PEC 3 – DESARROLLO BACKEND CON PHP LARAVEL

Abstract

Desarrollo de web de noticias utilizando el framework de PHP Laravel

Mansori Amar Sara

smansori@uoc.edu

1. Contenidos

1. Contenidos 1
2. Instalación de Laravel de forma local 2
   2.1 Instalación a través de composer 2
   2.2 Comparativa entre Laravel y CodeIgniter o gestores de contenidos 2
3. Migraciones y modelos 2
   3.1 Crear la base de datos 2
   3.2 Creación de migraciones y modelos 2
   3.3 Crear elementos con Tinker 6
4. Generar contenido ficticios 6
   4.1 Creación de los Factory 6
   4.2 Creación de los seeder 7
   4.3 Generación de datos ficticios 8
5. Impementación del frontend 9
6. API REST 12
   6.1 api / noticias / < page > 12
   6.2 api / noticia / < id > 12
   6.3 api / categoria / < id >/ < page > 13
7. Migrar prototipo al servidor de pruebas 13
8. Diferencias entre Laravel y CodeIgniter 14

2. Instalación de Laravel de forma local

2.1 Instalación a través de composer

Tras comprobar que el equipo tiene tanto PHP como Composer instalados, se pueden crear proyectos de Laravel utilizando el CLI de Artisan, e inicializar el servidor local de desarrollo con el comando "serve":

"composer create-project laravel/laravel name-of-the-app"

"php artisan serve" -> Hace que esté disponible en http://localhost:8000

2.2 Comparativa entre Laravel y CodeIgniter o gestores de contenidos

Laravel me ha parecido más parecido a CodeIgniter que a gestores de contenidos como Wordpress o Laravel, ya que no es tan opinionado como estos y permite una mayor capacidad de customización (además de que ambos siguen el Modelo Vista Controlador).

Por un lado, la parte positiva de Laravel es este nivel de customización o capacidad de desarrollar al gusto del programador, aunque por otro lado también se hace más complejo debido a que debe hacerse por linea de comandos y requiere cierto conocimiento de programación.

En cuanto a las diferencias entre Laravel y CodeIgniter en términos de su instalación, me ha sorprendido la cantidad de opciones disponibles en Laravel, pudiendo instalarse de forma local o dockerizada, y habiendo diferentes maneras de hacerlo dentro de cada una de las opciones mencionadas, lo que me hace considerar que Laravel se pueda adaptar mejor a las necesidades de cada desarrollador.

3. Migraciones y modelos

3.1 Crear la base de datos

En primer lugar deberemos crear la base de datos (en nuestro caso se llamará news) en la que posteriormente añadiremos las tablas, filas y columnas necesarias. Para eso, podemos hacerlo desde la interfaz gráfica o inicializando desde la terminal SQL y usando los siguientes comandos:

- mysql -uroot -p
- create database news

3.2 Creación de migraciones y modelos

Una vez la base de datos esté creada y corroboremos que tiene el mismo nombre que la base de datos referenciadas en el archivo de entorno (.env) de nuestro proyecto de Laravel, necesitaremos crear tres migraciones y dos modelos para el ejercicio propuesto: modelo Article con migración create_article_table, modelo Category con migración create_category_table y migración para crear una tabla que albergue las relaciones many to many entre Category y Article.

Para crear estas migraciones en Laravel, podemos ejecutar los comandos del php artisan por separado, o crear un modelo y una migración relacionada usando un flag.

- php artisan make:model Article -m

- php artisan make:model Article -m
- php artisan make:migration create_article_category_table

Tras crear estos archivos, derebemos realizar ciertas modificaciones para que reflejen los campos y valores que queremos que tenga nuestra base de datos.

