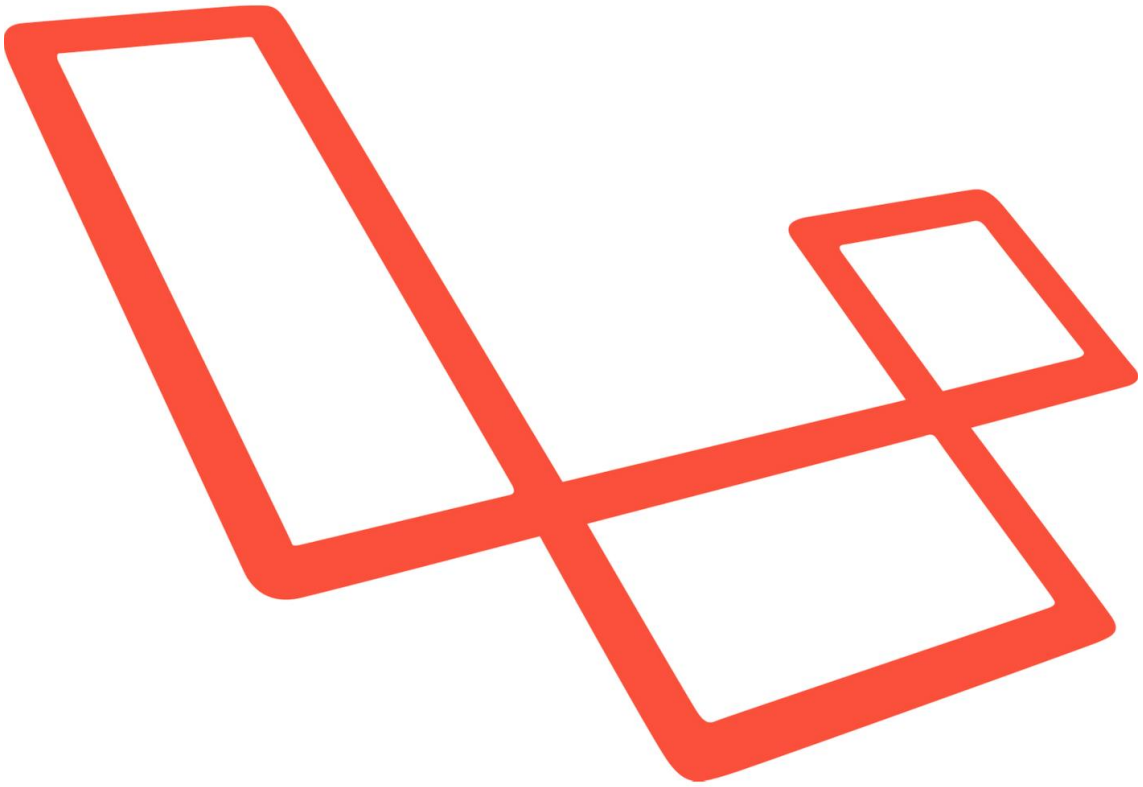

LARAVEL

TIPS Y BUENAS PRÁCTICAS

RECOPILACIÓN DE CONSEJOS Y BUENAS PRÁCTICAS PARA TRABAJAR CON EL FRAMEWORK

NigmaCode.com

ÍNDICE

Buenas prácticas en Laravel (1-3)

Una tarea por función

Las funciones deben tener una única responsabilidad, es decir, si queremos realizar varias tareas podemos crear una función para cada una de ellas. De esta forma, nuestro código es mucho más legible y fácil de mantener.

Ejemplos:

Mal hecho:

```
public function checkProduct()
    {
        if($this->isStock() && $this->isAvailable()){
            return 'Nombre: '.$this->name.' precio: '.$this->price;
        }else{
            return 'El producto '.$this->name.' no esta disponible';
        }
    }
```

Bien hecho:

```
    public function checkProduct()
    {
        return $this->checkIsPurchasable() ? $this->isPurchasable() : $this->isNotPurchasable();
    }
    public function checkIsPurchasable()
    {
        return $this->isStock() && $this->isAvailable();
    }
    public function isPurchasable()
    {
        return 'Nombre: '.$this->name.' precio: '.$this->price;
    }
    public function isNotPurchasable()
    {
        return 'El producto '.$this->name.' no esta disponible';
    }
```

Validaciones

Las validaciones son esenciales en Laravel, pero no todos las creamos como deberíamos.

Si eres de los que crean una validación en el mismo controlador, deberías comenzar a crear una clase que extienda de request y una vez hecho esto, enviar este request por parámetro a la función de nuestro controlador.

