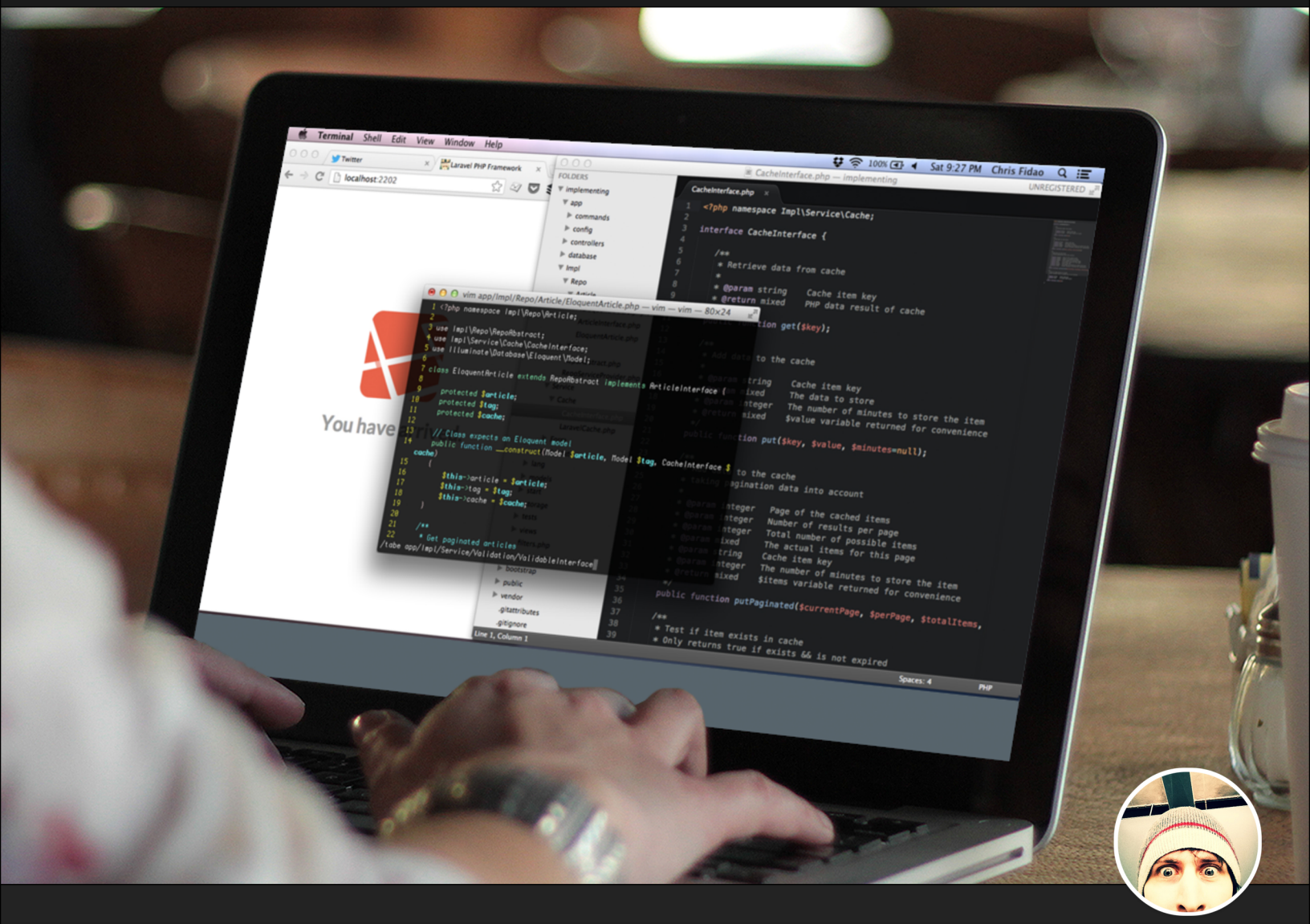

Implementing Laravel

Chris Fidao

Implementing Laravel

Chris Fidao

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

This version was published on 2013-10-21

Contents

Thanks

Thanks to Natalie for her patience, my reviewers for helping make this so much better and the team at Digital Surgeons for their support (and Gumby[1])!

[1] http://gumbyframework.com

Introduction

Who Is This For?

Who Will Get the Most Benefit?

This book is written for those who know the fundamentals of Laravel and are looking to see more advanced examples of implementing their knowledge in a testable and maintainable manner.

From the lessons here, you will see how to apply architectural concepts to Laravel in a variety of ways, with the hope that you can use and adapt them for your own needs.

What To Know Ahead of Time

We all have varying levels of knowledge. This book is written for those who are familiar with Laravel 4 and its core concepts. It therefore assumes some knowledge of the reader.

Taylor Otwell's book *Laravel: From Apprentice to Artisan*[2] is a great prerequisite. Although I'll cover these on a basic level, readers will hopefully already have a basic understanding of the principles of SOLID and Laravel's IoC container.

[2] https://leanpub.com/laravel

A Note on Opinions

Knowing the benefits (and pitfalls!) of Repository, Dependency Injection, Container, Service Locator patterns and other tools from our architectural tool set can be both liberating and exciting.

The use of those tools, however, can be plagued with unexpected and often nuanced issues.

As such, there are many opinions about how to go about crafting "good code" with such tools.

As I use many real examples in this book, I have implicitly (and sometimes explicitly!) included my own opinions in this book. Always, however, inform your own opinion with both what you read and your own experience!

"When all you have is a hammer..."

Overuse of any of these tools can cause it's own issues. The chapters here are examples of how you *can* implement the architectural tools available to us. Knowing when *not* to use them is also an important decision to keep in mind.

SOLID

Since I'll mention SOLID principles in passing throughout this book, I'll include a very brief explanation of them here, mostly taken from the Wikipedia entry[3] with some extra explanation in context of Laravel.

Single Responsibility Principle

A class (or unit of code) should have one responsibility.

Open/Closed Principle

A class should be open for extension but closed for modification.

You can extend a class or implement and interface, but you should not be able to modify a class directly. This means you should extend a class and use the new extension rather than change a class directly.

Additionally, this means setting class attributes and methods as private or protected properly so they cannot be modified by external code.

Liskov Substitution Principle

Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

In PHP, this often means creating interfaces for your code to implement. You can then change (switch-out) implementations of the interfaces. Doing so should be possible without having to change how your application code interacts with the implementation. The interface serves as a contract, guaranteeing that certain methods will be available.

Interface Segregation Principle

Many client-specific interfaces are better than one general-purpose interface.

In general, it's preferable to create an interface and implement it many times over than create a general-purpose class which attempts to work in all situations.

[3] http://en.wikipedia.org/wiki/SOLID_(object-oriented_design)

Dependency Inversion Principle

One should depend upon abstractions rather than concrete classes.

You should define class dependencies as an interface rather than a concrete class. This allows you to switch an implementation of the interface out without having to change the class using the dependency.

Core Concepts

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

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

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

The Container

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

Basic Usage

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

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

File: app/routes.php

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

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

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

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

Facades

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

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

Getting More Advanced

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

File: app/routes.php

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

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

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

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

```
Hello, World!
```

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

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

Inversion of Control

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

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

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

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

File: app/routes.php

```
interface GreetableInterface {

    public function greet();

}

class HelloWorld implements GreetableInterface {

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

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

File: app/routes.php

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

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

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

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

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

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

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

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

Real-World Usage

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

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

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

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

Illuminate\Pagination\PaginationServiceProvider.php

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

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

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

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

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

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

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

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

Dependency Injection

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

What is Dependency Injection?

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

Take this constructor method, for example:

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

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

This is opposite of direct instantiation:

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

If you find yourself asking *why* Dependency Injection is used, this Stack Overflow answer[4] is a great place to start. I'll cover some benefits of it in the following examples.

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

Adding Controller Dependencies

This is a very common use case within Laravel.

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

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

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

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

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

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

File: app/routes.php

```
interface GreetableInterface {

    public function greet();

}

class HelloWorld implements GreetableInterface {

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

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

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

File: app/controllers/ContainerController.php

```
<?php

class ContainerController extends BaseController {

    protected $greeter;

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

```
    }

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

}
```

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

```
Hello, World!
```

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

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

Interfaces as Dependencies

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

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

Let's see what that would look like:

File: app/controllers/ContainerController.php

```
<?php

class ContainerController extends BaseController {

    protected $greeter;

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

```

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

}
```

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

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

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

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

File: app/routes.php

```
interface GreetableInterface {

    public function greet();

}

class HelloWorld implements GreetableInterface {

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

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

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

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

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

Why Dependency Injection?

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

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

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

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

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

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

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

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

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

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

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

File: app/controllers/EmailController.php

```
class EmailController extends BaseController {

    protected $emailer;

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

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

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

}
```

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

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

First, make an implementation of `Emailer` which uses SendGrid!

```
class SendGridEmailer implements Emailer {

    protected $sendgrid;

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

    public function send($to, $from, $subject, $message)
    {
        $mail = $this->sendgrid->mail->instance();

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

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

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

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

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

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

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

Wrapping Up

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

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

Setting Up Laravel

The Sample Application

This book will use a sample application. We will build upon a simple blog application - everybody's favorite weekend learning project.

The application is viewable on Github at fideloper\Implementing-Laravel[5]. There you can view code and see working examples from this book.

Here is an overview of some information about the application.

Database

We'll have a database with the following tables and fields:

- **articles** - id, user_id, status_id, title, slug, excerpt, content, created_at, updated_at, deleted_at
- **statuses** - id, status, slug, created_at, updated_at
- **tags** - id, tag, slug
- **articles_tags** - article_id, tag_id
- **users** - id, email, password, created_at, updated_at, deleted_at

You'll find migrations for these tables, as well as some seeds, in the `app/database/migrations` directory on Github.

Models

Each of the database tables will have a corresponding Eloquent model class, inside of the `app/models` directory.

- `models/Article.php`
- `models/Status.php`
- `models/Tag.php`
- `models/User.php`

[5] https://github.com/fideloper/implementing-laravel

Relationships

Users can create articles and so there is a "one-to-many" relationship between users and articles; Each user can write multiple articles, but each article only has one user. This relationship is created within both the `User` and `Article` models.

Like users, there is a "one to many" relationship between statuses and articles - Each article is assigned a status, and a status can be assigned to many articles.

Finally, the Articles and Tags have a relationship. Each article can have one or many tags. Each tag can be assigned to one or many articles. Therefore, there is a "many to many" relationship between articles and tags. This relationship is defined within the `Article` and `Tag` models, and uses the `Articles_Tags` pivot table.

