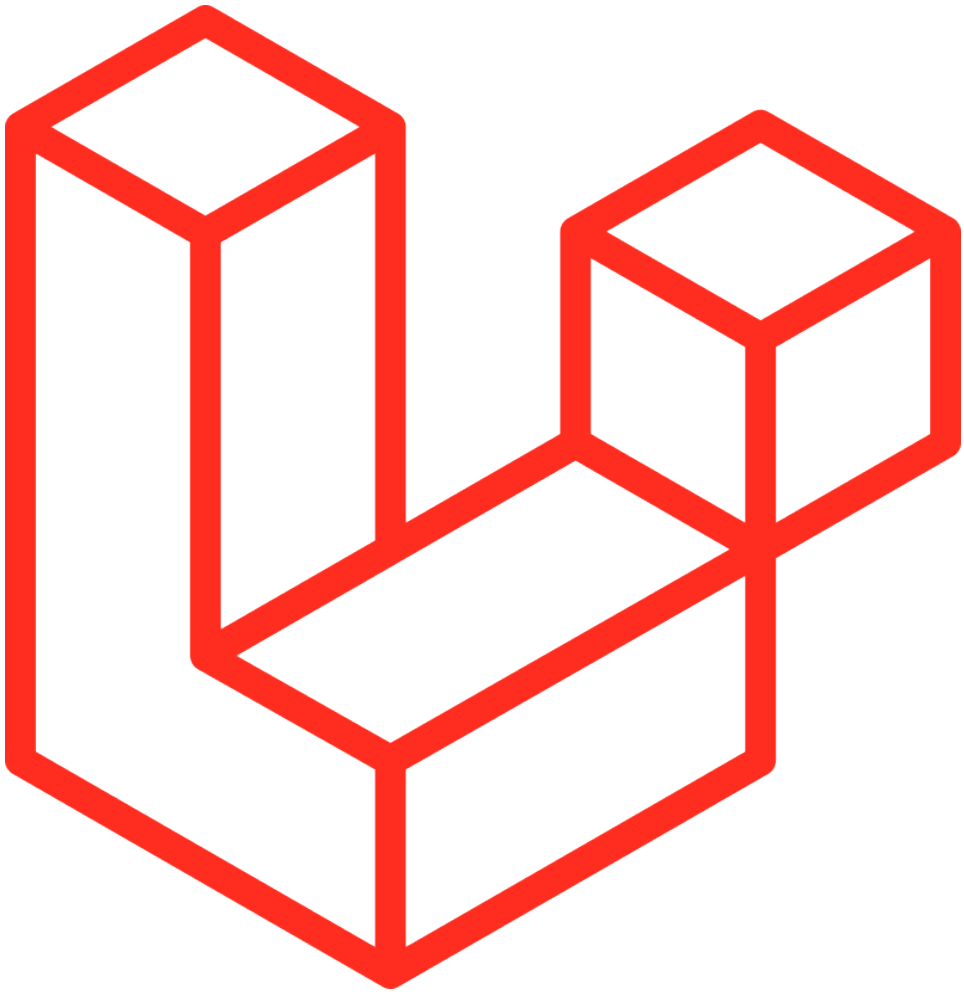


Introducción a Laravel

Rubén Gómez Olivencia

2024-10-29

- Github: https://github.com/yuki

Puedes encontrar la última versión de este libro en formato **HTML** en el siguiente link, así como otros libros que he creado. Para descargar el código fuente en formato **Markdown** visita el repositorio en GitHub.

Información

Por favor, ponte en contacto conmigo si encuentras algún fallo, falta de ortografía o quieres mejorar de alguna manera este libro. Gracias.

I Introducción

II Crear entorno Laravel

III Funcionalidad básica

Introducción

1. Introducción a Laravel

Laravel es un *framework* para crear aplicaciones y servicios web haciendo uso del lenguaje de programación PHP, buscando la simplicidad y evitar el "*spaghetti code*". Hace uso de la arquitectura "modelo-vista-controlador" (MVS) y es un proyecto de código abierto.

1.1. Características

Entre las características que tiene Laravel, se pueden destacar:

- Sistema de enrutamiento, también RESTful.
- Motor de plantillas web llamado Blade. Nos permite:
  - Crear plantillas que pueden incluir otras plantillas.
  - Hacer uso de PHP dentro de las plantillas.
  - Permite cachear las plantillas hasta que se modifiquen.
- Creador de queries a la base de datos llamada Fluent.
- Eloquent como ORM (*object-relational mapper*).
- Uso de "*migrations*" para crear la base de datos a modo de sistema de control de versiones.
- Sistema de enrutado de la aplicación para relacionar rutas de acceso con controladores.
- Posibilidad de usar "semillas" (en inglés "*seeds*") en la base de datos para importar datos, ya sea de test o datos iniciales necesarios.
- Permite hacer uso de paquetes de Composer.
- Soporte para usar servicios de caché.
- Posibilidad de paginación automática.