Ejemplo:

Mal hecho:

```
    public function store(Request $request)
{
    $request->validate([
        'name' => 'required|max:255',
        'price' => 'required',
        'publish_at' => 'nullable|date',
    ]);

    ....
}
```

Bien hecho:

```
public function store(ProductRequest $request)
{
    ....
}

class ProductRequest extends Request
{
    public function rules()
    {
        return [
            'name' => 'required|max:255',
            'price' => 'required',
            'publish_at' => 'nullable|date',
        ];
    }
}
```

Utiliza Eloquent

Eloquent nos permite generar consultas mucho más legibles y fácil de mantener. Tiene muchísimas herramientas ya integradas que nos facilita las consultas en Laravel:

Mal hecho:

```
SELECT *
FROM `articles`
WHERE EXISTS (SELECT *
              FROM `users`
              WHERE `articles`.`user_id` = `users`.`id`
              AND EXISTS (SELECT *
                          FROM `profiles`
                          WHERE `profiles`.`user_id` = `users`.`id`)
              AND `users`.`deleted_at` IS NULL)
AND `verified` = '1'
AND `active` = '1'
ORDER BY `created_at` DESC
```

Bien hecho:

```
Article :: has ( ' user.profile ' ) -> verified () -> latest () -> get ();
```

No hace falta asignar de uno a uno el valor del request

A la hora de insertar en nuestra base de datos un valor, podemos hacer uso de la función create de nuestro modelo, para asignarle todo lo que nos devuelve el request.

Ejemplo:

Mal hecho:

```
$product = new Product;
$article->name = $request-> name;
$article->price = $request-> price;
$article->stock = $request-> stock;
```

Bien hecho:

```
$category->product()->create($request->all());
```

No utilices las consultas en las plantillas blade

Si utilizamos las consultas en la plantilla blade echaremos a perder la optimización de nuestro proyecto con esta acción, ¿Por qué? Te lo mostramos:

Ejemplo

Mal hecho:

Para 500 usuarios haremos 501 consultas:

```
@foreach (Product::all() as $product)
    {{ $product->name }}
@endforeach
```

Bien hecho:

Para 500 usuarios haremos 1 consulta:

```
$products = Product::all();

...

@foreach ($products as $ product)
    {{ $product->name }}
@endforeach
```

Utiliza helpers y crea los tuyos propios

Los helpers son funciones globales que trae Laravel ya incorporadas, algunas de estas funciones las utiliza el propio framework en sí.

En esta tabla, te mostramos los tipos de helpers que hay y un ejemplo de cada uno:

Tipos	Definición	Ejemplos
Array	Funcionalidades definidas de arrays	**array_collapse**: que nos permite juntar varios arrays en uno.
Paths	Facilita el trabajo con rutas de la app	**database_path**: Para obtener la ruta del directorio del archivo database
Strings	Trabajar con strings	**str_limit**: Comprueba si una cadena acaba o comienza por un valor dado.
Urls	Abarca pocos helpers pero es realmente importante para el propio framework	contiene funciones como **route()**
Miscellaneous	Los helpers miscellaneous son helpers sin una categoría especifica	Podemos encontrar helpers como **csrf_token()**, para el trabajo de formularios o el de **cookie()**

Encuentra más información sobre los helpers predefinidos aquí: Helpers de Laravel

Para aprender a crear helpers propios, visita nuestro artículo: Helpers personalizados en Laravel

Las llaves en Laravel

A la hora de crear funciones en Laravel, las llaves de apertura se escriben en la línea siguiente y su contenido indentado.

Mal hecho:

```
    public function checkProduct(){
    return $this->checkIsPurchasable() ? $this->isPurchasable() : $this->isNotPurchasable(); }
```

Bien hecho:

```
public function checkProduct()
{
        return $this->checkIsPurchasable() ? $this->isPurchasable() : $this->isNotPurchasable();
}
```

Haz uso de los grupos de rutas para añadirles un middelware

Gracias a los grupos de rutas, podemos añadir un grupo de middleware a varias rutas de una vez.

Ejemplo:

```
Route::group(['middleware' => ['auth']], function ()
{
       Route::get(/, 'ExampleController@index')->name('index');

       Route::get(/users, 'ExampleController@users')->name('users');

       . . .
}
```

Mantener tus proyectos actualizados

Es importante que vayas actualizando tus proyectos a la última versión estable del framework.

Deberías hacer esto regularmente, ya que actualizar nuestro proyecto de versión en versión no supondrá un cambio muy grande, en cambio, si nuestro proyecto está seis versiones por detrás de la actual, podría darnos muchos quebraderos de cabeza.