Testability and Maintainability

I'll use the phrase "testable and maintainable" a lot. In most contexts, you can assume:

1. **Testable** code practices SOLID principles in a way that allows us to unit test - to test one specific unit of code (or class) without bringing in its dependencies. This is most notable in the use of Dependency Injection which directly allows the use of mocking to abstract away class dependencies.
2. **Maintainable** code looks to the long-term cost in development time. This means that making changes to the code should be easy, even after months or years. Popular examples of a change that should be easy is switching out one email provider for another or one data-store for another. This is most directly realized through the use of interfaces and making use of inversion of control.

Architectural Notes

This book will cover creating an application library which contains most application code for the sample blog. The structure of the application library makes some assumptions about how we go about building the application code.

First, you'll note that I do not put Controllers into my application code. This is on purpose!

Laravel is a web-framework, designed to handle an HTTP request and route to your application code. The Request, Router and Controller classes are all designed to operate at the HTTP level.

Our application, however, does not need any such requirement. We're simply writing application logic which revolves around our business goals.

An HTTP request being routed to a controller function can be seen as a convenient way for a request to reach our application code (we're all on the internet, after all). However, our application does not necessarily need to "know" about the code calling our application, or HTTP at all.

This is an extension of the concept "separation of concerns". While we certainly are unlikely to use our applications out of context of the internet, it is useful to think of your web framework as an implementation detail of your application, rather than the core of it.

Think of it this way: our applications are not an implementation of the Laravel framework. Instead, Laravel is an implementation of our applications.

Taken to an extreme, an application would be able to be implemented by any framework or code capable of implementing the interfaces we define.

Extremes, however, are not pragmatic, and so we use Laravel to accomplish most of our application goals. Laravel is the means to most of our ends - it does things extremely well!

This also shapes the application library structure I build up to in this book. I'll create a series of directories reflecting application code functions, such as a data repository and cache services. These functional areas tend to be interfaced, which we implement using various Illuminate packages.

Installation

Let's start at the beginning.

This chapter will cover how to install Laravel. If you're reading this, you probably already know how to do this! However I feel it's worth including as I'll cover some extra steps I take to ward off potential pitfalls.

Install Composer

Composer is necessary to handle Laravel's dependencies. I typically install Composer[6] globally, so it can be run anywhere on my development machine.

Installing Composer globally in the command line

```
$ curl -sS https://getcomposer.org/installer | php
// Assumes /usr/local/bin is in your PATH
$ sudo mv composer.phar /usr/local/bin/composer
```

Now you can run Composer from anywhere.

Create a New Project

Once installed, we can use Composer to create a new Laravel project. We'll call our sample project "Implementing Laravel". Run this command:

Creating a new Laravel project with Composer

```
$ composer create-project laravel/laravel "Implementing Laravel"
```

This will clone the `laravel/laravel` project from Github and install its dependencies, as if you've cloned the git repository and run `$ composer install` manually. It won't, however, start a Git repository for you - you'll have to do that yourself with `$ git init`.

[6] http://getcomposer.org/

Config

Next we'll do some application setup and configuration.

Permissions

Periodically, depending on your development environment, you may find yourself needing to set the permissions of the `app/storage` directory. This is where Laravel puts logs, sessions, caches and other optimizations. PHP needs to be able to write to this directory.

Making your storage directory world-writable

```
$ chmod -R 0777 app/storage
```

In production, the web server often runs as user (and group) `www-data`. Since we don't necessarily want to make our files world-writable in production, we can instead make sure they are writable by `www-data` specifically.

Making your storage directory production-ready

```
$ chgrp -R www-data app/storage  # Change all to group www-data
$ chmod -R g+w app/storage       # Make them group-writable
```

Environments

Environments can and often should be set differently per server - usually at a minimum you may want to setup a local development server, a staging server and a production server.

Let's create a "local" environment. Start by creating a directory `app/config/local`.

For now, all we need to do is add in our local database connection. Create the file `app/config/local/database.php`.

Here's an example of what that might look like. Note that we didn't copy the complete `app/config/database.php` file, we only defined what values we needed to override for our local environment.

File: app/config/local/database.php

```php
<?php

return array(

	'connections' => array(

		'mysql' => array(
			'driver'    => 'mysql',
			'host'      => 'localhost',
			'database'  => 'implementinglaravel',
			'username'  => 'root',
			'password'  => 'root',
			'charset'   => 'utf8',
			'collation' => 'utf8_unicode_ci',
			'prefix'    => '',
		),

	)
);
```

The next step is to make sure Laravel chooses the "local" environment when we run it from our development server. By default, the environment is determined by what URL you use to access the application. If your URL is "localhost", you can set "localhost" as the "machine name" used to specify the "local" environment:

File: bootstrap/start.php

```php
// Assign "local" environment to "localhost"
$env = $app->detectEnvironment(array(

	'local' => array('localhost'),

));

// Or perhaps any URL ending in .dev:
$env = $app->detectEnvironment(array(

	'local' => array('*.dev'),
```

```
12
13 ));
14
15 // Or a combination thereof:
16 $env = $app->detectEnvironment(array(
17
18         'local' => array('localhost', '*.dev'),
19
20 ));
```

Environmental Variables

Rather than URL, you may want to use an environmental variable.

An environmental variable is a variable set in your *server* configuration, rather than in PHP code.

I opt for this method as it allows us to change the URL of your development environment without effecting the environment. This gives you the ability to allow a team of developers to use the same URL to access the application, without worrying about environment collision. It also allows you to set the environment inside of any automated configuration and provisioning tool such as Chef or Puppet.

In Apache, you can add an environmental variable into your .htaccess file or virtual host configuration:

```
1 SetEnv LARA_ENV local
```

If you're using Nginx and PHP-FPM, you can pass an environmental variable to fastcgi:

```
1 location ~ \.php$ {
2         # Other Parametes . . .
3         fastcgi_split_path_info ^(.+\.php)(/.+)$;
4         fastcgi_pass unix:/var/run/php5-fpm.sock;
5         fastcgi_index index.php;
6         fastcgi_param LARA_ENV local; # Here's our environment!
7         include fastcgi_params;
8 }
```

After you set an environmental variable, we need to tell Laravel how to detect it:

File: bootstrap/start/php

```
$env = $app->detectEnvironment(function()
{
        // Default to development if no env found
        return getenv('LARA_ENV') ?: 'development';
});
```

Wrapping Up

After you get your environment set up, you should be good to go! You can now:

- Install Composer globally
- Set needed file permissions
- Set configuration for your specific environment
- Set your environment with environmental variables

Default Environment

Many people are inclined to set their production credentials directly in the default `app/config/database.php` file. This is not recommended.

While setting up a separate environment for your production server(s) may seem like extra work, consider what may happen if your environment detection should fail and an action such as a migration or seeding falls back to production credentials! The extra safety is well worth it.

You may also want to consider omitting your "production" environment in your code repository altogether! If you are using git, you can add `app/config/production` to your `.gitignore` file to do so. You'll need to add in the configuration to your production server manually, but you avoid exposing your production credentials to your code repository.

Application Setup

If you've ever asked yourself "Where should I put my code?", this chapter will answer that question.

Always create an application library.

An application-specific library works well for code which isn't generic enough to warrant its own package, and also isn't a direct use of Laravel's core classes, such as a controller. An example is business logic code or integration of third-party libraries.

Basically, anything that you want to keep out of your controllers (most code!) should belong in your application library.

Setting Up the Application Library

To accomplish this, we'll start by creating a namespace for the application. In our example application, I'll choose the easy-to-type name "Impl", short for "Implementing Laravel". This will be both our application's top-level namespace and the directory name. We will create the directory `app/Impl` to house this application code.

Here's what the folder structure will look like:

```
Implementing Laravel
|- app
|--- commands
|--- config
|--- [ and so on ]
|--- Impl
|------ Exception
|------ Repo
|------ Service
```

As you likely know, the namespace, file name and directory structure matters - they allow us to autoload the PHP files based on the PSR-0 autoloading standard[7].

For example, a sample class in this structure might look like this:

[7] http://www.php-fig.org/psr/0/

File: app/Impl/Repo/EloquentArticle.php

```php
<?php namespace Impl\Repo;

class EloquentArticle {

        public function all() { . . . }

}
```

Following PSR-0, this file would go here:

```
Implementing Laravel
|- app
|--- Impl
|------Repo
|---------EloquentArticle.php
```

Autoloading

We now have a home for the `Impl` application library. Let's tell Composer to autoload those classes with the PSR-0 specification. To accomplish this, edit `composer.json`, and add to the "autoload" section:

File: composer.json

```
{
        "require": {
                "laravel/framework": "4.0.*"
        },
        "autoload": {
                "classmap": [
                        . . .
                ],
                "psr-0": {
                        "Impl": "app"
                }
        },
        "minimum-stability": "dev"
}
```

After adding in the `PSR-0` section to autoload the `Impl` library, we need to tell Composer to be aware of them:

```
$ composer dump-autoload
```

Using Composer's dump-autoload[8] is a way of telling Composer to find new classes in a classmap package (Laravel's controllers, models, etc). For PSR-0 autoloading, it can also rebuild composer's optimized autoloader, saving time when the application is run.

Now anywhere in our code, we can instantiate the `Impl\Repo\EloquentArticle` class and PHP will know to autoload it!

File: app/routes.php

```
Route::get('/', function()
{
        $articleRepo = new Impl\Repo\EloquentArticle;

        return View::make('home')->with('articles', $articleRepo->all());
}
```

Wrapping Up

We saw how to create a separate application library to house our application code. This library is where our business logic, extensions, IoC bindings and more will be added.

Location, Location, Location

Your application library can go anywhere. By convention, I add mine into the `app` directory, but you are certainly not limited to that location. You may want to use another location, or even consider creating a package out of your application library that can be added as a Composer dependency!

[8] http://getcomposer.org/doc/03-cli.md#dump-autoload

Useful Patterns

Getting to the heart of the matter, this chapter will review a few useful architectural patterns in Laravel. We will explore how we can employ the container, interfaces and dependency injection to increase our code's testability and maintainability.

The Repository Pattern

What Is It?

The repository pattern is a way of abstracting your business logic away from your data source. It's an extra layer on top of your data retrieval code which can be used in a number of ways.

Why Do We Use It?

The goal of a repository is to increase code maintainability and to form your code around your application's use cases.

Many developers think of it as a tool for larger-scale applications only, however I often find myself using it for most applications. The pros outweigh the cons of writing extra code.

Let's go over some of the benefits:

Dependency Inversion

This is an expression of the SOLID principles. The repository pattern allows us to create many substitutable implementations of an interface whose purpose is handling our data.

The most cited use case is to "switch out your data source". This describes the ability to switch out a data store, such as MySQL, to something else, such as a NoSQL database, without affecting your application code.

This is accomplished by creating an interface for your data retrieval. You can then create one or many implementations of that interface.

For instance, for the logic around our article repository, we will create an implementation using Eloquent. If we ever needed to switch out our data source to MongoDB, we can create a MongoDB implementation and switch it out. As our application code expects an interface, rather than a concrete class, it does not know the difference when you switch out one implementation of the interface for another.