En las migraciones (que estarán en el directorio “database/migrations”) añadiremos en la función que se le pasa como segundo parámetro al método create() del Schema, todas las columnas que queremos que tenga nuestra tabla, indicando el tipo de columna en base a los datos que va a albergar (basándonos en los tipos de columna de Laravel).

```php
<?php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('categories', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('slug');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('categories');
    }
};
```

Figure 1. "create-categories-table"

```php
<?php       "sara.mansori", 18 hours ago • first commit …

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('articles', function (Blueprint $table) {
            $table->id();
            $table->string('title');
            $table->text('author');
            $table->string('image');
            $table->timestamp('published_at')->nullable();
            $table->text('content');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('articles');
    }
};
```

Figure 2. "create-articles-table"

```php
return new class extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('article_category', function (Blueprint $table) {
            $table->id();

            $table->unsignedBigInteger('category_id');
            $table->unsignedBigInteger('article_id');

            $table->foreign('category_id')->references('id')->on('categories')->onDelete('cascade');
            $table->foreign('article_id')->references('id')->on('articles')->onDelete('cascade');

            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('article_category');
    }
};       "sara.mansori", 18 hours ago • first commit …
```

Figure 3. "create-article-category-table"

En los modelos (que están en el directorio “App/Models/”, teniendo en cuenta que la relación entre los modelos es de many to many (un artículo puede tener muchas categorías y una categoría pertenecer a muchos artículos), desarrollaremos un método para cada modelo que nos permita acceder a las categorías desde el modelo Article y a los artículos que contengan la categoría del modelo Category.

También podemos introducir una variable llamada $fillable que indica qué valores pueden ser asignados a través de lo que se conoce como “mass assignment” o asignación masiva, debido a que esta forma de asignar valores puede presentar vulnerabilidades, esto previene que se asignen campos que no está previsto que cambien).

```php
<?php
namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Article extends Model
{
    use HasFactory;

    protected $fillable = [
        'title', 'author', 'content'
    ];

    public function categories()
    {
        return $this->belongsToMany('App\Models\Category');
    }
}
```

Figure 4. Model\Article

```php
<?php
namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Category extends Model
{
    use HasFactory;

    public function articles()
    {
        return $this->belongsToMany('App\Models\Article');
    }
}
```

Figure 5. Model\Category

Para enlazar la base de datos creadas con las migraciones y modelos creados podemos utilizar el comando:

- php artisan migrate

Y eso creara en nuestra base de datos todas las columnas con los valores definidos en las migraciones.

3.3 Crear elementos con Tinker

Para comprobar que nuestros modelosy migraciones están correctamente creados, podemos utilizar una herramienta llamada Tinker que nos permite acceder al modelo y crear elementos a través de la terminal.

Para ello tendremos que introducir el comando: php artisan tinker

Una vez que estemos dentro del modo de tinker podemos utilizar comandos como los siguientes para popular nuestra base de datos:

- $article = new App\Models\Article; (guardamos una instancia del modelo Article)
- $article->title='Título de la noticia'; (le asignamos los diferentes valores a las columnas declaradas en la migración)
- $article->author='Anónimo';
- $article->content='Lorem ipsum dolor sit amet.
- $article->save()' (guardamos la fila en la base de datos)

En caso de que quisiéramos asignarle alguna categoría, tenemos que tener en cuenta la existencia de la pivot table que relaciona los Article con las Category. Para ello haríamos lo siguiente (a continuación del comando anterior o seleccionando el artículo de la base de datos al que le queremos asignar una categoría con por ejemplo Article::find(id):

- $article->categories()->attach(category_id) -> si queremos asignarle una categoría
- $article->categories()->attach([category1_id, category2_id…]) -> si queremos asignarle varias categorías a la vez

4. Generar contenido ficticios

Para generar contenidos ficticios se usará un Factory y Faker (que ya viene instalado con Laravel). En este caso, se creará una Factory para Article puesto que será el modelo del que se necesiten crear contenidos ficticios, mientras que la creación de los elementos del modelo Category se realizará directamente desde el fichero del seed.