Utiliza los seeder para añadir datos falsos en desarrollo

Todos los programadores, necesitamos testear nuestra aplicación y de las acciones por excelencia que hacemos para llevar esto acabo es la inserción de datos falsos. El framework cuenta con los seeders que son un modo de crear tantas inserciones de datos como queramos en la tabla de nuestra base de datos en cuestión de segundos, haciendo mucho más productivos nuestros desarrollos.

No crees tablas directamente en el gestor de base de datos (PhpmyAdmin, etc)

Utiliza siempre las migraciones, ya que de esta forma llevaremos un control de nuestras tablas, desde su creación hasta futuras modificaciones de campos.

La lógica de la base de datos en nuestros modelos

La lógica relacionada con la base de datos en nuestros modelos y no tanto en nuestros controladores:

Mal hecho:

```
public function index()
{
    $clients = Client::verified()
        ->with(['orders' => function ($q) {
            $q->where('created_at', '>', Carbon::today()->subWeek());
        }])
        ->get();

    return view('index', ['clients' => $clients]);
}
```

Bien hecho:

```
public function index()
    {
        return view('index', ['clients' => $this->client->getWithNewOrders()]);
    }

    class Client extends Model
    {
        public function getWithNewOrders()
        {
            return $this->verified()
                ->with(['orders' => function ($q) {
                    $q->where('created_at', '>', Carbon::today()->subWeek());
                }])
                ->get();
        }
    }
```

Cuidado con la nomenclatura

Cuidado a la hora de la creación de clases, funciones y especialmente en los modelos, ya que Laravel espera que una tabla de la base de datos llamada "**productos**" esté relacionada con un modelo que se llame "**Producto".**

Utiliza Laravel Mix

Laravel Mix es el sustituto de **Laravel Elixir** y, al contrario de éste, está basado en webpack.

Entre otras, nos permite trabajar con el procesado de los JavaScript y CSS. Podemos juntar todos nuestros archivos css y javascript en uno solo y minificarlo para producción

Ejemplo:

```
.styles([
    'resources/assets/web/css/web.css',
    'resources/assets/web/css/prism.css',
    'resources/assets/web/css/hamburgers.min.css',
    'resources/assets/web/css/responsive.css',
    'resources/assets/web/css/animate.css',
    ], 'public/css/web.css')
```

De esta forma, todos los CSS definidos pasarían a generarse en uno solo llamado **web.css**

Llama a tus rutas mediante sus nombres

Para facilitar el mantenimiento de nuestra aplicación y ahorrarnos un quebradero de cabeza a posteriori, dale nombre a tus rutas y utilízalo para hacer referencia a éstas. De esta forma, si en un futuro cambiamos la url, nuestra aplicación no daría error ya que el nombre sigue siendo el mismo.

Añadiendo nombre "**space**" a la ruta:

```
Route::get('/espacios', 'SpaceController@index)->name('spaces');
```

Llamando a la ruta mediante el nombre:

```
<a href="{{ route('spaces') }}">Espacios</a>
```

Minimiza los complementos

Hay muchos complementos en Laravel que nos permite agregar fácilmente más funcionalidad a nuestras aplicaciones, pero en ocasiones tenemos más complementos de los que utilizamos. Es recomendable eliminar los complementos del compositor que no esté utilizando

Tips para trabajar con Laravel (1,4-6)

Incrementos y decrementos

En eloquent, puedes utilizar increment para incrementar directamente un campo del modelo:

```
$article = Article::find($article_id);
$article->increment('read_count');
```

Metodos XorY

Eloquent, cuenta con funciones que combinan métodos para ahorrarnos líneas de código:

Ejemplos:

findOrFail:

Si queremos que nuestra aplicación retorne un error si no encuentra a un usuario, en vez de hacer esto:

```
$user = User::find($id);
if (!$user) { abort (404); }
```

Podríamos hacer esto:

```
$user = User::findOrFail($id);
```

firstOrCreate:

Si quisiéramos comprobar si un email y en el caso de que no dejar al usuario registrarse, en vez de hacer esto:

```
$user = User::where('email', $email)->first();
if (!$user) {
  User::create([
    'email' => $email
  ]);
}
```

Podríamos hacer esto:

```
$user = User::firstOrCreate(['email' => $email]);
```

Relación con condicionales

Sabemos cómo crear una relación mediante HasMany:

```
public function users() {
    return $this->hasMany('App\User');
}
```

¿Pero sabes que puedes añadir a estas relaciones condiciones como un where?

```
public function approvedUsers() {
    return $this->hasMany('App\User')->where('approved', 1);
}
```