This becomes *very* powerful if you use your data repository in many places of your application (you likely do). You likely interact with your data on almost every call to your application, whether it's directly in a controller, in a command, in a queue job or in form-processing code.

If you can be sure that each area of your application always has the methods it needs, you're on your way to a much easier future.

But really, how often do you change data sources? Chances are that you rarely change from your core SQL-based data source. There are, however, other reasons for still using the repository pattern.

Planning

I mentioned earlier in the book that my point of view of application coding is one centering around the business logic. Creating an interface is useful for planning your code around your business needs (use cases).

When defining the methods each implementation will use, you are planning the use cases for your domain. Will users be creating articles? Or will they only be reading articles? How do administrators interact differently than regular users?

Defining interfaces gives you a clearer picture of how your application will be used from a domain-perspective rather than a data-perspective.

Business Logic Orientation

Building upon the idea of planning around your business domain is actually expressing your business domain in code. Remember that each Eloquent model represents a single database table. Your business logic does not.

For example, an article in our sample application is more than one row in our `articles` table. It also encompasses an author from the `users` table, a status from the `statuses` table and a set of tags, represented in the `tags` and `articles_tags` tables. An article is a composite business entity; It's a business entity which contains other entities, rather than simply containing attributes such as its title, content and publication date.

The repository pattern allows us to express an article as more than a single row in a table. We can combine and mesh together our Eloquent models, relationships and the built-in query builder in whatever way we need in order to convert the raw data into true representations of our business entities.

Data Layer Logic

The data repository becomes a very convenient place to add in other logic around your data retrieval. Rather than add in extra logic to our Eloquent models, we can use our repository to house it.

For instance, you may need to cache your data. The data repository is a great place to add in your caching layer. The next chapter will show how to do that in a maintainable way.

Remote Data

It's important to remember that your data can come from many sources - not necessarily your databases.

Many modern web applications are mash-ups - They consume multiple APIs and return useful data to the end user. A data repository can be a way to house the logic of retrieving API information and combining it into an entity that your application can easily consume or process.

This is also useful in a Service Oriented Architecture (SOA), where we may need to make API calls to service(s) within our own infrastructure but don't have direct access to a database.

Example

Let's see what that looks like with a practical example.

In this example, we'll start with the central portion of any blog, the articles.

The Situation

As noted, the relevant portions of our database looks like this:

- **articles** - id, user_id, status_id, title, slug, excerpt, content, created_at, updated_at, deleted_at
- **tags** - id, tag, slug
- **articles_tags** - article_id, tag_id

We have an `Articles` table, where each row represents an article. We have a `Tags` table where each row represents one tag. Finally, we have an `Articles_Tags` table which we use to assign tags to articles. In Laravel, this is known as a "Pivot Table", and is necessary for representing any Many to Many relationship.

As mentioned previously, the `Articles` and `Tags` tables have corresponding models in `app/models`, which define their relationship and make use of the pivot table.

```
app
|-models
|--- Article.php
|--- Tag.php
```

Now the simplest, yet ultimately least maintainable, way of getting our articles is to use Eloquent models directly in a controller. For example, let's see the logic for the home page of our blog, which will display our 10 latest articles, with pagination.

app/controllers/ContentController.php

```
<?php

ContentController extends BaseController {

        // Home page route
        public function home()
        {
                // Get 10 latest articles with pagination
                $articles = Articles::with('tags')
                                ->orderBy('created_at', 'desc')
                                ->paginate(10);

                return View::make('home')
                                ->with('articles', $articles);
        }

}
```

This is simple, but can be improved. Some of the issues with this pattern:

- **Cannot change data sources** - With Eloquent, we can actually change data sources between various types of SQL. However, the repository pattern will let us change to any data storage - arrays, NoSQL database, from a cache (which we'll see later on) - without changing any code elsewhere in our application.
- **Not Testable** - We cannot test this code without hitting the database. The Repository Pattern will let us test our code without doing so.
- **Poor business logic** - We have to put any business logic around our data and models in this controller, greatly reducing reusability.

In short, we'll make our controllers messy and end up repeating code. Let's restructure this to improve the situation.

Restructuring

We'll be doing quite a few things here:

1. Getting away from using models directly
2. Making use of interfaces
3. Implementing Dependency Injection into our controllers
4. Using Laravel's IoC container to load the correct classes into our controllers

The models directory

The first thing we'll do is to get away from using models directly, and use our application's namespaced and auto-loaded directory, `Impl`.

Here's the directory structure we'll use:

```
app
|- Impl
|--- Repo
|------ Article
|------ Tag
|- models
|--- Article.php
|--- Tag.php
```

Interfaces

We'll create interfaces quite often in our application code. Interfaces are contracts - they enforce the use of their defined methods in their implementations. This allows us to safely use *any* repository which implements an interface without fear of its methods changing.

They also force you to ask yourself how the class will interact with other parts of your application.

Let's create our first:

File: app/Impl/Repo/Article/ArticleInterface.php

```php
<?php namespace Impl\Repo\Article;

interface ArticleInterface {

    /**
     * Get paginated articles
     *
     * @param int  Current Page
     * @param int  Number of articles per page
     * @return object  Object with $items and $totalItems for pagination
     */
    public function byPage($page=1, $limit=10);

    /**
     * Get single article by URL
     *
```

```
17          * @param string  URL slug of article
18          * @return object  Object of article information
19          */
20         public function bySlug($slug);
21
22         /**
23          * Get articles by their tag
24          *
25          * @param string  URL slug of tag
26          * @param int  Current Page
27          * @param int  Number of articles per page
28          * @return object  Object with $items and $totalItems for pagination
29          */
30         public function byTag($tag, $page=1, $limit=10);
31
32     }
```

Next we'll create an article repository to implement this interface. But first, we have a decision to make.

How we implement our interface depends on what our data source is. If we're using a flavor of SQL, chances are that Eloquent supports it. However, if we are consuming an API or using a NoSQL database, we may need to create an implementation to work for those.

Since I'm using MySQL, I'll leverage Eloquent, which will handily deal with relationships and make managing our data easy.

File: app/Impl/Repo/Article/EloquentArticle.php

```
1   <?php namespace Impl\Repo\Article;
2
3   use Impl\Repo\Tag\TagInterface;
4   use Illuminate\Database\Eloquent\Model;
5
6   class EloquentArticle implements ArticleInterface {
7
8       protected $article;
9       protected $tag;
10
11
12      // Class dependency: Eloquent model and
13      // implementation of TagInterface
14      public function __construct(Model $article, TagInterface $tag)
```

```
15      {
16          $this->article = $article;
17          $this->tag = $tag;
18      }
19
20      /**
21       * Get paginated articles
22       *
23       * @param int  Current Page
24       * @param int  Number of articles per page
25       * @return StdClass Object with $items and $totalItems for pagination
26       */
27      public function byPage($page=1, $limit=10)
28      {
29          $result = new \StdClass;
30          $result->page = $page;
31          $result->limit = $limit;
32          $result->totalItems = 0;
33          $result->items = array();
34
35          $articles = $this->article->with('tags')
36                          ->where('status_id', 1)
37                          ->orderBy('created_at', 'desc')
38                          ->skip( $limit * ($page-1) )
39                          ->take($limit)
40                          ->get();
41
42          // Create object to return data useful
43          // for pagination
44          $result->items = $articles->all();
45          $result->totalItems = $this->totalArticles();
46
47          return $data;
48      }
49
50      /**
51       * Get single article by URL
52       *
53       * @param string  URL slug of article
54       * @return object object of article information
55       */
56      public function bySlug($slug)
```

```
57      {
58          // Include tags using Eloquent relationships
59          return $this->article->with('tags')
60                      ->where('status_id', 1)
61                      ->where('slug', $slug)
62                      ->first();
63      }
64
65      /**
66       * Get articles by their tag
67       *
68       * @param string  URL slug of tag
69       * @param string  Tag
70       * @param int     Current Page
71       * @param int     Number of articles per page
72       * @return StdClass Object with $items and $totalItems for pagination
73       */
74      public function byTag($tag, $page=1, $limit=10)
75      {
76          $foundTag = $this->tag->where('slug', $tag)->first();
77
78          $result = new \StdClass;
79          $result->page = $page;
80          $result->limit = $limit;
81          $result->totalItems = 0;
82          $result->items = array();
83
84          if( !$foundTag )
85          {
86              return $result;
87          }
88
89          $articles = $this->tag->articles()
90                          ->where('articles.status_id', 1)
91                          ->orderBy('articles.created_at', 'desc')
92                          ->skip( $limit * ($page-1) )
93                          ->take($limit)
94                          ->get();
95
96          $result->totalItems = $this->totalByTag();
97          $result->items = $articles->all();
98
```

```
99          return $result;
100     }
101
102     /**
103      * Get total article count
104      *
105      * @return int  Total articles
106      */
107     protected function totalArticles()
108     {
109         return $this->article->where('status_id', 1)->count();
110     }
111
112     /**
113      * Get total article count per tag
114      *
115      * @param  string  $tag  Tag slug
116      * @return int     Total articles per tag
117      */
118     protected function totalByTag($tag)
119     {
120         return $this->tag->bySlug($tag)
121                 ->articles()
122                 ->where('status_id', 1)
123                 ->count();
124     }
125
126 }
```

Here's our file structure again, with the `ArticleInterface` and `EloquentArticle` files:

```
app
|- Impl
|--- Repo
|------ Article
|--------- ArticleInterface.php
|--------- EloquentArticle.php
|------ Tag
|--- models
|------ Article.php
|------ Tag.php
```

With our new implementation, we can revisit our controller:

app/controllers/ContentController.php

```php
<?php

use Impl\Repo\Article\ArticleInterface;

class ContentController extends BaseController {

	protected $article;

	// Class Dependency: Subclass of ArticleInterface
	public function __construct(ArticleInterface $article)
	{
		$this->article = $article;
	}

	// Home page route
	public function home()
	{
		$page = Input::get('page', 1);
		$perPage = 10;

		// Get 10 latest articles with pagination
		// Still get "arrayable" collection of articles
		$pagiData = $this->article->byPage($page, $perPage);

		// Pagination made here, it's not the responsibility
		// of the repository. See section on cacheing layer.
		$articles = Paginator::make(
			$pagiData->items,
			$pagiData->totalItems,
			$perPage
		);

		return View::make('home')->with('articles', $articles);
	}

}
```

Wait, What?

You might have a few questions on what I did here in my implementation.

First, we don't return a `Pagination` object by way of the query builder's `paginate()` method. This is on purpose. Our repository is meant to simply return a set of articles and shouldn't have knowledge of the Pagination class nor its generated HTML links.

Instead, we support pagination by using `skip()` and `take()` to make use of MySQL's `LIMIT` and `OFFSET` directly.