4.1 Creación de los Factory

Podemos crear un archivo Factory utilizando el comando de artisan:

- Php artisan make:factory Article

Una vez creado (se guardará en el directorio “database/factories”, deberemos modificarlo de la siguiente forma para indicar que elementos queremos que se creen y de qué forma cuando se llame a la Factory:

```php
<?php       'sara.mansori', 18 hours ago • first commit

namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;

/**
 * @extends \Illuminate\Database\Eloquent\Factories\Factory<\App\Models\Article>
 */

class ArticleFactory extends Factory
{
    /**
     * Define the model's default state.
     *
     * @return array<string, mixed>
     */

    public function definition()

    {
        $randomNumber = rand(0, 100);
        return [
            'title' => $this->faker->sentence($nbWords = 6, $variableNbWords = true),
            'author' => $this->faker->name(),
            'content' => '<p>' . $this->faker->text() . '</p>',
            'image' => "https://picsum.photos/600/300?random=$randomNumber",
            'published_at' => $this->faker->dateTime()
        ];
    }
}
```

Figure 6. ArticleFactory

Como se puede observara en el código, se ha usado la dependencia faker para generar contenidos ficticios, a excepción de la imagen, puesto que parece que la funcionalidad de faker de imágenes ya no está operativa y se ha optado por usar como alternativa la página de picsum, que genera imágenes aleatorias cada vez que se accede a ella. Para que las imágenes de los artículos no fuesen la misma cada vez que se cargaba la página, se guardan en la base de datos cada una con un número aleatorio del 0 al 100.

4.2 Creación de los seeder

Ahora que tenemos los Factory se puede proceder a desarrollar los archivos de seed para sembrar nuestra base de datos.

Dentro del directorio “database/seeders” podemos ver un fichero: “DatabaseSeeder.php”. Este es el fichero que tendremos que modificar para indicarle cómo queremos popular nuestra base de datos cada vez que se ejecute el comando seed.

```php
class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {

        Category::create([
            'name' => 'Economy',
            'slug' => 'economy'
        ]);

        Category::create([
            'name' => 'Sports',
            'slug' => 'sports'
        ]);

        Category::create([
            'name' => 'Lifestyle',
            'slug' => 'lifestyle'
        ]);

        $articles = Article::factory(100)->create();
        $categories = Category::all();

        // Seed Categories

        $articles->each(function ($article) use ($categories) {
            $article->categories()->attach(
                $categories->random(rand(1, 3))->pluck('id')->toArray()
            );
        });
    }
}
```

Figure 7. DatabaseSeeder.php

En este archivo, dentro del método run(), introduciremos la creación de las diferentes categorías que queramos (similar a cómo lo hicimos anteriormente con Tinker).

Para los artículos, sin embargo, utilizaremos la Factory creada, llamando al modelo Article, y al método factory, pasándole como parámetro un integer con la cantidad de entradas que queremos generar e indicándole que los cree con el método create().

También queremos asignar de manera aleatoria una o varias categorías a cada artículo creado, así que una vez se han almacenado todos los artículos creados y las categorías en las variables correspondientes $articles y $categories, iteramos por cada uno de los artículos y llamamos al método categories() para a su vez llamar al método attach() y asignarle el id de categorías aleatorias.

4.3 Generación de datos ficticios

Tras esto, para generar los datos ficticios nos dirigimos a la terminal, desde donde podemos usar el siguiente comando para realizar de nuevo las migraciones necesarias, además de limpiar las anteriores y hacer seed de nuestro archivo DatabaseSeeder.php para generar los contenidos ahí indicados:

- php artisan migrate:fresh –seed

5. Impementación del frontend

Para desarrollar el frontend de la aplicación se tendrán que desarrollar dos cosas principalmente: las rutas y las vistas.

Las rutas las podemos encontrar dentro del directorio “routes”, y estas en concreto se configurarán dentro del archivo “web.php”.

- **Listado de noticias con su título, imagen y fecha:**

  Es la página principal a la que se accede nada más acceder a la página. Esto significa que le corresponde la ruta “/”.

```php
Route::get('/', function () {
    return view('articles', [
        'articles' => Article::all()->take(5)
    ]);
});
```

Figure 8. web.php

La configuramos indicándole que cuando se acceda a “/”, se renderizará la vista “articles” (que todavía no hemos creado) y que le pasará “articles”, que contendrá los primeros 5 artículos de la base de datos.

