



Implementing Laravel

CHRIS FIDAO

Implementing Laravel (ES)

implementando Laravel

Chris Fidao y Judas Borbón

Este libro está a la venta en <http://leanpub.com/implementinglaravel-es>

Esta versión se publicó en 2013-10-16



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2013 Chris Fidao y Judas Borbón

Índice general

Agradecimientos	i
Introducción	ii
¿A quién va dirigido el libro?	iii
¿Quién le sacará mayor provecho?	iii
Algo que saber de antemano	iii
Una nota sobre opiniones	iv
SOLID	1
Conceptos fundamentales	3
El contenedor	4
Uso básico	4
Algo más avanzado	5
Inversión de control	5
Uso en una situación real	7
Inyección de dependencias	9
¿Qué es la inyección de dependencias?	9
Agregando dependencias a los controladores	10
Interfaces como dependencias	11
¿Por qué la inyección de dependencias?	13
En resumen.	17
Preparando Laravel	18
La aplicación de ejemplo	19
Base de datos	19
Modelos	19
Relaciones	20

ÍNDICE GENERAL

Comprobabilidad y mantenibilidad	20
Notas de arquitectura	20
Instalación	22
Instalación de Composer	22
Creación de un nuevo proyecto	22
Configuración	23
En resumen	26
Preparación de la aplicación	27
Preparando la biblioteca de la aplicación	27
Autocarga	28
En resumen	29
Patrones útiles	30
El patrón repositorio	31
¿Qué es?	31
¿Para qué se usa?	31
Ejemplo	33
Uso de caché con el patrón repositorio	44
¿Qué es?	44
¿Para qué se usa?	44
Ejemplo	44
Validación como servicio	57
¿Qué es?	57
¿Para qué se usa?	57
Ejemplo	57
Reestructurando	59
¿Qué hemos logrado?	65
Procesamiento de formularios	66
¿Qué es?	66
¿Dónde lo utilizamos?	66
Ejemplo	66
Reestructurando	67
Trabajo pesado	71
En resumen	75
Resultados finales	76
¿Qué hemos logrado?	78
Manejo de errores	79

ÍNDICE GENERAL

Uso de paquetes	85
Uso de paquetes: notificaciones	86
Preparación.	86
Implementación	87
Juntando las piezas	89
En acción	91
¿Qué hemos logrado?	92
Conclusión	93
Recapitulación	94
Instalación	94
Preparación de la aplicación	94
Patrón repositorio	94
Caché en el repositorio	94
Validación	94
Procesamiento de formularios	94
Manejo de errores	95
Librerías externas	95
¿Qué hemos logrado?	96
El futuro	96

Agradecimientos

Gracias a Natalie por su paciencia, a mis revisores por ayudarme a mejorar esto y al equipo de Digital Surgeons por su apoyo (y por Gumby¹).

¹<http://gumbyframework.com>

Introducción

¿A quién va dirigido el libro?

¿Quién le sacará mayor provecho?

El libro va dirigido a aquellos que conocen los aspectos fundamentales de Laravel y buscan ejemplos más avanzados de cómo implementar su conocimiento de una forma más orientada a las pruebas y el mantenimiento.

En estas lecciones, aprenderán cómo aplicar conceptos estructurales en Laravel de diversas maneras, con el fin de que puedan usarlos y adaptarlos a sus propias necesidades.

Algo que saber de antemano

Todos tenemos distintos niveles de conocimiento. El libro es para quienes están familiarizados con los aspectos básicos de Laravel 4. Por lo tanto, resulta de gran ayuda que el lector cuente con algunos conocimientos del tema.

El libro de Taylor Otwell *Laravel: From Apprentice to Artisan*² es un gran prerequisito. Y aunque abordaré dichos temas de manera general, espero que los lectores tengan una comprensión básica de los principios SOLID y el contenedor IoC de Laravel.

²<https://leanpub.com/laravel>

Una nota sobre opiniones

Conocer los beneficios (y dificultades!) del patrón Repositorio, la Inyección de Dependencias, el Localizador de Servicios y otras herramientas de estructuración puede ser muy liberador y emocionante.

El uso de estas herramientas, sin embargo, puede estar plagado de problemas inesperados.

Es por ello que hay muchas opiniones sobre cómo elaborar “buen código” con dichas herramientas.

Ya que uso muchos ejemplos reales en el libro, he incluído implícitamente (y en ocasiones explícitamente) mi opinión. Sin embargo, siempre hay que formarnos una opinión propia con lo que leemos y en base a nuestra experiencia.



“Cuando lo único que tienes es un martillo...”

El uso excesivo de estas herramientas puede traer problemas. Los capítulos muestran ejemplos de cómo se *pueden* implementar las herramientas de las que se disponen. Saber cuando *no* usarlas también es algo importante que hay que tener en cuenta.

SOLID

Ya que durante el libro mencionaré los principios SOLID, incluiré una breve explicación, tomada en gran parte de [Wikipedia³](#), con una explicación extra en el contexto de Laravel.

Principio de Responsabilidad Única (Single Responsibility Principle)

Una clase (o unidad de código) debe tener una sola responsabilidad.

Principio Abierto/Cerrado (Open/Closed Principle)

Una clase debe estar abierta para su extensión pero cerrada para su modificación.

Se puede extender una clase o implementar una interfaz, pero no se debería poder modificar una clase directamente. Esto significa que se debería de poder extender y usar la nueva extensión en lugar de alterar la clase directamente.

Adicionalmente, significa definir públicos o privados los atributos y métodos de una clase para que no puedan ser modificados por código externo.

Principio de Sustitución de Liskov (Liskov Substitution Principle)

Los objetos de un programa deben poder reemplazarse con instancias de subtipos sin alterar el funcionamiento del programa.

En PHP, normalmente esto significa crear interfaces para luego poder intercambiar sus implementaciones. Esto debería ser posible sin tener que cambiar la forma en que la aplicación interactúa con la implementación. La interfaz sirve como un contrato que garantiza que ciertos métodos estarán disponibles.

Principio de Segregación de la Interfaz (Interface Segregation Principle)

Muchas interfaces específicas son mejores que una interfaz de propósito general.

Generalmente, es preferible crear una interfaz e implementarla varias veces que crear una clase de propósito general que intente funcionar en todo tipo de situaciones.

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

Principio de Inversión de Dependencias (Dependency Inversion Principle)

Uno debería depender de abstracciones, no de clases concretas.

Deberíamos definir las dependencias de una clase en forma de interfaz. Esto permite intercambiar la implementación sin tener que modificar la clase que usa las dependencias.

Conceptos fundamentales

A lo largo del libro usaremos algunas de las características más poderosas de Laravel.

Antes de comenzar, es importante conocer por lo menos el contenedor de Laravel y cómo facilita el uso de la Inyección de Dependencias.

Este capítulo aborda el contenedor de Laravel, su uso de la Inversión de Control y la Inyección de Dependencias.

El contenedor

La clase `Illuminate\Foundation\Application` es la que encapsula todo Laravel. Esta clase es un *contenedor* - puede “contener” datos, objetos, clases y hasta funciones anónimas.

Uso básico

Para ver cómo funciona el contenedor, hagamos un ejercicio en el archivo de rutas.

El contenedor de Laravel implementa `ArrayAccess`, así que podemos acceder a él como un arreglo. Veamos cómo acceder a manera de arreglo asociativo.

Archivo: `app/routes.php`

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

Si nos dirigimos a la ruta `/container`, obtendremos el siguiente resultado:

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

Como podemos ver, aunque `Application` es una clase con atributos y métodos, ¡también se puede acceder como si fuera un arreglo!



Fachadas

¿Confundido con lo que hace `App::getFacadeRoot()`? La clase `App` es una fachada. Esto nos permite usarla en cualquier lugar, de manera estática. Sin embargo, en realidad no se trata de una clase estática. `getFacadeRoot` obtiene la verdadera instancia de la clase, lo cual tuvimos que hacer para usarla como un arreglo en el ejemplo anterior.

Se puede ver esta y otras fachadas en el espacio de nombres `Illuminate\Support\Facades`.

Algo más avanzado

Hagamos algo más elegante con el contenedor y asignémosle una función anónima:

Archivo: app/routes.php

```
1 Route::get('/container', function()
2 {
3     // Se obtiene la instancia de la aplicación
4     $app = App::getFacadeRoot();
5
6     $app['say_hi'] = function()
7     {
8         return "Hello, World!";
9     };
10
11    return $app['say_hi'];
12});
```

De nuevo, si nos dirigimos a la ruta /container veremos lo siguiente:

```
1 Hello, World!
```

Aunque en apariencia es algo muy simple, en realidad es algo bastante poderoso. De hecho, es la base de cómo los paquetes Illuminate interactúan unos con otros para hacer funcionar al framework Laravel.

Más adelante veremos cómo los Proveedores de Servicios vinculan elementos al contenedor, uniendo los diferentes paquetes Illuminate.

Inversión de control

La clase Container de Laravel tiene más cosas bajo la manga además de actuar como un arreglo. También funciona como un contenedor de Inversión de Control.

La Inversión de Control es una técnica que permite definir cómo una aplicación implementa una clase o interfaz. Por ejemplo, si nuestra aplicación depende de la interfaz FooInterface y queremos implementar la clase ConcreteFoo, es en el contenedor de Inversión de Control donde hay que definir la implementación.

Veamos un ejemplo básico de cómo funciona usando la ruta /container una vez más.

Primero vamos a preparar algunas clases - una interfaz y una clase que la implemente. Para mantener el ejemplo sencillo, ambas pueden estar en el archivo app/routes.php:

Archivo: app/routes.php

```
1 interface GreetableInterface {
2
3     public function greet();
4
5 }
6
7 class HelloWorld implements GreetableInterface {
8
9     public function greet()
10    {
11        return 'Hello, World!';
12    }
13 }
```

Ahora usemos estas clases junto con el contenedor y veamos qué sucede. Primero quiero presentarles el concepto de “vinculación”.

Archivo: app/routes.php

```
1 Route::get('/container', function()
2 {
3     // Se obtiene la instancia de la aplicación
4     $app = App::getFacadeRoot();
5
6     $app->bind('GreetableInterface', function()
7     {
8         return new HelloWorld;
9     });
10
11     $greeter = $app->make('GreetableInterface');
12
13     return $greeter->greet();
14 });
```

En lugar de acceder como arreglo usando `$app['GreetableInterface']` hemos usado el método `bind()`.

Esto hace uso del contenedor de Inversión de Control de Laravel para devolver la clase `HelloWorld` cada vez que se solicita `GreetableInterface`.

¡De este modo es posible “intercambiar” implementaciones! Por ejemplo, en lugar de `HelloWorld`, podríamos crear una implementación llamada `GoodbyeCruelWorld` y hacer que el contenedor la devuelva cada que se solicite `GreetableInterface`.

Esto encamina nuestras aplicaciones hacia la mantenibilidad. Al usar el contenedor, podemos (de forma ideal) intercambiar implementaciones en un solo lugar sin afectar otras áreas de la aplicación.

Uso en una situación real

¿Dónde hay que colocar estas vinculaciones? Si no queremos saturar los archivos `start.php`, `filters.php`, `routes.php`, entonces podemos hacer uso de los Proveedores de Servicios.

Los Proveedores de Servicios sirven específicamente para registrar vinculaciones en el contenedor de Laravel. De hecho, casi todos los paquetes Illuminate usan un Proveedor de servicios justo para eso.

Veamos un ejemplo de cómo los Proveedores de Servicios se usan dentro de un paquete Illuminate. Vamos a examinar el paquete de paginación.

Primero, el método `register()` del Proveedor de servicios de paginación:

`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 }
```



El método `register()` se invoca automáticamente en cada uno de los Proveedores de servicios alojados en el archivo `app/config/app.php`.

¿Entonces, qué es lo que ocurre en el método `register()`? Primero que nada, se registra la instancia del “paginador” en el contenedor. Esto hará que `$app['paginator']` y `App::make('paginator')` estén disponibles en otras áreas de la aplicación.

Luego, se define una instancia ‘paginador’ como resultado del valor devuelto por una función anónima, tal y como lo hicimos en el ejemplo ‘say_hi’.



No hay que confundir el uso de `$this->app->share()`. El método Share simplemente hace que la función anónima pueda usarse como un singleton (instancia única), similar a llamar a `$this->app->instance('paginator', new Environment)`.

La función anónima crea un objeto `Pagination\Environment`, lo configura y lo devuelve.

Tal vez ya lo hayan notado, pero ¡el Proveedor de Servicios crea otras vinculaciones en la aplicación! La clase `PaginationEnvironment` claramente accede a ciertas dependencias mediante su constructor - un objeto para peticiones `$app['request']`, un objeto para vistas `$app['view']` y un traductor `$app['translator']`. Por suerte, estas vinculaciones son creadas en otros paquetes Illuminate y se definen en distintos Proveedores de Servicios.

Podemos observar entonces cómo los diferentes paquetes Illuminate interactúan entre sí. Al encontrarse vinculados al contenedor de la aplicación, se pueden utilizar en otros paquetes (¡o en nuestro propio código!) sin que nuestro código quede atado a una clase en particular.

Inyección de dependencias

Ahora que hemos visto el funcionamiento del contenedor, veamos cómo podemos usarlo para implementar la Inyección de Dependencias en Laravel.

¿Qué es la inyección de dependencias?

La Inyección de Dependencias es el acto de agregar (inyectar) cualquier dependencia a una clase, en lugar de instanciarla en algún lugar dentro de ella. Es común definir las dependencias en el constructor como parámetros de un tipo específico.

Tomen como ejemplo el siguiente constructor:

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

Al definir el parámetro de tipo `HelloWorld`, indicamos explícitamente que se trata de una dependencia de la clase.

Esto es lo contrario a la instanciación directa:

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



Si se preguntan *para qué* se usa la Inyección de Dependencias, [esta respuesta de Stack Overflow⁴](#) es un buen lugar para empezar. Explicaré algunos otros de sus beneficios en ejemplos subsecuentes.

Enseguida, veremos un ejemplo de Inyección de Dependencias en acción, usando el contenedor de Inversión de Control de Laravel.

⁴<http://stackoverflow.com/questions/130794/what-is-dependency-injection>

Agregando dependencias a los controladores

Este es un caso de uso común en Laravel.

Normalmente, cuando un controlador requiere de ciertas clases en el constructor, es necesario agregar esas dependencias cuando se crea la clase. Sin embargo, ¿qué sucede cuando se definen dependencias en un controlador de Laravel? Tendríamos que instanciar el controlador en alguna parte:

