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

Author's Note

Since creating the Laravel framework, I have received numerous requests for a book containing guidance on building well-architected, complex applications. As each application is unique, such a book requires that the counsel remains general, yet practical and easily applicable to a variety of projects.

So, we will begin by covering the foundational elements of dependency injection, then take an in-depth look at service providers and application structure, as well as an overview of the SOLID design principles. A strong knowledge of these topics will give you a firm foundation for all of your Laravel projects.

If you have any further questions about advanced architecture on Laravel, or wish to see something added to the book, please e-mail me. I plan to expand the book further based on community feedback, so your ideas matter!

Finally, thank you so much for being a part of the Laravel community. You have all helped make PHP development more enjoyable and exciting for thousands of people around the world. Code happy!

Taylor Otwell

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.

The IoC Container

Basic Binding

Now that we've learned about dependency injection, let's explore inversion of control containers. IoC containers make managing your class dependencies much more convenient, and Laravel ships with a very powerful container. The IoC container is the central piece of the Laravel framework, and it is what allows all of the framework's components to work together. In fact, the Laravel `Application` class extends the `Container` class!

IoC Container

Inversion of control containers make dependency injection more convenient. How to resolve a given class or interface is defined once in the container, which manages resolving and injecting those objects throughout your application.

In a Laravel application, the IoC container can be accessed via the `App` facade. The container has a variety of methods, but we'll start with the most basic. Let's continue to use our `BillerInterface` and `BillingNotifierInterface` from the previous chapter, and assume that our application is using Stripe[1] to process payments. We can bind the Stripe implementation of the interface to the container like this:

```
App::bind('BillerInterface', function()
{
        return new StripeBiller(App::make('BillingNotifierInterface'));
})
```

Notice that within our `BillerInterface` resolver, we also resolve a `BillingNotifierInterface` implementation. Let's define that binding as well:

```
App::bind('BillingNotifierInterface', function()
{
        return new EmailBillingNotifier;
});
```

[1] https://stripe.com

So, as you can see, the container is a place to store Closures that resolve various classes. Once a class has been registered with the container, we can easily resolve it from anywhere in our application. We can even resolve other container bindings within a resolver.

Have Acne?

The Laravel IoC container is a drop-in replacement for the Pimple[2] IoC container by Fabien Potencier. So, if you're already using Pimple on a project, feel free to upgrade to the Illuminate Container[3] component for a few more features!

Once we're using the container, we can switch interface implementations with a single line change. For example, consider the following:

```
class UserController extends BaseController {

        public function __construct(BillerInterface $biller)
        {
                $this->biller = $biller;
        }

}
```

When this controller is instantiated via the IoC container, the `StripeBiller`, which includes the `EmailBillingNotifier`, will be injected into the instance. Now, if we want to change our notifier implementation, we can simply change the binding to this:

```
App::bind('BillingNotifierInterface', function()
{
        return new SmsBillingNotifier;
});
```

Now, it doesn't matter where the notifier is resolved in our application, we will now always get the `SmsBillingNotifier` implementation. Utilizing this architecture, our application can be rapidly shifted to new implementations of various services.

Being able to change implementations of an interface with a single line is amazingly powerful. For example, imagine we want to change our SMS service from a legacy provider to Twilio. We can develop a new Twilio implementation of the notifier and swap our binding. If we have problems with the transition to Twilio, we can quickly change back to the legacy provider by making a single IoC binding change. As you can see, the benefits of using dependency injection go beyond what is

[2] https://github.com/fabpot/pimple
[3] https://github.com/illuminate/container

immediately obvious. Can you think of more benefits for using dependency injection and an IoC container?

Sometimes you may wish to resolve only one instance of a given class throughout your entire application. This can be achieved via the `singleton` method on the container class:

```
App::singleton('BillingNotifierInterface', function()
{
        return new SmsBillingNotifier;
});
```

Now, once the container has resolved the billing notifier once, it will continue to use that same instance for all subsequent requests for that interface.

The `instance` method on the container is similar to ‘singleton; however, you are able to pass an already existing object instance. The instance you give to the container will be used each time the container needs an instance of that class:

```
$notifier = new SmsBillingNotifier;

App::instance('BillingNotifierInterface', $notifier);
```

Now that we’re familiar with basic container resolution using Closures, let’s dig into its most powerful feature: the ability to resolve classes via reflection.

Stand Alone Container

Working on a project that isn’t built on Laravel? You can still utilize Laravel’s IoC container by installing the `illuminate/container` package via Composer!

Reflective Resolution

One of the most powerful features of the Laravel container is its ability to automatically resolve dependencies via reflection. Reflection is the ability to inspect a classes and methods. For example, the PHP `ReflectionClass` class allows you to inspect the methods available on a given class. The PHP method `method_exists` is also a form of reflection. To play with PHP’s reflection class, try the following code on one of your classes:

```
$reflection = new ReflectionClass('StripeBiller');

var_dump($reflection->getMethods());

var_dump($reflection->getConstants());
```

By leveraging this powerful feature of PHP, the Laravel IoC container can do some interesting things! For instance, consider the following class:

```
class UserController extends BaseController {

        public function __construct(StripeBiller $biller)
        {
                $this->biller = $biller;
        }

}
```

Note that the controller is type-hinting the `StripeBiller` class. We are able to retrieve this type-hint using reflection. When the Laravel container does not have a resolver for a class explicitly bound, it will try to resolve the class via reflection. The flow looks like this:

1. Do I have a resolver for `StripeBiller`?
2. No resolver? Reflect into `StripeBiller` to determine if it has dependencies.
3. Resolve any dependencies needed by `StripeBiller` (recursive).
4. Instantiate new `StripeBiller` instance via `ReflectionClass->newInstanceArgs()`.

As you can see, the container is doing a lot of heavy lifting for you, which saves you from having to write resolvers for every single one of your classes. This is one of the most powerful and unique features of the Laravel container, and having a strong grasp of this capability is very beneficial when building large Laravel applications.

Now, let's modify our controller a bit. What if it looked like this?

```
class UserController extends BaseController {

        public function __construct(BillerInterface $biller)
        {
                $this->biller = $biller;
        }

}
```

Assuming we have not explicitly bound a resolver for `BillerInterface`, how will the container know what class to inject? Remember, interfaces can't be instantiated since they are just contracts. Without us giving the container any more information, it will be unable to instantiate this dependency. We need to specify a class that should be used as the default implementation of this interface, and we may do so via the `bind` method:

```
App::bind('BillerInterface', 'StripeBiller');
```

Here, we are passing a string instead of a Closure, and this string tells the container to always use the `StripeBiller` class anytime it needs an implementation of the `BillerInterface` interface. Again, we're gaining the ability to switch implementations of services with a simple one-line change to our container binding. For example, if we need to switch to Balanced Payments as our billing provider, we simply write a new `BalancedBiller` implementation of `BillerInterface`, and change our container binding:

```
App::bind('BillerInterface', 'BalancedBiller');
```

Automatically, our new implementation will be used throughout our application!

When binding implementations to interfaces, you may also use the `singleton` method so the container only instantiates one instance of the class per request cycle:

```
App::singleton('BillerInterface', 'StripeBiller');
```

Master The Container

Want to learn even more about the container? Read through its source! The container is only one class: `Illuminate\Container\Container`. Read over the source code to gain a deeper understanding of how the container works under the hood.

Interface As Contract

Strong Typing & Water Fowl

In the previous chapters, we covered the basics of dependency injection: what it is; how it is accomplished; and several of its benefits. The examples in the previous chapters also demonstrated the injection of *interfaces* into our classes, so before proceeding further, it will be beneficial to talk about interfaces in depth, as many PHP developers are not familiar with them.

Before I was a PHP programmer, I was a .NET programmer. Do I love pain or what? In .NET, interfaces are everywhere. In fact, many interfaces are defined as part of the core .NET framework itself, and for good reason: many of the .NET languages, such as C# and VB.NET, are *strongly typed*. In general, you must define the *type* of object or primitive you will be passing into a method. For example, consider the following C# method:

```
public int BillUser(User user)
{
        this.biller.bill(user.GetId(), this.amount)
}
```

Note that we were forced to define not only what *type* of arguments we will be passing to the method, but also what the method itself will be returning. C# is encouraging *type safety*. We will not be allowed to pass anything other than an User object into our `BillUser` method.