This means we defer the creation of a paginator class instance to our controller. Yes, we actually added more code to our controller!

The reason I choose not to incorporate the paginator class into the repository is because it uses HTTP input to get the current page number and generates HTML for page links. This implicitly adds these functionalities as dependencies on our data repository, where they don't belong. Determining the current page number, and generating presentation (HTML) is not logic a data repository should be responsible for.

By keeping the pagination functionality out of our repository, we're also actually keeping our code more maintainable. This would become clear if we used an implementation of the repository that doesn't happen to be an Eloquent model. In that case, it likely wouldn't return an instance of the paginator class. Our view may look for the paginator's `links()` method and find it doesn't exist!

Tying It Together

We have one step to go before our code works.

As noted, we set up some dependencies in our controllers and repositories. Class `EloquentArticle` expects `Eloquent\Model` and class `ContentController` expects an implementation of `ArticleInterface` on instantiation.

The last thing we have to do is use Laravel's IoC container and Service Providers to pass these dependencies into the classes when they are requested.

To accomplish this in our application library, we'll create a Service Provider which will tell the application to instantiate the correct classes when needed.

File: app/Impl/Repo/RepoServiceProvider.php

```
1  <?php namespace Impl\Repo;
2
3  use Article; // Eloquent article
4  use Impl\Repo\Tag\EloquentTag;
5  use Impl\Repo\Article\EloquentArticle;
6  use Illuminate\Support\ServiceProvider;
7
8  class RepoServiceProvider extends ServiceProvider {
9
```

```
10      /**
11       * Register the binding
12       *
13       * @return void
14       */
15      public function register()
16      {
17          $this->app->bind('Impl\Repo\Tag\TagInterface', function($app)
18          {
19              return new EloquentTag( new Tag );
20          });
21
22          $this->app->bind('Impl\Repo\Article\ArticleInterface', function($app)
23          {
24              return new EloquentArticle(
25                  new Article,
26                  $app->make('Impl\Repo\Tag\TagInterface')
27              );
28          });
29      }
30  }
```

Now, when an instance of `ArticleInterface` is asked for in our controller, Laravel's IoC container will know to run the closure above, which returns a new instance of `EloquentArticle` (with its dependency, an instance of the `Article` model).

Add this service provider to `app/config/app.php` and you're all set!

Going Further

You may have noticed that I mentioned, but did not create, a Tag repository. This is left as an exercise for the reader, and is shown in the sample application code.

You'll need to define an interface and create an Eloquent implementation. Then the code above will function with the `TagInterface` dependency, which is registered in the `RepoServiceProvider`.

If you're wondering if it's okay to require a Tag repository inside of your Article repository, the answer is most certainly "yes". We created interfaces so that you're guaranteed that the proper methods are always available.

Furthermore, repositories are there to follow your business logic, not your database schema. Your business entities often have complex relationships between then. Using multiple Eloquent models and other repositories is absolutely necessary in order to create and modify your business-logic entities.

What have we gained?

Well, we gained more code, but we have some great reasons!

Data Sources

We're now in a position where we can change our data source. If we need to someday change from MySQL to another SQL-based server we can likely keep using `EloquentArticle` and just change our database connection in `app/config/database.php`. This is something Eloquent and many ORMs make easy for us.

However, if we need to change to a NoSQL database or even add in another data source on top of Eloquent (an API call, perhaps), we can create a new implementation without having to change code throughout our application to support this change.

As an example, if we were to change to MongoDB, we would create a `MongoDbArticle` implementation and change the class bound in the `RepoServiceProvider` - similar to how we changed the email providers in the Container chapter.

Testing

We used dependency injection in two places: Our controller and our `EloquentArticle` classes. We can now test these implementations without hitting the database by mocking an instance of `ArticleInterface` in our controller and `Eloquent/Model` in our repository.

Business Logic

We can express the true business-logic between our entities by including other repositories into our Article repository! For example, an article contains tags, and so it makes sense that our Article repository can use Tags as part of its logic.

Interfaces

You may ask yourself if you really need to use interfaces for all of your repositories. Using interfaces does add overhead. Any additions or changes to your repository, such as new public methods or changes to method parameters, should also be represented in your interface. You may find yourself editing multiple files for minor changes on the onset of your project.

This is a decision you may want to consider if you find yourself not needing an interface. Smaller projects are candidates for skipping interfaces. Larger or long-term projects can benefit greatly.

In any case, there are still many benefits to using a data repository.

Caching with the Repository Pattern

What Is It?

A cache is a place to put data that can later be retrieved quickly. A typical use case would be to cache the result of a database query and store it in memory (RAM). This allows us to retrieve the result of the query much quicker the next time we need it - we save a trip to the database and the time it takes for the database to process the query. Because the data is stored in memory, it is extremely fast.

While a database is a persistent data store, a cache is a *temporary* data storage. By design, cached data cannot be counted on to be present.

Why Do We Use It?

Caching is often added to reduce the number of times the database or other services need to be accessed by your application. If you have an application with large data sources or complex queries and processing, caching can be an indispensable tool for keeping your application fast and responsive.

Example

Now that we've seen the repository pattern in action, let's add a caching layer onto it.

To accomplish caching, we'll do something a little advanced but ultimately more maintainable. We'll be using the Decorator pattern[9] in order to "decorate" our data repository with a caching mechanism.

This will also set up our data repositories for any future decorators, perhaps loggers or profilers.

The Decorator Pattern

A decorator is a class which "wraps" a class (the "component class"), giving it the ability to add functionality around any component class method.

To accomplish this, the decorators extends (or implements) the same base class as the wrapped component class. This lets us call the same methods on the decorator as we would on the component class.

[9] http://en.wikipedia.org/wiki/Decorator_pattern

The ultimate benefit is that we can add more and more behaviors around a component class without having to actually change that component class. Additionally, we can add whichever behavior we need, as we need them, at run-time.

Let's see how that works in practice.

The Situation

The last chapter introduced the `Article` data repository. We can continue our trend of abstraction and wrap the repository with a cache decorator.

Caching works by checking if the data we want already exists in the cache. If it does, we return it to the calling code. If it does not exist, or is expired, we retrieve the data from our persistent data source (often a database), and then store it in the cache for the next request to use. Finally, we return the data to the calling code.

One issue that's common in Laravel is caching paginated results. Closures (anonymous functions) aren't able to be serialized without some mucking about. Luckily this is not an issue since we did not use the Paginator class in our repository!

Let's see how we can add caching cleanly. First, we'll create a caching service and then we'll use it within a cache decorator.

The Structure

Before we jump into the decorator pattern, we need to first create our caching service.

As usual, we'll start by building an interface. This, once again, serves as a contract - our code will expect classes to implement these interfaces so they know that certain methods will always be available.

Here's the directory structure we'll use:

```
app
|- Impl
|--- Service
|------ Cache
|--------- CacheInterface.php
|--------- LaravelCache.php
|--- Repo
```

Now we'll create the interface.

File: app/Impl/Service/Cache/CacheInterface.php

```php
<?php namespace Impl\Service\Cache;

interface CacheInterface {

        /**
         * Retrieve data from cache
         *
         * @param string         Cache item key
         * @return mixed         PHP data result of cache
         */
        public function get($key);

        /**
         * Add data to the cache
         *
         * @param string    Cache item key
         * @param mixed     The data to store
         * @param integer   The number of minutes to store the item
         * @return mixed    $value variable returned for convenience
         */
        public function put($key, $value, $minutes=null);

        /**
         * Test if item exists in cache
         * Only returns true if exists && is not expired
         *
         * @param string    Cache item key
         * @return bool     If cache item exists
         */
        public function has($key);

}
```

This interface incorporates the usual caching mechanisms. We could have used Laravel's cache class directly (which has its own interface) but, as you'll see, creating our own implementation can give us some extra configurability.

Let's create an implementation. As we'll be using Laravel's Cache package, we'll create a "Laravel" implementation.

I won't create a Memcached, File or any other specific cache storage implementation here because Laravel already abstracts away the ability to change the cache driver at will. If you're asking yourself why I add *another* layer of abstraction on top of Laravel's, it's because I'm striving to abstract away any specific implementation from my application! This goes towards maintainability (the ability to switch implementations without affecting other parts of the application) and testability (the ability to unit test with mocking).

```
1  <?php namespace Impl\Service\Cache;
2
3  use Illuminate\Cache\CacheManager;
4
5  class LaravelCache implements CacheInterface {
6
7          protected $cache;
8          protected $cachekey;
9          protected $minutes;
10
11         public function __construct(CacheManager $cache, $cachekey, $minutes=null)
12         {
13                 $this->cache = $cache;
14                 $this->cachekey = $cachekey;
15                 $this->minutes = $minutes;
16         }
17
18         public function get($key)
19         {
20                 return $this->cache->section($this->cachekey)->get($key);
21         }
22
23         public function put($key, $value, $minutes=null)
24         {
25                 if( is_null($minutes) )
26                 {
27                         $minutes = $this->minutes;
28                 }
29
30                 return $this->cache->section($this->cachekey)->put($key, $value, $minutes);
31         }
32
33         public function has($key)
34         {
35                 return $this->cache->section($this->cachekey)->has($key);
```

```
36             }
37
38 }
```

Let's go over what's happening here. This class has some dependencies:

1. An instance of Laravel's Cache
2. A cache key
3. A default number of minutes to cache data

We pass our code an instance of Laravel's Cache in the constructor method (Dependency Injection) in order to make this class unit-testable - we can mock the `$cache` dependency.

We use a cache key so each instance of this class can have a unique key. We can also later change the key to invalidate any cache created in this class, should we need to.

Finally we can set a default number of minutes to cache any item in this class. This default can be overridden in the `put()` method.

I typically use Memcached for caching. The default "file" driver does **NOT** support the used `section()` method, and so you'll see an error if you use the default "file" driver with this implementation.

A Note on Cache Keys

A good use of cache keys is worth mentioning. Each item in your cache has a unique key used to retrieve the data. By convention, these keys are often "namespaced". Laravel adds a global namespace of "Laravel" by default for each key. That's editable in `app/config/cache.php`. Should you ever need to invalidate your entire cache you can change that key. This is handy for large pushes to the code which require much of the data stored in the cache to update.

On top of Laravel's global cache namespace, the implementation above adds in a custom namespace ($cachekey). That way, any instance of this `LaravelCache` class can have its own local namespace which can also be changed. You can then quickly invalidate the cache for the keys handled by any particular instance of `LaravelCache`.

See more on namespacing in this presentation[10] by Ilia Alshanetsky, creator of Memcached.