```
1 $ctrl = new ContainerController( new HelloWorld );
```

Muy bien, pero en Laravel no es necesario instanciar el controlador directamente - el enrutador se encarga de eso.

Aún así, ¡es posible inyectar dependencias en el controlador con el uso del contenedor de Laravel!

Siguiendo con las mismas clases GreetableInterface y HelloWorld de ejemplos anteriores, imaginemos que vinculamos a un controlador la ruta /container:

Archivo: app/routes.php

```
1 interface GreetableInterface {
2
3     public function greet();
4
5 }
6
7 class HelloWorld implements GreetableInterface {
8
9     public function greet()
10    {
11        return 'Hello, World!';
12    }
13 }
14
15 Route::get('/container', 'ContainerController@container');
```

Ahora en el controlador podemos establecer HelloWorld como parámetro:

Archivo: app/controllers/ContainerController.php

```
1 <?php
2
3 class ContainerController extends BaseController {
4
5     protected $greeter;
6
7     // Dependencia: HelloWorld
8     public function __construct(HelloWorld $greeter)
9     {
10         $this->greeter = $greeter;
11     }
12
13     public function container()
14     {
15         return $this->greeter->greet();
16     }
17
18 }
```

Si nos dirigimos a la ruta /container deberíamos ver, una vez más:

```
1 Hello, World!
```

Noten que NO vinculamos nada al contenedor. “Simplemente funcionó” - ¡una instancia de tipo HelloWorld se pasó como parámetro al controlador!

Esto se debe a que el contenedor de Inversión de Control de Laravel intenta resolver automáticamente cualquier dependencia que se haya definido en el constructor de un controlador. ¡Laravel inyectará las dependencias por nosotros!

Interfaces como dependencias

No hemos terminado aún. ¡Esta es la parte hacia donde queríamos llegar!

¿Qué tal si en lugar de especificar HelloWorld como dependencia del controlador especificáramos la interfaz GreetableInterface?

Veamos cómo quedaría:

Archivo: app/controllers/ContainerController.php

```
1 <?php
2
3 class ContainerController extends BaseController {
4
5     protected $greeter;
6
7     // Dependencia: GreetableInterface
8     public function __construct(GreetableInterface $greeter)
9     {
10         $this->greeter = $greeter;
11     }
12
13     public function container()
14     {
15         echo $this->greeter->greet();
16     }
17
18 }
```

Si intentamos ejecutarlo obtendremos un error:

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

Por supuesto que GreetableInterface no se puede instanciar, se trata de una interfaz. Sin embargo, podemos ver que Laravel intenta instanciarla con tal de resolver la dependencia de la clase.

Solucionemos eso - usaremos el método bind() del contenedor para indicarle a Laravel que utilice una instancia de HelloWorld cuando el contenedor descubra que nuestro controlador depende de una instancia de GreetableInterface:

Archivo: app/routes.php

```
1 interface GreetableInterface {  
2  
3     public function greet();  
4  
5 }  
6  
7 class HelloWorld implements GreetableInterface {  
8  
9     public function greet()  
10    {  
11        return 'Hello, World!';  
12    }  
13 }  
14  
15 // iVinculamos HelloWorld para cuando se requiera  
16 // una instancia de GreetableInterface!  
17 App::bind('GreetableInterface', 'HelloWorld');  
18  
19 Route::get('/container', 'ContainerController@container');
```

Ahora ejecutemos la ruta /container para ver de nuevo el mensaje Hello, World!



Noten que no utilizamos una función anónima para vincular HelloWorld - Es posible simplemente indicar el nombre de la clase que deseamos. Una función anónima es útil cuando la implementación tiene a su vez sus propias dependencias, las cuales hay que pasar en su constructor.

¿Por qué la inyección de dependencias?

¿Por qué íbamos a querer especificar una interfaz como dependencia en lugar de una clase concreta?

Lo queremos así porque necesitamos que cualquier clase que le pasemos al constructor sea una subclase de la interfaz. De esta manera podemos usar cualquier implementación - el método que necesitamos siempre estará disponible.

Dicho de otra manera: **podemos cambiar la implementación a voluntad, sin afectar otras secciones de nuestra aplicación.**

Aquí tenemos un ejemplo. Algo que he tenido que hacer muchas veces en aplicaciones reales.



No copien y peguen el código de este ejemplo. He omitido algunos detalles de configuración para simplificarlo, tales como claves API.

Supongamos que nuestra aplicación envía correos utilizando AWS de Amazon. Para lograrlo, hemos definido una interfaz `Emailer` y una implementación, llamada `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    {  
17        $this->aws->addTo($to)  
18            ->setFrom($from)  
19            ->setSubject($subject)  
20            ->setMessage($message);  
21            ->sendEmail();  
22    }  
23 }
```

Vinculamos `Emailer` a la implementación `AwsEmailer`:

```
1 App::bind('Emailer', function()  
2 {  
3     return new AwsEmailer( new AwsSDK );  
4 });
```

Un controlador usa como dependencia la interfaz `Emailer`:

Archivo: app/controllers/EmailController.php

```
1 class EmailController extends BaseController {
2
3     protected $emailer;
4
5     // Dependencia: Emailer
6     public function __construct(Emailer $emailer)
7     {
8         $this->emailer = $emailer;
9     }
10
11    public function email()
12    {
13        $this->emailer->send(
14            'ex-to@example.com',
15            'ex-from@example.com',
16            'Peanut Butter Jelly Time!',
17            "It's that time again! And so on!"
18        );
19
20        return Redirect::to('/');
21    }
22
23 }
```

Ahora supongamos que, más adelante, nuestra aplicación crece y necesita más funcionalidad de la que provee AWS. Después de investigar y evaluar opciones decidimos usar SendGrid.

¿Cómo procedemos a modificar la aplicación para que use SendGrid? Ya que hemos usado interfaces y el contenedor de Laravel, ¡hacer el cambio es muy fácil!

Primero hay que crear una implementación de `Emailer` que haga uso de SendGrid.

```

1  class SendGridEmailer implements Emailer {
2
3      protected $sendgrid;
4
5      public function __construct(SendGridSDK $sendgrid)
6      {
7          $this->sendgrid = $sendgrid;
8      }
9
10     public function send($to, $from, $subject, $message)
11     {
12         $mail = $this->sendgrid->mail->instance();
13
14         $mail->addTo($to)
15             ->setFrom($from)
16             ->setSubject($subject)
17             ->setText( strip_tags($message) )
18             ->setHtml($message)
19             ->send();
20
21         $this->sendgrid->web->send($mail);
22     }
23 }
```

Después (y por último!), hay que indicarle a la aplicación que use SendGrid en lugar de Aws. Como hemos utilizado `bind()` en el contenedor, cambiar la implementación de `Emailer` de `AwsEmailer` a `SendGridEmailer` es tan sencillo como hacer esta *única* modificación:

```

1 // De
2 App::bind('Emailer', function()
3 {
4     return new AwsEmailer( new AwsSDK );
5 });
6
7 // A
8 App::bind('Emailer', function()
9 {
10    return new SendGridEmailer( new SendGridSDK );
11});
```

Noten que hemos hecho todo esto sin cambiar una sola línea de código en otro lugar de la aplicación. Al requerir el uso de la interfaz `Emailer` como dependencia, se garantiza que cualquier clase que se inyecte tendrá disponible el método `send()`.

Podemos observar esto en nuestro ejemplo. El controlador aún hace la llamada a `$this->emailer->send()` sin requerir ninguna modificación al hacer el cambio de implementación de `AwsEmailer` a `SendGridEmailer`.

En resumen.

La Inyección de Dependencias y la Inversión de Control son patrones utilizados una y otra vez en el desarrollo con Laravel.

Como veremos después, vamos a definir muchas interfaces para hacer nuestro código más fácil de mantener y para facilitar las pruebas. El contenedor de Inversión de Control de Laravel lo facilita.

Preparando Laravel

La aplicación de ejemplo

El libro utilizará una aplicación de ejemplo. Vamos a crear un simple blog - el proyecto de fin de semana favorito de cualquiera.

La aplicación se puede consultar en Github [fideloper\Implementing-Laravel⁵](#). Allí se puede ver el código y ejemplos funcionales del libro.

He aquí información general sobre la aplicación.

Base de datos

Tendremos una base de datos con las siguientes tablas y campos:

- **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

En el directorio `app/database/migrations` de GitHub encontrarán las migraciones para las tablas, así como algunas semillas.

Modelos

Cada una de las tablas tendrá su clase de Eloquent correspondiente, dentro del directorio `app/models`.

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

⁵<https://github.com/fideloper/implementing-laravel>

Relaciones

Los usuarios pueden crear artículos, por eso hay una relación “uno a muchos” entre usuarios y artículos; cada usuario puede escribir múltiples artículos, pero cada artículo pertenece a un solo usuario. La relación se crea tanto en el modelo User como en el modelo Article.

Al igual que con los usuarios, hay una relación “uno a muchos” entre los estados y los artículos - Cada artículo tiene asignado un estado, y un estado puede ser asignado a muchos artículos.

Por último, los artículos y las etiquetas tienen una relación. Cada artículo puede tener una o más etiquetas. Cada etiqueta puede ser asignada a uno o más artículos. Por lo tanto, hay una relación “muchos a muchos” entre artículos y etiquetas. Esta relación se define en los modelos Article y Tag y usa la tabla pivot Articles_Tags.

Comprobabilidad y mantenibilidad

Utilizaré muy seguido las frase “comprobable y mantenible”. En la mayoría de los casos se puede suponer lo siguiente:

1. El código **Comprobable** sigue los principios SOLID de manera que nos permita realizar pruebas de unidad - para probar una unidad de código (o clase) específica sin tener que incluir sus dependencias. Esto se aprecia mejor en el uso de la Inyección de Dependencias, la cual permite simular objetos directamente para abstraer las dependencias de una clase.
2. El código **Mantenible** prevee el costo a largo plazo del tiempo de desarrollo. Esto significa que las modificaciones al código deben ser fáciles de hacer después de que hayan pasado meses, incluso años. Ejemplos populares de una modificación que debe ser fácil es el cambio de un proveedor de correo a otro o el cambio de un proveedor de almacenamiento de datos. Se aprecia en el uso de interfaces y la inversión de control.

Notas de arquitectura

El libro abordará la creación de una biblioteca que contiene la mayor parte del código para el blog de ejemplo. La estructura de la biblioteca hace algunas asunciones de cómo se construye la aplicación.

Primero, notarán que no uso controladores en el código. ¡Es intencional!

Laravel es un framework para la web, diseñado para manejar peticiones HTTP y enrutarlas al código de la aplicación. La Petición, Enrutador y Controladores están diseñados para operar al nivel de HTTP.

Nuestra aplicación, sin embargo, no necesita de dichos requerimientos. Simplemente creamos la lógica, que gira entorno a las reglas del negocio.

Una manera conveniente de que una petición HTTP llegue hasta el código de nuestra aplicación es enrutarla hacia un controlador (estamos hablando de internet, después de todo). Sin embargo, la aplicación no tiene que “saber” necesariamente sobre el código de la aplicación o sobre HTTP.

Esto es una extensión del concepto de “separación de intereses”. Mientras que no es probable que usemos nuestras aplicaciones fuera del contexto de internet, es útil pensar en el framework como un detalle de implementación de la aplicación y no como la base de ella.

Piénselo de esta manera: nuestras aplicaciones no son una implementación del framework Laravel. En su lugar, Laravel es una implementación de nuestras aplicaciones.

Llevado al extremo, una aplicación podría ser implementada en cualquier framework o en cualquier código capaz de implementar las interfaces que definamos.

Los extremos, sin embargo, no son prácticos, así que usemos Laravel para cumplir con la mayoría de los objetivos de nuestra aplicación.

Esto también le da forma a la estructura de la biblioteca construída en el libro. Crearé una serie de directorios que reflejen funciones de la aplicación, tales como repositorio de datos y servicios de caché. Estas áreas funcionales tienden a ser interfaces, las cuales vamos a implementar usando varios paquetes Illuminate.

Instalación

Comencemos por el principio.

Este capítulo abordará la instalación de Laravel. Si estás leyendo este libro, ¡es probable que ya sepas cómo hacerlo! Sin embargo, siento que vale la pena incluirlo, ya que cubriré algunos pasos extra que ayudan a evitar algunas posibles dificultades.

Instalación de Composer

Composer es necesario para manejar las dependencias de Laravel. Yo normalmente instalo [Composer](#)⁶ globalmente con el fin de poder ejecutarlo desde cualquier ubicación.

Instalación de Composer por línea de comandos

```
1 $ curl -sS https://getcomposer.org/installer | php
2 // Supone que la ruta /usr/local/bin se encuentra en el PATH
3 $ sudo mv composer.phar /usr/local/bin/composer
```

Ahora se puede ejecutar Composer desde donde sea.

Creación de un nuevo proyecto

Una vez instalado, podemos usar Composer para crear un nuevo proyecto de Laravel. Nombraremos a nuestro proyecto “Implementando Laravel”. Ejecuten este comando:

Creando un nuevo proyecto en Laravel con Composer

```
1 $ composer create-project laravel/laravel "Implementando Laravel"
```



Esto clonará el proyecto laravel/laravel desde GitHub e instalará sus dependencias, como si hubiéramos clonado el repositorio de git y ejecutado `$ composer install` manualmente. Sin embargo, no inicializará un repositorio de Git - tendremos que hacerlo nosotros mismos con `$ git init`.