However, PHP is generally a *duck typed* language. In a duck typed language, an object's available methods determine the way it may be used, rather than its inheritance from a class or implementation of an interface. Let's look at an example:

```
public function billUser($user)
{
        $this->biller->bill($user->getId(), $this->amount);
}
```

In PHP, we did not have to tell the method what type of argument to expect. In fact, we can pass any type, so long as the object responds to the `getId` method. This is an example of duck typing. If the object looks like a duck, and quacks like a duck, it must be a duck. Or, in this case, if the object looks like a user, and acts like a user, it must be one.

But, does PHP have *any* strongly typed style features? Of course! PHP has a mixture of both strong and duck typed constructs. To illustrate this, let's re-write our `billUser` method:

```
public function billUser(User $user)
{
        $this->biller->bill($user->getId(), $amount);
}
```

After adding the `User` type-hint to our method signature, we can now gurantee that all objects passed to `billUser` either are a `User` instance, or extend the `User` class.

There are advantages and disadvantages to both typing systems. In strongly typed languages, the compiler can often provide you with thorough compile-time error checking, which is extremely helpful. Method inputs and outputs are also much more explicit in a strongly typed language.

At the same time, the explicit nature of strong typing can make it rigid. For example, dynamic methods such as `whereEmailOrName` that the Eloquent ORM offers would be impossible to implement in a strongly typed language like C#. Try to avoid heated discussions on which paradigm is better, and remember the advantages and disadvantages of each. It is not "wrong" to use the strong typing available in PHP, nor is it wrong to use duck typing. What is wrong is exclusively using one or the other without considering the particular problem you are trying to solve.

A Contract Example

Interfaces are contracts. Interfaces do not contain any code, but simply define a set of methods that an object must implement. If an object implements an interface, we are guaranteed that every method defined by the interface is valid and callable on that object. Since the contract guarantees the implementation of certain methods, type safety becomes more flexible via *polymorphism*.

Polywhat?

Polymorphism is a big word that essentially means an entity can have multiple forms. In the context of this book, we mean that an interface can have multiple implementations. For example, a `UserRepositoryInterface` could have a MySQL and a Redis implementation, and both implementations would qualify as a `UserRepositoryInterface` instance.

To illustrate the flexibility that interfaces introduce into strongly typed languages, let's write some simple code that books hotel rooms. Consider the following interface:

```
interface ProviderInterface {
        public function getLowestPrice($location);
        public function book($location);
}
```

When our user books a room, we'll want to log that in our system, so let's add a few methods to our `User` class:

```
class User {

        public function bookLocation(ProviderInterface $provider, $location)
        {
                $amountCharged = $provider->book($location);

                $this->logBookedLocation($location, $amountCharged);
        }

}
```

Since we are type hinting the `ProviderInterface`, our `User` can safely assume that the `book` method will be available. This gives us the flexibility to re-use our `bookLocation` method regardless of the hotel provider the user prefers. Finally, let's write some code that harnesses this flexibility:

```
$location = 'Hilton, Dallas';

$cheapestProvider = $this->findCheapest($location, array(
        new PricelineProvider,
        new OrbitzProvider,
));

$user->bookLocation($cheapestProvider, $location);
```

Wonderful! No matter what provider is the cheapest, we are able to simply pass it along to our `User` instance for booking. Since our `User` is simply asking for an object instances that abides by the `ProviderInterface` contract, our code will continue to work even if we add new provider implementations.

Forget The Details

Remember, interfaces don't actually *do* anything. They simply define a set of methods that an implementing class **must** have.

Interfaces & Team Development

When your team is building a large application, pieces of the application progress at different speeds. For example, imagine one developer is working on the data layer, while another developer is working on the front-end and web / controller layer. The front-end developer wants to test his controllers, but the back-end is progressing slowly. However, what if the two developers can agree on an interface, or contract, that the back-end classes will abide by, such as the following:

```
interface OrderRepositoryInterface {
        public function getMostRecent(User $user);
}
```

Once this contract has been agreed upon, the front-end developer can test his controllers, even if no real implementation of the contract has been written! This allows components of the application to be built at different speeds, while still allowing for proper unit tests to be written. Further, this approach allows entire implementations to change without breaking other, unrelated components. Remember, ignorance is bliss. We don't want our classes to know *how* a dependency does something, but only that it *can*. So, now that we have a contract defined, let's write our controller:

```
class OrderController {

        public function __construct(OrderRepositoryInterface $orders)
        {
                $this->orders = $orders;
        }

        public function getRecent()
        {
                $recent = $this->orders->getMostRecent(Auth::user());

                return View::make('orders.recent', compact('recent'));
        }

}
```

The front-end developer could even write a "dummy" implementation of the interface, so the application's views can be populated with some fake data:

```
class DummyOrderRepository implements OrderRepositoryInterface {

        public function getMostRecent(User $user)
        {
                return array('Order 1', 'Order 2', 'Order 3');
        }

}
```

Once the dummy implementation has been written, it can be bound in the IoC container so it is used throughout the application:

```
App::bind('OrderRepositoryInterface', 'DummyOrderRepository');
```

Then, once a "real" implementation, such as `RedisOrderRepository`, has been written by the back-end developer, the IoC binding can simply be switched to the new implementation, and the entire application will begin using orders that are stored in Redis.

Interface As Schematic

Interfaces are helpful for developing a "skeleton" of defined functionality provided by your application. Use them during the design phase of a component to facilitate design discussion amongst your team. For example, define a `BillingNotifierInterface` and discuss its methods with your team. Use the interface to agree on a good API before you actually write any implementation code!

Service Providers

As Bootstrapper

A Laravel service provider is a class that registers IoC container bindings. In fact, Laravel ships with over a dozen service providers that manage the container bindings for the core framework components. Almost every component of the framework is registered with the container via a provider. A list of the providers currently being used by your application can be found in the `providers` array within the `app/config/app.php` configuration file.

A service provider must have at least one method: `register`. The `register` method is where the provider binds classes to the container. When a request enters your application and the framework is booting up, the `register` method is called on the providers listed in your configuration file. This happens very early in the application life-cycle so that all of the Laravel services will be available to you when your own bootstrap files, such as those in the `start` directory, are loaded.

Register Vs. Boot

Never attempt to use services in the `register` method. This method is only for binding objects into the IoC container. All resolution and interaction with the bound classes must be done inside the `boot` method.

Some third-party packages you install via Composer will ship with a service provider. The package's installation instructions will typically tell you to register their provider with your application by adding it to the `providers` array in your configuration file. Once you've registered the package's provider, it is ready for use.

Package Providers

Not all third-party packages require a service provider. In fact, no package *requires* a service provider, as service providers generally just boostrap components so they are ready for use immediately. They are simply a convenient place to organize bootstrap code and container bindings.

Deferred Providers

Not every provider that is listed in your `providers` configuration array will be instantiated on every request. This would be detrimental to performance, especially if the services the provider registers

are not needed for the request. For example, the `QueueServiceProvider` services are not needed on every request, but only on requests where the queue is actually utilized.

In order to only instantiate the providers that are needed for a given request, Laravel generates a "service manifest" that is stored in the `app/storage/meta` directory. This manifest lists all of the service providers for the application, as well as the names of the container bindings they register. So, when the application asks the container for the `queue` container binding, Laravel knows that it needs to instantiate and run the `QueueServiceProvider` because it is listed as providing the `queue` service in the manifest. This allows the framework to lazy-load service providers for each request, greatly increasing performance.

Manifest Generation

When you add a service provider to the `providers` array, Laravel will automatically regenerate the service manifest file during the next request to the application.

As you have time, take a look through the service manifest so you are aware of its contents. Understanding how this file is structured will be helpful if you ever need to debug a deferred service provider.

As Organizer

One of the keys to building a well architected Laravel application is learning to use service providers as an organizational tool. When you are registering many classes with the IoC container, all of those bindings can start to clutter your `app/start` files. Instead of doing container registrations in those files, create service providers that register related services.

Get It Started

Your application's "start" files are all stored in the `app/start` directory. These are "bootstrap" files that are loaded based on the type of request entering your application. Start files named after an environment are loaded after the global `start.php` file. Finally, the `artisan.php` start file is loaded when any console command is executed.

Let's explore an example. Perhaps our application is using Pusher[4] to push messages to clients via WebSockets. In order to decouple ourselves from Pusher, it will be beneficial to create an `EventPusherInterface`, and a `PusherEventPusher` implementation. This will allow us to easily change WebSocket providers down the road as requirements change or our application grows.