Setting Up the Decorator

I'm going to create an abstract Article decorator, which any future article decorator will extend. Let's see what that looks like:

[10] http://ilia.ws/files/tnphp_memcached.pdf

```php
<?php namespace Impl\Repo\Article;

abstract class AbstractArticleDecorator implements ArticleInterface {

    protected $nextArticle;

    public function __construct(ArticleInterface $nextArticle)
    {
        $this->nextArticle = $nextArticle;
    }

    public function byId($id)
    {
        return $this->nextArticle->byId($id);
    }

    public function byPage($page=1, $limit=10, $all=false)
    {
        return $this->nextArticle->byPage($page, $limit, $all);
    }

    public function bySlug($slug)
    {
        return $this->nextArticle->bySlug($slug);
    }

    public function byTag($tag, $page=1, $limit=10)
    {
        return $this->nextArticle->byTag($tag, $page, $limit);
    }

}
```

There's a few things to point out here:

This abstract class implements `ArticleInterface`, just like our `EloquentArticle` class. This is important because when we use the cache decorator, we'll be treating it as if it's an instance of `EloquentArticle` itself.

The constructor method takes an instance of `ArticleInterface`. **That means it can be passed `EloquentArticle` or another decorator**.

Each method of the abstract class passes the parameters and function call through to the `nextArticle` object. This is a "pass-through" - it's not adding any functionality.

This is simply so any extending decorator class can choose which methods to wrap functionality around. For example, if our interface defined `create()` and `update()` methods, the cache decorator could skip implementing any caching around them while still making those methods callable. You'll see those two methods in the code examples on GitHub.

Using the Implementation

Now that we have an implementation of `CacheInterface`, and a `AbstractArticleDecorator` we can create our cache decorator.

Here's the file structure for the article repository, including our new decorator:

```
app
|- Impl
|--- Repo
|------ Article
|--------- AbstractArticleDecorator.php
|--------- ArticleInterface.php
|--------- CacheDecorator.php
|--------- EloquentArticle.php
|------ Tag
```

Let's create the decorator.

```
<?php namespace Impl\Repo\Article;

use Impl\Service\Cache\CacheInterface;

class CacheDecorator extends AbstractArticleDecorator {

    protected $cache;

    public function __construct(
        ArticleInterface $nextArticle, CacheInterface $cache)
    {
        parent::__construct($nextArticle);
        $this->cache = $cache;
    }

    /**
     * Attempt to retrieve from cache
     * by ID
```

```
19          */
20         public function byId($id)
21         {
22             $key = md5('id.'.$id);
23 
24             if( $this->cache->has($key) )
25             {
26                 return $this->cache->get($key);
27             }
28 
29             $article = $this->nextArticle->byId($id);
30 
31             $this->cache->put($key, $article);
32 
33             return $article;
34         }
35 
36         /**
37          * Attempt to retrieve from cache
38          */
39         public function byPage($page=1, $limit=10)
40         {
41             $key = md5('page.'.$page.'.'.$limit);
42 
43             if( $this->cache->has($key) )
44             {
45                 return $this->cache->get($key);
46             }
47 
48             $paginated = $this->nextArticle->byPage($page, $limit);
49 
50             $this->cache->put($key, $paginated);
51 
52             return $paginated;
53         }
54 
55         /**
56          * Attempt to retrieve from cache
57          */
58         public function bySlug($slug)
59         {
60             $key = md5('slug.'.$slug);
```

```
61
62          if( $this->cache->has($key) )
63          {
64              return $this->cache->get($key);
65          }
66
67          $article = $this->nextArticle->bySlug($slug);
68
69          $this->cache->put($key, $article);
70
71          return $article;
72      }
73
74      /**
75       * Attempt to retrieve from cache
76       */
77      public function byTag($tag, $page=1, $limit=10)
78      {
79          $key = md5('tag.'.$tag.'.'.$page.'.'.$limit);
80
81          if( $this->cache->has($key) )
82          {
83              return $this->cache->get($key);
84          }
85
86          $paginated = $this->nextArticle->byId($tag, $page, $limit);
87
88          $this->cache->put($key, $paginated);
89
90          return $paginated;
91      }
92
93  }
```

Let's go over what's happening in the cache decorator.

As stated above, the first dependency is yet another instance of `ArticleInterface`. We also added the dependency `CacheInterface`, which will be an instance of the cache service we created above.

If we look at the methods of the cache decorator, we'll see that they create a cache key and check if there's a valid item in the cache by that key. If there is, we return it (never actually using the `nextArticle` variable). If there is not a valid item, we call the same method with the same parameters on the `nextArticle` object.

Essentially, if an item isn't found in the cache, the cache decorator asks the `nextArticle` variable for the requested item.

The `nextArticle` object will either be another decorator or the component `EloquentArticle`. No matter how many decorators we use, we'll ultimately get the result of the database call. *The decorator pattern allows us to chain as many decorators as we need around a component class!*

Finally, note that we now had to take the `$page` and `$limit` into account when creating the cache key. Since we need to create unique cache keys for **all** variations of our data, we now need to take that into account - it wouldn't do to accidentally return the same set of articles for each page!

Tying It Together

Just as with our Article repository, our last step is to manage our new dependencies within our Service Providers.

File: app/Impl/Repo/RepoServiceProvider.php

```
1  <?php namespace Impl\Repo;
2
3  use Article;
4  use Impl\Service\Cache\LaravelCache;
5  use Impl\Repo\Article\CacheDecorator;
6  use Impl\Repo\Article\EloquentArticle;
7  use Illuminate\Support\ServiceProvider;
8
9  class RepoServiceProvider extends ServiceProvider {
10
11     /**
12      * Register the service provider.
13      *
14      * @return void
15      */
16     public function register()
17     {
18         $app = $this->app;
19
20         $app->bind('Impl\Repo\Article\ArticleInterface', function($app)
21         {
22             // Assign the Article repo to a variable
23             $article =  new EloquentArticle(
24                 new Article,
25                 $app->make('Impl\Repo\Tag\TagInterface')
```

```
            );

            // Wrap the Article repo in the
            // CacheDecorator and return it
            return new CacheDecorator(
                $article,
                // Our new Cache service class:
                new LaravelCache($app['cache'], 'articles', 10)
            );
        });
    }

}
```

How did I know to use `$app['cache']` to retrieve Laravel's Cache Manager class? I took a look at Illuminate\Support\Facades\Cache[11] and saw which key was used for Laravel's cache class within its IoC container!

Reviewing Laravel's Service Providers will give you invaluable insight into how Laravel works!

For the `EloquentArticle` repository, you can see that I am using the cache key 'articles'. This means that any cache key used for our Articles will be: `"Laravel.articles.".$key`. For instance, the cache key for an article retrieved by URL slug will be: `"Laravel.articles".md5("slug.".$slug)`.

In this way, we can:

1. Invalidate the entire cache by changing the global "Laravel" namespace in our app config
2. Invalidate the "article" cache by changing the "articles" namespace in our Service Provider
3. Invalidate the article cache's "slug" items by changing the "slug" string in our class method
4. Invalidate a specific article by changing the URL slug of our article.

We have **multiple** levels of granularity in what cached items we can manually invalidate, should the need arise!

Consider moving your cache key namespaces to a configuration file so they can be managed in one location.

How many levels of granularity you choose to use is a design decision worth taking some time to consider.

[11] https://github.com/laravel/framework/blob/master/src/Illuminate/Support/Facades/Cache.php#L10

Similar to our original repository, we cache the information relevant to handle pagination. This has the benefit of NOT making the code specific to any one pagination implementation. Instead we just store what any pagination library is likely going to need (total number of items, the current page, how many items per page) and move the responsibility of creating the Paginator object to the controller.

Final Steps

Now we can update our controller to take these changes into account:

File: app/controllers/ContentController.php

```
// Home page route
public function home()
{
        // Get page, default to 1 if not present
        $page = Input::get('page', 1);

        // Include which $page we are currently on
        $pagiData = $this->article->byPage($page);

        $articles = Paginator::make(
                $pagiData->items,
                $pagiData->totalItems,
                $pagiData->perPage
        );

        return View::make('home')->with('articles', $articles);
}
```

What Have We Gained?

We've cached database calls in a testable, maintainable way.

Cache Implementations

We can now switch out which cache implementations we use in our application. We can keep **all** of our code the same and use Laravel's config to switch between Redis, Memcached or other cache drivers. Alternatively, we can create our own implementation and define its use in the Service Provider.

Separation of Concerns

We've gone through some hurdles to not couple our code to Laravel's libraries, while still keeping the ability to leverage them. We can still swap out any cache implementation and use any pagination implementation, all-the-while still being able to cache the database query results.

Further still, rather than adding caching into our Article repository directory, we enabled our code to add layers of extra behaviors around it via decorators. The Cache decorator allowed us to add in a layer of caching without changing our previous code *at all*.

In the future, we can use other decorators to add additional functionality - perhaps logging or profiling.

Testing

Using the principles of Dependency Injection, we can still unit test each of our new classes by mocking their dependencies.

Validation as a Service

Validation and form processing is a tedious, constantly recreated wheel. Because forms and user input are so often at the heart of an application's business logic, their needs promise to always change from project to project.

Here we will discuss an attempt to make validation less painful.

What Is It?

Validation "as a service" describes moving your validation logic into a "service" which your application can use. Similar to how we started with a database repository and added a "caching service" onto it, we can start with a form and add a "validation service".

Why Do We Use It?

The goal of adding validation as a service is simply to keep a separation of concerns, while writing testable code. Lastly, we also want to make creating form validation super-easy. In this chapter, we'll create an extendable validation base class which we can use for each form. Then we'll see that in use in a controller.

In the next chapter, we'll get fancier by taking the validation completely away from the controller, and into a form-processor.

Let's start simple.

Example

The Situation

Here's some validation we might find in a controller. Let's say we need to validate user input when creating a new article in the admin area of our example blog:

A Resourceful controller

```php
<?php

class ArticleController extends BaseController {

	// GET /article/create
	public function create()
	{
		return View::make('admin.article_create', array(
			'input' => Input::old()
		));
	}

	// POST /article
	public function store()
	{
		$input = Input::all();
		$rules = array(
			'title' => 'required',
			'user_id' => 'required|exists:users,id',
			'status_id' => 'required|exists:statuses,id',
			'excerpt' => 'required',
			'content' => 'required',
			'tags' => 'required',
		)

		$validator = Validator::make($input, $rules);

		if( ! $validator->fails() )
		{
			// OMITTED - CREATING NEW ARTICLE CODE
		} else {
			return View::make('admin.article_create')
					->withInput($input)
					->withErrors($validator->getErrors());
		}
	}

}
```

As you can see, we have all this *stuff* in our controller to validate user input, and the above example

doesn't even show the code actually processing the form! Additionally, what if we need the same validation rules for updating the article? We'd have to repeat code in our update method.

In an effort to make the controller skinnier and make our code more reusable and testable, let's see how to move validation out of our controllers.