⁶<http://getcomposer.org/>

Configuración

Luego prepararemos y configuraremos la aplicación.

Permisos

Periódicamente, dependiendo del entorno de desarrollo, quizá nos encontremos con la necesidad de establecer los permisos del directorio `app/storage`. Aquí es donde Laravel coloca registros, sesiones, caché y otro tipo de optimizaciones. PHP necesita poder escribir en este directorio.

Otorgando permisos de escritura al directorio storage

```
1 $ chmod -R 0777 app/storage
```

En un ambiente de producción, el servidor web frecuentemente se ejecuta con el usuario (y grupo) `www-data`. Ya que no necesariamente queremos que en producción nuestros archivos tengan permisos de escritura, en su lugar haremos que específicamente `www-data` pueda escribir en ellos.

permisos del directorio storage para producción

```
1 $ chgrp -R www-data app/storage      # cambiar el grupo a www-data
2 $ chmod -R g+w app/storage           # hacer que el grupo pueda escribir
```

Entornos

Los entornos pueden y deberían ser configurados de manera diferente para cada servidor - usualmente deberíamos preparar por lo menos un servidor local de desarrollo, un servidor de pruebas y un servidor de producción.

Vamos a crear un entorno “local”. Comencemos por crear un directorio llamado `app/config/local`.

Por ahora, todo lo que necesitamos hacer es agregar los datos de conexión de la base de datos. Creemos un archivo llamado `app/config/local/database.php`.

Aquí tenemos un ejemplo de cómo debería lucir. Noten que no copié por completo el archivo `app/config/database.php`, solamente hay que definir los valores que necesitamos sobreescribir para nuestro entorno local.

Archivo: app/config/local/database.php

```
1 <?php
2
3 return array(
4
5     'connections' => array(
6
7         'mysql' => array(
8             'driver'      => 'mysql',
9             'host'        => 'localhost',
10            'database'   => 'implementinglaravel',
11            'username'   => 'root',
12            'password'   => 'root',
13            'charset'    => 'utf8',
14            'collation'  => 'utf8_unicode_ci',
15            'prefix'     => '',
16        ),
17
18    )
19 );
```

El siguiente paso es asegurarnos que Laravel elija el entorno “local” cuando lo ejecutemos en nuestro servidor de desarrollo. Por defecto, el entorno se determina mediante la URL que se usa para acceder a la aplicación. Si la URL es “localhost”, podemos establecer “localhost” como el “nombre del equipo”

Archivo: bootstrap/start.php

```
1 // Asignación del entorno "local" a "localhost":
2 $env = $app->detectEnvironment(array(
3
4     'local' => array('localhost'),
5
6 ));
7
8 // O quizá cualquier URL que termine en .dev:
9 $env = $app->detectEnvironment(array(
10
11     'local' => array('*.*.dev'),
```

```
13  ));
14
15 // O una combinación de ambos:
16 $env = $app->detectEnvironment(array(
17
18     'local' => array('localhost', '*.dev'),
19
20 ));
```

Variables de entorno

Es posible que en lugar de una URL, deseemos usar una variable de entorno.

Una variable de entorno es una variable establecida en la configuración del *servidor*, en lugar de en el código PHP.

Yo prefiero este método ya que permite cambiar la URL del entorno de desarrollo sin que este se vea afectado. Esto permite que un equipo de desarrolladores pueda usar la misma URL para acceder a la aplicación, sin tener que preocuparse por alguna colisión. Además permite establecer el entorno a través de cualquier herramienta de configuración y provisionamiento automatizado como Chef o Puppet.

En Apache, podemos agregar una variable de entorno en el archivo .htaccess o en la configuración del host virtual:

```
1 SetEnv LARA_ENV local
```

Si usamos Nginx y PHP-FPM, podemos enviar la variable de entorno como parámetro de fastcgi:

```
1 location ~ \.php$ {
2     # Otras opciones . . .
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; # ¡Nuestro entorno!
7     include fastcgi_params;
8 }
```

Después de establecer la variable de entorno, necesitamos indicarle a Laravel cómo detectarla:

Archivo: bootstrap/start.php

```
1 $env = $app->detectEnvironment(function()
2 {
3     // Utiliza "development" si no encuentra el entorno
4     return getenv('LARA_ENV') ?: 'development';
5 });
```

En resumen

Después de preparar el entorno, ¡deberíamos estar listos para comenzar! Ahora sabemos cómo:

- Instalar Composer de forma global
- Establecer permisos de archivo necesarios
- Establecer la configuración específica del entorno
- Establecer el entorno con variables



Entorno por defecto

Mucha gente configura sus datos directamente en el archivo `app/config/database.php`. Esto no se recomienda.

Aunque configurar un entorno independiente del servidor de producción puede parecer trabajo extra, ¡hay que considerar lo que pasaría si la detección del entorno falla y una migración ó alimentación de datos se ejecuta por accidente en producción! La seguridad adicional bien lo vale.

¡Quizá también sea bueno omitir del todo el entorno “producción” en el repositorio de código! Si usamos git, es posible agregar `app/config/production` al archivo `.gitignore`. Será necesario agregar manualmente la configuración al preparar el servidor de producción, pero así evitamos exponer información en el repositorio.

Preparación de la aplicación

Si alguna vez se han preguntado “¿Dónde pongo mi código?”, este capítulo responderá esa pregunta.

Siempre creen una biblioteca para la aplicación.

Una biblioteca específica para la aplicación funciona bien para el código que no es lo suficientemente genérico para justificar ser un paquete; código que no hace uso directo de las clases de Laravel, tales como los controladores. Un ejemplo es la lógica del negocio o la integración de una librería externa.

Básicamente, cualquier cosa que deseemos excluir de los controladores (*¡la mayor parte del código!*) pertenece a la biblioteca de la aplicación.

Preparando la biblioteca de la aplicación

Para hacerlo, comenzaremos por crear un espacio de nombres. Para nuestra aplicación de ejemplo usaremos el nombre “`Impl`”, que es “`Implementing Laravel`” pero acortado y fácil de escribir.

Aquí se muestra cómo luce la estructura de directorios:

```
1 Implementando Laravel
2   | - app
3   | --- commands
4   | --- config
5   | --- [ y demás directorios ]
6   | --- Impl
7   |     | ----- Exception
8   |     | ----- Repo
9   |     | ----- Service
```

Como puede que ya sepan, el espacio de nombres, el nombre de archivo y la estructura de directorios es importante - nos permite autocargar los archivos de PHP basados en el [estándar de autocarga PSR-0](#)⁷.

Por ejemplo, una clase en esta estructura podría lucir así:

⁷<http://www.php-fig.org/psr/0/>

Archivo: app/Impl/Repo/EloquentArticle.php

```
1 <?php namespace Impl\Repo;
2
3 class EloquentArticle {
4
5     public function all() { . . . }
6
7 }
```

Siguiendo PSR-0, este archivo se encontraría aquí:

```
1 Implementing Laravel
2 | - app
3 | --- Impl
4 | -----Repo
5 | -----EloquentArticle.php
```

Autocarga

Ahora tenemos un lugar para la biblioteca `Impl`. Digámosle a Composer que autocargue las clases con especificación PSR-0. Para conseguirlo, editemos `composer.json` y agreguemos lo siguiente a la sección “autoload”:

File: composer.json

```
1 {
2     "require": {
3         "laravel/framework": "4.0.*"
4     },
5     "autoload": {
6         "classmap": [
7             .
8             .
9             ],
10        "psr-0": {
11            "Impl": "app"
12        }
13    },
14    "minimum-stability": "dev"
15 }
```

Después de agregar `Impl` a la sección PSR-0 de “autoload”, necesitamos decirle a Composer que la detecte:

```
1 $ composer dump-autoload
```



El uso de `dump-autoload`⁸ es una forma de indicarle a Composer que busque nuevos archivos en el mapa de clases (Controladores de Laravel, Modelos, etc). Para la autocarga PSR-0, también puede reconstruir el autocargador optimizado, lo que ahorra tiempo cuando se ejecuta la aplicación.

¡Ahora podemos instanciar la clase `Impl\Repo\EloquentArticle` en cualquier lugar de nuestro código y PHP se encargará de autocargarla!

Archivo: `app/routes.php`

```
1 Route::get('/', function()
2 {
3     $articleRepo = new Impl\Repo\EloquentArticle;
4
5     return View::make('home')->with('articles', $articleRepo->all());
6 }
```

En resumen

Hemos visto cómo crear una biblioteca que albergue el código de nuestra aplicación. Esta biblioteca es donde agregaremos la lógica de negocio, extensiones, las vinculaciones al contenedor, entre otras cosas.



Ubicación, Ubicación, Ubicación

La biblioteca puede ir en cualquier lugar. Por conveniencia, yo normalmente la agrego en el directorio `app`, pero no está limitada a esa ubicación. Tal vez quieran usar otra ubicación ¡o incluso consideren convertirla en un paquete que pueda ser agregado como una dependencia de Composer!

⁸<http://getcomposer.org/doc/03-cli.md#dump-autoload>

Patrones útiles

Llegando al meollo del asunto, este capítulo abordará algunos patrones útiles de arquitectura. Vamos a explorar cómo podemos emplear el contenedor, las interfaces y la inyección de dependencias para incrementar la mantenibilidad de nuestro código.

El patrón repositorio

¿Qué es?

El patrón repositorio es una forma de abstraer la lógica del negocio para separarla de la fuente de datos. Es una capa externa extra encima de la obtención de datos, la cuál se puede usar de diferentes maneras.

¿Para qué se usa?

El objetivo de un repositorio es incrementar la mantenibilidad y dar forma al código entorno a los casos de uso de la aplicación.

Muchos desarrolladores creen que es una herramienta que se usa únicamente en aplicaciones de gran escala, sin embargo me he visto usándola para la mayoría de mis aplicaciones. Sus ventajas compensan las desventajas del código extra que hay que escribir.

Veamos algunos de sus beneficios:

Inversión de dependencias

Esta es una expresión de los principios SOLID. El patrón repositorio permite crear distintas implementaciones de una interfaz, cuyo propósito es el manejo de datos.

El caso de uso más común es el “intercambio de fuente de datos”. Se trata de la capacidad de cambiar el almacenamiento de datos, tal como MYSQL, por alguno otro como una base de datos NoSQL, sin afectar el código de la aplicación.

Esto se logra al crear una interfaz que se encargue de obtener la información. Luego, se puede crear una o varias implementaciones de dicha interfaz.

Por ejemplo, para la lógica que gira entorno al repositorio, vamos a crear una implementación utilizando Eloquent. Si alguna vez necesitamos cambiar la fuente de datos a MongoDB, podemos crear la implementación correspondiente. Ya que nuestra aplicación espera una interfaz en lugar de una clase concreta, no distingue la diferencia entre una implementación y otra.

Esto se vuelve *muy* poderoso si usamos el repositorio de datos en distintos lugares de nuestra aplicación (es probable que sí). Por lo regular interactuamos con los datos en casi cualquier llamada a la aplicación, ya sea directamente en un controlador, desde un comando, en una cola de procesos o en el código que procesa un formulario.

Si tenemos la certeza de que cada área de nuestra aplicación siempre cuenta con los métodos que necesita, vamos por buen camino hacia un futuro más sencillo.

Pero en realidad, ¿qué tan seguido cambiamos la fuente de datos? Es probable que rara vez cambiemos la fuente datos basada en SQL. Sin embargo, hay otras razones para utilizar el patrón de repositorio.

Planeación

He mencionado antes en el libro que mi punto de vista se basa en escribir el código entorno a la lógica de negocio. Crear una interfaz es útil para planear el código entorno a las necesidades del negocio (casos de uso).

Al definir los métodos de cada implementación estamos planeando los casos de uso para nuestro dominio. ¿Los usuarios van a crear artículos o solamente van a leerlos? ¿Qué tan diferente es la interacción de un administrador comparada con los usuarios normales?

Definir interfaces nos brinda una imagen más clara de cómo se utilizará la aplicación desde una perspectiva de dominio, en lugar de una perspectiva de datos.

Orientación a la lógica de negocio

Construir la aplicación entorno al dominio del negocio en realidad es expresar con código el dominio del negocio. Recuerden que cada modelo de Eloquent representa una tabla en la base de datos. La lógica del negocio no.

Por ejemplo, un artículo en nuestra aplicación es más que un registro en la tabla `articles`. También abarca a un usuario en la tabla `users`, un estado en la tabla `statuses` y una serie de etiquetas, representadas en las tablas `tags` y `articles_tags`. Un artículo es una entidad de negocio compuesta; es una entidad de negocio que contiene otras entidades en lugar de simplemente contener atributos como título, contenido y fecha de publicación.

El patrón repositorio nos permite presentar un artículo como más que un simple registro de una tabla. Podemos combinar y mezclar nuestros modelos de Eloquent, relaciones y el constructor de consultas de la forma que sea necesaria con tal de convertir los datos puros en verdaderas representaciones de nuestras entidades de negocio.

Lógica de la capa de datos

El repositorio de datos se vuelve un lugar muy conveniente para agregar lógica entorno a la obtención de datos. En lugar de agregar lógica extra a los modelos de Eloquent, podemos usar nuestro repositorio para alojarla.

Por ejemplo, quizás necesitemos guardar en caché nuestros datos. El repositorio de datos es un gran lugar para agregar la capa de caché. El siguiente capítulo mostrará cómo hacerlo de una forma mantenable.

Datos remotos

Es importante recordar que los datos pueden provenir de muchas fuentes - no necesariamente una base de datos.