Estas características las iremos utilizando para crear nuestro primer proyecto y para posteriormente aprender a crear una API que podrá ser accedida desde cualquier tipo de aplicación: un interfaz web, una aplicación móvil, desde línea de comandos...

2. Modelo-Vista-Controlador

La arquitectura **Modelo-Vista-Controlador** es un patrón de diseño que separa las funciones que el software realiza en tres capas principales:

- **Modelo de datos**: Es la representación de la información que la que la aplicación interactúa, tanto para obtener la información como para ser actualizada.

  El modelo de datos normalmente equivale al diseño de base de datos, donde cada modelo representa a una entidad en un diseño de base de datos relacional.

El controlador es el encargado de realizar las peticiones al modelo, ya sean actualizaciones o la obtención de información.

- **Controlador**: Responde a acciones del usuario (o eventos), que normalmente desencadenan en una acción al modelo de datos (ya sea obtención de datos, actualización, borrado...).

  El controlador hace de intermediario entre la vista y el modelo.

- **Vista**: Es la parte que muestra al usuario los datos obtenidos y con la que este interactúa. Esta interacción generará posibles acciones que irán al controlador para volver a empezar el ciclo.

2.1. Interacción de los componentes

Aunque existen distintas implementaciones de la arquitectura Modelo-Vista-Controlador, el flujo de acciones suele ser similar al siguiente:

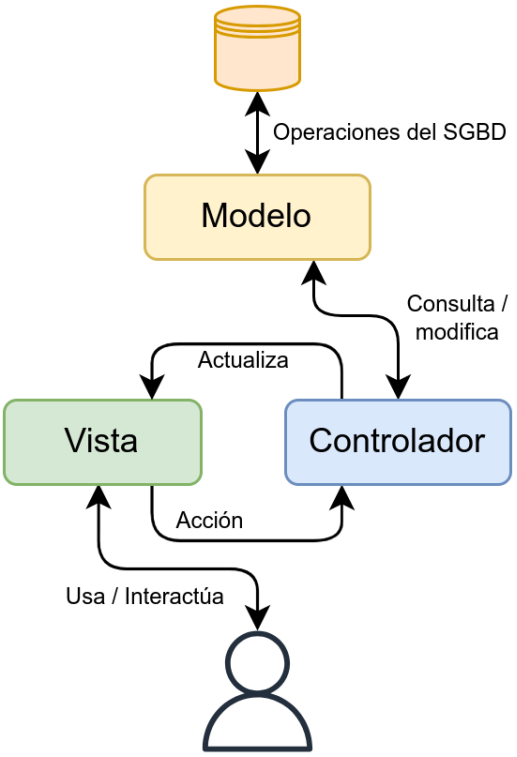
1. El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, el usuario pulsa un botón, enlace, etc.).

2. El controlador recibe (por parte de los objetos de la interfaz-vista) la notificación de la acción solicitada por el usuario. El controlador gestiona el evento que llega, frecuentemente a través de un gestor de eventos (*handler*) o *callback*.

3. El controlador realiza una petición al modelo, ya sea para solicitar información o para actualizarla. El modelo debe confirmar si la acción se ha realizado de manera correcta o no.

4. El controlador delega en la vista la información obtenida para que sea visualizada.

5. La interfaz se mantiene a la espera de una nueva interacción para comenzar de nuevo el ciclo.



Tal como se ha dicho, pueden existir distintas implementaciones, pero de manera generalizada y simplificada este sería el esquema básico de interacción.

3. ORM

Los sistemas ORM (del inglés *Object-Relational mapping*, o “mapeo objeto-relacional”) es una técnica de programación para convertir los datos de una base de datos relacional en objetos cuando son utilizados en un lenguaje de programación orientado a objetos.

Este sistema “simula” una base de datos orientada a objetos (que no están muy extendidas) haciendo uso de las bases de datos relacionales que son ampliamente utilizadas y son más conocidas. De esta manera, también es un sistema menos a mantener.

Estos sistemas ORM suelen ocultar cómo se generan las peticiones a la base de datos, ya que para el programador lo único que hace es interactuar con objetos. El motor que usa Laravel se llama Eloquent.