Restructuring

We'll turn to our `Impl` application library as always. Validation is treated as a **Service** to our application and so we'll create our Validation code under the `Impl\Service` namespace. Here's what we'll see in there:

```
app
|- Impl
|- [ . . . ]
|--- Service
|------ Validation
|--------- AbstractLaravelValidator.php
|--------- ValidableInterface.php
```

The Interface

We'll start, as we usually do, by creating an interface. What methods will our controllers be using on our validation class?

We know we'll need to gather input, test input validity and retrieve error messages. Let's start with those needs:

File: app/Impl/Service/Validation/ValidableInterface.php

```php
<?php namespace Impl\Service\Validation;

interface ValidableInterface {

        /**
         * Add data to validation against
         *
         * @param array
         * @return \Impl\Service\Validation\ValidableInterface
         */
        public function with(array $input);

```

```
13          /**
14           * Test if validation passes
15           *
16           * @return boolean
17           */
18          public function passes();
19
20          /**
21           * Retrieve validation errors
22           *
23           * @return array
24           */
25          public function errors();
26
27  }
```

So, any validation class used in our application should implement this interface. From this, we can see how we'll invoke an implementation of it - likely something like this:

```
1  // Somewhere in your code
2  if( ! $validator->with($input)->passes() )
3  {
4          return $validator->errors();
5  }
```

An Abstraction

Now, usually after an interface we can go straight into creating concrete implementations of it. We'll do that, but before we do, we're going to add one more layer.

Validation is a service where it might make sense to generalize. We're going to make an abstract class which uses the Laravel Validator and can then be extended and modified to be easily used in any form. Once done, we'll be able to extend this abstract class, change a few parameters, and be on our way for any validation scenario.

Our abstract class will implement our interface and use Laravel's validator library:

File: app/Imple/Service/Validation/AbstractLaravelValidator.php

```php
<?php namespace Impl\Service\Validation;

use Illuminate\Validation\Factory as Validator;

abstract class AbstractLaravelValidator implements ValidableInterface {

    /**
     * Validator
     *
     * @var \Illuminate\Validation\Factory
     */
    protected $validator;

    /**
     * Validation data key => value array
     *
     * @var Array
     */
    protected $data = array();

    /**
     * Validation errors
     *
     * @var Array
     */
    protected $errors = array();

    /**
     * Validation rules
     *
     * @var Array
     */
    protected $rules = array();

    /**
     * Custom validation messages
     *
     * @var Array
     */
    protected $messages = array();
```

```
41
42      public function __construct(Validator $validator)
43      {
44          $this->validator = $validator;
45      }
46
47      /**
48       * Set data to validate
49       *
50       * @return \Impl\Service\Validation\AbstractLaravelValidation
51       */
52      public function with(array $data)
53      {
54          $this->data = $data;
55
56          return $this;
57      }
58
59      /**
60       * Validation passes or fails
61       *
62       * @return Boolean
63       */
64      public function passes()
65      {
66          $validator = $this->validator->make(
67              $this->data,
68              $this->rules,
69              $this->messages
70          );
71
72          if( $validator->fails() )
73          {
74              $this->errors = $validator->messages();
75              return false;
76          }
77
78          return true;
79      }
80
81      /**
82       * Return errors, if any
```

```
     *
     * @return array
     */
    public function errors()
    {
        return $this->errors;
    }

}
```

Next, we'll see how we can simply extend this class to meet the needs for any input validation.

Implementation

Let's create a validator class for the example at the beginning of this chapter:

File: app/Impl/Service/Validation/ArticleFormValidator.php

```
<?php namespace Impl\Service\Validation;

class ArticleFormValidator extends AbstractLaravelValidator {

    /**
     * Validation rules
     *
     * @var Array
     */
    protected $rules = array(
        'title' => 'required',
        'user_id' => 'required|exists:users,id',
        'status_id' => 'required|exists:statuses,id',
        'excerpt' => 'required',
        'content' => 'required',
        'tags' => 'required',
    );

    /**
     * Validation messages
     *
     * @var Array
     */
```

```
    protected $messages = array(
        'user_id.exists' => 'That user does not exist',
        'status_id.exists' => 'That status does not exist',
    );

}
```

That's it! Let's see this in action back in our controller:

A Resourceful controller

```
<?php

use Impl\Service\Validation\ArticleLaravelValidator;

class ArticleController extends BaseController {

        // Class Dependency: Concrete class ArticleLaravelValidator
        // Note that Laravel resolves this for us
        // We will not need a Service Provider
        public function __construct(ArticleLaravelValidator $validator)
        {
                $this->validator = $validator;
        }

        // GET /article/create
        public function create()
        {
                return View::make('admin.article_create', array(
                        'input' => Input::old()
                ));
        }

        // POST /article
        public function store()
        {
                if( $this->validator->with( Input::all() )->passes() )
                {
                        // FORM PROCESSING
                } else {
                        return View::make('admin.article_create')
```

```
31                                          ->withInput( Input::all() )
32                                          ->withErrors($validator->errors());
33                  }
34          }
35
36  }
```

What Did We Gain?

We were able to make our controllers a bit skinnier by taking out the boiler-plate validation and moving it off into our application library.

Additionally, we created a very reusable abstract class. For any new form, we can extend this abstract class, define our rules, and be done with it!

Up next we'll see how we can get even more stream-lined by moving all of the form processing out of the controller altogether.

Form processing

One half of handling user input is Validation. The other half is, of course, form processing.

What Is It?

Let's go over what "form processing" is. If you have a form, and a user submits info, the steps taken in code generally are:

1. Gather user input
2. Validate user input
3. Perform some operation with user input
4. Generate a response

Where Do We Use It?

A controller is an excellent place for items 1 and 4 above. We've seen how validation (step 2) can be handled as a service. Processing the form (step 3) can *also* be handled by a service within our application code!

Moving the form processing out of the controller and into a set of service classes gives us a better opportunity to unit test it. We can take away the dependency of it having to run in context of an HTTP request, allowing us to mock its dependencies.

In short, moving the form code out of the controller gives us a more reusable, more testable and finally more maintainable way to handle our form code.

Example

We have validation in place already from the previous chapter, so let's see how we can roll up Validation and Form processing into a better home.

The Situation

We'll be creating a Form service, which makes use of our Repository class to handle the CRUD, and Validation service to handle user input validation.

From the previous chapter on Validation, we ended up with a controller which has validation setup as a service:

A Resourceful controller

```
// POST /article
public function store()
{
    if( $this->validator->with( Input::all() )->passes() )
    {
        // FORM PROCESSING
    } else {
        return View::make('admin.article_create')
                ->withInput( Input::all() )
                ->withErrors($validator->errors());
    }
}
```

What we can do now is setup the Form processing as a service.

After doing so, here's what our new directory structure will look like:

```
app
|- Impl
|--- Service
|------ Form
|--------- Article
|------------ ArticleForm.php
|------------ ArticleFormLaravelValidator.php
|--------- FormServiceProvider.php
```

Restructuring

This is a rare occasion where we **won't** start by creating an implementation. The form class will *use* our interfaced classes, but won't itself implement a specific interface. This is because there aren't other implementations of the form to take on. It's pure PHP - it will take input in the form of an array and *orchestrate* the use of the validation service and data repository.

Validation

Let's start with using the validation from the last chapter. In this example, we'll create a form for creating *and* updating articles. In both cases in this example, the validation rules are the same. Lastly, note that we're moving the article form validator class into the `Form` service directory.

File: app/Impl/Service/Form/Article/ArticleFormLaravelValidator.php

```php
<?php namespace Impl\Service\Form\Article;

use Impl\Service\Validation\AbstractLaravelValidator;

class ArticleFormLaravelValidator extends AbstractLaravelValidator {

    /* Same as previous chapter, with new namespace & location */

}
```

Form

Now let's create our `ArticleForm` class. As noted, this class will orchestrate the creating and editing of articles, using validation, data repositories and data input.

File: app/Impl/Service/Form/Article/ArticleForm.php

```php
<?php namespace Impl\Service\Form\Article;

use Impl\Service\Validation\ValidableInterface;
use Impl\Repo\Article\ArticleInterface;

class ArticleForm {

        /**
         * Form Data
         *
         * @var array
         */
        protected $data;

        /**
          * Validator
          *
          * @var \Impl\Service\Form\Contracts\ValidableInterface
          */
        protected $validator;

```

```
22          /**
23           * Article repository
24           *
25           * @var \Impl\Repo\Article\ArticleInterface
26           */
27          protected $article;
28
29          public function __construct(
30                  ValidableInterface $validator, ArticleInterface $article)
31          {
32                  $this->validator = $validator;
33                  $this->article = $article;
34          }
35
36          /**
37           * Create an new article
38           *
39           * @return boolean
40           */
41          public function save(array $input)
42          {
43                  // Code to go here
44          }
45
46          /**
47           * Update an existing article
48           *
49           * @return boolean
50           */
51          public function update(array $input)
52          {
53                  // Code to go here
54          }
55
56          /**
57           * Return any validation errors
58           *
59           * @return array
60           */
61          public function errors()
62          {
63                  return $this->validator->errors();
```

```
64          }
65
66          /**
67           * Test if form validator passes
68           *
69           * @return boolean
70           */
71          protected function valid(array $input)
72          {
73                  return $this->validator->with($input)->passes();
74          }
75
76  }
```

So here's the shell of our Article form. We can see that we're injecting the `ValidableInterface` and `ArticleInterface` dependencies, and have a method for returning validation errors.

Let's see what creating and updating an article looks like:

A Resourceful controller

```
1   /**
2    * Create an new article
3    *
4    * @return boolean
5    */
6   public function save(array $input)
7   {
8           if( ! $this->valid($input) )
9           {
10                  return false;
11          }
12
13          return $this->article->create($input);
14  }
15
16  /**
17   * Update an existing article
18   *
19   * @return boolean
20   */
21  public function update(array $input)
```

```
{
        if( ! $this->valid($input) )
        {
                return false;
        }

        return $this->article->update($input);
}
```

Well, that was easy! Our form is handling the processing, but actually not doing the heavy lifting of creating or updating an article. Instead, it's passing that responsibility off to our interfaced repository classes which do the hard work for us. The form class is merely orchestrating this process.

Heavy Lifting

Now we actually have the form processing in place. It takes user input, runs the validation, returns errors and sends the data off for processing.

The last step is to do the heavy lifting of creating/updating the articles.

Let's go back to our Article interface. Since we know we need to create and update articles, we can make those into the required methods `create()` and `update()`. Let's see what that looks like:

File: app/Impl/Repo/Article/ArticleInterface.php

```
<?php namespace Impl\Repo\Article;

interface ArticleInterface {

        /* Previously covered methods omitted */

        /**
         * Create a new Article
         *
         * @param array  Data to create a new object
         * @return boolean
         */
        public function create(array $data);

        /**
         * Update an existing Article
         *
```

```
18          * @param array  Data to update an Article
19          * @return boolean
20          */
21         public function update(array $data);
```

Now that we know how our code expects to interact with our repository, let's roll our sleeves up and start the dirty work.