Validaciones de fechas con desde-hasta

En las validaciones con fechas podemos utilizar desde y hasta para validar su valor, como por ejemplo:

```
$rules = [
    'start_date' => 'required|date',
    'end_date' => 'required|date|after:start_date',
]
```

Utilizando el método find con varios id

No solo podemos utilizar el método find con id, sino que con un array podemos buscar tantos como queramos:

```
$users = User::find([1,2,3]);
```

Y de la misma forma en métodos como destroy:

```
User::destroy([1, 2, 3]);
```

Condicional personalizada

Este es uno de los tips que más nos gusta de esta recopilación, normalmente, utilizamos el where en eloquent de esta forma:

```
$users = User::where('activated', 1)->get();
```

Pero también podemos hacerla de esta otra, juntando la cláusula where con el nombre del campo:

```
$users = User::whereActivated(1)->get();
```

Evitar un error de relación

Si tenemos una tabla post que se relaciona con la tabla author, una forma fácil de mostrar el nombre del autor en la plantilla blade sería de esta forma:

```
{{ $post->author->name }}
```

¿Pero y si borramos el autor porque no lo tenemos controlado por algún motivo? Nos daría un error ya que ese autor ya no existe, pero no te preocupes, una forma sencilla de evitar este error sería de esta forma:

```
{{ $post->author->name ?? '' }}
```

Replicar una fila

Con el método replicate, podemos replicar un objeto de la siguiente forma:

```
$post = Posts::find(1);
$newPost = $post->replicate();
$newPost->save();
```

Un solo comando para generar el modelo, la migración y el controlador con funciones por defecto

```
php artisan make:model Post -mcr
```

Anular update_at al guardar

Laravel, por defecto, guardar la última fecha de modificación de la fila en cuestión ¿Pero sabías que podemos anular esta función en el momento que queramos?

```
$user = User::find($id);
$user->updated_at = '2019-01-01 10:00:00';
$user->save(['timestamps' => false]);
```

Utilizando arrays con los Seeders

Podemos utilizar arrays en el método call para mejorar la legibilidad y estructura de nuestros proyectos:

En vez de:

```
class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
        $this->call(UserSeed::class);
        $this->call(PostSeed::class);
        $this->call(CommentsSeed::class);
    }
}
```

Podemos hacer esto:

```
class DatabaseSeeder extends Seeder
{
    private $seeders =[
        UserSeed::class,
        PostSeed::class,
        CommentsSeed::class,
    ];
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
        $this->call($this->$seeders);
    }
}
```

Cambiar el idioma de faker

Ya que hemos entrado en el tema de los seeders ¿Sabías que puedes cambiar el idioma de los datos fake que generes en tu aplicación? Simplemente dirígete app.php en la carpeta config y cambia el valor de la key al idioma que desees, como por ejemplo el Español:

```
    'locale' => 'en',

    //Añadimos esta linea
    'faker_locale' => 'es_Es'
```

Obteniendo el número de filas afectadas

En ocasiones realizamos una actualización pero no sabemos a cuantas filas ha afectado, de esta forma podemos obtener el resultado:

```
$result = $post->whereNull('category_id')->update(['category_id' => 1]);
```

El helper str_limit

Ya sabemos que los helpers son funciones dadas por el propio framework para ahorrarnos trabajo en tareas que solemos realizar en proyectos. Un ejemplo de esto es el helper str_limit que nos da la opción limitar el tamaño del string, mostrando los caracteres que le indiquemos de esta forma y añadiendo al final "…"

```
str_limit('Lorem Ipsum is simply dummy text, 7);
```

Lo que no es tan conocido, es el tercer parámetro que podemos añadirle para que sustituya esos puntos suspensivos:

```
str_limit('Lorem Ipsum is simply dummy text, 7, '-');
```

El helper route

Aprovechando que estamos hablando de helpers, vamos a dar también un tip sobre el helper route, donde mostrará la ruta que le digamos a partir de su nombre, y si esta ruta acepta parámetro podremos pasarle un segundo argumento. Lo que es posible que no sepas es que si queremos pasar un Id en vez de hacerlo así:

```
$post = Post::find(1);

route('nombre_de_la_ruta', $post->id);
```

Bastaría haciéndolo así:

```
route('nombre_de_la_ruta', $post);
```

Obteniendo elementos aleatorios con Eloquent

Podemos utilizar el método **inRandomOrder()** para obtener elementos aleatorios de nuestra base de datos:

```
//Un elemento
$post = Post::inRandomOrder()->first();

//Varios elementos
$post = Post::inRandomOrder()->get();
```

Especificando los campos en el método All

Cuando utilizamos el método **all()** en eloquent, nos devuelve todos los datos del modelo, pero podemos especificar que campos obtener:

```
$post = Post::all(['id', 'title', 'content']);
```