Muchos *frameworks* de programación hacen uso de sistemas ORM, por lo que dependerá de cuál usemos podremos hacer uso de uno o podremos elegir entre varios. La wikipedia tiene una página donde se listan distintos ORMs separados por lenguajes de programación.

Crear entorno Laravel

1. Crear primer proyecto en Laravel

A la hora de crear un proyecto en Laravel lo primero que deberíamos hacer es visitar la documentación, ya que nos dará distintas opciones dependiendo del sistema operativo en el que nos encontremos. Aparte, podremos ver si ha habido cambios desde la última vez que hayamos creado un proyecto.

1.1. Servicios a utilizar

Antes de crear el proyecto, debemos tomar una serie de decisiones para nuestro *stack* de aplicación. Laravel cuenta con distintos servicios, algunos de ellos necesarios y otros optativos, por lo que deberemos tenerlos en cuenta.

Los servicios entre los que deberemos decidir son:

- **Sistema Gestor de Base de Datos a utilizar**: Laravel permite el uso de distintos sistemas de bases de datos relacionales como son MySQL, PostgreSQL y MariaDB. Por defecto hace uso de **MySQL**.
- **Sistema de caché**: Podemos hacer uso de distintos sistemas para cachear desde la sesión a información obtenida de la base de datos y también HTML. Por defecto, **Laravel cachea la sesión en el sistema de ficheros**, pero eso puede ser lento, por lo que se permite hacer uso de sistemas **clave-valor** para el almacenamiento de información para acelerar el rendimiento de la aplicación web. Se puede elegir Memcached o Redis entre otros.

Otros servicios que podemos instalar y que nos darán ciertas funcionalidades son:

- **Mailpit**: Es un sistema para controlar los emails que envía nuestra aplicación durante el desarrollo. En lugar de enviarlos a las cuentas finales, se quedan almacenados y se pueden visualizar a través de una web que a modo de buzón de correo. También ofrece una API.
- Uso de MinIO para simular el **almacenamiento en la nube S3**. De esta manera no tendremos que crear un Bucket de pruebas.
- Sistema de **búsqueda *full-text*** en la base de datos gracias a Scout y haciendo uso del backend MeiliSearch.
- Creación y automatización de **tests** utilizando Selenium.

Estas son algunas de los servicios que podríamos configurar antes de comenzar a crear nuestra aplicación. Para comenzar de manera sencilla nos centraremos únicamente en la elección de la base de datos, dejando el resto de servicios para más adelante.

1.2. Instalación mediante Sail y Docker

En la documentación de Laravel nos explica cómo realizar la instalación de distintos modos teniendo en cuenta el sistema operativo, los servicios iniciales que nos interesan y el sistema de instalación que mejor se adapte a nuestro entorno.

El sistema es similar utilizando GNU/Linux, Windows y MacOS, con la salvedad de que en Windows

deberíamos instalar Docker Desktop y *Windows Subsystem for Linux* (WSL). De manera generalizada, es necesario tener instalado:

- Entorno GNU/Linux
- Docker y Docker Compose (ambos integrados en **Docker Desktop**)

Para realizar la instalación sólo vamos a elegir tener el servicio de MySQL, para simplificarlo, tal como se ha comentado previamente. Para ello, deberemos ejecutar lo siguiente en el directorio donde nos interese crear el directorio del proyecto.

Usamos el instalador de Laravel

```
ruben@vega:~$ curl -s "https://laravel.build/example-app?with=mysql" | bash
```

Este comando lo que va a hacer es descargarse un script que va a ejecutar lo siguiente:

- Se va a asegurar que Docker está corriendo
- Va a levantar un contenedor con la imagen "laravelsail/php82-composer" que nos va a crear un directorio llamado `example-app` con un proyecto limpio de Laravel usando MySQL como SGBD.
- Si no tenemos la imagen de MySQL la descarga.

1.2.1. Ventajas

Este sistema de instalación permite realizar un despliegue sencillo en un equipo donde tengamos instalado Docker, con todas las ventajas que ello ofrece, aparte de la posibilidad de elegir los servicios que necesitemos inicialmente.

Podríamos resumir las ventajas en la siguiente lista:

- Instalación rápida con un único comando.
- Ventajas de usar Docker: todos los desarrolladores usan el mismo contenedor/entorno de desarrollo.
- No es necesario tener nada más que Docker instalado en el equipo anfitrión (ni PHP, composer, servicios web, …).