[4] http://pusher.com

```
interface EventPusherInterface {
        public function push($message, array $data = array());
}

class PusherEventPusher implements EventPusherInterface {

        public function __construct(PusherSdk $pusher)
        {
                $this->pusher = $pusher;
        }

        public function push($message, array $data = array())
        {
                // Push message via the Pusher SDK...
        }

}
```

Next, let's create an `EventPusherServiceProvider`:

```
use Illuminate\Support\ServiceProvider;

class EventPusherServiceProvider extends ServiceProvider {

        public function register()
        {
                $this->app->singleton('PusherSdk', function()
                {
                        return new PusherSdk('app-key', 'secret-key');
                });

                $this->app->singleton('EventPusherInterface', 'PusherEventPusher');
        }

}
```

Great! Now we have a clean abstraction over event pushing, as well as a convenient place to register this, and other related bindings, in the container. Finally, we just need to add the `EventPusherServiceProvider` to our `providers` array in the `app/config/app.php` configuration file. Now we are ready to inject the `EventPusherInterface` into any controller or class within our application.

Should You Singleton?

You will need to evaluate whether to bind classes with `bind` or `singleton`. If you only want one instance of the class to be created per request cycle, use `singleton`. Otherwise, use `bind`.

Note that a service provider has an `$app` instance available via the base `ServiceProvider` class. This is a full `Illuminate\Foundation\Application` instance, which inherits from the `Container` class, so we can call all of the IoC container methods we are used to. If you prefer to use the `App` facade inside the service provider, you may do that as well:

```
1  App::singleton('EventPusherInterface', 'PusherEventPusher');
```

Of course, service providers are not limited to registering certain kinds of services. We could use them to register our cloud file storage services, database access services, a custom view engine such as Twig, etc. They are simply bootstrapping and organizational tools for your application. Nothing more.

So, don't be scared to create your own service providers. They are not something that should be strictly limited to distributed packages, but rather are great organizational tools for your own applications. Be creative and use them to bootstrap your various application components.

Booting Providers

After all providers have been registered, they are "booted". This will fire the `boot` method on each provider. A common mistake when using service providers is attempting to use the services provided by another provider in the `register` method. Since, within the `register` method, we have no gurantee all other providers have been loaded, the service you are trying to use may not be available yet. So, service provider code that uses other services should always live in the `boot` method. The `register` method should **only** be used for, you guessed it, registering services with the container.

Within the boot method, you may do whatever you like: register event listeners, include a routes file, register filters, or anything else you can imagine. Again, use the providers as organizational tools. Perhaps you wish to group some related event listeners together? Placing them in the `boot` method of a service provider would be a great approach! Or, you could include an "events" or "routes" PHP file:

```
public function boot()
{
        require_once __DIR__.'/events.php';

        require_once __DIR__.'/routes.php';
}
```

Now that we've learned about dependency injection and a way to organize our projects around providers, we have a great foundation for building well-architected Laravel applications that are maintainable and testable. Next, we'll explore how Laravel itself uses providers, and how the framework works under the hood!

Don't Box Yourself In

Remember, don't assume that service providers are something that only packages use. Create your own to help organize your application's services.

Providing The Core

By now you have probably noticed that your application already has many service providers registered in the `app` configuration file. Each of these service providers bootstraps a part of the framework core. For example, the `MigrationServiceProvider` bootstraps the various classes that run migrations, as well as the migration Artisan commands. The `EventServiceProvider` bootstraps and registers the event dispatcher class. Some of these providers are larger than others, but they each bootstrap a piece of the core.

Meet Your Providers

One of the best ways to improve your understanding of the Laravel core is to read the source for the core service providers. If you are familiar with the function of service providers and what each core service provider registers, you will have an incredibly better understanding of how Laravel works under the hood.

Most of the core providers are deferred, meaning they do not load on every request; however, some, such as the `FilesystemServiceProvider` and `ExceptionServiceProvider` load on every request to your application since they bootstrap very foundational pieces of the framework. One might say that the core service providers and the application container *are* Laravel. They are what ties all of the various pieces of the framework together into a single, cohesive unit. These providers are the building blocks of the framework.

As mentioned previously, if you wish to gain a deeper understanding of how the framework works, read the source code of the core service providers that ship with Laravel. By reading through these

providers, you will gain a solid understanding of how the framework is put together, as well as what each provider offers to your application. Furthermore, you will be better prepared to contribute to Laravel itself!

Application Structure

Introduction

Where does this class belong? This question is extremely common when building applications on a framework. Many developers ask this question because they have been told that "Model" means "Database". So, developers have their controllers that interact with HTTP, models which do *something* with the database, and views which contain their HTML. But, what about classes that send e-mail? What about classes that validate data? What about classes that call an API to gather information? In this chapter, we'll cover good application structure in the Laravel framework and break down some of the common mental roadblocks that hold developers back from good design.

MVC Is Killing You

The biggest roadblock towards developers achieving good design is a simple acronym: M-V-C. Models, views, and controllers have dominated web framework thinking for years, in part because of the popularity of Ruby on Rails. However, ask a developer to define "model". Usually, you'll hear a few mutters and the word "database". Supposedly, the model *is* the database. It's where all your database *stuff* goes, whatever that means. But, as you quickly learn, your application needs a lot more logic than just a simple database access class. It needs to do validation, call external services, send e-mails, and more.

What Is A Model?

The word "model" has become so ambiguous that it has no meaning. By developing with a more specific vocabulary, it will be easier to separate our application into smaller, cleaner classes with a clearly defined responsiblity.

So, what is the solution to this dilemma? Many developers start packing logic into their controllers. Once the controllers get large enough, they need to re-use business logic that is in other controllers. Instead of extracting the logic into another class, most developers mistakenly assume they *need* to call controllers from within other controllers. This pattern is typically called "HMVC". Unfortunately, this pattern often indicates poor application design, and controllers that are much too complicated.

HMVC (Usually) Indicates Poor Design

Feel the need to call controllers from other controllers? This is often indicative of poor application design and too much business logic in your controllers. Extract the logic into a third class that can be injected into any controller.

There is a better way to structure applications. We need to wash our minds clean of all we have been taught about *models*. In fact, let's just delete the model directory and start fresh!

Bye, Bye Models

Is your `models` directory deleted yet? If not, get it out of there! Let's create a folder within our `app` directory that is simply named after our application. For this discussion, let's call our application `QuickBill`, and we'll continue to use some of the interfaces and classes we've discussed before.

Remember The Context

Remember, if you are building a very small Laravel application, throwing a few Eloquent models in the `models` directory is perfectly fine. In this chapter, we're primarily concerned with discovering more "layered" architecture suitable to large and complex projects.

So, we should have an `app/QuickBill` directory, which is at the same level in the application directory structure as `controllers` and `views`. We can create several more directories within `QuickBill`. Let's create a `Repositories` directory and a `Billing` directory. Once these directories are established, remember to register them for PSR-0 auto-loading in your `composer.json` file:

```
"autoload": {
        "psr-0": {
                "QuickBill": "app/"
        }
}
```

For now, let's put our Eloquent classes at the root of the `QuickBill` directory. This will allow us to conveniently access them as `QuickBill\User`, `QuickBill\Payment`, etc. In the `Repositories` folder would belongs classes such as `PaymentRepository` and `UserRepository`, which would contain all of our data access functions such as `getRecentPayments`, and `getRichestUser`. The `Billing` directory would contain the classes and interfaces that work with third-party billing services like Stripe and Balanced. The folder structure would look something like this:

```
// app
        // QuickBill
                // Repositories
                        -> UserRepository.php
                        -> PaymentRepository.php
                // Billing
                        -> BillerInterface.php
                        -> StripeBiller.php
                // Notifications
                        -> BillingNotifierInterface.php
                        -> SmsBillingNotifier.php
                User.php
                Payment.php
```

What About Validation

Where to perform validation often stumps developers. Consider placing validation methods on your "entity" classes (like `User.php` and `Payment.php`). Possible method names might be: `validForCreation` or `hasValidDomain`. Alternatively, you could create a `UserValidator` class within a `Validation` namespace and inject that validator into your repository. Experiment with both approaches and see what you like best!