Usando Auth en las plantillas blade

Podemos dinamizar nuestras aplicaciones con Auth, que nos permitirá controlar si un usuario ha iniciado sesión o no, y mostrarle algo que otro usuario que no esté logueado no vería.

```
@if (Auth::check())
   <!--Contenido para el usuario logeado -->
@endif
```

O también:

```
@auth
   <!--Contenido para el usuario logeado -->
@endauth
```

Controlar en que ruta está el usuario

Podemos controlar mediante **Request** si el usuario está en una ruta u otra, esto nos puede servir, por ejemplo, para darle al elemento de la barra de navegación un estilo para que se muestre como seleccionado:

```
<a class="{{ (\Request::route()->getName() == 'web-courses') ? 'active' : '' }}"
href="{{ route('web-courses') }}">
Cursos
</a>
```

Uso de $loop en un foreach

Podemos hacer uso de la variable $loop en nuestro blade para controlar nuestro foreach en su primera y última entrada:

```
@foreach ($posts as $post)
    @if ($loop->first)
        <!--Primera entrada en el bucle -->
    @endif

    @if ($loop->last)
        <!--última entrada en el bucle -->
    @endif
@endforeach
```

Concatenar en blade

Puedes concatenar variables en tu plantilla blade de la misma forma que en tus archivos php:

```
{{ $user->forenames . ' ' . $user->surname }}
```

whereNull y whereNotNull

Utiliza estos métodos para ponerle condicionales de valores nulos a tus consultas.

```
// Post::where('category', null)->get();
Post::whereNull('category')->get();

// Post::where('category', '!=', null)->get();
Post::whereNotNull('category')->get();
```

Utiliza el nuevo asistente de depuración

A partir de la versión 5.7.7 de Laravel, tendremos disponible telescope, el nuevo asistente de depuración de Laravel que nos permite reducir nuestros tiempos de desarrollo gracias a toda la información que nos proporciona mientras desarrollamos nuevos proyectos.

Si quieres saber más sobre telescope, te invitamos a que eches un vistazo a nuestro post sobre éste: Telescope en Laravel

Puedes comprobar si un archivo existe

Podemos comprobar desde blade si un archivo existe, como por ejemplo, cuando queremos mostrar la foto de perfil de un usuario. De esta forma podríamos comprobar antes de importarla si existe y poner una por defecto en el caso que no la tengamos.

```
@if(is_file('/path/to/file.ext'))
   <!-- Acción -->
@else
    <!-- Acción -->
@endif
```

Podemos hacer uso del helper **public_path** en este ejemplo, para facilitarnos la tarea:

```
@if(is_file(public_path('img/uploads/avatars/' . $user->image)))
    <img src="{{ asset('img/uploads/avatars/' . $user->image) }}">
@endif
```

Traduce los mensajes de error

Para traducir los mensajes de error del framework, simplemente debemos ir al archivo app.php, que se encuentra en la carpeta config y modificar la variable locale:

```
'locale' => es,
```

Una vez hecho esto, crearemos una carpeta llamada **es** en resourses/lang y dentro de ésta un archivo llamado **validation.php**.

Por último, copiaremos el código de este repositorio en el archivo **validation.php**

Referencias bibliográficas

1. *Propia.*
2. *AdebsAlert. Laravel : The Right Way (Best Practices) [Internet]. AdebsAlert. 2017 [citado 13 de enero de 2019]. Disponible en: https://medium.com/@adebsalert/laravel-the-right-way-best-practices-2346cd6c5d89*
3. *alexeymezenin. Laravel best practices. Contribute to alexeymezenin/laravel-best-practices development by creating an account on GitHub [Internet]. 2019 [citado 13 de enero de 2019]. Disponible en: https://github.com/alexeymezenin/laravel-best-practices*
4. *20 Laravel Eloquent Tips and Tricks [Internet]. Viblo. [citado 13 de enero de 2019]. Disponible en: https://viblo.asia/p/20-laravel-eloquent-tips-and-tricks-bJzKm03B59N*
5. *Kumar P. 12 Tips for Laravel Performance Optimization [Internet]. The Official Cloudways Blog. 2018 [citado 13 de enero de 2019]. Disponible en: https://www.cloudways.com/blog/laravel-performance-optimization/*
6. *Vallejo C. Los mejores 30 Tips de Laravel [Internet]. Claudio Vallejo. 2018 [citado 13 de enero de 2019]. Disponible en: https://medium.com/@cvallejo/los-mejores-30-tips-de-laravel-d0c96a1c900e*