of the container is necessary. However, we should note that an IoC container is simply a convenience mechanism for achieving a software design pattern: *dependency injection*. A container is not necessary to perform dependency injection, it simply makes the task easier.

First, let's explore why dependency injection is beneficial. Consider the following class and method:

```
class UserController extends BaseController {

        public function getIndex()
        {
                $users = User::all();

                return View::make('users.index', compact('users'));
        }

}
```

While this code is concise, we are unable to test it without hitting an actual database. In other words, the Eloquent ORM is *tightly coupled* to our controller. We have no way to use or test this controller without also using the entire Eloquent ORM, including hitting a live database. This code also violates a software design principle commonly called *separation of concerns*. Simply put: our controller knows too much. Controllers do not need to know *where* data comes from, but only how to access it. The controller doesn't need to know that the data is available in MySQL, but only that it is available *somewhere*.

Separation Of Concerns

Every class should have a single responsibility, and that responsibility should be entirely encapsulated by the class.

So, it will be beneficial for us to decouple our web layer (controller) from our data access layer completely. This will allow us to migrate storage implementations easily, as well as make our code easier to test. Think of the "web" as just a transport layer into your "real" application.

Imagine that your application is like a monitor with a variety of cable ports. You can access the monitor's functionality via HDMI, VGA, or DVI. Think of the Internet as just a cable into your application. The bulk of a monitor's functionality is independent of the cable. The cable is just a transport mechanism just like HTTP is a transport mechanism for your application. So, we don't want to clutter up our transport mechanism (the controller) with application logic. This will allow any transport layer, such as an API or mobile application, to access our application logic.

So, instead of coupling our controller to the Eloquent ORM, let's inject a repository class.

Build A Contract

First, we'll define an interface and a corresponding implementation:

```
interface UserRepositoryInterface {

        public function all();

}

class DbUserRepository implements UserRepositoryInterface {

        public function all()
        {
                return User::all()->toArray();
        }

}
```

Next, we'll inject an implementation of this interface into our controller:

```
class UserController extends BaseController {

        public function __construct(UserRepositoryInterface $users)
        {
                $this->users = $users;
        }

        public function getIndex()
        {
                $users = $this->users->all();

                return View::make('users.index', compact('users'));
```

```
13          }
14
15  }
```

Now our controller is completely ignorant of where our user data is being stored. In this case, ignorance is bliss! Our data could be coming from MySQL, MongoDB, or Redis. Our controller doesn't know the difference, nor should it care. Just by making this small change, we can test our web layer independent of our data layer, as well as easily switch our storage implementation.

Respect Boundaries

Remember to respect responsibility boundaries. Controllers and routes serve as a mediator between HTTP and your application. When writing large applications, don't clutter them up with your domain logic.

To solidify our understanding, let's write a quick test. First, we'll mock the repository and bind it to the application IoC container. Then, we'll ensure that the controller properly calls the repository:

```php
public function testIndexActionBindsUsersFromRepository()
{
        // Arrange...
        $repository = Mockery::mock('UserRepositoryInterface');
        $repository->shouldReceive('all')->once()->andReturn(array('foo'));
        App::instance('UserRepositoryInterface', $repository);

        // Act...
        $response = $this->action('GET', 'UserController@getIndex');

        // Assert...
        $this->assertResponseOk();
        $this->assertViewHas('users', array('foo'));
}
```

Are You Mocking Me

In this example, we used the `Mockery` mocking library. This library provides a clean, expressive interface for mocking your classes. Mockery can be easily installed via Composer.

Taking It Further

Let's consider another example to solidify our understanding. Perhaps we want to notify customers of charges to their account. We'll define two interfaces, or contracts. These contracts will give us the flexibility to change out their implementations later.

```php
interface BillerInterface {
        public function bill(array $user, $amount);
}

interface BillingNotifierInterface {
        public function notify(array $user, $amount);
}
```

Next, we'll build an implementation of our `BillerInterface` contract:

```
class StripeBiller implements BillerInterface {

        public function __construct(BillingNotifierInterface $notifier)
        {
                $this->notifier = $notifier;
        }

        public function bill(array $user, $amount)
        {
                // Bill the user via Stripe...

                $this->notifier->notify($user, $amount);
        }

}
```

By separating the responsibilities of each class, we're now able to easily inject various notifier implementations into our billing class. For example, we could inject a `SmsNotifier` or an `EmailNotifier`. Our biller is no longer concerned with the implementation of notifying, but only the contract. As long as a class abides by its contract (interface), the biller will gladly accept it. Furthermore, not only do we get added flexibility, we can now test our biller in isolation from our notifiers by injecting a mock `BillingNotifierInterface`.

Be The Interface

While writing interfaces might seem to a lot of extra work, they can actually make your development more rapid. Use interfaces to mock and test the entire back-end of your application before writing a single line of implementation!

So, how do we *do* dependency injection? It's simple:

```
$biller = new StripeBiller(new SmsNotifier);
```

That's dependency injection. Instead of the biller being concerned with notifying users, we simply pass it a notifier. A change this simple can do amazing things for your applications. Your code immediately becomes more maintainable since class responsibilities are clearly delineated. Also, testability will skyrocket as you can easily inject mock dependencies to isolate the code under test.

But what about IoC containers? Aren't they necessary to do dependency injection? Absolutely not! As we'll see in the following chapters, containers make dependency injection easier to manage, but they are not a requirement. By following the principles in this chapter, you can practice dependency injection in any of your projects, regardless of whether a container is available to you.

Too Much Java?

A common criticism of use of interfaces in PHP is that it makes your code too much like "Java". What people mean is that it makes the code very verbose. You must define an interface and an implementation, which leads to a few extra key-strokes.

For small, simple applications, this criticism is probably valid. Interfaces are often unnecessary in these applications, and it is "OK" to just couple yourself to an implementation you know won't change. There is no need to use interfaces if you are certain your implementation will not change. Architecture astronauts will tell you that you can "never be certain". But, let's face it, sometimes you are.

Interfaces are very helpful in large applications, and the extra key-strokes pale in comparison to the flexibility and testability you will gain. The ability to quickly swap implementations of a contract will "wow" your manager, and allow you to write code that easily adapts to change.

So, in conclusion, keep in mind that this book presents a very "pure" architecture. If you need to scale it back for a small application, don't feel guilty. Remember, we're all trying to "code happy". If you're not enjoying what you're doing or you are programming out of guilt, step back and re-evaluate.