Muchas aplicaciones web modernas son híbridas - Consumen múltiples APIs y devuelven datos útiles para el usuario final. Un repositorio de datos puede ser un modo de albergar la lógica que obtiene información de una API y combinarla en una entidad que la aplicación puede consumir o procesar fácilmente.

También es de utilidad en la Arquitectura Orientada a Servicios (Service Oriented Architecture), en la que tal vez necesitemos hacer llamadas API a servicios que se encuentren dentro de nuestra propia infraestructura pero que no cuentan con acceso a la base de datos.

Ejemplo

Veamos un ejemplo práctico de cómo luciría:

En este ejemplo, comenzaron con la parte central de cualquier blog: los artículos.

La Situación

Cómo he mencionado, las porciones relevantes de nuestra base de datos luce de esta manera:

- **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

Tenemos una table `Articles`, en la que cada registro representa un artículo. Tenemos una tabla `Tags` donde cada registro representa una etiqueta. Finalmente, tenemos una tabla `Articles_Tags` que usamos para asignar etiquetas a los artículos. En Laravel, esto se conoce como “tabla pivote”, y es necesaria para representar una relación “muchos a muchos”.

Como se mencionó antes, las tablas `Articles` y `Tags` tienen sus modelos correspondientes en `app/models`, donde se definen sus relaciones y se hace uso de la tabla pivote.

```
1 app
2 |-models
3 |--- Article.php
4 |--- Tag.php
```

La forma más simple de obtener artículos, y aun así la menos mantenible, es el uso de modelos Eloquent directamente en un controlador. Por ejemplo, veamos la lógica para la página inicial de nuestro blog, la cuál va a desplegar nuestros 10 artículos más recientes, paginados.

app/controllers/ContentController.php

```
1 <?php
2
3 ContentController extends BaseController {
4
5     // Ruta a la página inicial
6     public function home()
7     {
8         // Obtener 10 artículos más recientes con paginación
9         $articles = Articles::with('tags')
10            ->orderBy('created_at', 'desc')
11            ->paginate(10);
12
13         return View::make('home')
14             ->with('articles', $articles);
15     }
16
17 }
```

Bastante simple, pero puede mejorarse. Algunos de los problemas de esta solución son:

- **No se puede cambiar la fuente de datos** - Con Eloquent, podemos cambiar la fuente de datos entre distintos tipos de SQL. Sin embargo, el patrón de repositorio nos permitirá cambiar a cualquier almacenamiento de datos - arreglos, base de datos NoSQL, caché (que veremos más adelante) - sin cambiar código en ninguna otra parte de nuestra aplicación.
- **No se puede probar** - No podemos probar el código sin tener que consultar la base de datos. El Patrón Repositorio nos permitirá probar nuestro código sin necesidad de eso.
- **Lógica de negocio pobre** - Tenemos que colocar la lógica de negocio entorno a nuestros datos y modelos del controlador, reduciendo así la reusabilidad.

Resumiendo, hacemos un desorden en nuestros controladores y terminamos repitiendo código. Vamos a reestructurarlo para mejorar la situación.

Reestructuración

Vamos a hacer bastantes cosas:

1. Alejarnos del uso directo de los modelos
2. Hacer uso de interfaces
3. Implementar la Inyección de Dependencias en nuestros controladores
4. Usar el contenedor de Inversión de Control de Laravel para cargar las clases correctas en nuestros controladores

El directorio de modelos

Lo primero que haremos es alejarnos del uso directo de los modelos y vamos a usar nuestro directorio autocargado y con nombres de espacio, `Impl`.

Esta es la estructura de directorios que usaremos:

```
1 app
2 | - Impl
3 | --- Repo
4 | ----- Article
5 | ----- Tag
6 | - models
7 | --- Article.php
8 | --- Tag.php
```

Interfaces

Crearemos interfaces muy seguido en nuestra aplicación. Las interfaces son contratos - establecen el uso de los métodos definidos en sus implementaciones. Esto nos permite utilizar cualquier repositorio que implemente la interfaz sin temor a que cambien sus métodos.

Además, hacen que nos preguntemos cómo interactuará una clase con las demás partes de la aplicación.

Vamos a crear la nuestra primero:

Archivo: `app/Impl/Repo/Article/ArticleInterface.php`

```
1 <?php namespace Impl\Repo\Article;
2
3 interface ArticleInterface {
4
5     /**
6      * Obtener artículos paginados
7      *
8      * @param int Página Actual
9      * @param int Número de artículos por página
10     * @return object Objeto con $items y $totalItems para paginación
11     */
12    public function byPage($page=1, $limit=10);
13
14    /**
15     * Obtener un artículo a través de su URL
```

```
16     *
17     * @param string URL del artículo
18     * @return object Objeto con información del artículo
19     */
20    public function bySlug($slug);
21
22    /**
23     * Obtener artículos por etiqueta
24     *
25     * @param string URL de la etiqueta
26     * @param int Página Actual
27     * @param int Número de artículos por página
28     * @return object Objeto con $items y $totalItems para paginación
29     */
30    public function byTag($tag, $page=1, $limit=10);
31
32 }
```

Luego crearemos un repositorio de artículos para implementar esta interfaz. Pero primero, tenemos que tomar una decisión.

El cómo implementar nuestra interfaz depende de nuestra fuente de datos. Si usamos algún tipo de SQL, es posible que Eloquent lo soporte. Sin embargo, si vamos a consumir una API o utilizar una base de datos NoSQL, tal vez necesitemos crear una implementación para estos casos.

Ya que estoy usando MySQL, voy a aprovechar Eloquent, el cual hará más fácil el uso de relaciones y la manipulación de datos.

Archivo: 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     // Dependencia de clase: mdoelo de Eloquent e
13     // implementación de TagInterface
```

```
14     public function __construct(Model $article, TagInterface $tag)
15     {
16         $this->article = $article;
17         $this->tag = $tag;
18     }
19
20 /**
21 * Obtener artículos paginados
22 *
23 * @param int Página Actual
24 * @param int Número de artículos por página
25 * @return Objeto StdClass con $items y $totalItems para paginación
26 */
27 public function byPage($page=1, $limit=10)
28 {
29     $articles = $this->article->with('tags')
30             ->where('status_id', 1)
31             ->orderBy('created_at', 'desc')
32             ->skip( $limit * ($page-1) )
33             ->take($limit)
34             ->get();
35
36     // Crear objeto para devolver datos útiles
37     // para paginación
38     $data = new \StdClass();
39     $data->items = $articles->all();
40     $data->totalItems = $this->totalArticles();
41
42     return $data;
43 }
44
45 /**
46 * Obtener un artículo a través de su URL
47 *
48 * @param string URL del artículo
49 * @return object objeto con información del artículo
50 */
51 public function bySlug($slug)
52 {
53     // incluir etiquetas mediante relaciones de Eloquent
54     return $this->article->with('tags')
55             ->where('status_id', 1)
```

```
56             ->where('slug', $slug)
57             ->first();
58     }
59
60 /**
61 * Obtener artículos por etiqueta
62 *
63 * @param string URL de la etiqueta
64 * @param int Página Actual
65 * @param int Número de artículos por página
66 * @return StdClass Objeto con $items y $totalItems para paginación
67 */
68 public function byTag($tag, $page=1, $limit=10)
69 {
70     $foundTag = $this->tag->bySlug($tag);
71
72     if( !$foundTag )
73     {
74         // StdClass vacía para cumplir con el valor esperado por @return
75         // si no se encontraron etiquetas
76         $data = new \StdClass();
77         $data->items = array();
78         $data->totalItems = 0;
79
80         return $data;
81     }
82
83     $articles = $this->tag->articles()
84             ->where('articles.status_id', 1)
85             ->orderBy('articles.created_at', 'desc')
86             ->skip( $limit * ($page-1) )
87             ->take($limit)
88             ->get();
89
90     // Crear objeto para devolver datos útiles
91     // para paginación
92     $data = new \StdClass();
93     $data->items = $articles->all();
94     $data->totalItems = $this->totalByTag();
95
96     return $data;
97 }
```

```
98
99     /**
100    * Obtener número total de artículos
101   *
102   * @return int  Artículos totales
103  */
104 protected function totalArticles()
105 {
106     return $this->article->where('status_id', 1)->count();
107 }
108
109 /**
110 * Obtener número total de artículos por etiqueta
111 *
112 * @param string $tag  slug de etiqueta
113 * @return int      total de artículos por etiqueta
114 */
115 protected function totalByTag($tag)
116 {
117     return $this->tag->bySlug($tag)
118         ->articles()
119         ->where('status_id', 1)
120         ->count();
121 }
122
123 }
```

Este es nuestra estructura, con los archivos ArticleInterface y EloquentArticle:

```
1 app
2 |- Impl
3 |--- Repo
4 |----- Article
5 |----- ArticleInterface.php
6 |----- EloquentArticle.php
7 |----- Tag
8 |--- models
9 |----- Article.php
10 |----- Tag.php
```

Con nuestra nueva implementación lista, podemos volver a nuestro controlador:

```
app/controllers/ContentController.php
1 <?php
2
3 use Impl\Repo\Article\ArticleInterface;
4
5 class ContentController extends BaseController {
6
7     protected $article;
8
9     // Dependencia de clase: Subclase de ArticleInterface
10    public function __construct(ArticleInterface $article)
11    {
12        $this->article = $article;
13    }
14
15    // Ruta a la página inicial
16    public function home()
17    {
18        $page = Input::get('page', 1);
19        $perPage = 10;
20
21        // Obtener 10 artículos más recientes con paginación
22        // Aún así se obtiene una colección de artículos en forma de arreglo
23        $pagiData = $this->article->byPage($page, $perPage);
24
25        // La paginación se hace aquí, no es responsabilidad
26        // del repositorio. Véase la sección capa de caché.
27        $articles = Paginator::make(
28            $pagiData->items,
29            $pagiData->totalItems,
30            $perPage
31        );
32
33        return View::make('home')->with('articles', $articles);
34    }
35
36 }
```

Un momento, ¿qué sucedió?

Quizá tengan algunas preguntas de lo que se hizo en esta implementación.

Primero, no devolvemos un objeto `Pagination` a través del método `paginate()` del constructor de consultas. Esto ha sido intencional. Nuestro repositorio simplemente sirve para devolver un conjunto de artículos y no debería tener conocimiento de la clase de paginación ni de los enlaces HTML que genera.

En su lugar, paginamos usando `skip()` y `take()` los cuales hacen uso directo de las cláusulas `LIMIT` y `OFFSET` de MySQL.

Esto significa que delegamos a nuestro controlador la creación de una instancia de paginación. ¡Sí, hemos agregado más código al controlador!

La razón por la cuál decidí no incorporar la clase paginadora en el repositorio es porque utiliza datos de entrada HTTP para obtener el número de página actual y generar enlaces HTML. Esto agrega funcionalidad implícita a nuestro repositorio de datos en forma de dependencias, donde no corresponden. Determinar el número de página actual y generar la presentación (HTML) no es responsabilidad de un repositorio de datos.

Al mantener la paginación fuera del repositorio, hacemos que el código sea más fácil de mantener. Esto resultaría más claro si hubieramos usado una implementación que no fuera un modelo de Eloquent. En ese caso, no devolvería una instancia de la clase de paginación. Nuestra vista buscaría el método `links()` del paginador ¡sólo para darse cuenta que no existe!

Uniendo las piezas

Nos queda un paso más antes de que nuestro código funcione:

Cómo han notado, preparamos algunas dependencias en nuestros controladores y repositorios. La clase `EloquentArticle` espera un `Eloquent\Model` y la clase `ContentController` espera una implementación de `ArticleInterface` cuando se instancia.

Lo último que debemos hacer es utilizar el contenedor de Inversión de Control de Laravel y los Proveedores de Servicios para mandar estas dependencias hacia las clases que las necesitan.

Para llevarlo a cabo, vamos a crear un Proveedor de Servicios para indicarle a la aplicación cuáles son las clases que debe instanciar cuando se necesiten.

Archivo: `app/Impl/Repo/RepoServiceProvider.php`