By just getting rid of the `models` directory, you can often break down mental roadblocks to good design, allowing you to create a directory structure that is more suitable for your application. Of course, each application you build will have some similarities, since each complex application will need a data access (repository) layer, several external service layers, etc.

Don't Fear Directories

Don't be afraid to create more directories to organize your application. Always break your application into small components, each having a very focused responsibility. Thinking outside of the "model" box will help. For example, as we previously discussed, you could create a `Repositories` directory to hold all of your data access classes.

It's All About The Layers

As you may have noticed, a key to solid application design is simply separating responsibilities, or creating layers of responsibility. Controllers are responsible for receiving an HTTP request and calling the proper business layer classes. Your business / domain layer *is* your application. It contains the classes that retrieve data, validate data, process payments, send e-mails, and any other function

of your application. In fact, your domain layer doesn't need to know about "the web" at all! The web is simply a transport mechanism to access your application, and knowledge of the web and HTTP need not go beyond the routing and controller layers. Good architecture can be challenging, but will yield large profits of sustainable, clear code.

For example, instead of accessing the web request instance in a class, you could simply pass the web input from the controller. This simple change alone decouples your class from "the web", and the class can easily be tested without worrying about mocking a web request:

```
1  class BillingController extends BaseController {
2
3          public function __construct(BillerInterface $biller)
4          {
5                  $this->biller = $biller;
6          }
7
8          public function postCharge()
9          {
10                 $this->biller->chargeAccount(Auth::user(), Input::get('amount'));
11
12                 return View::make('charge.success');
13         }
14
15 }
```

Our `chargeAccount` method is now much easier to test, since we no longer have to use the `Request` or `Input` class inside of our `BillerInterface` implementation as we are simply passing the charged amount into the method as an integer.

Separation of responsibilities is one of the keys to writing maintainable applications. Always be asking if a given class knows more than it should. You should frequently ask yourself: "Should this class care about *X*?" If the answer is "no", extract the logic into another class that can be injected as a dependency.

Single Reason To Change

A helpful method of determining whether a class has too many responsibilities is to examine your reasons for changing code within that class. For example, should we need to change code within a `Biller` implementation when tweaking our notification logic? Of course not. The `Biller` implementations are concerned with billing, and should only work with notification logic via a contract. Maintaining this mindset as you are working on your code will help you quickly identify areas of an application that can be improved.

Where To Put "Stuff"

When developing applications with Laravel, you may sometimes wonder where to put "stuff". For example, where should you put "helper" functions? Where should you put event listeners? Where should you put view composers? It may be surprising for you to know that the answer is: "wherever you want!" Laravel does not have many conventions regarding where things belong on the file system. However, as this answer is not often satisfactory, we'll explore some possible locations for such things before moving on.

Helper Functions

Laravel ships with a file full of helpers functions (`support/helpers.php`). You may wish to create a similar file containing helper functions relevant to your own application and coding style. A great place to include these functions are the "start" files. Within your `start/global.php` file, which is included on every request to the application, you may simply require your own `helpers.php` file:

```
// Within app/start/global.php

require_once __DIR__.'/../helpers.php';
```

Event Listeners

Since event listeners obviously do not belong in the `routes.php` file, and can begin to clutter the "start" files, we need another location to place this code. A great option is a service provider. As we've learned, service providers are not strictly for registering bindings in the IoC container. They can be used to do all sorts of work. By grouping related event registrations inside of a service provider, the code stays neatly tucked away behind the scenes of your main application code. View composers, which are a type of event, may also be neatly grouped within service providers.

For example, a service provider that registers events might look something like this:

```
<?php namespace QuickBill\Providers;

use Illuminate\Support\ServiceProvider;

class BillingEventsProvider extends ServiceProvider {

        public function boot()
        {
                Event::listen('billing.failed', function($bill)
                {
                        // Handle failed billing event...
```

```
                });
        }

}
```

After creating the provider, we would simply add it our `providers` array within the `app/config/app.php` configuration file.

Wear The Boot

Remember, in the example above, we are using the `boot` method for a reason. The `register` method in a service provider is **only** intended for binding classes into the container.

Error Handlers

If your application has many custom error handlers, they may start to take over your "start" files. Again, like event handlers, these are best moved into a service provider. The service provider might be named something like `QuickBillErrorProvider`. Within the `boot` method of this provider you may register all of your custom error handlers. Again, this keeps boilerplate code out of the main files of your application. An error handler provider would look something like this:

```
<?php namespace QuickBill\Providers;

use App, Illuminate\Support\ServiceProvider;

class QuickBillErrorProvider extends ServiceProvider {

        public function register()
        {
                //
        }

        public function boot()
        {
                App::error(function(BillingFailedException $e)
                {
                        // Handle failed billing exceptions...
                });
        }

}
```

The Small Solution

Of course, if you only have one or two simple error handlers, keeping them in the "start" files is a reasonable and quick solution.

The Rest

In general, classes may be neatly organized using a PSR-0 structure within a directory of your application. Imperative code such as event listeners, error handlers, and other "registration" type operations may be placed inside of a service provider. With what we have learned so far, you should be able to make an educated decision on where to place a piece of code. But, don't hesitate to experiment. The beauty of Laravel is that you can make conventions that work best for you. Discover a structure that works best for your applications, and make sure to share your insights with others!

For example, as you probably noticed above, you might create a `Providers` namespace for all of your application's custom service providers, creating a directory structure like so:

```
// app
        // QuickBill
                // Billing
                // Extensions
                        // Pagination
                                -> Environment.php
                // Providers
                        -> EventPusherServiceProvider.php
                // Repositories
                User.php
                Payment.php
```

Notice that in the example above we have a `Providers` and an `Extensions` namespace. All of your application's service providers could be stored in the `Providers` directory in namespace, and the `Extensions` namespace is a convenient place to store extensions made to core framework classes.

Applied Architecture: Decoupling Handlers

Introduction

Now that we have discussed various aspects of sound application architecture using Laravel 4, let's dig into some more specifics. In this chapter, we'll discuss tips for decoupling various handlers like queue and event handlers, as well as other "event-like" structures such as route filters.

Don't Clog Your Transport Layer

Most "handlers" can be considered *transport layer* components. In other words, they receive calls through something like queue workers, a dispatched event, or an incoming request. Treat these handlers like controllers, and avoid clogging them up with the implementation details of your application.

Decoupling Handlers

To get started, let's jump right into an example. Consider a queue handler that sends an SMS message to a user. After sending the message, the handler logs that message so we can keep a history of all SMS messages we have sent to that user. The code might look something like this:

```
class SendSMS {

        public function fire($job, $data)
        {
                $twilio = new Twilio_SMS($apiKey);

                $twilio->sendTextMessage(array(
                        'to' => $data['user']['phone_number'],
                        'message' => $data['message'],
                ));

                $user = User::find($data['user']['id']);

                $user->messages()->create([
                        'to' => $data['user']['phone_number'],
                        'message' => $data['message'],
                ]);

                $job->delete();
        }

}
```

Just by examining this class, you can probably spot several problems. First, it is hard to test. The `Twilio_SMS` class is instantiated inside of the `fire` method, meaning we will not be able to inject a mock service. Secondly, we are using Eloquent directly in the handler, thus creating a second testing problem as we will have to hit a real database to test this code. Finally, we are unable to send SMS messages outside of the queue. All of our SMS sending logic is tightly coupled to the Laravel queue.

By extracting this logic into a separate "service" class, we can decouple our application's SMS sending logic from Laravel's queue. This will allow us to send SMS messages from anywhere in our application. While we are decoupling this process from the queue, we will also refactor it to be more testable.