1.2.2. Desventajas

Aunque las ventajas durante el desarrollo de aplicaciones son notables, también pueden existir algunas desventajas:

- El servidor web que se arranca por defecto no es el más recomendado para despliegues en producción, obteniendo mejor rendimiento con el servidor web Nginx.
- En un principio puede resultar "raro" desarrollar dentro de un contenedor.

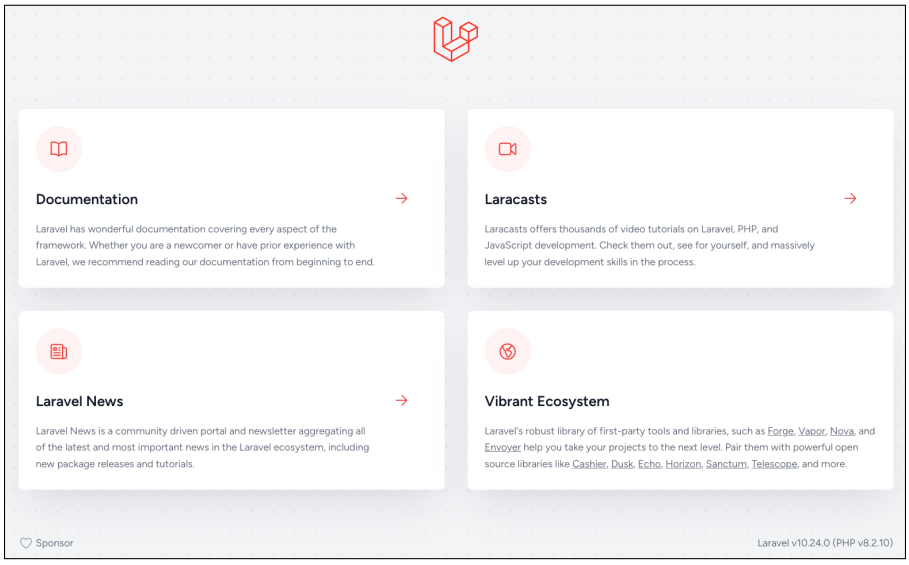
1.3. Iniciar servicios

Una vez terminada la descarga del código y tras realizar las acciones que necesita, el propio asistente nos avisa de qué tenemos que realizar para que nuestro entorno arranque.

Arrancamos los servicios

```
ruben@vega:~$ cd example-app && ./vendor/bin/sail up -d
[+] Running 2/2
  Container example-app-mysql-1          Started    0.4s
  Container example-app-laravel.test-1   Started    0.7s
```

Y con ello podemos ir al puerto 80 a través de nuestro navegador y veremos la página principal para comprobar que todo ha ido bien.



2. Variables de entorno

Todo proyecto de Laravel cuenta con unas variables de entorno del proyecto. Es un fichero de configuración situado en la raíz del proyecto que se llama `.env` y en él se encuentran las credenciales para acceder a la base de datos, servidor SMTP, ...

Dado que hemos generado el entorno a través del asistente, se ha rellenado con la configuración por defecto, entre las que nos encontramos que la aplicación tiene **el modo debug activado**. Durante el desarrollo nos va a ayudar para poder hacer *debugging* mientras realizamos acciones, pero esta opción debería estar deshabilitada al poner la aplicación en producción.

Por último, es conveniente recordar que este fichero **nunca debería estar en un repositorio público**, ya que contiene información sensible como lo son los credenciales de acceso a bases de datos o servicios externos.

¡Cuidado!

Cuidado con versionar el fichero ".env", ya que contiene información sensible

3. Usar Visual Studio Code con Laravel

Visual Studio Code es un entorno de desarrollo integrado (IDE) desarrollado por Microsoft y con licencia MIT, lo que lo hace Software Libre y que cualquiera pueda ver el código fuente, así como realizar modificaciones.

El problema es que Microsoft no ha liberado todo el código fuente, y los binarios que ofrece para descargar hacen uso de ese software, así como la inclusión de sistemas de telemetría. Es por eso que existe un proyecto llamado VSCodium que ofrece los binarios libres de ese código.

Entre las ventajas que ofrece este IDE podemos destacar:

- Se puede programar para muchos lenguajes de programación, no está especializado en uno sólo.
- Es extensible mediante *plugins*. Hoy en día existen infinidad de extensiones para todo tipo de desarrollos.
- Es multiplataforma.
- Altamente configurable.
- Configurando la cuenta de GitHub, se puede sincronizar las configuraciones entre distintos dispositivos.
- Existe una versión online.