```
1 <?php namespace Impl\Repo;
2
3 use Article; // artículo de Eloquent
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 * Registro de vinculación
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 }
```

Ahora, cada que nuestro controlador solicite una instancia de `ArticleInterface`, el contenedor de Laravel sabrá que tiene que ejecutar la función anónima definida anteriormente, la cual devuelve una instancia de `EloquentArticle` (junto con su dependencia, una instancia del modelo `Article`).

¡Agreguemos este proveedor de servicios en `app/config/app.php` y tendremos todo listo!



Yendo Más Lejos

Quizá hayan notado que he mencionado, pero no creado, un repositorio Tag. Esto queda como ejercicio para el lector, que se incluye en el código de la aplicación de ejemplo.

Se tendrá que definir una interfaz y crear una implementación de Eloquent. Entonces el código funcionará con la dependencia `TagInterface`, la cual se registra en `RepoServiceProvider`.

Si se preguntan si es correcto requerir de un repositorio de etiquetas dentro de un repositorio de artículos, la respuesta indudablemente es “sí”. Hemos creado interfaces para garantizar que los métodos apropiados siempre estén disponibles.

Además, los repositorios existen para seguir la lógica de negocio, no el esquema de la base de datos. Las entidades de negocio por lo regular tienen relaciones complejas entre ellas. El uso de múltiples modelos de Eloquent y otros repositorios es absolutamente necesario para crear y modificar las entidades de lógica de negocio.

¿Qué hemos obtenido?

Pues bien, hemos obtenido más código, ¡pero tenemos muy buenas razones para ello!

Fuentes de datos

Nos encontramos en una posición en la que podemos cambiar nuestra fuente de datos. Si algún día necesitamos cambiar de MySQL a otro servidor basado en SQL podemos seguir usando `EloquentArticle` y simplemente cambiar la conexión a la base de datos en `app/config/database.php`. Esto es algo que Eloquent y muchos ORM nos facilitan.

Sin embargo, si necesitamos cambiar a una base de datos NoSQL o incluso agregar otra fuente de datos además de Eloquent (una API, tal vez), podemos crear una nueva implementación sin tener que cambiar el código a lo largo de la aplicación.

Como ejemplo, si quisiéramos cambiar a MongoDB, crearíamos una implementación llamada `MongoDbArticle` y cambiaríamos la vinculación de clase en `RepoServiceProvider` - similar al cambio de proveedores de correo en el capítulo del Contenedor.

Pruebas

Hemos usado inyección de dependencias en dos lugares: en nuestro controlador y en `EloquentArticle`. Ahora podemos probar estas implementaciones simulando una instancia de `ArticleInterface` en nuestro controlador y simulando una instancia de `Eloquent/Model` en nuestro repositorio, sin tener que consultar la base de datos real.

Lógica de negocio

¡Podemos expresar la verdadera lógica de negocio entre nuestras entidades al incluir otros repositorios en nuestro repositorio de artículos! Por ejemplo, un artículo contiene etiquetas, así que tiene sentido que nuestro repositorio de artículos pueda usar etiquetas como parte de su lógica.



Interfaces

Quizá se pregunten si realmente necesitamos usar interfaces para todos nuestros repositorios; el uso de interfaces agrega complejidad. Cualquier modificación al repositorio, como añadir nuevos métodos públicos o cambiar parámetros en los métodos, debe verse reflejado en la interfaz. Así que es posible que se necesite editar varios archivos por algún cambio menor.

Esta es una decisión que se debe considerar si nos damos cuenta que no requerimos una interfaz. Los proyectos pequeños son candidatos para omitir el uso de interfaces. Los proyectos más grandes son los que se verán más beneficiados.

En cualquier caso, los repositorios de datos tienen muchos beneficios.

Uso de caché con el patrón repositorio

¿Qué es?

El caché es un lugar para colocar datos que pueden ser obtenidos después más rápidamente. Un típico caso de uso sería guardar en caché el resultado de una consulta a la base de datos y almacenarla en memoria (RAM). Esto nos permite obtener el resultado de la consulta mucho más rápido la próxima vez que lo necesitemos - nos ahorramos una consulta a la base de datos y el tiempo que lleva procesarla. Ya que los datos están almacenados, es extremadamente rápido.

Mientras que una base de datos es una almacén permanente de información, el caché es un almacén *temporal*. Por cómo está diseñado el caché, no se puede contar con que los datos siempre estén presentes.

¿Para qué se usa?

El caché normalmente se agrega para reducir el número de veces que la aplicación accede a una base de datos u otro servicio. Si tenemos una aplicación con grandes cantidades de información o consultas y procesamiento complejos, el caché puede resultar indispensable para que la aplicación se mantenga rápida y responsive.

Ejemplo

Ahora que hemos visto el patrón de repositorio en acción, imaginemos que le agregamos una capa de caché.

Similar al Patrón Repositorio, podemos usar Inyección de Dependencias y aprovechar el contenedor de Laravel para crear una capa de caché mantenible.

Nos basaremos en la sección del Patrón Repositorio y usaremos ArticleRepository.

La situación

Tenemos un repositorio Article que separa la obtención de datos de la lógica de negocio. Podemos continuar usando este patrón y agregar una capa de servicio - el caché.

El caché verifica si ya ha almacenado antes los datos. Si es así, los devuelve al código que los solicita. Si no existen o ya han expirado, se obtienen de la fuente de datos (por lo regular una base de datos) y se almacenan en caché para que se utilicen en la siguiente petición. Por último, se devuelven los datos al código que los requiere.



Algo común en Laravel es el caché de resultados paginados. Las funciones anónimas no se pueden serializar sin tener que usar algún truco. Por fortuna eso no es problema ¡ya que no hemos utilizado la clase de paginación en nuestro repositorio!

Veamos cómo agregar el caché limpiamente.

La estructura

Como es usual, comenzamos por crear una interfaz. Ésta, una vez más, sirve como un contrato - nuestro código espera que las clases que recibe implementen estas interfaces, por lo que sabe que ciertos métodos siempre estarán disponibles.

Esta es la estructura de directorios que usaremos:

```
1 app
2 | - Impl
3 | --- Service
4 | ----- Cache
5 | ----- CacheInterface.php
6 | ----- LaravelCache.php
7 | --- Repo
```

Ahora crearemos la interfaz.

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

```
1 <?php namespace Impl\Service\Cache;
2
3 interface CacheInterface {
4
5     /**
6      * Obtener datos desde caché
7      *
8      * @param string           Índice de elemento en caché
9      * @return mixed           Resultado de datos en caché
10 */
11
```

```
11     public function get($key);  
12  
13     /**  
14      * Agregar datos al caché  
15      *  
16      * @param string    Índice de elemento en caché  
17      * @param mixed     Los datos a almacenar  
18      * @param integer   Número de minutos que se almacenarán los datos  
19      * @return mixed    Variable $value que se devuelve por conveniencia  
20      */  
21     public function put($key, $value, $minutes=null);  
22  
23     /**  
24      * Agregar datos al caché  
25      * tomando en cuenta la paginación  
26      *  
27      * @param integer   Página de elementos en caché  
28      * @param integer   Número de resultados por página  
29      * @param integer   Número total de posibles elementos  
30      * @param mixed     Elementos de la página actual  
31      * @param string    Índice de elemento en caché  
32      * @param integer   Número de minutos que se almacenarán los datos  
33      * @return mixed    Variable $items que se devuelve por conveniencia  
34      */  
35     public function putPaginated(  
36         $currentPage,  
37         $perPage,  
38         $totalItems,  
39         $items,  
40         $key,  
41         $minutes=null  
42     );  
43  
44     /**  
45      * Verificar si un elemento existe en caché  
46      * Devuelve verdadero sólo si existe y no ha expirado  
47      *  
48      * @param string    Índice de elemento en caché  
49      * @return bool     Indica si existe el elemento  
50      */  
51     public function has($key);  
52
```

53 }

Ya tenemos nuestra interfaz. Esto refleja en gran parte el mecanismo de caché, excepto que agregamos un método `putPaginated()` para auxiliarnos al guardar datos paginados.

Enseguida creamos una implementación de esta interfaz. Ya que vamos a usar el paquete de Caché de Laravel, crearemos una implementación llamada “Laravel”.



No vamos a crear una implementación específica para Memcached, archivos ni otro tipo de almacenamiento ya que Laravel nos permite cambiar el manejador de caché a voluntad. Tal vez se pregunten por qué agregamos *otra* capa de abstracción encima de Laravel. ¡Esto se debe a qué la intención es separar de la aplicación cualquier implementación específica! Esto va encaminado a la mantenibilidad (la capacidad de intercambiar implementaciones sin afectar otras partes de la aplicación) y a las pruebas (la capacidad de probar usando objetos simulados).

```
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, $minutes);
31     }
32
33     public function putPaginated($currentPage, $perPage, $totalItems,
34                                   $items, $key, $minutes=null)
35     {
36         $cached = new \StdClass;
37
38         $cached->currentPage = $currentPage;
39         $cached->items = $items;
40         $cached->totalItems = $totalItems;
41         $cached->perPage = $perPage;
42
43         $this->put($key, $cached, $minutes);
44
45         return $cached;
46     }
47
48     public function has($key)
49     {
50         return $this->cache->section($this->cachekey)->has($key);
51     }
52
53 }
```

Repasemos lo que sucede aquí. Esta clase tiene algunas dependencias:

1. Una instancia del caché de Laravel
2. Un índice de caché
3. El número de minutos por defecto para almacenar en caché

Le enviamos a nuestro código una instancia del caché de Laravel en el constructor (Inyección de Dependencias) para que sea posible hacer pruebas de unidad con la clase - podemos simular la dependencia \$cache

Usamos un índice para caché con tal de que la clase tenga un índice único. Esto nos permite obtener un elemento en caché mediante su índice. Después podemos cambiar el índice para invalidar cualquier caché creado en la clase, de ser necesario.

Finalmente podemos establecer el número de minutos que se almacenará en caché un elemento. Este valor se puede sobreescribir en el método put().



Normalmente uso Memcached para el caché. El manejador para “archivos” NO soporta el método `section()`, así que ocurrirá un error si se utiliza el manejador para “archivos” con esta implementación.

Una nota sobre Los índices de caché

Vale la pena mencionar el buen uso de los índices de caché. Cada elemento tiene un índice único utilizado para obtener los datos. Por convención, dichos índices usan “espacio de nombres” frecuentemente. Laravel agrega un espacio de nombres “Laravel” a cada índice. Esto se puede configurar en `app/config/cache.php`. Si alguna vez es necesario invalidar el caché completo se puede cambiar dicho índice. Útil para grandes cambios en el código que requieren que gran parte de la información en caché sea actualizada.

Esta implementación agrega un espacio de nombres personalizado encima del espacio de nombres global de caché de Laravel, . De esta forma, cualquier instancia de la clase `LaravelCache` puede tener su propio espacio de nombres local que también puede ser modificado. Es posible entonces invalidar el caché de los índices manejados por cualquier instancia de `LaravelCache` en particular.

Véase más sobre nombres de espacio en [esta presentación](#)⁹ de Ilia Alshanetsky, creador de Memcached.

Uso de la implementación

Ahora que tenemos una implementación de `CacheInterface`, podemos usarla en nuestro repositorio. Veamos cómo luciría:

Archivo: `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}
```

⁹http://ilia.ws/files/tnphp_memcached.pdf

```
13 // Dependencias de la clase: un Modelo de Eloquent
14 // una subclase de TagInterface y
15 // una subclase de CacheInterface
16 public function __construct(
17     Model $article, TagInterface $tag, CacheInterface $cache)
18 {
19     $this->article = $article;
20     $this->tag = $tag;
21     $this->cache = $cache;
22 }
23
24 /**
25 * Obtener artículos paginados
26 *
27 * @param int Página
28 * @param int Número de artículos por página
29 * @return Illuminate\Support\Collection - puede ser usado como arreglo
30 */
31 public function byPage($page=1, $limit=10)
32 {
33     // Se construye el índice de caché, único por página y límite
34     $key = md5('page.'.$page.'.'.$limit);
35
36     if( $this->cache->has($key) )
37     {
38         return $this->cache->get($key);
39     }
40
41     $articles = $this->article->with('tags')
42             ->where('status_id', 1)
43             ->orderBy('created_at', 'desc')
44             ->skip( $limit * ($page-1) )
45             ->take($limit)
46             ->get();
47
48     // Se almacena en caché para la siguiente petición
49     $cached = $this->cache->putPaginated(
50         $page,
51         $limit,
52         $this->totalArticles(),
53         $articles->all(),
54         $key
```

```
55     );
56
57     return $cached;
58 }
59
60 /**
61 * Obtener artículo por URL
62 *
63 * @param string URL del artículo
64 * @return object objeto con información del artículo
65 */
66 public function bySlug($slug)
67 {
68     // Se construye el índice de caché, único por página y límite
69     $key = md5('slug.' . $slug);
70
71     if ( $this->cache->has($key) )
72     {
73         return $this->cache->get($key);
74     }
75
76     // El elemento no está en caché, hay que obtenerlo
77     $article = $this->article->with('tags')
78         ->where('slug', $slug)
79         ->where('status_id', 1)
80         ->first();
81
82     // Se almacena en caché para la siguiente petición
83     $this->cache->put($key, $article);
84
85     return $article;
86 }
87
88 /**
89 * Obtener artículos por etiqueta
90 *
91 * @param string URL de la etiqueta
92 * @param int Número de artículos por página
93 * @return Illuminate\Support\Collection - puede ser usado como arreglo
94 */
95 public function byTag($tag, $page=1, $limit=10)
96 {
```

```
97     // Se construye el índice de caché, único por página y límite
98     $key = md5('tag.' . $tag . '.' . $page . '.' . $limit);
99
100    if( $this->cache->has($key) )
101    {
102        return $this->cache->get($key);
103    }
104
105    // El elemento no está en caché, hay que obtenerlo
106    $foundTag = $this->tag->where('slug', $tag)->first();
107
108    if( !$foundTag )
109    {
110        // Probablemente debido a un error. No se devuelven etiquetas.
111        return false;
112    }
113
114    $articles = $this->tag->articles()
115        ->where('articles.status_id', 1)
116        ->orderBy('articles.created_at', 'desc')
117        ->skip( $limit * ($page-1) )
118        ->take($limit)
119        ->get();
120
121    // Se almacena en caché para la siguiente petición
122    $cached = $this->cache->putPaginated(
123        $page,
124        $limit,
125        $this->totalByTag(),
126        $articles->all(),
127        $key
128    );
129
130    return $cached;
131 }
132
133 // El resto de los métodos ha sido omitido para mantener breve el ejemplo
134
135 }
```

Repasemos cómo se utiliza esto. Tendremos una clase `LaravelCache` que será inyectada en nuestro repositorio `Article`. Después, en cada método, verificamos si el elemento está en caché. Si lo está,

se devuelve. De lo contrario, se obtiene el elemento, se almacena en caché para la siguiente petición, y por último, se devuelve.

Fíjense cómo tuvimos que tomar en cuenta a `$page` y `$limit` a la hora de crear el índice caché. Ya que necesitamos crear índice de caché únicos para **todas** las variantes de nuestros datos, tenemos que tomarlo en cuenta - ¡no resultaría útil devolver el mismo conjunto de artículos en cada página!

Ahora podemos actualizar nuestro controlador para que tenga en cuenta estos cambios:

Archivo: app/controllers/ContentController.php

```
1 // Ruta a la página inicial
2 public function home()
3 {
4     // Obtener página, 1 por defecto en caso de no estar presente
5     $page = Input::get('page', 1);
6
7     // Incluir el número de página en que nos encontramos
8     $pagiData = $this->article->byPage($page);
9
10    if( $pagiData !== false )
11    {
12        $articles = Paginator::make(
13            $pagiData->items,
14            $pagiData->totalItems,
15            $pagiData->perPage
16        );
17
18        return View::make('home')->with('articles', $articles);
19    }
20
21    // Vista alternativa en caso de no haber artículos
22    return View::make('home.empty');
23 }
```

Uniendo las piezas

Al igual que con el repositorio de artículos, el último paso es manejar las nuevas dependencias dentro del Proveedor de Servicios.