So, let's examine an alternative:

```
class User extends Eloquent {

        /**
         * Send the User an SMS message.
         *
         * @param  SmsCourierInterface  $courier
         * @param  string  $message
         * @return SmsMessage
         */
        public function sendSmsMessage(SmsCourierInterface $courier, $message)
        {
                $courier->sendMessage($this->phone_number, $message);

                return $this->sms()->create([
                        'to' => $this->phone_number,
                        'message' => $message,
                ]);
        }

}
```

In this refactored example, we have extracted the SMS sending logic into the `User` model. We are also injecting a `SmsCourierInterface` implementation into the method, allowing us to better test that aspect of the process. Now that we have refactored this logic, let's re-write our queue handler:

```
class SendSMS {

        public function __construct(UserRepository $users, SmsCourierInterface $courier)
        {
                $this->users = $users;
                $this->courier = $courier;
        }

        public function fire($job, $data)
        {
                $user = $this->users->find($data['user']['id']);

                $user->sendSmsMessage($this->courier, $data['message']);

                $job->delete();
        }

}
```

As you can see in this refactored example, our queue handler is now much lighter. It essentially serves as a *translation layer* between the queue and your *real* application logic. That is great! It means that we can easily send SMS messages outside of the queue context. Finally, let's write some tests for our SMS sending logic:

```
class SmsTest extends PHPUnit_Framework_TestCase {

        public function testUserCanBeSentSmsMessages()
        {
                /**
                 * Arrange...
                 */
                $user = Mockery::mock('User[sms]');
                $relation = Mockery::mock('StdClass');
                $courier = Mockery::mock('SmsCourierInterface');

                $user->shouldReceive('sms')->once()->andReturn($relation);

                $relation->shouldReceive('create')->once()->with(array(
                        'to' => '555-555-5555',
                        'message' => 'Test',
                ));

                $courier->shouldReceive('sendMessage')->once()->with(
                        '555-555-5555', 'Test'
                );

                /**
                 * Act...
                 */
                $user->sms_number = '555-555-5555';
                $user->sendSmsMessage($courier, 'Test');
        }

}
```

Other Handlers

We can improve many other types of "handlers" using this same approach to decoupling. By restricting all handlers to being simple *translation layers*, you can keep your heavy business logic neatly organized and decoupled from the rest of the framework. To drive the point home further,

let's examine a route filter that verifies that the current user of our application is subscribed to our "premium" pricing tier.

```
Route::filter('premium', function()
{
        return Auth::user() && Auth::user()->plan == 'premium';
});
```

On first glance, this route filter looks very innocent. What could possibly be wrong with a filter that is so small? However, even in this small filter, we are leaking implementation details of our application into the code. Notice that we are manually checking the value of the `plan` variable. We have tightly coupled the representation of "plans" in our business layer into our routing / transport layer. Now, if we change how the "premium" plan is represented in our database or user model, we will need to change this route filter!

Instead, let's make a very simple change:

```
Route::filter('premium', function()
{
        return Auth::user() && Auth::user()->isPremium();
});
```

A small change like this has great benefits and very little cost. By deferring the determination of whether a user is on the premium plan to the model, we have removed all implementation details from our route filter. Our filter is no longer responsible for knowing how to determine if a user is on the premium plan. Instead, it simply asks the User model. Now, if the representation of premium plans changes in the database, there is no need to update the route filter!

Who Is Responsible?

Again we find ourselves exploring the concept of *responsibility*. Remember, always be considering a class' responsibility and knowledge. Avoid making your transport layer, such as handlers, responsible for knowledge abour your application and busines logic.

Extending The Framework

Introduction

Laravel offers many extension points for you to customize the behavior of the framework's core components, or even replace them entirely. For example, the hashing facilities are defined by a `HasherInterface` contract, which you may implement based on your application's requirements. You may also extend the `Request` object, allowing you to add your own convenient "helper" methods. You may even add entirely new authentication, cache, and session drivers!

Laravel components are generally extended in two ways: binding new implementations in the IoC container, or registering an extension with a `Manager` class, which are implementations of the "Factory" design pattern. In this chapter we'll explore the various methods of extending the framework and examine the necessary code.

Methods Of Extension

Remember, Laravel components are typically extended in one of two ways: IoC bindings and the `Manager` classes. The manager classes serve as an implementation of the "factory" design pattern, and are responsible for instantiating driver based facilities such as cache and session.

Managers & Factories

Laravel has several `Manager` classes that manage the creation of driver-based components. These include the cache, session, authentication, and queue components. The manager class is responsible for creating a particular driver implementation based on the application's configuration. For example, the `CacheManager` class can create APC, Memcached, Native, and various other implementations of cache drivers.

Each of these managers includes an `extend` method which may be used to easily inject new driver resolution functionality into the manager. We'll cover each of these managers below, with examples of how to inject custom driver support into each of them.

Learn About Your Managers

Take a moment to explore the various `Manager` classes that ship with Laravel, such as the `CacheManager` and `SessionManager`. Reading through these classes will give you a more thorough understanding of how Laravel works under the hood. All manager classes extend the `Illuminate\Support\Manager` base class, which provides some helpful, common functionality for each manager.

Cache

To extend the Laravel cache facility, we will use the `extend` method on the `CacheManager`, which is used to bind a custom driver resolver to the manager, and is common across all manager classes. For example, to register a new cache driver named "mongo", we would do the following:

```
Cache::extend('mongo', function($app)
{
        // Return Illuminate\Cache\Repository instance...
});
```

The first argument passed to the `extend` method is the name of the driver. This will correspond to your `driver` option in the `app/config/cache.php` configuration file. The second argument is a Closure that should return an `Illuminate\Cache\Repository` instance. The Closure will be passed an `$app` instance, which is an instance of `Illuminate\Foundation\Application` and an IoC container.

To create our custom cache driver, we first need to implement the `Illuminate\Cache\StoreInterface` contract. So, our MongoDB cache implementation would look something like this:

```
class MongoStore implements Illuminate\Cache\StoreInterface {

        public function get($key) {}
        public function put($key, $value, $minutes) {}
        public function increment($key, $value = 1) {}
        public function decrement($key, $value = 1) {}
        public function forever($key, $value) {}
        public function forget($key) {}
        public function flush() {}

}
```

We just need to implement each of these methods using a MongoDB connection. Once our implementation is complete, we can finish our custom driver registration:

```
use Illuminate\Cache\Repository;

Cache::extend('mongo', function($app)
{
        return new Repository(new MongoStore);
});
```

As you can see in the example above, you may use the base `Illuminate\Cache\Repository` when creating custom cache drivers. There is typically no need to create your own repository class.

If you're wondering where to put your custom cache driver code, consider making it available on Packagist! Or, you could create an `Extensions` namespace within your application's primary folder. For example, if the application is named `Snappy`, you could place the cache extension in `app/Snappy/Extensions/MongoStore.php`. However, keep in mind that Laravel does not have a rigid application structure and you are free to organize your application according to your preferences.

Where To Extend

If you're ever wondering where to put a piece of code, always consider a service provider. As we've discussed, using a service provider to organize framework extensions is a great way to organize your code.

Session

Extending Laravel with a custom session driver is just as easy as extending the cache system. Again, we will use the `extend` method to register our custom code:

```
Session::extend('mongo', function($app)
{
        // Return implementation of SessionHandlerInterface
});
```

Note that our custom cache driver should implement the `SessionHandlerInterface`. This interface is included in the PHP 5.4+ core. If you are using PHP 5.3, the interface will be defined for you by Laravel so you have forward-compatibility. This interface contains just a few simple methods we need to implement. A stubbed MongoDB implementation would look something like this:

```
class MongoHandler implements SessionHandlerInterface {

        public function open($savePath, $sessionName) {}
        public function close() {}
        public function read($sessionId) {}
        public function write($sessionId, $data) {}
        public function destroy($sessionId) {}
        public function gc($lifetime) {}

}
```

Since these methods are not as readily understandable as the cache `StoreInterface`, let's quickly cover what each of the methods do:

- The `open` method would typically be used in file based session store systems. Since Laravel ships with a `native` session driver that uses PHP's native file storage for sessions, you will almost never need to put anything in this method. You can leave it as an empty stub. It is simply a fact of poor interface design (which we'll discuss later) that PHP requires us to implement this method.
- The `close` method, like the `open` method, can also usually be disregarded. For most drivers, it is not needed.
- The `read` method should return the string version of the session data associated with the given `$sessionId`. There is no need to do any serialization or other encoding when retrieving or storing session data in your driver, as Laravel will perform the serialization for you.
- The `write` method should write the given `$data` string associated with the `$sessionId` to some persistent storage system, such as MongoDB, Dynamo, etc.
- The `destroy` method should remove the data associated with the `$sessionId` from persistent storage.
- The `gc` method should destroy all session data that is older than the given `$lifetime`, which is a UNIX timestamp. For self-expiring systems like Memcached and Redis, this method may be left empty.

Once the `SessionHandlerInterface` has been implemented, we are ready to register it with the Session manager:

```
Session::extend('mongo', function($app)
{
        return new MongoHandler;
});
```

Once the session driver has been registered, we may use the `mongo` driver in our `app/config/session.php` configuration file.