File: app/Impl/Repo/Article/EloquentArticle.php

```
1  <?php namespace Impl\Repo\Article;
2
3  use Impl\Repo\Tag\TagInterface;
4  use Impl\Service\Cache\CacheInterface;
5  use Illuminate\Database\Eloquent\Model;
6
7  class EloquentArticle implements ArticleInterface {
8
9      protected $article;
10     protected $tag;
11     protected $cache;
12
13     // Class expects an Eloquent model
14     public function __construct(
15         Model $article, TagInterface $tag, CacheInterface $cache)
16     {
17         $this->article = $article;
18         $this->tag = $tag;
19         $this->cache = $cache;
20     }
21
22     // Previous implementation code omitted
23
24     /**
25      * Create a new Article
26      *
27      * @param array  Data to create a new object
28      * @return boolean
29      */
30     public function create(array $data)
31     {
```

```
        // Create the article
        $article = $this->article->create(array(
            'user_id' => $data['user_id'],
            'status_id' => $data['status_id'],
            'title' => $data['title'],
            'slug' => $this->slug($data['title']),
            'excerpt' => $data['excerpt'],
            'content' => $data['content'],
        ));

        if( ! $article )
        {
            return false;
        }

        // Helper method
        $this->syncTags($article, $data['tags']);

        return true;
    }

    /**
     * Update an existing Article
     *
     * @param array  Data to update an Article
     * @return boolean
     */
    public function update(array $data)
    {
        $article = $this->article->find($data['id']);

        if( ! $article )
        {
            return false;
        }

        $article->user_id = $data['user_id'];
        $article->status_id = $data['status_id'];
        $article->title = $data['title'];
        $article->slug = $this->slug($data['title']);
        $article->excerpt = $data['excerpt'];
        $article->content = $data['content'];
```

```
74          $article->save();
75
76          // Helper method
77          $this->syncTags($article, $data['tags']);
78
79          return true;
80      }
81
82      /**
83       * Sync tags for article
84       *
85       * @param \Illuminate\Database\Eloquent\Model  $article
86       * @param array  $tags
87       * @return void
88       */
89      protected function syncTags(Model $article, array $tags)
90      {
91          // Return tags after retrieving
92          // existing tags & creating new tags
93          $tags = $this->tag->findOrCreate( $tags );
94
95          $tagIds = array();
96          $tags->each(function($tag) use ($tagIds)
97          {
98              $tagIds[] = $tag->id;
99          });
100
101         // Assign set tags to article
102         $this->article->tags()->sync($tagIds);
103     }
104
105 }
```

So, we now have an implementation with Eloquent to create or update an article. We also have a method `findOrCreate()` in the tags repository which (you guessed it) finds tags or creates them if they don't exist already. This allows us to add new tags as we need when creating or editing an article.

The implementation of the `Tags` repository method `findOrCreate` can be seen in the book's Github repository.

The steps taken to assign tags to each article is repeated often, and so the protected method `syncTags` was created to handle it. This method does not need to be part of the interface as it's an internal method only used by methods within the class!

Wrapping Up

Although we don't need a Service Provider to define what form class to create (there's only one concrete `ArticleForm` class), we do need to define what dependencies get injected into it. These are Laravel's validator class, and the implementation of `ArticleInterface`.

We'll do that in a Service Provider. Let's see what that looks like:

File: app/Impl/Service/Form/FormServiceProvider.php

```
<?php namespace Impl\Service\Form;

use Illuminate\Support\ServiceProvider;
use Impl\Service\Form\Article\ArticleForm;
use Impl\Service\Form\Article\ArticleFormLaravelValidator;

class FormServiceProvider extends ServiceProvider {

    /**
     * Register the binding
     *
     * @return void
     */
    public function register()
    {
        $app = $this->app;

        $app->bind('Impl\Service\Form\Article\ArticleForm', function($app)
        {
            return new ArticleForm(
                    new ArticleFormLaravelValidator( $app['validator'] ),
                    $app->make('Impl\Repo\Article\ArticleInterface')
            );
        });
    }

}
```

The last step is to add this to your `app/config/app.php` with the other Service Providers.

The Final Results

Now we can see what our controller finally looks like! We've moved all form processing and validation logic out of the controller. We can now test and reuse those independently of the controller. The only logic remaining in the controller, other than calling our form processing class, is using Laravel's Facades for redirecting and responding to the requests.

A Resourceful controller

```
<?php

use Impl\Repo\Article\ArticleInterface;
use Impl\Service\Form\Article\ArticleForm;

class ArticleController extends BaseController {

    protected $articleform;

    public function __construct(
            ArticleInterface $article, ArticleForm $articleform)
    {
            $this->article = $article;
            $this->articleform = $articleform;
    }

    /**
     * Create article form
     * GET /admin/article/create
     */
    public function create()
    {
        View::make('admin.article_create', array(
            'input' => Session::getOldInput()
        ));
    }

    /**
     * Create article form processing
     * POST /admin/article
```

```
31          */
32         public function store()
33         {
34             // Form Processing
35             if( $this->articleform->save( Input::all() ) )
36             {
37                 // Success!
38                 return Redirect::to('admin.article')
39                         ->with('status', 'success');
40             } else {
41
42                 return Redirect::to('admin.article_create')
43                         ->withInput()
44                         ->withErrors( $this->articleform->errors() )
45                         ->with('status', 'error');
46             }
47         }
48
49         /**
50          * Create article form
51          * GET /admin/article/{id}/edit
52          */
53         public function edit()
54         {
55             View::make('admin.article_edit', array(
56                 'input' => Session::getOldInput()
57             ));
58         }
59
60         /**
61          * Create article form
62          * PUT /admin/article/{id}
63          */
64         public function update()
65         {
66             // Form Processing
67             if( $this->articleform->update( Input::all() ) )
68             {
69                 // Success!
70                 return Redirect::to('admin.article')
71                         ->with('status', 'success');
72             } else {
```

```
73
74              return Redirect::to('admin.article_edit')
75                      ->withInput()
76                      ->withErrors( $this->articleform->errors() )
77                      ->with('status', 'error');
78          }
79      }
80
81  }
```

What Did We Gain?

We now made our form processing more testable and maintainable:

- We can test the `ArticleForm` class with a liberal use of mocks.
- We can change the form's validation rules easily.
- We can swap out data storage implementations without changing the form code.
- We gain code reusability: We can call the form code outside of the context of an HTTP request and we can reuse the data repository and validation classes!

Building up our application using the repository pattern and utilizing dependency injection and other SOLID practices has, in total, enabled us to make form processing less insufferable!

Error Handling

What response a user sees as the result of a request is driven by Laravel. Laravel acts as the intermediary between our application code and an HTTP request, and so has the responsibility of generating the response to a request. Our application merely supplies Laravel with the data to respond with.

Going further, how Laravel responds to *errors* depends on the context. Laravel is making requests on our application and returning a representation of the application's response. In a browser, an HTML representation is returned. In an API call, a JSON object might be returned. Lastly, in a CLI call, a plain-text response may be most appropriate.

To that end, there really isn't a need to do a lot of extra work for error handling. We can intercept errors and decide how to show them (or not) to the user using Laravel's built-in mechanisms.

However, there is the issue of business logic errors. You may need to handle your errors within your business logic separately from Laravel-specific errors. For example, you may need to be alerted about failures in a process important to your application, while finding it safe to ignore most 404 HTTP errors.

In this chapter, we'll see how we can set your application up to throw certain Exceptions and use a custom error handler to perform any needed actions.

Specifically, we'll cover how to catch business-logic exceptions and send a notification as a result.

What is Error Handling?

Error handling is, rather obviously, how your application responds to an error.

There are two main things to consider when handling an error:

1. Taking action on the error
2. Displaying a meaningful response about the error

Taking action on an error usually involves logging and alerting. You want a log of any error so that they can be audited and for debugging. You may also need some sort of alert mechanism for severe errors which stop critical business logic from performing.

Displaying a meaningful response is, as mentioned, all about context. In development, we want errors to display to the developer. If they are using a console, a console-readable error is appropriate. If they are using a browser, then a browser-based error response is appropriate. If we are in production, then a "friendly" error response is appropriate for our users.

How Laravel Handles Errors

Laravel uses the Chain of Responsibility Pattern[12] for handling errors. What this means for error handling is that Laravel will send any Exception to each registered error handler (in order of registration) until a handler successfully "handles" the error. In Laravel, an error is considered "handled" when it returns any non-null value.

This gives you the opportunity to actually capture a specific Exception, handle it in any way (perhaps logging it), and then letting Laravel pass the Exception off to a handler which ultimately chooses what to display to the end-user (if anything is presented at all).

Setting Up Laravel to Respond to Errors

As we have discussed, Laravel should be ultimately responsible for the display of errors. To do so, I put any error handle returning a response in the `routes.php` file, essentially treating it like a defined route.

The trick here is to decide when to show an error and when to display something "friendly" to a user.

File: app/routes.php

```
1  /*
2      With error handling, you should define your least specific errors first.
3
4      In the example below, an incorrect URL will be handled by the
5  type-hinted `NotFoundHttpException` handler first, then by the generic
6  `Exception` handler.
7
8  Reversing the order of these will result in the `NotFoundHttpException`
9  handler being skipped when debugging is off!
10 */
11
12 use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;
13
14 // Other Exceptions
15 App::error(function(\Exception $e)
16 {
17
18     if( Config::get('app.debug') === true )
19     {
```

[12] http://en.wikipedia.org/wiki/Chain-of-responsibility_pattern

```
20          return null; // Fallback to Laravel's error handler
21      }
22
23      return View::make('error');
24
25  });
26
27  // 404
28  App::error(function(NotFoundHttpException $e)
29  {
30
31      if( Config::get('app.debug') === true )
32      {
33          return null; // Fallback to Laravel's error handler
34      }
35
36      return View::make('404');
37
38  });
```

Note that what order you define the errors matters. Laravel will roll through each error handler until a (non-null) response is returned, and so you want your most-specific errors defined last. Above, we see the `NotFoundHttpException` registered *after* the generic `Exception` handler because we want any "not found" error to be handled in a specific way before the generic exception handler gets to it.

Setting Up Our Application to Respond to Errors

So, now that we can handle errors shown to web users, let's decide how to handle errors coming from our application code.

First, we'll define an interface. We already know that we want to "handle" an error, so we'll start there:

File: app/Impl/Exception/HandlerInterface.php

```
<?php namespace Impl\Exception;

interface HandlerInterface {

        /**
         * Handle Impl Exceptions
         *
         * @param \Impl\Exception\ImplException
         * @return void
         */
        public function handle(ImplException $exception);

}
```

So, any error handler will "handle" an exception - and not just any, but a `ImplException` and its subclasses.