Archivo: 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\EloquentArticle;
6 use Illuminate\Support\ServiceProvider;
7
8 class RepoServiceProvider extends ServiceProvider {
9
10    /**
11     * Registro del proveedor de servicios
12     *
13     * @return void
14     */
15    public function register()
16    {
17        $app = $this->app;
18
19        $app->bind('Impl\Repo\Article\ArticleInterface', function($app)
20        {
21            // Ahora con las nuevas dependencias de caché
22            return new EloquentArticle(
23                new Article,
24                $app->make('Impl\Repo\Tag\TagInterface'),
25                new LaravelCache($app['cache'], 'articles', 10)
26            );
27        });
28    }
29}
30 }
```



¿Cómo es que supe que podía usar `$app['cache']` para obtener la clase manejadora de caché de Laravel? Eché un vistazo a `Illuminate\Support\Facades\Cache`¹⁰ ¡y me di cuenta cuál índice era utilizado para la clase de caché dentro del contenedor de Inversión de Control!

¡Revisar los Proveedores de Servicios de Laravel puede proporcionar información muy valiosa sobre su funcionamiento!

¹⁰<https://github.com/laravel/framework/blob/master/src/Illuminate/Support/Facades/Cache.php#L10>

Para el repositorio `EloquentArticle` podemos ver que se utiliza el índice ‘articles’. Esto significa que cualquier índice de caché usado en nuestros artículos estará formado así: “`Laravel.articles.” . $key`. Por ejemplo, el índice de caché para un artículo obtenido por URL sería: “`Laravel.articles”.md5(“slug.” . $slug)`”.

De esta manera es posible:

1. Invalidar el caché completo cambiando el espacio de nombres “Laravel” en el archivo de configuración
2. Invalidar el caché de artículos cambiando el nombre de espacios para artículos en el Proveedor de Servicios
3. Invalidar los elementos un artículo cambiando nuestro método de clase
4. Invalidar un artículo específico cambiando la URL del artículo

De esta forma, tenemos **múltiples** niveles de granularidad sobre los elementos que podemos invalidar manualmente, ¡en caso de que lo necesitemos!



Hay que considerar en mover los espacios de nombres de los índices a un archivo de configuración que pueda administrarse desde un solo lugar.

El número de niveles de granularidad es una decisión que puede llevar algo de tiempo en tomar.

Similar a lo que hicimos en el repositorio original, almacenamos la información necesaria para el manejo de paginación. Esto tiene la ventaja de que el código NO depende de una implementación específica de paginación. En su lugar, almacenamos lo que una librería de paginación pudiera necesitar (el número total de elementos, la página actual, el número de elementos por página) y movemos al controlador la responsabilidad de crear el objeto paginador.

¿Qué hemos obtenido?

Hemos almacenado en caché las llamadas a la base de datos de forma mantenible y fácil de probar.

Implementaciones de caché

Ahora es posible intercambiar las implementaciones de caché que usamos en la aplicación. Podemos mantener igual **todo** nuestro código y usar la configuración de Laravel para cambiar entre Redis, Memcached y otros manejadores de caché. Como alternativa, podemos crear nuestra propia implementación y definir su uso en el Proveedor de Servicios.

Separación de intereses

Nos hemos encontrado con algunos obstáculos para hacer que nuestro código no dependa de las librerías de Laravel, mientras que al mismo tiempo permitimos que las pueda utilizar. Podemos cambiar la implementación de caché e implementar cualquier paginación y aún así es posible almacenar en caché los resultados de las consultas a la base de datos.

Pruebas

Al usar los principios de la Inyección de Dependencias es posible realizar pruebas de unidad a cada una de nuestras nuevas clases, ya que podemos simular sus dependencias.

Validación como servicio

La validación y el procesamiento de formularios es tedioso y repetitivo. Ya que los formularios y las entradas de información son frecuentemente el núcleo de la lógica de negocio, tienden a cambiar siempre de un proyecto a otro.

Vamos a discutir una forma de hacer la validación más amena.

¿Qué es?

La validación “como servicio” trata sobre mover la lógica de la validación a un “servicio” que la aplicación puede usar. Similar a cómo comenzamos con un repositorio de base de datos y le agregamos un “servicio de caché”, podemos comenzar con un formulario y agregar un “servicio de validación”.

¿Para qué se usa?

El propósito de agregar validación como servicio es simplemente mantener la separación de intereses y a la vez escribir código mantenible. Por último, también se desea que la validación de formularios resulte muy sencilla. En este capítulo vamos a crear una clase para validación de la que otras clases pueden heredar y utilizar en formularios. Luego veremos cómo usarla en un controlador.

En el siguiente capítulo haremos algo más elegante: remover por completo la validación del controlador y colocarla en un procesador de formularios.

Comencemos con algo simple.

Ejemplo

La situación

Aquí tenemos una validación que podemos encontrar en un controlador. Digamos que necesitamos validar datos de entrada cuando se crea un artículo en el área de administración de nuestro blog:

Un controlador de recursos

```
1 <?php
2
3 class ArticleController extends BaseController {
4
5     // GET /article/create
6     public function create()
7     {
8         return View::make('admin.article_create', array(
9             'input' => Input::old()
10            ));
11    }
12
13    // POST /article
14    public function store()
15    {
16        $input = Input::all();
17        $rules = array(
18            'title' => 'required',
19            'user_id' => 'required|exists:users,id',
20            'status_id' => 'required|exists:statuses,id',
21            'excerpt' => 'required',
22            'content' => 'required',
23            'tags' => 'required',
24        )
25
26        $validator = Validator::make($input, $rules);
27
28        if( ! $validator->fails() )
29        {
30            // CÓDIGO PARA CREAR ARTÍCULO ( OMITIDO PARA BREVEDAD )
31        } else {
32            return View::make('admin.article_create')
33                ->withInput($input)
34                ->withErrors($validator->getErrors());
35        }
36    }
37
38 }
```

Como podemos ver, tenemos todo *esto* en nuestro controlador para validar datos de entrada, ¡y eso

que el ejemplo ni siquiera muestra el código que procesa el formulario! Adicionalmente, ¿qué tal si necesitamos la misma validación cuando se crea el artículo? Tendríamos que repetir el código para ese método.

En un intento de hacer más liviano el controlador y el código más reutilizable, veamos cómo mover la validación fuera del controlador.

Reestructurando

Vamos a recurrir de nuevo a nuestra biblioteca de aplicación. La validación se trata como un **Servicio** de nuestra aplicación y el código se va a crear en el espacio de nombres `Impl\Service`. Esto es lo que contendrá:

```
1 app
2 | - Impl
3 | - [ . . . ]
4 | --- Service
5 | ----- Validation
6 | ----- AbstractLaravelValidator.php
7 | ----- ValidableInterface.php
```

La interfaz

Vamos a comenzar, como usualmente lo hacemos, creando una interfaz. ¿Qué métodos de nuestra clase de validación usarán los controladores?

Sabemos que necesitamos obtener entradas de datos, validarlos y recuperar mensajes de error. Comencemos por estas necesidades:

Archivo: `app/Impl/Service/Validation/ValidableInterface.php`

```
1 <?php namespace Impl\Service\Validation;
2
3 interface ValidableInterface {
4
5     /**
6      * Agregar datos a validar
7      *
8      * @param array
9      * @return \Impl\Service\Validation\ValidableInterface
10     */
11    public function with(array $input);
```

```
12
13     /**
14      * Verificar si pasa la validación
15      *
16      * @return boolean
17      */
18     public function passes();
19
20    /**
21     * Recuperar errores de validación
22     *
23     * @return array
24     */
25     public function errors();
26
27 }
```

Entonces, cualquier clase para validación que se utilice en nuestra aplicación debe implementar esta interfaz. Podemos observar cómo invocaremos a dicha implementación - algo así:

```
1 // En algún lugar del código
2 if( ! $validator->with($input)->passes() )
3 {
4     return $validator->errors();
5 }
```

Una abstracción

Después de la interfaz usualmente podemos crear una implementación concreta. Haremos eso, pero antes, vamos a agregar una capa más.

La validación es un servicio en el que tiene sentido generalizar. Vamos a crear una clase abstracta que utilice el validador de Laravel y que pueda extenderse y modificarse para que sea utilizada fácilmente en cualquier formulario. Una vez hecho eso vamos a poder extender esta clase base, cambiar algunos parámetros y estar listos para cualquier escenario de validación.

Nuestra clase abstracta implementará nuestra interfaz y hará uso de la librería de validación de Laravel:

Archivo: app/Impl/Service/Validation/AbstractLaravelValidator.php

```
1 <?php namespace Impl\Service\Validation;  
2  
3 use Illuminate\Validation\Factory as Validator;  
4  
5 abstract class AbstractLaravelValidator implements ValidableInterface {  
6  
7     /**  
8      * Validador  
9      *  
10     * @var \Illuminate\Validation\Factory  
11     */  
12     protected $validator;  
13  
14     /**  
15      * Datos de validación arreglo índice => valor  
16      *  
17      * @var Array  
18      */  
19     protected $data = array();  
20  
21     /**  
22      * Errores de validación  
23      *  
24      * @var Array  
25      */  
26     protected $errors = array();  
27  
28     /**  
29      * Reglas de validación  
30      *  
31      * @var Array  
32      */  
33     protected $rules = array();  
34  
35     /**  
36      * Mensajes de validación personalizados  
37      *  
38      * @var Array  
39      */  
40     protected $messages = array();
```

```
41
42     public function __construct(Validator $validator)
43     {
44         $this->validator = $validator;
45     }
46
47     /**
48      * Establecer datos a validar
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      * La validación pasa o falla
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      * Devolver errores, si los hay
```

```
83     *
84     * @return array
85     */
86    public function errors()
87    {
88        return $this->errors;
89    }
90
91 }
```

Enseguida, veremos cómo podemos extender fácilmente esta clase para cumplir con las necesidades de cualquier validación.

Implementación

Vamos a crear una clase de validación para el ejemplo del principio de este capítulo:

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

```
1 <?php namespace Impl\Service\Validation;
2
3 class ArticleFormValidator extends AbstractLaravelValidator {
4
5     /**
6      * Reglas de validación
7      *
8      * @var Array
9      */
10    protected $rules = array(
11        'title' => 'required',
12        'user_id' => 'required|exists:users,id',
13        'status_id' => 'required|exists:statuses,id',
14        'excerpt' => 'required',
15        'content' => 'required',
16        'tags' => 'required',
17    );
18
19    /**
20     * Mensajes de validación
21     *
22     * @var Array
```

```
23     */
24     protected $messages = array(
25         'user_id.exists' => 'That user does not exist',
26         'status_id.exists' => 'That status does not exist',
27     );
28 }


---


```

¡Y eso es todo! Vayamos de vuelta a nuestro controlador para verlo en acción:

Un controlador de recursos

```
1 <?php
2
3 use Impl\Service\Validation\ArticleLaravelValidator;
4
5 class ArticleController extends BaseController {
6
7     // Dependencia: clase concreta ArticleLaravelValidator
8     // Noten que Laravel lo resuelve por nosotros
9     // No vamos a necesitar un Proveedor de Servicios
10    public function __construct(ArticleLaravelValidator $validator)
11    {
12        $this->validator = $validator;
13    }
14
15    // GET /article/create
16    public function create()
17    {
18        return View::make('admin.article_create', array(
19            'input' => Input::old()
20        ));
21    }
22
23    // POST /article
24    public function store()
25    {
26        if( $this->validator->with( Input::all() )->passes() )
27        {
28            // PROCESAMIENTO DEL FORMULARIO
29        } else {

```

```
30         return View::make('admin.article_create')
31             ->withInput( Input::all() )
32             ->withErrors($validator->errors());
33     }
34 }
35
36 }
```

¿Qué hemos logrado?

Pudimos lograr que nuestros controladores sean un poco más livianos al remover el código de validación y moverlo a la biblioteca de la aplicación.

Adicionalmente, hemos creado una clase abstracta reutilizable. Para cualquier formulario, podemos extender esta clase, definir nuestras propias reglas, ¡y listo!

Más adelante veremos cómo hacerlo más dinámico al mover el procesamiento del formulario fuera del controlador.

Procesamiento de formularios

La mitad del manejo de entradas de información es validación. La otra mitad es, por supuesto, el procesamiento de formularios.

¿Qué es?

Veamos lo que es el “procesamiento de formularios”. Si tenemos un formulario, un usuario envía información, los pasos siguientes son por lo regular:

1. Obtener entradas de información
2. Validar las entradas de información
3. Realizar operaciones con las entradas de información
4. Generar una respuesta

¿Dónde lo utilizamos?

Un controlador es un excelente lugar para los elementos 1 y 4. Hemos visto cómo la validación (paso 2) puede ser manejada como un servicio. ¡El procesamiento del formulario (paso 3) también puede ser manejado como un servicio dentro de nuestra aplicación!

Mover el procesamiento de formularios fuera del controlador, hacia un conjunto de clases de servicio, nos brinda una mejor oportunidad de efectuar pruebas de unidad. Podemos quitar la dependencia de tener que ejecutarse en el contexto de una petición HTTP, lo que permite simular sus dependencias.

En pocas palabras, mover el código del formulario fuera del controlador nos ofrece una manera más reutilizable y mantenible de manejarlo.

Ejemplo

Ya tenemos la validación del capítulo anterior. Veamos cómo podemos encontrar un mejor lugar para el procesamiento de formularios y la validación.

La situación

Vamos a crear un servicio de formularios, el cual hace uso de nuestra clase Repositorio para manejar las operaciones básicas y el servicio de validación para manipular las entradas de datos.

En el capítulo anterior de validación, terminamos con un controlador que contiene la validación como servicio:

Un controlador de recursos