Share Your Knowledge

Remember, if you write a custom session handler, share it on Packagist!

Authentication

Authentication may be extended the same way as the cache and session facilities. Again, we will use the `extend` method we have become familiar with:

```
Auth::extend('riak', function($app)
{
        // Return implementation of Illuminate\Auth\UserProviderInterface
});
```

The `UserProviderInterface` implementations are only responsible for fetching a `UserInterface` implementation out of a persistent storage system, such as MySQL, Riak, etc. These two interfaces allow the Laravel authentication mechanisms to continue functioning regardless of how the user data is stored or what type of class is used to represent it.

Let's take a look at the `UserProviderInterface`:

```
interface UserProviderInterface {

        public function retrieveById($identifier);
        public function retrieveByCredentials(array $credentials);
        public function validateCredentials(UserInterface $user, array $credentials);

}
```

The `retrieveById` function typically receives a numeric key representing the user, such as an auto-incrementing ID from a MySQL database. The `UserInterface` implementation matching the ID should be retrieved and returned by the method.

The `retrieveByCredentials` method receives the array of credentials passed to the `Auth::attempt` method when attempting to sign into an application. The method should then "query" the underlying persistent storage for the user matching those credentials. Typically, this method will run a query with a "where" condition on `$credentails['username']`. **This method should not attempt to do any password validation or authentication.**

The `validateCredentials` method should compare the given `$user` with the `$credentials` to authenticate the user. For example, this method might compare the `$user->getAuthPassword()` string to a `Hash::make` of `$credentials['password']`.

Now that we have explored each of the methods on the `UserProviderInterface`, let's take a look at the `UserInterface`. Remember, the provider should return implementations of this interface from the `retrieveById` and `retrieveByCredentials` methods:

```
interface UserInterface {

        public function getAuthIdentifier();
        public function getAuthPassword();

}
```

This interface is simple. The `getAuthIdentifier` method should return the "primary key" of the user. In a MySQL back-end, again, this would be the auto-incrementing primary key. The `getAuthPassword` should return the user's hashed password. This interface allows the authentication system to work with any User class, regardless of what ORM or storage abstraction layer you are using. By default, Laravel includes a `User` class in the `app/models` directory which implements this interface, so you may consult this class for an implementation example.

Finally, once we have implemented the `UserProviderInterface`, we are ready to register our extension with the `Auth` facade:

```
Auth::extend('riak', function($app)
{
        return new RiakUserProvider($app['riak.connection']);
});
```

After you have registered the driver with the `extend` method, you switch to the new driver in your `app/config/auth.php` configuration file.

IoC Based Extension

Almost every service provider included with the Laravel framework binds objects into the IoC container. You can find a list of your application's service providers in the `app/config/app.php` configuration file. As you have time, you should skim through each of these provider's source code. By doing so, you will gain a much better understanding of what each provider adds to the framework, as well as what keys are used to bind various services into the IoC container.

For example, the `PaginationServiceProvider` binds a `paginator` key into the IoC container, which resolves into a `Illuminate\Pagination\Environment` instance. You can easily extend and override this class within your own application by overriding this IoC binding. For example, you could create a class that extend the base `Environment`:

```
namespace Snappy\Extensions\Pagination;

class Environment extends \Illuminate\Pagination\Environment {

        //

}
```

Once you have created your class extension, you may create a new `SnappyPaginationProvider` service provider class which overrides the paginator in its `boot` method:

```
class SnappyPaginationProvider extends PaginationServiceProvider {

        public function boot()
        {
                App::bind('paginator', function()
                {
                        return new Snappy\Extensions\Pagination\Environment;
                });

                parent::boot();
        }

}
```

Note that this class extends the `PaginationServiceProvider`, not the default `ServiceProvider` base class. Once you have extended the service provider, swap out the `PaginationServiceProvider` in your `app/config/app.php` configuration file with the name of your extended provider.

This is the general method of extending any core class that is bound in the container. Essentially every core class is bound in the container in this fashion, and can be overridden. Again, reading through the included framework service providers will familiarize you with where various classes are bound into the container, and what keys they are bound by. This is a great way to learn more about how Laravel is put together.

Request Extension

Because it is such a foundational piece of the framework and is instantiated very early in the request cycle, extending the `Request` class works a little differently than the previous examples.

First, extend the class like normal:

```php
<?php namespace QuickBill\Extensions;

class Request extends \Illuminate\Http\Request {

        // Custom, helpful methods here...

}
```

Once you have extended the class, open the `bootstrap/start.php` file. This file is one of the very first files to be included on each request to your application. Note that the first action performed is the creation of the Laravel `$app` instance:

```php
$app = new \Illuminate\Foundation\Application;
```

When a new application instance is created, it will create a new `Illuminate\Http\Request` instance and bind it to the IoC container using the `request` key. So, we need a way to specify a custom class that should be used as the "default" request type, right? And, thankfully, the `requestClass` method on the application instance does just this! So, we can add this line at the very top of our `bootstrap/start.php` file:

```php
use Illuminate\Foundation\Application;

Application::requestClass('QuickBill\Extensions\Request');
```

Once you have specified the custom request class, Laravel will use this class anytime it creates a `Request` instance, conveniently allowing you to always have an instance of your custom request class available, even in unit tests!

Single Responsibility Principle

Introduction

The “SOLID” design principles, articulated by Robert “Uncle Bob” Martin, are five principles that provide a good foundation for sound application design. The five principles are:

- The Single Responsibility Principle
- The Open Closed Principle
- The Liskov Substitution Principle
- The Interface Segregation Principle
- The Dependency Inversion Principle

Let’s explore each of these principles in more depth, and look at some code examples illustrating each principle. As we will see, each principle compliments the others, and if one principle falls, most, if not all, of the others do as well.

In Action

The Single Responsibility Principle states that a class should have one, and only one, reason to change. In other words, a class’ scope and responsibility should be narrowly focused. As we have said before, ignorance is bliss when it comes to class responsibilites. A class should do its job, and should not be affected by changes to any of its dependencies.

Consider the following class:

```
class OrderProcessor {

    public function __construct(BillerInterface $biller)
    {
        $this->biller = $biller;
    }

    public function process(Order $order)
    {
        $recent = $this->getRecentOrderCount($order);

        if ($recent > 0)
        {
            throw new Exception('Duplicate order likely.');
        }

        $this->biller->bill($order->account->id, $order->amount);

        DB::table('orders')->insert(array(
            'account'    => $order->account->id,
            'amount'     => $order->amount;
            'created_at' => Carbon::now();
        ));
    }

    protected function getRecentOrderCount(Order $order)
    {
        $timestamp = Carbon::now()->subMinutes(5);

        return DB::table('orders')
                        ->where('account', $order->account->id)
                        ->where('created_at', '>=', $timestamps)
                        ->count();
    }

}
```

What are the responsibilites of the class above? Obviously, its name implies that it is responsible for processing orders. But, based on the `getRecentOrderCount` method, we can also see that it is responsible for examining an account's order history in the database in order to detect duplicate orders. This extra validation responsiblity means that we must change our order processor when our data store changes, as well as when our order validation rules change.

We should extract this responsiblity into another class, such as an `OrderRepository`:

```
class OrderRepository {

    public function getRecentOrderCount(Account $account)
    {
        $timestamp = Carbon::now()->subMinutes(5);

        return DB::table('orders')
                        ->where('account', $account->id)
                        ->where('created_at', '>=', $timestamp)
                        ->count();
    }

    public function logOrder(Order $order)
    {
        DB::table('orders')->insert(array(
            'account'    => $order->account->id,
            'amount'     => $order->amount;
            'created_at' => Carbon::now();
        ));
    }

}
```

Then, we can inject our repository into the `OrderProcessor`, alleviating it of the responsibility of researching an account's order history:

```
class OrderProcessor {

    public function __construct(BillerInterface $biller,
                                OrderRepository $orders)
    {
        $this->biller = $biller;
        $this->orders = $orders;
    }

    public function process(Order $order)
    {
        $recent = $this->orders->getRecentOrderCount($order->account);

        if ($recent > 0)
        {
            throw new Exception('Duplicate order likely.');
        }

        $this->biller->bill($order->account->id, $order->amount);

        $this->orders->logOrder($order);
    }

}
```

Now that we have abstracted our order data gathering responsibilities, we no longer have to change our `OrderProcessor` when the method for retrieving and logging orders changes. Our class responsibilities are more focused and narrow, providing for cleaner, more expressive code, and a more maintainable application.