Let's define that base exception as well. This will be used as the base class for any app-specific exceptions thrown.

File: app/Impl/Exception/ImplException.php

```
<?php namespace Impl\Exception;

class ImplException extends \Exception {}
```

Now we can decide what we want our handler to do. Let's say for a specific exception, we want our application to notify us of the error.

Implementation

Since we have a `HandlerInterface`, the next step is to create an implementation of it. We want it to notify us of an error, so we'll call it `NotifyHandler`.

File: app/Impl/Exception/NotifyHandler.php

```php
<?php namespace Impl\Exception;

use Impl\Service\Notification\NotifierInterface;

class NotifyHandler implements HandlerInterface {

    protected $notifier;

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

    /**
     * Handle Impl Exceptions
     *
     * @param \Impl\Exception\ImplException
     * @return void
     */
    public function handle(ImplException $exception)
    {
        $this->sendException($exception);
    }

    /**
     * Send Exception to notifier
     * @param  \Exception $exception Send notification of exception
     * @return void
     */
    protected function sendException(\Exception $e)
    {
        $this->notifier->notify('Error: '.get_class($e), $e->getMessage());
    }

}
```

Now we have a class to notify us of an exception! Our last step is to set this as a handler for our application Exceptions.

You may have noticed there is a dependency on this implementation. For now we can assume that the `NotifierInterface` class, whose implementation is not shown here, is a class which will send us an alert of some kind. In the next chapter, we'll cover using a third-party library which will be used to send notifications.

Putting it Together.

Our last step is to use this error handler when a `ImplException` (or subclass) is thrown. For that, we use Laravel's error handling capabilities.

File: app/Impl/Exception/ExceptionServiceProvider.php

```
<?php namespace Impl\Exception;

use Illuminate\Support\ServiceProvider;

class ExceptionServiceProvider extends ServiceProvider
{
        public function register()
        {
                $app = $this->app;

                // Bind our app's exception handler
                $app['impl.exception'] = $app->share(function($app)
                {
                        return new NotifyHandler( $app['impl.notifier'] );
                });
        }

        public function boot()
        {
                $app = $this->app;

                // Register error handler with Laravel
                $app->error(function(ImplException $e) use ($app)
                {
                        $app['impl.exception']->handle($e);
                });
        }
}
```

What's happening here? First, the `register` method registers `impl.exception` as the error handler for our application.

Then the `boot()` method uses Laravel's built-in error handling and registers our `impl.exception` handler for any exception raised of base class `ImplException`.

Now, when any `ImplException` is thrown, our handler will capture it and handle the error. Since we are not returning a `Response` from our handler, Laravel will continue to cycle through it's own error handlers, which we can utilize to return an appropriate response back to the end-user.

Here again we see that we're using `$app['impl.notifier']`, which is not yet defined in our application. In the next chapter, we'll cover using a third-party library to send the notifications.

Using Packages

Using a Package: Notifications

In our previous chapter, we setup an error handler which sends us a notification upon an application error.

In this chapter, we'll create the notification functionality by using a third-party Composer package.

We're going to send an SMS message to our phones when an exception is raised.

Setup

We need a tool which can send notifications of some kind. In that sense, our application needs a notification service.

Since we identified it as a service to our application, we have an idea of where it will go:

```
app
|- Impl
|--- Service
|------ Notification
```

Next, as usual, we'll make an interface. What does our notifier need to do? Well, it needs a message to send, to know who to send it to, as well as to know who sent the notification. Let's start there:

File: app/Impl/Service/Notification/NotifierInterface.php

```php
<?php namespace Impl\Service\Notification;

interface NotifierInterface {

        /**
         * Recipients of notification
         * @param  string $to The recipient
         * @return Impl\Service\Notification\NotifierInterface
         */
        public function to($to);

        /**
         * Sender of notification
```

```
14          * @param  string $from The sender
15          * @return Impl\Service\Notification\NotifierInterface
16          */
17         public function from($from);
18
19         /**
20          * Send notification
21          * @param  string $subject Subject of notification
22          * @param  string $message Notification content
23          * @return void
24          */
25         public function notify($subject, $message);
26
27 }
```

Implementation

Next, we need to implement this interface. We've decided to send an SMS to ourselves. To do this, we can use the service Twilio.

Twilio has a PHP SDK available as a Composer package. To add it to our project, we can add it to the `composer.json` file:

```
1 // Add to composer.json as a requirement
2 // or run this in our project root:
3 $ php composer require twilio/sdk:dev-master
```

After installing the SDK, we can start using it. Let's create an `SmsNotifier` which uses Twilio.

File: app/Impl/Service/Notification/SmsNotifier.php

```
1  <?php Impl\Service\Notification;
2
3  use Services_Twilio;
4
5  class SmsNotifier implements NotifierInterface {
6
7      /**
8      * Recipient of notification
9      * @var string
10     */
```

```
11      protected $to;
12
13      /**
14      * Sender of notification
15      * @var string
16      */
17      protected $from;
18
19      /**
20      * Twilio SMS SDK
21      * @var \Services_Twilio
22      */
23      protected $twilio;
24
25      public function __construct(Services_Twilio $twilio)
26      {
27          $this->twilio = $twilio;
28      }
29
30      /**
31       * Recipients of notification
32       * @param  string $to The recipient
33       * @return Impl\Service\Notificaton\SmsNotifier
34       */
35      public function to($to)
36      {
37          $this->to = $to;
38
39          return $this;
40      }
41
42      /**
43       * Sender of notification
44       * @param  string $from The sender
45       * @return Impl\Service\Notificaton\NotifierInterface
46       */
47      public function from($from)
48      {
49          $this->from = $from;
50
51          return $this;
52      }
```

```
53
54      /**
55       * Send notification
56       * @param  string $subject Subject of notification
57       * @param  string $message Notification content
58       * @return void
59       */
60      public function notify($subject, $message)
61      {
62          $this->twilio
63                ->account
64                ->sms_messages
65                ->create(
66                  $this->from,
67                  $this->to,
68                  $this->subject."\n".$this->message
69                );
70      }
71
72  }
```

So that's really the bulk of the work. We defined an interface and implemented an SMS notifier, which uses Twilio's PHP SDK.

Note that if we needed to switch which SMS provider we use, we could take more steps towards abstracting that out by creating "Transports". For instance, one `SmsTransport` would be Twilio, but could be any other SMS provider. For simplicity, the SmsNotifier simply assumes Twilio and creates no abstraction towards separate "transports".

Check out how Laravel uses Swift Mailer (in the `Illuminate\Mail` package) for sending emails as an example of how transports can be used.

Tying it Together

The last step is to add configuration for our Twilio account, and putting everything together in a Service Provider.

For configuration, we can create a new Twilio configuration file:

File: app/config/twilio.php

```
<?php

return array(

        'from' => '555-1234',

        'to' => '555-5678',

        'account_id' => 'abc1234',

        'auth_token' => '1111111',

);
```

Now this configuration file will be available to us in our application. Next we can create the Service Provider to tie everything together.

File: app/Impl/Service/Notification/NotificationServiceProvider.php

```
<?php namespace Impl\Service\Notification;

use Services_Twilio;
use Illuminate\Support\ServiceProvider;

class NotificationServiceProvider extends ServiceProvider {

    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
        $app = $this->app;

        $app['impl.notifier'] = $app->share(function($app)
        {
            $config = $app['config'];
```

```
20
21              $twilio = new Services_Twilio(
22                  $config->get('twilio.account_id'),
23                  $config->get('twilio.auth_token')
24              );
25
26              $notifier = SmsNotifier( $twilio );
27
28              $notifier->from( $config['twilio.from'] )
29                       ->to( $config['twilio.to'] );
30
31              return $notifier;
32          });
33      }
34
35  }
```

So, we used Twilio's SDK, passed it our api credentials from configuration, set our "to" and "from" from configuration and passed that off to our `SmsNotifier`.

Register this Service Provider in the `app/config/app.php` file and we're good to go.

In Action

We can see this in action back in our error handler. Remember, in our `ExceptionServiceProvider`, we created an error handler which used the SMS notifier:

File: app/Impl/Exception/ExceptionServiceProvider.php

```
1   public function register()
2   {
3       $app = $this->app;
4
5       $app['impl.exception'] = $app->share(function($app)
6       {
7           return new NotifyHandler( $app['impl.notifier'] );
8       });
9   }
10
11  public function boot()
12  {
```

```
13      $app = $this->app;
14
15      $app->error(function(ImplException $e) use ($app)
16      {
17          $app['impl.exception']->handle($e);
18      });
19  }
```

Now `$app['impl.notifier']` exists for our `NotifyHandler` to use as its notifier implementation!

You can test this by throwing a `ImplException` in your code and waiting to receive a text message.

```
1  throw new ImplException('Test message');
```

Furthermore, if you want to use the SMS notifier anywhere else in your code, you can!

```
1  $sms = App::make('notification.sms');
2
3  $sms->to('555-5555')
4          ->notify($subject, $message);
```

What Did We Gain?

We saw an example of how to install and use a third-party Composer package. In this example we used Twilio's SMS SDK in an SMS-based implementation of our Notification service.

Packagist[13] is a go-to resource for discovering Composer packages. Whenever you find yourself about to code any functionality, you should check there first to see if something already exists. Chances are a well-coded, tested package exists!

[13] http://packagist.org

Conclusion

Review

We covered a lot of ground in this book! Some of the topics included:

Installation

Installing Laravel, including environment setup and production considerations.

Application Setup

Using an application library.

Repository Pattern

The how and why of using repositories as an interface to your data storage.

Caching In the Repository

We added a service layer into our code repository. In this example, we added a layer of caching.

Validation

We created validation as a service, helping us abstract out the work of validation and implement validation classes specific to our needed use cases.

Form Processing

We created classes for orchestrating the validation of input and the interaction with our data repositories. These form classes are decoupled from HTTP requests and are able to be used outside of a controller. This also makes them easier to test.

Error Handling

We went over some considerations of how and when to use error handling, and saw an example of how it might be done to catch application-specific errors.

Third Party Libraries

We used Twilio's SDK to build a notification library so could send SMS notifications within our error handler.

What Did You Gain?

Most of this book lays the ground work for building a full featured application.

What I hope you gained from reading this is insight on how SOLID principles can be used in Laravel. The use of interfaces, the IoC container and the Service Providers all provide a powerful way to create a highly testable and maintainable PHP application.

The Future

I'm hoping the early editions of this book serve as a base on which to build.

The answer to many questions such as "How do I use queues effectively?", "How do I integrate a search engine?" or "What's an advanced usage of Service Providers" will rely and expand on the fundamentals covered here.

With any luck, this will be just the beginning.

Happy coding!