```
1 // POST /article
2 public function store()
3 {
4     if( $this->validator->with( Input::all() )->passes() )
5     {
6         // PROCESAMIENTO DE FORMULARIO
7     } else {
8         return View::make('admin.article_create')
9             ->withInput( Input::all() )
10            ->withErrors($validator->errors());
11    }
12 }
```

Lo que podemos hacer ahora es preparar el procesamiento de formularios como servicio.

Después de hacer esto, veamos cómo luce nuestra nueva estructura de directorios:

```
1 app
2 | - Impl
3 | --- Service
4 | ----- Form
5 | ----- Article
6 | ----- ArticleForm.php
7 | ----- ArticleFormLaravelValidator.php
8 | ----- FormServiceProvider.php
```

Reestructurando

Esta es una de esas veces en las que **no** comenzaremos por crear una implementación. La clase formulario *usará* clases que implementan una interfaz, pero no implementará una interfaz por sí misma. Esto se debe a que no hay otras implementaciones de formulario que usar. Es PHP puro - tomará las entradas de datos en forma de arreglo y *orquestará* el uso del servicio de validación y el repositorio de datos.

Validación

Comencemos usando la validación del capítulo pasado. En este ejemplo vamos a crear un formulario para la creación y actualización de artículos. En ambos casos las reglas de validación son las mismas. Por último, noten que movemos el validador de formularios del artículo hacia el directorio de servicios Form.

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

```
1 <?php namespace Impl\Service\Form\Article;
2
3 use Impl\Service\Validation\AbstractLaravelValidator;
4
5 class ArticleFormLaravelValidator extends AbstractLaravelValidator {
6
7     /* Igual que en el capítulo anterior,
8      con un nuevo espacio de nombres y ubicación */
9
10 }
```

Formulario

ahora vamos a crear nuestra clase ArticleForm. Esta clase va a orquestar la creación y edición de artículos, usando validación, repositorios y entrada de datos.

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

```
1 <?php namespace Impl\Service\Form\Article;
2
3 use Impl\Service\Validation\ValidableInterface;
4 use Impl\Repo\Article\ArticleInterface;
5
6 class ArticleForm {
7
8     /**
9      * Datos del Formulario
10     *
11     * @var array
12     */
13     protected $data;
14
15     /**
16      * Validador
17      *
18      * @var \Impl\Service\Form\Contracts\ValidableInterface
19      */
20     protected $validator;
```

```
21
22     /**
23      * Repositorio de Artículos
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      * Creación de un nuevo artículo
38      *
39      * @return boolean
40      */
41     public function save(array $input)
42     {
43         // Aquí va el código
44     }
45
46     /**
47      * Actualización de un artículo existente
48      *
49      * @return boolean
50      */
51     public function update(array $input)
52     {
53         // Aquí va el código
54     }
55
56     /**
57      * Se devuelve cualquier error de validación
58      *
59      * @return array
60      */
61     public function errors()
62     {
```

```
63         return $this->validator->errors();
64     }
65
66     /**
67      * Probar si el validador pasa
68      *
69      * @return boolean
70      */
71     protected function valid(array $input)
72     {
73         return $this->validator->with($input)->passes();
74     }
75
76 }
```

Este es el caparazón de nuestro formulario de Artículo. Podemos ver que se inyectan las dependencias `ValidableInterface` y `ArticleInterface`. Además hay un método para devolver errores de validación.

Veamos cómo luce la creación y actualización de un artículo:

Un controlador de recursos

```
1 /**
2  * Creación de un nuevo artículo
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  * Actualización de un artículo existente
18  *
19  * @return boolean
```

```
20  /*
21  public function update(array $input)
22  {
23      if( ! $this->valid($input) )
24      {
25          return false;
26      }
27
28      return $this->article->update($input);
29 }
```

¡Eso fue fácil! Nuestro formulario se encarga del procesamiento, pero en realidad no hace el trabajo pesado de crear y actualizar artículos. En su lugar, le pasa la responsabilidad a nuestras clases que implementan una interfaz, las cuales hacen el trabajo por nosotros. La clase para formulario simplemente orquesta el proceso.

Trabajo pesado

Ya tenemos el procesamiento de formulario en su lugar. Toma entradas de datos, ejecuta la validación, devuelve errores y envía los datos para su procesamiento.

El último paso es realizar el trabajo pesado de creación/actualización de artículos.

Volvamos a nuestra interfaz de artículos. Ya que sabemos que necesitamos crear y actualizar artículos, podemos escribir los métodos necesarios `create()` y `update()`. Veamos cómo luciría:

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

```
1 <?php namespace Impl\Repo\Article;
2
3 interface ArticleInterface {
4
5     /* El código explicado anteriormente se ha omitido */
6
7     /**
8      * Creación de un nuevo artículo
9      *
10     * @param array Datos para crear un nuevo objeto
11     * @return boolean
12     */
13    public function create(array $data);
14 }
```

```
15     /**
16      * Actualización de un artículo existente
17      *
18      * @param array Datos para actualizar un artículo
19      * @return boolean
20      */
21     public function update(array $data);
```

Noten que sabemos cómo espera interactuar nuestro código con nuestro repositorio, vamos a enrollarnos las mangas para comenzar con el trabajo sucio.

Archivo: 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    // La clase espera un modelo de Eloquent
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    // El código explicado anteriormente se ha omitido
23
24    /**
25     * Creación de un nuevo artículo
26     *
27     * @param array Datos para crear un nuevo objeto
28     * @return boolean
```

```
29     */
30     public function create(array $data)
31     {
32         // Creación del artículo
33         $article = $this->article->create(array(
34             'user_id' => $data['user_id'],
35             'status_id' => $data['status_id'],
36             'title' => $data['title'],
37             'slug' => $this->slug($data['title']),
38             'excerpt' => $data['excerpt'],
39             'content' => $data['content'],
40         ));
41
42         if( ! $article )
43         {
44             return false;
45         }
46
47         // Método auxiliar
48         $this->syncTags($article, $data['tags']);
49
50         return true;
51     }
52
53 /**
54 * Actualización de un artículo existente
55 *
56 * @param array Datos para actualizar un artículo
57 * @return boolean
58 */
59 public function update(array $data)
60 {
61     $article = $this->article->find($data['id']);
62
63     if( ! $article )
64     {
65         return false;
66     }
67
68     $article->user_id = $data['user_id'];
69     $article->status_id = $data['status_id'];
70     $article->title = $data['title'];
```

```
71     $article->slug = $this->slug($data['title']);
72     $article->excerpt = $data['excerpt'];
73     $article->content = $data['content'];
74     $article->save();
75
76     // Método auxiliar
77     $this->syncTags($article, $data['tags']);
78
79     return true;
80 }
81
82 /**
83 * Sincronización de etiquetas para artículo
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     // Devolver etiquetas después de obtener
92     // las etiquetas existentes y crear nuevas
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    // Asignar etiquetas al artículo
102    $this->article->tags()->sync($tagIds);
103 }
104
105 }
```

Ahora tenemos una implementación de Eloquent para crear o actualizar un artículo. También tenemos un método `findOrCreate()` en el repositorio de etiquetas el cual (como ya adivinaron) encuentra o crea etiquetas si todavía no existen. Esto nos permite agregar nuevas etiquetas conforme se necesite cada que creemos o actualicemos un artículo.

La implementación del método `findOrCreate` del repositorio Tags se encuentra en el repositorio de Github del libro.



Los pasos necesarios para asignar etiquetas a cada artículo se repiten frecuentemente, por eso se creó un método protegido syncTags que haga eso. Dicho método no tiene que ser parte de la interfaz ya que es un método interno que utilizan los métodos dentro de la clase!

En resumen

Aunque no es necesario un Proveedor de Servicios para definir que clase de formulario crear (hay solamente una clase concreta ArticleForm), se deben definir las dependencias que se van a inyectar. Estas son: la clase para validación de Laravel y la implementación de ArticleInterface.

Lo haremos en un Proveedor de Servicios. Veamos cómo luciría:

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

```
1 <?php namespace Impl\Service\Form;
2
3 use Illuminate\Support\ServiceProvider;
4 use Impl\Service\Form\Article\ArticleForm;
5 use Impl\Service\Form\Article\ArticleFormLaravelValidator;
6
7 class FormServiceProvider extends ServiceProvider {
8
9     /**
10      * Registro de vinculaciones
11      *
12      * @return void
13      */
14     public function register()
15     {
16         $app = $this->app;
17
18         $app->bind('Impl\Service\Form\Article\ArticleForm', function($app)
19         {
20             return new ArticleForm(
21                 new ArticleFormLaravelValidator( $app['validator'] ),
22                 $app->make('Impl\Repo\Article\ArticleInterface')
23             );
24         });
25     }
26
27 }
```

El último paso es agregar esto al archivo `app/config/app.php` junto con los demás Proveedores de Servicios.

Resultados finales

¡Finalmente podemos ver cómo luce nuestro controlador! Hemos movido todo el procesamiento de formularios y la lógica de validación fuera del controlador. Ahora es posible probarlos y reutilizarlos independientemente. La lógica restante en el controlador, además de llamar a nuestra clase de procesamiento de formularios, utiliza las fachadas de Laravel para redirigir y responder a las peticiones.

Un controlador de recursos

```
1 <?php
2
3 use Impl\Repo\Article\ArticleInterface;
4 use Impl\Service\Form\Article\ArticleForm;
5
6 class ArticleController extends BaseController {
7
8     protected $articleform;
9
10    public function __construct(
11        ArticleInterface $article, ArticleForm $articleform)
12    {
13        $this->article = $article;
14        $this->articleform = $articleform;
15    }
16
17 /**
18 * Creación del formulario de artículo
19 * GET /admin/article/create
20 */
21 public function create()
22 {
23     View::make('admin.article_create', array(
24         'input' => Session::getOldInput()
25     ));
26 }
27
28 /**
```

```
29     * Creación del procesamiento de formulario de artículo
30     * POST /admin/article
31     */
32     public function store()
33     {
34         // Procesamiento de formulario
35         if( $this->articleform->save( Input::all() ) )
36         {
37             // ¡Éxito!
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 * Creación del formulario de artículo
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 * Creación del formulario de artículo
62 * PUT /admin/article/{id}
63 */
64     public function update()
65     {
66         // Procesamiento de formulario
67         if( $this->articleform->update( Input::all() ) )
68         {
69             // ¡Éxito!
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 }
```

¿Qué hemos logrado?

Hemos hecho que nuestro procesamiento de formularios sea mantenible y pueda probarse:

- Podemos probar la clase ArticleForm con un uso liberal de objetos simulados.
- Podemos cambiar fácilmente las reglas de validación del formulario.
- Podemos intercambiar implementaciones del almacenamiento de datos sin cambiar el código del formulario.
- Ganamos reusabilidad en el código. Podemos llamar al código del formulario fuera del contexto de una petición HTTP. Podemos (¡y lo hacemos!) reutilizar el repositorio de datos y las clases de validación!

Construir nuestra aplicación usando el patrón repositorio y utilizando la inyección de dependencias y otras prácticas de SOLID, nos ha permitido lograr que el procesamiento de formularios sea más tolerable.

Manejo de errores

Laravel maneja las respuestas que un usuario recibe como resultado de una petición. Laravel actúa como un intermediario entre nuestra aplicación y la petición HTTP, y tiene la responsabilidad de generar la respuesta a una petición. Nuestra aplicación simplemente proporciona a Laravel los datos que necesita.

El cómo Laravel responde a los *errores* depende del contexto. Laravel realiza peticiones a nuestra aplicación y devuelve una representación de la respuesta. En un navegador, se devuelve una representación en HTML. En la llamada a una API, puede que la respuesta sea un objeto JSON. Por último, en un comando de la terminal, una respuesta en texto plano puede que sea más apropiada.

No es necesario mucho trabajo extra para el manejo de errores. Podemos interceptarlos y decidir cómo mostrarlos (o no mostrarlos) al usuario mediante el mecanismo incorporado de Laravel.

Sin embargo, está el asunto de los errores de lógica de negocio. Quizá sea necesario manejar por separado los errores de lógica de negocios y los específicos de Laravel. Por ejemplo, tal vez sea necesario mostrar alertas sobre fallas en un proceso importante de la aplicación, mientras que no haya problema si se ignoran los errores 404 de HTTP.

En este capítulo, vamos a ver cómo podemos hacer que la aplicación lance ciertas excepciones y cómo usar un manejador de errores personalizado para efectuar cualquier acción necesaria.

Específicamente, vamos a ver cómo atrapar excepciones de lógica de negocio y enviar notificaciones como resultado.

¿Qué es el manejo de errores?

El manejo de errores es, obviamente, cómo responde la aplicación frente a un error.

Hay dos cosas a considerar al manejar errores:

1. Realizar alguna acción frente al error.
2. Desplegar información detallada sobre el error.

Realizar una acción frente a un error usualmente consiste en registrar y alertar. Queremos registro de cualquier error para revisión y depuración. Tal vez también sea necesario algún mecanismo de alerta para errores severos que puedan detener la ejecución de un proceso crítico.

Desplegar información detallada, como ya se ha mencionado, depende del contexto. Durante el desarrollo, queremos que los errores se muestren al desarrollador. Si utilizan una consola, un error legible en consola es apropiado. Si utilizan un navegador, un error en el navegador es apropiado. Si se trata del ambiente de producción, entonces un error “amigable” es apropiado para mostrar a los usuarios.

Preparando a Laravel para que responda a errores

Laravel debe manejar el despliegue de errores. Yo coloco los manejadores de errores de Laravel en el archivo `routes.php` para interceptar las peticiones y devolver una respuesta. Igual que al enrutar peticiones GET, POST y de otro tipo.