Keep in mind, the Single Responsibility Principle isn't just about less lines of code, it's about writing classes that have a narrow responsibility, and a cohesive set of available methods, so make sure that all of the methods in a class are aligned with the overall responsibility of that class. After building a library of small, clear classes with well defined responsibilities, our code will be more decoupled, easier to test, and friendlier to change.

Open Closed Principle

Introduction

Over the lifetime of an application, more time is spent adding to the existing codebase rather than constantly adding new features from scratch. As you are probably aware, this can be a tedious and frustrating process. Anytime you modify code, you risk introducing new bugs, or breaking old functionality completely. Ideally, we should be able to modify an existing codebase as quickly and easily as writing brand new code. If we correctly design our application according to the Open Closed principle, we can just do that!

Open Closed Principle

The Open Closed principle of SOLID design states that code is open for extension but closed for modification.

In Action

To demonstrate the Open Closed principle, let's continue working with our `OrderProcessor` from the previous chapter. Consider the following section of the `process` method:

```
    $recent = $this->orders->getRecentOrderCount($order->account);

    if ($recent > 0)
    {
        throw new Exception('Duplicate order likely.');
    }
```

This code is quite readable, and even easy to test since we are properly using dependency injection. However, what if our business rules regarding order validation change? What if we get new rules? In fact, what if, as our business grows, we get *many* new rules? Our `process` method will quickly grow into a beast of spaghetti code that is hard to maintain. Because this code must be changed each time our business rules change, it is open for modification and violates the Open Closed principle. Remember, we want our code to be open for *extension*, not modification.

Instead of performing all of our order validation directly inside the `process` method, let's define a new interface: `OrderValidator`:

```
interface OrderValidatorInterface {
        public function validate(Order $order);
}
```

Next, let's define an implementation that protects against duplicate orders:

```
class RecentOrderValidator implements OrderValidatorInterface {

        public function __construct(OrderRepository $orders)
        {
                $this->orders = $orders;
        }

        public function validate(Order $order)
        {
                $recent = $this->orders->getRecentOrderCount($order->account);

                if ($recent > 0)
                {
                throw new Exception('Duplicate order likely.');
                }
        }

}
```

Great! Now we have a small, testable encapsulation of a single business rule. Let's create another implementation that verifies the account is not suspended:

```
class SuspendedAccountValidator implememts OrderValidatorInterface {

        public function validate(Order $order)
        {
                if ($order->account->isSuspended())
                {
                        throw new Exception("Suspended accounts may not order.")
                }
        }

}
```

Now that we have two different implementations of our `OrderValidatorInterface`, let's use them within our `OrderProcessor`. We'll simply inject an array of validators into the processor instance, which will allow us to easily add and remove validation rules as our codebase evolves.

```
class OrderProcessor {

    public function __construct(BillerInterface $biller,
                                OrderRepository $orders,
                                array $validators = array())
    {
        $this->biller = $biller;
        $this->orders = $orders;
        $this->validators = $validators;
    }

}
```

Next, we can simply loop through the validators in the `process` method:

```
public function process(Order $order)
{
        foreach ($this->validators as $validator)
        {
                $validator->validate($order);
        }

        // Process valid order...
}
```

Finally, we will register our `OrderProcessor` class in the application IoC container:

```
App::bind('OrderProcessor', function()
{
        return new OrderProcessor(
                App::make('BillerInterface'),
                App::make('OrderRepository'),
                array(
                        App::make('RecentOrderValidator'),
                        App::make('SuspendedAccountValidator'),
                ),
        );
});
```

With these few changes, which took only minimal effort to build from our existing codebase, we can now add and remove new validation rules without modifying a single line of existing code.

Each new validation rule is simply a new implementation of the `OrderValidatorInterface`, and is registered with the IoC container. Instead of unit testing a large, unwieldy `process` method, we can now test each validation rule in isolation. Our code is now *open* for extension, but *closed* for modification.

Leaky Abstractions

Watch out for dependencies that leak implementation details. An implementation change in a dependency should not require any changes by its consumer. When changes to the consumer are required, it is said that the dependency is "leaking" implementation details. When your abstractions are leaky, the Open Closed principle has likely been broken.

Before proceeding further, remember that this principle is not a law. It does not state that every piece of code in your application must be "pluggable". For instance, a small application that retrieves a few records out of a MySQL database does not warrant a strict adherence to every design principle you can imagine. Do not blindly apply a given design principle out of guilt as you will likely create an over-designed, cumbersome system. Keep in mind that many of these design principles were created to address common architectural problems in large, robust applications. That being said, don't use this paragraph as an excuse to be lazy!

Liskov Substitution Principle

Introduction

Don't worry, the Liskov Substitution Principle is a lot easier to understand than it sounds. This principle states that you should be able to use any implementation of an abstraction in any place that accepts that abstraction. But, let's make this a little simpler. In plain English, the principle states that if a class uses an implementation of an interface, it must be able to use any implementation of that interface without requiring any modifications.

Liskov Substitution Principle

This principle states that objects should be replacable with instances of their subtypes without altering the correctness of that program.

In Action

To illustrate this principle, let's continue to use our `OrderProcessor` example from the previous chapter. Take a look at this method:

```
1  public function process(Order $order)
2  {
3          // Validate order...
4
5          $this->orders->logOrder($order);
6  }
```

Note that after the `Order` is validated, we log the order using the `OrderRepositoryInterface` implementation. Let's assume that when our order processing business was young, we stored all of our orders in CSV format on the filesystem. Our only `OrderRepositoryInterface` implementation was a `CsvOrderRepository`. Now, as our order rate grows, we want to use a relational database to store them. So, let's look at a possible implementation of our new repository:

```
class DatabaseOrderRepository implements OrderRepositoryInterface {

        protected $connection;

        public function connect($username, $password)
        {
                $this->connection = new DatabaseConnection($username, $password);
        }

        public function logOrder(Order $order)
        {
                $this->connection->run('insert into orders values (?, ?)', array(
                        $order->id, $order->amount,
                ));
        }

}
```

Now, let's examine how we would have to consume this implementation:

```
public function process(Order $order)
{
        // Validate order...

        if ($this->repository instanceof DatabaseOrderRepository)
        {
                $this->repository->connect('root', 'password');
        }

        $this->repository->logOrder($order);
}
```

Notice that we are forced to check if our `OrderRepositoryInterface` is a database implementation from within our consuming processor class. If it is, we must connect to the database. This may not seem like a problem in a very small application, but what if the `OrderRepositoryInterface` is consumed by dozens of other classes? We would be forced to implement this "bootstrap" code in every consumer. This will be a headache to maintain and is prone to bugs, and if we forget to update a single consumer our application will break.

The example above clearly breaks the Liskov Substitution Principle. We were unable to inject an implementation of our interface without changing the consumer to call the `connect` method. So, now that we have identified the problem, let's fix it. Here is our new `DatabaseOrderRepository` implementation:

```
class DatabaseOrderRepository implements OrderRepositoryInterface {

        protected $connector;

        public function __construct(DatabaseConnector $connector)
        {
                $this->connector = $connector;
        }

        public function connect()
        {
                return $this->connector->bootConnection();
        }

        public function logOrder(Order $order)
        {
                $connection = $this->connect();

                $connection->run('insert into orders values (?, ?)', array(
                        $order->id, $order->amount,
                ));
        }

}
```

Now our `DatabaseOrderRepository` is managing the connection to the database, and we can remove our "bootstrap" code from the consuming `OrderProcessor`:

```
public function process(Order $order)
{
        // Validate order...

        $this->repository->logOrder($order);
}
```

With this modification, we can now use our `CsvOrderRepository` or `DatabaseOrderRepository` without modifying the `OrderProcessor` consumer. Our code adheres to the Liskov Substitution Principle! Take note that many of the architecture concepts we have discussed are related to *knowledge*. Specifically, the knowledge a class has of its "surroundings", such as the peripheral code and dependencies that help a class do its job. As you work towards a robust application architecture, limiting class *knowledge* will be a recurring and important theme.

Also note the consequence of violating the Liskov Substitution principle with regards to the other principles we have covered. By breaking this principle, the Open Closed principle must also be broken, as, if the consumer must check for instances of various child classes, the consumer must be changed each time there is a new child class.