Para desarrollar la vista de articles, crearemos un fichero llamado “articles.blade.php” dentro del directorio “resources/views” (Laravel asocia automáticamente el nombre de la vista al fichero con extensión .blade.php)

Como la página va a tener un “title” y el css se va a compartir entre todos, se ha creado también un directorio “components” en el que se guarda un fichero llamado “layout.blade.php”, que contiene el layout con estos elementos (y $slot guardará todo el contenido que se ponga dentro de la referencia a este layout que será a través de las etiquetas <x-layout></x-layout>):

```php
<!doctype html>

<title>News site</title>
<link rel="stylesheet" href="/app.css">

<body>

    {{ $slot }}

</body>
```

Figure 9. layout.blade.php

```blade
<x-layout>

    @foreach ($articles as $article)

    <article>

        <img src="{{$article->image}}">

        <h1>
            <a href="/articles/{{ $article->id }}">
                {{ $article->title }}
            </a>
        </h1>

        <p>
            Published at:

            <span>{{ $article->published_at }}</span>

        </p>

    </article>

    @endforeach


</x-layout>
```

Figure 10. articles.blade.php

En esta vista iteraremos por todos los artículos pasados a la vista, renderizando para cada uno de ellos, e incluyendo un anchor tag que lleve a la ruta que configuraremos a continuación, la cual renderizará la página de detalle del artículo.

- **Página única de noticias con sus campos:**

  Similar a la página anterior, tendremos que configurar la ruta. En este accedera cada vez que se acceda a la routa “/articles/id”, donde el id será el id de la noticia a mostrar.

  En este caso, la función recibe como parámetro Article $article, porque Laravel busca dentro de los elementos del modelo Article, aquel que coincida con el id del parámetro de la ruta con el mismo nombre (solo funciona si se llama igual y es el id), guardándolo en la variable $article.

  Una vez se acceda a esta routa, se renderizará la vista ‘article’ (que aún tenemos que crear), pasándole el $article.

```php
Route::get('/articles/{article}', function (Article $article) {

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

  La vista la crearemos en el mismo lugar que la anterior, dentro del directorio “resources/views”, creando por lo tanto un fichero llamado “article.blade.php”:

```blade
<x-layout>

    <article>

        <img src="{{$article->image}}">

        <h1>
            {{ $article->title }}
        </h1>

        <p>
            By {{ $article->author }}
        </p>

        <p>
            Categories:

            @foreach ($article->categories as $category)
            <a href="/categories/{{ $category->slug }}">{{ $category->name }}</a> |
            @endforeach

        </p>

        <div>
            {!! $article->content !!}
        </div>
    </article>

    <a href="/">Go back</a>

</x-layout>
```

Figure 11. article.blade.php

Aquí pondremos la información del article que querramos mostrar en la vista, iterando dentro de las categorias que tenemos disponibles gracias al método desarrollado en el modelo para poder mostrarlas.

Cada una de estas categorías a su vez tendrá un anchor tag que llevará a la ruta que configuraremos en el siguiente paso:

- **Página de categoría:**

  En este caso también desarrollaremos la ruta como en los casos anteriores, pero en este caso como la ruta va a incluir la propiedad “slug” de la categoría y no el id, se lo debemos indicar o si no no funcionaría la asignación de Category $category.

```php
Route::get('/categories/{category:slug}', function (Category $category) {

    return view('articles', [
        'articles' => $category->articles->take(5)
    ]);
});
```

En este caso, cuando se acceda a la ruta “/articles/category_slug”, se renderizará la vista ya creada “articles”, pero en vez de mostrar los cinco artículos de la ruta configurada en el primer paso, se mostrarán solo los primeros cinco artículos que contengan la categoría que tenga un id que coincida con el pasado como parámetro.

6. API REST

Para desarrollar las rutas de la api, tendremos que crear controllers para cada uno de los modelos cuya información nos interese mostrar a través de la misma, en este caso Article y Category.

Para crear los controllers, podemos ejecutar los comando:

- "php artisan make:controller Api\\ArticleController --model=Article"
- "php artisan make:controller Api\\CategoryController --model=Category"

6.1 api / noticias / < page >

Configuraremos las rutas de la api dentro del directorio "routes", en el ficher "api.php":

```
Route::apiResource('noticias/{page?}/', ArticleController::class);
```

Y a su vez tendremos que configurar el comportamiento del método básico (index()) del controller de la siguiente manera, para poder controlar la paginación y devolver un JSON con la información que se desea mostrar:

```
public function index($page = 1)
{
    $paginate = 10;
    $skip = ($page * $paginate) - $paginate;
    $prevUrl = $nextUrl = '';
    if ($skip > 0) {
        $prevUrl = $page - 1;
    }
    $articles = Article::orderBy('id', 'asc')->skip($skip)->take($paginate)->get();
    if ($articles->count() > 0) {

        if ($articles->count() >= $paginate) {
            $nextUrl = $page + 1;
        }

        return response()->json([
            'status' => true,
            'message' => 'Article List',
            'articles' => $articles,
            'prevUrl' => "/api/noticias/$prevUrl",
            'nextUrl' => "/api/noticias/$nextUrl"
        ]);
    }

    return redirect('/api/noticias/page/1');
}
```

6.2 api / noticia / < id >

```
Route::apiResource('noticia', ArticleController::class);
```

En este caso, Laravel detecta el parámetro que hay detrás de /noticia como el id, y llama al método del controller show(), que es el que deberemos de modificar para que busque el artículo cuyo id coincida con el id pasado como parámetro.

```php
public function show($id)
{

    $article = Article::find($id);

    if (is_null($article)) {
        return $this->sendError('Article not found');
    }
    return response()->json([
        "success" => true,
        "message" => "Article retrieved succesfully",
        "data" => $article
    ]);
}
```

6.3 api / categoria / < id >/ < page >

```php
Route::apiResource('categoria/{id}/{page?}', CategoryController::class, ['only' => ['index']]);
```

En este caso tenemos dos parámetros: id (que representa el id de la categoría) y page (que representa la paginación).

En este caso se llama también al controller, pero solo se tiene en cuenta el método index(), puesto que no se van a mostrar páginas de detalle con show(). Por tanto solo habrá que actualizar el método index() tendro del controller de Category de forma similar a como lo hicimos con el controller de Article.

```php
public function index($id, $page = 1)
{
    $paginate = 10;
    $skip = ($page * $paginate) - $paginate;
    $prevUrl = $nextUrl = '';
    if ($skip > 0) {
        $prevUrl = $page - 1;
    }
    $articles = Category::find($id)->articles()->orderBy('id', 'asc')->skip($skip)->take($paginate)->get();
    if ($articles->count() > 0) {

        if ($articles->count() >= $paginate) {
            $nextUrl = $page + 1;
        }

        return response()->json([
            'status' => true,
            'message' => 'Article List',
            'articles' => $articles,
            'prevUrl' => "/api/categoria/$id/$prevUrl",
            'nextUrl' => "/api/categoria/$id/$nextUrl"
        ]);
    }