El truco es decidir cuando mostrar un error y cuando mostrar un mensaje “amigable” al usuario.

Archivo: `app/routes.php`

```
1  /*
2  En el manejo de errores es preferible definir primero los errores más generales.
3
4  En el siguiente ejemplo, una URL incorrecta será interceptada primero por el
5  manejador `NotFoundHttpException` y luego por el manejador genérico `Exception`.
6
7  Si se invierte el orden, el manejador `NotFoundHttpException` será omitido
8  si la depuración está desactivada!
9 */
10
11 use Symfony\Component\HttpFoundation\Exception\NotFoundHttpException;
12
13 // Otras excepciones
14 App::error(function(\Exception $e)
15 {
16
17     if( Config::get('app.debug') === true )
18     {
19         return null; // Delegar al manejador de Laravel
20     }
21
22     return View::make('error');
23
24 });
25
26 // 404
```

```
27 App::error(function(NotFoundHttpException $e)
28 {
29
30     if( Config::get('app.debug') === true )
31     {
32         return null; // Delegar al manejador de Laravel
33     }
34
35     return View::make('404');
36
37 }));
```

Noten que el orden en que se definen los errores importa. Hay que ordenar los manejadores desde el más general al más específico - Laravel revisará cada manejador de errores hasta devolver una respuesta.

Preparando nuestra aplicación para que responda a errores

Ahora que podemos hacer que los errores se muestren a los usuarios de web, vamos a decidir cómo manejar los errores que provengan de nuestra aplicación.

Primero, vamos a definir una interfaz. Ya sabemos que deseamos “manejar” un error, así que comencemos por ahí:

Archivo: app/Impl/Exception/HandlerInterface.php

```
1 <?php namespace Impl\Exception;
2
3 interface HandlerInterface {
4
5     /**
6      * Handle Impl Exceptions
7      *
8      * @param \Impl\Exception\ImplException
9      * @return void
10     */
11    public function handle(ImplException $exception);
12
13 }
```

Cualquier manejador de error “manejará” una excepción - y no cualquier excepción sino `ImplException` y sus subclases.

Vamos a definir una excepción base. Será utilizada como la clase base para cualquier excepción específica de la aplicación.

Archivo: app/Impl/Exception/ImplException.php

```
1 <?php namespace Impl\Exception;  
2  
3 class ImplException extends \Exception {}
```

Ahora podemos decidir lo que deseamos que hagan nuestros manejadores. Digamos que para una excepción en específico, queremos que nuestra aplicación nos notifique del error.

Implementación

Como ya tenemos nuestra interfaz HandlerInterface, el siguiente paso es crear una implementación de ella. Queremos que nos notifique sobre errores, así que la vamos a llamar NotifyHandler.

Archivo: app/Impl/Exception/NotifyHandler.php

```
1 <?php namespace Impl\Exception;  
2  
3 use Impl\Service\Notification\NotifierInterface;  
4  
5 class NotifyHandler implements HandlerInterface {  
6  
7     protected $notifier;  
8  
9     public function __construct(NotifierInterface $notifier)  
10    {  
11        $this->notifier = $notifier;  
12    }  
13  
14    /**  
15     * Manejar excepciones Impl  
16     *  
17     * @param \Impl\Exception\ImplException  
18     * @return void  
19     */  
20    public function handle(ImplException $exception)  
21    {
```

```
22     $this->sendException($exception);
23 }
24
25 /**
26 * Enviar excepción al notificador
27 * @param \Exception $exception Enviar notificación de excepción
28 * @return void
29 */
30 protected function sendException(\Exception $e)
31 {
32     $this->notifier->notify('Error: '.get_class($e), $e->getMessage());
33 }
34
35 }
```

¡Ahora tenemos una clase que notifica sobre excepciones! El último paso es establecerlo como manejador para las excepciones de nuestra aplicación.



Tal vez se hayan dado cuenta que existe una dependencia en esta implementación. Por ahora vamos a suponer que la clase NotifierInterface, cuya implementación no se muestra, es una clase que nos enviará una alerta de algún tipo. En el siguiente capítulo se explicará el uso de una librería externa utilizada para enviar notificaciones.

Juntando las piezas.

El último paso es usar este manejador de errores cuando se dispare una excepción del tipo `ImplException` (o de alguna de sus subclases). Para ello, utilizamos la capacidad de manejo de errores de Laravel.

Archivo: `app/Impl/Exception/ExceptionServiceProvider.php`

```
1 <?php namespace Impl\Exception;
2
3 use Illuminate\Support\ServiceProvider;
4
5 class ExceptionServiceProvider extends ServiceProvider
6 {
7     public function register()
8     {
9         $app = $this->app;
```

```
10
11     // Vinculación del manejador de errores de la aplicación
12     $app['impl.exception'] = $app->share(function($app)
13     {
14         return new NotifyHandler( $app['impl.notifier'] );
15     });
16 }
17
18 public function boot()
19 {
20     $app = $this->app;
21
22     // Registro del manejador de errores de Laravel
23     $app->error(function(ImplException $e) use ($app)
24     {
25         $app['impl.exception']->handle($e);
26     });
27 }
28 }
```

¿Qué ocurrió aquí? Primero, el método `register` registra a `impl.exception` como el manejador de errores de nuestra aplicación.

Luego el método `boot()` usa el manejador de errores de Laravel y registra el manejador `impl.exception` para que atrape cualquier excepción generada por la clase base `ImplException`.

Ahora, cada que se genere una excepción `ImplException`, nuestro manejador la capturará y manejará el error. Ya que no estamos devolviendo un `Response` en el manejador, Laravel continuará recorriendo sus propios manejadores de errores, que se pueden utilizar para devolver una respuesta adecuada al usuario.



Podemos ver que de nuevo hemos usado `$app['impl.notifier']`, el cual no ha sido definido aún en nuestra aplicación. Eso lo explicaremos en el siguiente capítulo.

Uso de paquetes

Uso de paquetes: notificaciones

En el capítulo anterior, configuramos un manejador de errores que nos envía una notificación al ocurrir un error en la aplicación.

En este capítulo, vamos a crear la funcionalidad para notificaciones usando un paquete externo de Composer.

Vamos a enviar un mensaje de texto a nuestros teléfonos cuando se dispare una excepción.

Preparación.

Necesitamos una herramienta que pueda enviar notificaciones de algún tipo. En ese sentido, nuestra aplicación necesita un servicio de notificaciones.

Ya que hemos identificado que se trata de un servicio, tenemos una idea de dónde estará ubicado:

```
1 app
2 | - Impl
3 | --- Service
4 | ----- Notification
```

Luego, como usualmente hemos hecho, vamos a crear una interfaz. ¿Qué es lo que requiere nuestro notificador? Pues bien, necesita el mensaje que enviará, a quién se lo enviará, así como también necesita saber quién lo envía. Comencemos por ahí:

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

```
1 <?php namespace Impl\Service\Notification;
2
3 interface NotifierInterface {
4
5     /**
6      * Destinatario de la notificación
7      * @param string $to destinatario
8      * @return Impl\Service\Notification\NotifierInterface
9     */
10    public function to($to);
11}
```

```
12  /**
13   * Remitente de la notificación
14   * @param string $from The sender
15   * @return Impl\Service\Notification\NotifierInterface
16   */
17  public function from($from);
18
19  /**
20   * Enviar notificación
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 }
```

Implementación

Luego, necesitamos implementar la interfaz. Hemos decidido que nos enviaremos un mensaje de texto. Para esto, podemos usar el servicio de Twilio.

Twilio tiene disponible un SDK para PHP en forma de paquete de Composer. Para agregarlo a nuestro proyecto, podemos añadir lo siguiente al archivo `composer.json`:

```
1 // Agregar a composer.json como dependencia
2 // o ejecutar lo siguiente en la raíz del proyecto:
3 $ php composer require twilio/sdk:dev-master
```

Después de instalar el SDK, podemos comenzar a usarlo. Vamos a crear una clase `SmsNotifier` que utilice el servicio de Twilio.

Archivo: `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     * Destinatario de la notificación
9     * @var string
10    */
11   protected $to;
12
13 /**
14  * Remitente de la notificación
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  * Destinatario de la notificación
32  * @param string $to destinatario
33  * @return Impl\Service\Notificaton\SmsNotifier
34  */
35   public function to($to)
36 {
37     $this->to = $to;
38
39     return $this;
40 }
41
42 /**
43  * Remitente de la notificación
44  * @param string $from remitente
45  * @return Impl\Service\Notificaton\NotifierInterface
46  */
47   public function from($from)
48 {
49     $this->from = $from;
```

```
50
51     return $this;
52 }
53
54 /**
55 * Enviar notificación
56 * @param string $subject Asunto de la notificación
57 * @param string $message Contenido de la notificación
58 * @return void
59 */
60 public function notify($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 }
```

Y esa es toda la parte difícil del trabajo. Hemos definido una interfaz e implementado un notificador de mensajes de texto, el cual utiliza el SDK para PHP de Twilio.



Noten que si necesitamos cambiar el proveedor de mensajes, podríamos realizar más pasos para abstraerlo creando “transportes”. Por ejemplo, una clase `SmsTransport` podría ser Twilio, pero también podría ser otro proveedor de mensajes. Por simplicidad, `SmsNotifier` supone que se trata de Twilio y no crea ninguna abstracción para separar “transportes”.

Fíjense cómo Laravel utiliza Swift Mailer (paquete `Illuminate\Mail`) para el envío de correos a manera de ejemplo de cómo se utilizan los transportes.

Juntando las piezas

El último paso es agregar la configuración de nuestra cuenta de Twilio y unir todo mediante un Proveedor de Servicios.

En la carpeta de configuración, podemos crear un archivo para Twilio.

Archivo: app/config/twilio.php

```
1 <?php
2
3 return array(
4
5     'from' => '555-1234',
6
7     'to' => '555-5678',
8
9     'account_id' => 'abc1234',
10
11    'auth_token' => '11111111',
12
13 );
```

Ahora este archivo de configuración estará disponible en nuestra aplicación. Enseguida podemos crear un Proveedor de Servicios para unirlo todo.

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

```
1 <?php namespace Impl\Service\Notification;
2
3 use Services_Twilio;
4 use Illuminate\Support\ServiceProvider;
5
6 class NotificationServiceProvider extends ServiceProvider {
7
8     /**
9      * Registro del proveedor de servicios.
10     *
11     * @return void
12     */
13    public function register()
14    {
15        $app = $this->app;
16
17        $app['impl.notifier'] = $app->share(function($app)
18        {
19            $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 }
```

Hemos utilizado el SDK de Twilio, le enviamos las credenciales de api desde la configuración, establecemos el remitente y destinatario desde la configuración y se lo enviamos a nuestra clase `SmsNotifier`.

Hay que registrar este Proveedor de Servicios en el archivo `app/config/app.php` y quedará listo.

En acción

Podemos verlo en acción en nuestro manejador de errores. Recuerden que hemos creado un manejador llamado `ExceptionServiceProvider` que utiliza las notificaciones:

Archivo: `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 }
```

¡Ahora \$app['impl.notifier'] existe para que nuestra clase NotifyHandler lo use como su implementación de notificaciones!

Podemos probarlo al lanzar una excepción `ImplException` en el código y esperar a recibir un mensaje de texto.

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

Si más adelante deseamos usar el notificador de mensajes de texto en cualquier lugar del código, ¡es posible!

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

¿Qué hemos logrado?

Vimos un ejemplo de cómo instalar y usar una librería externa de Composer. En este ejemplo utilizamos el SDK de Twilio para mensajes de texto en una implementación de nuestro servicio de notificaciones.

Cuando se encuentren con la necesidad de programar cierta funcionalidad, deberían de revisar primero si ya hay algo en [Packagist¹¹](#). ¡Es posible que ya exista una librería bien hecha y probada!

¹¹<http://packagist.org>

Conclusión

Recapitulación

¡Hemos abarcado bastante en este libro! Algunos temas incluyeron:

Instalación

Cómo instalar Laravel, incluyendo entornos, preparación y consideraciones para ambientes de producción.

Preparación de la aplicación

Uso de una biblioteca para la aplicación.

Patrón repositorio

El cómo y por qué del uso de repositorios como interfaz para el almacenamiento de datos.

Caché en el repositorio

Agregamos una capa de servicio a nuestro repositorio. En este ejemplo agregamos una capa de caché.

Validación

Creamos validación como servicio, para ayudar a abstraer el trabajo de validación e implementar clases específicas para nuestros casos de uso.

Procesamiento de formularios

Creamos clases para orquestar la validación de entradas de datos y la interacción con nuestros repositorios. Estas clases para formularios están aisladas de las peticiones HTTP y pueden ser utilizadas fuera de los controladores. Esto hace que también sean fáciles de probar.

Manejo de errores

Abordamos algunas consideraciones de cómo y cuándo usar manejo de errores. Vimos un ejemplo de cómo se puede utilizar para atrapar algunos tipos de errores específicos.

Librerías externas

Utilizamos el SDK de Twilio para construir una librería de notificaciones para poder enviar mensajes de texto desde el manejador de errores.

¿Qué hemos logrado?

La mayor parte del libro provee la base para la construcción de aplicaciones completas.

Lo que espero les haya quedado claro es cómo se pueden usar los principios SOLID en Laravel. El uso de interfaces, el contenedor de Inversión de Control y los Proveedores de Servicios proporcionan una manera poderosa de crear una aplicación mantenible en PHP.

El futuro

Espero que las primeras ediciones de este libro sirvan como base para la construcción de aplicaciones.

La respuesta a muchas preguntas tales como “¿Cómo utilizo las colas de procesos de manera efectiva?”, “¿Cómo integro un motor de búsqueda?” o “¿Cuál es un uso avanzado de los Proveedores de Servicios?” partirá de los conceptos fundamentales que se han explicado aquí.

Con un poco de suerte, esto será sólo el comienzo.

¡Feliz programación!