Watch For Leaks

You have probably noticed that this principle is closely related to the avoidance of "leaky abstractions", which were discussed in the previous chapter. Our database repository's leaky abstraction was our first clue that the Liskov Substitution Principle was being broken. Keep an eye out for those leaks!

Interface Segregation Principle

Introduction

The Interface Segregation principle states that no implementation of an interface should be forced to depend on methods it does not use. Have you ever had to implement methods of an interface that you did not need? If so, you probably created blank methods in your implementation. This is an example of being forced to use an interface that breaks the Interface Segregation principle.

In practical terms, this principle demands that interfaces be granular and focused. Sound familiar? Remember, all five SOLID principles are related, such that by breaking one you often must break the others. When breaking the Interface Segregation principle, the Single Responsibility principle must also be broken.

Instead of having a "fat" interface containing methods not needed by all implementations, it is preferable to have several smaller interfaces that may be implemented individually as needed. By breaking fat interfaces into smaller, more focused contracts, consuming code can depend on the smaller interface, without creating dependencies on parts of the application it does not use.

Interface Segregation Principle

This principle states that no implementation of an interface should be forced to depend on methods it does not use.

In Action

To illustrate this principle, let's consider an example session handling library. In fact, we will consider PHP's own `SessionHandlerInterface`. Here are the methods defined by this interface, which is included with PHP beginning in version 5.4:

```
interface SessionHandlerInterface {
        public function close();
        public function destroy($sessionId);
        public function gc($maxLiftetime);
        public function open($savePath, $name);
        public function read($sessionId);
        public function write($sessionId, $sessionData);
}
```

Now that you are familiar with the methods defined by this interface, consider an implementation using Memcached. Would a Memcached implementation of this interface define functionality for each of these methods? Not only do we not need to implement all of these methods, we don't need half of them!

Since Memcached will automatically expire the values it contains, we do not need to implement the `gc` method of the interface, nor do we need to implement the `open` or `close` methods of the interface. So, we are forced to define "stubs" for these methods in our implementation that are just empty methods. To correct this problem, let's start by defining a smaller, more focused interface for session garbage collection:

```
interface GarbageCollectorInterface {
        public function gc($maxLifetime);
}
```

Now that we have a smaller interface, any consuming code can depend on this focused contract, which defines a very narrow set of functionality and does not create a dependency on the entire session handler.

To further understand the principle, let's reinforce our knowledge with another example. Imagine we have a `Contact` Eloquent class that is defined like so:

```
class Contact extends Eloquent {

        public function getNameAttribute()
        {
                return $this->attributes['name'];
        }

        public function getEmailAttribute()
        {
                return $this->attributes['email'];
        }

}
```

Now, let's assume that our application also employs a `PasswordReminder` class that is responsible for sending password reminder e-mails to users of the application. Below is a possible definition of the `PasswordReminder` class:

```
class PasswordReminder {

        public function remind(Contact $contact, $view)
        {
                // Send password reminder e-mail...
        }

}
```

As you probably noticed, our `PasswordReminder` is dependent upon the `Contact` class, which in turns means it is dependent on the Eloquent ORM. It is neither desirable or necessary to couple the password reminder system to a specific ORM implementation. By breaking the dependency, we can freely change our back-end storage mechanism or ORM without affecting the password reminder component of the application. Again, by breaking one of the SOLID principles, we have given a consuming class too much *knowledge* about the rest of the application.

To break the dependency, let's create a `RemindableInterface`. In fact, such an interface is included with Laravel, and is implemented by the `User` model by default:

```
interface RemindableInterface {
        public function getReminderEmail();
}
```

Once the interface has been created, we can implement it on our model:

```
class Contact extends Eloquent implements RemindableInterface {

        public function getReminderEmail()
        {
                return $this->email;
        }

}
```

Finally, we can depend on this smaller, more focused interface in the `PasswordReminder`:

```
class PasswordReminder {

        public function remind(RemindableInterface $remindable, $view)
        {
                // Send password reminder e-mail...
        }

}
```

By making this simple change, we have removed unnecessary dependencies from the password reminder component and made it flexible enough to use any class from any ORM, so long as that class implements the new `RemindableInterface`. This is exactly how the Laravel password reminder component remains database and ORM agnostic!

Knowledge Is Power

Again we have discovered the pitfalls of giving a class too much knowledge about application implementation details. By paying careful attention to how much knowledge we are giving a class, we can abide by all of the SOLID principles.

Dependency Inversion Principle

Introduction

We have reached our final destination in our overview of the five SOLID design principles! The final principle is the Dependency Inversion principle, and it states that high-level code should not depend on low-level code. Instead, high-level code should depend on an abstraction layer that serves as a "middle-man" between the high and low-level code. A second aspect to the principle is that abstractions should not depend upon details, but rather details should depend upon abstractions. If this all sounds extremely confusing, don't worry. We'll cover both aspects of this princple below.

Dependency Inversion Principle

This principle states that high-level code should not depend on low-level code, and that abstractions should not depend upon details.

In Action

If you have already read prior chapters of this book, you already have a good grasp of the Dependency Inversion principle! To illustrate the principle, let's consider the following class:

```
class Authenticator {

        public function __construct(DatabaseConnection $db)
        {
                $this->db = $db;
        }

        public function findUser($id)
        {
                return $this->db->exec('select * from users where id = ?', array($id));
        }

        public function authenticate($credentials)
        {
                // Authenticate the user...
        }

}
```

As you might have guessed, the `Authenticator` class is responsible for finding and authenticating users. Let's examine the constructor of this class. You will see that we are type-hinting a `DatabaseConnection` instance. So, we're tightly coupling our authenticator to the database, and essentially saying that users will always only be provided out of a relational SQL database. Furthermore, our high-level code (the `Authenticator`) is directly depending on low-level code (the `Database\Connection`).

First, let's discuss "high-level" and "low-level" code. Low-level code implements basic operations like reading files from a disk, or interacting with a database. High-level code encapsulates complex logic and relies on the low-level code to function, but should not be directly coupled to it. Instead, the high-level code should depend on an abstraction that sits on top of the low-level code, such as an interface. Not only that, but the low-level code should *also* depend upon an abstraction. So, let's write an interface that we can use within our `Authenticator`:

```
interface UserProviderInterface {
        public function find($id);
        public function findByUsername($username);
}
```

Next let's inject an implementation of this interface into our `Authenticator`:

```
class Authenticator {

        public function __construct(UserProviderInterface $users,
                                    HasherInterface $hash)
        {
                $this->hash = $hash;
                $this->users = $users;
        }

        public function findUser($id)
        {
                return $this->users->find($id);
        }

        public function authenticate($credentials)
        {
                $user = $this->users->findByUsername($credentials['username']);

                return $this->hash->make($credentials['password']) == $user->password;
        }

}
```

After making these changes, our `Authenticator` now relies on two high-level abstractions: `UserProviderInterface` and `HasherInterface`. We are free to inject any implementation of these interfaces into the `Authenticator`. For example, if our users are now stored in Redis, we can write a `RedisUserProvider` which implements the `UserProvider` contract. The `Authenticator` is no longer depending directly on low-level storage operations.

Furthermore, our low-level code now depends on the high-level `UserProviderInterface` abstraction, since it implements the interface itself:

```
class RedisUserProvider implements UserProviderInterface {

        public function __construct(RedisConnection $redis)
        {
                $this->redis = $redis;
        }

        public function find($id)
        {
                $this->redis->get('users:'.$id);
        }

        public function findByUsername($username)
        {
                $id = $this->redis->get('user:id:'.$username);

                return $this->find($id);
        }

}
```

Inverted Thinking

Applying this principle *inverts* the way many developers design applications. Instead of coupling high-level code directly to lower-level code in a "top-down" fashion, this principle states that **both** high and low-level code depend upon a high-level abstraction.

Before we "inverted" the dependencies of our `Authenticator`, it could not be used with anything other than a database storage system. If we changed storage systems, our `Authenticator` would also have to be modified, in violation of the Open Closed principle. Again, as we have seen before, multiple principles usually stand or fall together.

After forcing the `Authenticator` to depend upon an abstraction over the storage layer, we can use it with any storage system that implements our `UserProviderInterface` contract without any modifications to the `Authenticator` itself. The conventional dependency chain has been *inverted* and our code is much more flexible and welcoming of change!