    return redirect("/api/categoria/$id/1");
}
```

7. Migrar prototipo al servidor de pruebas

En primer lugar tendremos que exportar la base de datos a un fichero comprimido en formato .zip.sql, y subirlo a la base de datos en servidor.

Tras esto, se comprime también el proyecto de Laravel y se sube al servidor utilizando FileZilla, y descomprimiéndolo accediendo al servidor por la terminal y usando el comando unzip.

Una vez hecho esto, se modifica el archivo .env para cambiar el username, database y la password para que coincidan con la base de datos remota.

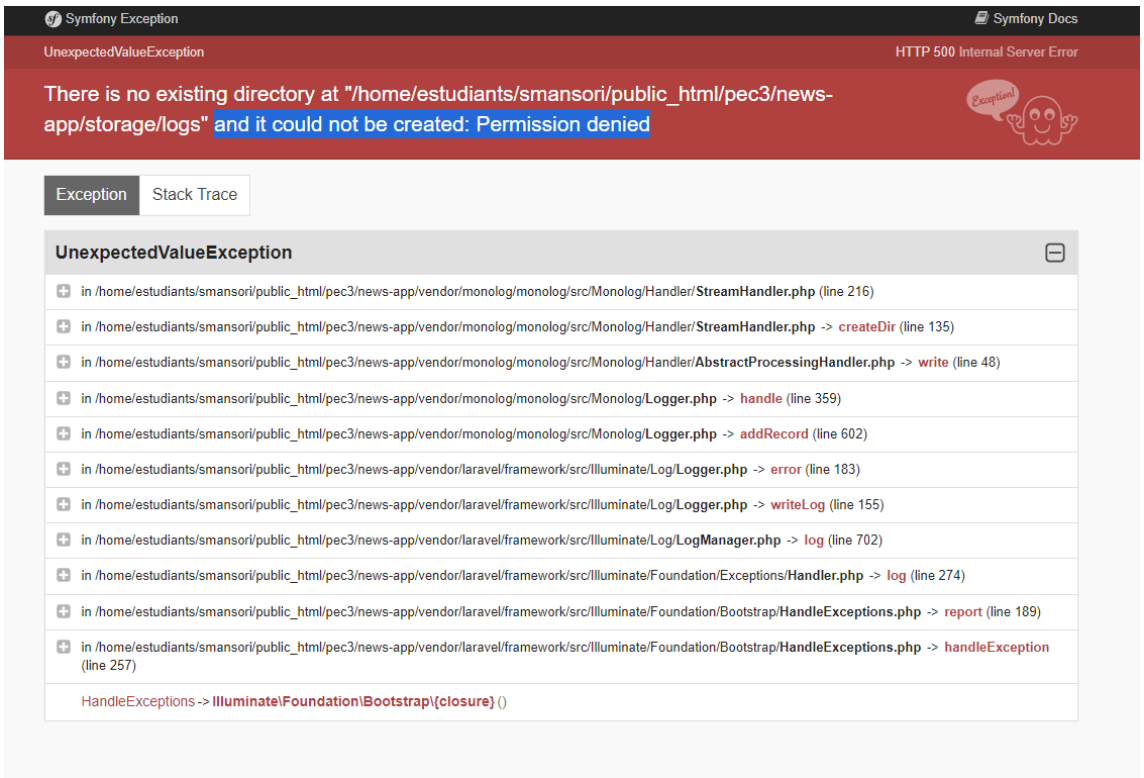
Sin embargo en este punto, a la hora de intentar acceder al sitio web, Symfony me arrojaba una Excepción que tras investigar parece ocurrir por un problema de permisos que pude solucionar cambiando los permisos de las carpetas storage y cache.

URLS WEB:

- https://eimtcms2.uoclabs.uoc.es/~smansori/pec3/news-app/public/
- https://eimtcms2.uoclabs.uoc.es/~smansori/pec3/news-app/public/articles/1
- https://eimtcms2.uoclabs.uoc.es/~smansori/pec3/news-app/public/categories/economy

URLS API

- https://eimtcms2.uoclabs.uoc.es/~smansori/pec3/news-app/public/api/noticias
- https://eimtcms2.uoclabs.uoc.es/~smansori/pec3/news-app/public/api/noticias/1
- https://eimtcms2.uoclabs.uoc.es/~smansori/pec3/news-app/public/api/noticia/1
- https://eimtcms2.uoclabs.uoc.es/~smansori/pec3/news-app/public/api/categoria/1/2



8. Diferencias entre Laravel y CodeIgniter

Al trabajar con ambos frameworks me he familiarizado con el tipo MVC (Modelo Vista Controlador) que utilizan ambos, lo cual ha hecho que el salto de uno a otro no haya sido tan grande.

En primer lugar me he dado cuenta buscando los errores o problemas que me iba encontrando, que para Laravel hay más documentación (o al menos más accesible) y más respuestas por parte de la comunidad que en CodeIgniter, lo que hacía que la resolución de dudas o el aprendizaje fuese más sencillo.

En cuanto a temás más técnicos, el uso de Eloquent, Migrations y el Seed de Laravel me han parecido herramientas muy potentes para manipular la base de datos mediante el código del programa.

Además, el modelo de plantillas (blade) me ha parecido que simplifica y hace más sencillo el desarrollo de las vistas, reduciendo la complejidad y poniendo a disposición del desarrollador azúcar sintáctico que permite que el desarrollo con php dentro del lenguaje de mercado sea más legible.

También es cierto que la instalación de Laravel es algo más compleja que la de CodeIgniter, por necesitar forzosamente el uso de composer, pero lo suple con las herramientas que ofrece a través del “artisan” y los numerosos métodos que facilitan el desarrollo haciendo que la experiencia de desarrollo sea en mi opinión superior a la de CodeIgniter.