3.1. Extensiones recomendadas

Para desarrollar con Laravel, aunque se puede extender a cualquier proyecto que haga uso de un entorno Docker, es recomendable utilizar una serie de extensiones para facilitar el desarrollo con ellos. De todas maneras, Visual Studio Code nos va a recomendar extensiones a medida que lo usemos, ya que observará el tipo de desarrollo que estamos realizando.

Entre las extensiones que se recomiendan están:

- Remote Development: Nos instala un grupo de extensiones para poder trabajar contra un servidor remoto.
- Laravel Extension Pack: Es una "meta-extensión", ya que incluye a otras extensiones creadas especialmente para ayudar durante el desarrollo de Laravel.
- PHP Extension Pack: Es un conjunto de extensiones que nos va a permitir trabajar de manera más cómoda durante el desarrollo de código PHP.

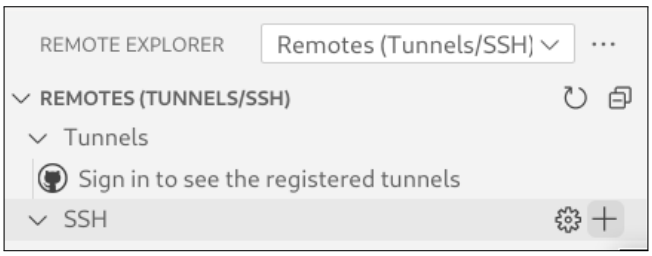
- Auto Close Tag: Muy útil durante el desarrollo de HTML, ya que cuando creamos una etiqueta, automáticamente nos crea la etiqueta de cerrado.
- VSCode icons: Aunque no es una extensión que nos ayude a programar, si que nos ayuda a identificar distintos ficheros, ya que añade iconos extra a nuestro entorno de trabajo.

Existe una infinidad de extensiones que nos pueden ayudar durante el desarrollo,

3.2. Desarrollo con conexión SSH a servidor remoto

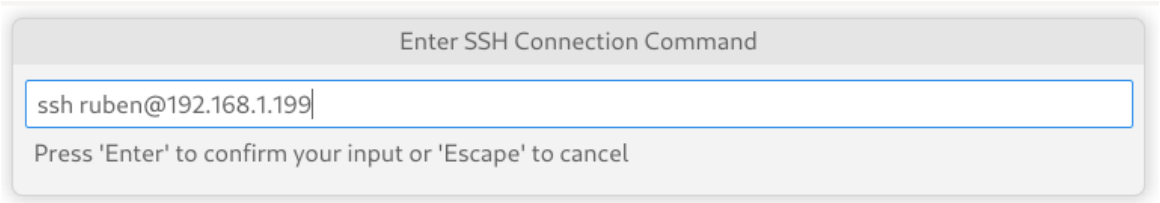
Si hacemos uso de una máquina virtual para el desarrollo, por no usar GNU/Linux en la máquina anfitriona donde hemos instalado el contenedor de Laravel, Visual Studio Code nos permite conectarnos por SSH a un servidor donde vayamos a realizar el desarrollo.

Para conectarnos usaremos la extensión recién instalada "Remote explorer", nos aseguramos que estamos en la opción "Remotes (Tunnels/SSH)" y crearemos una nueva conexión SSH al servidor a través del icono "+", en el que nos pedirá realizar la conexión SSH:



¡Atención!

Es recomendable hacer uso del sistema de certificados de clave pública/privada de SSH para realizar la conexión



Nos pedirá dónde queremos guardar la configuración, dejando la ruta por defecto, que es el fichero `.ssh/config` dentro de la "home" de nuestro usuario. Si hemos realizado la configuración de los certificados de clave pública/privada, no nos pedirá la contraseña.

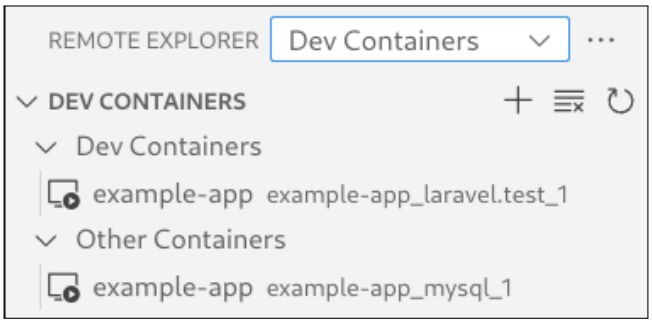
3.3. Desarrollo en un contenedor Docker

Gracias a la extensión "Remote Development" instalada previamente, vamos a poder trabajar **dentro de un contenedor**, en este caso de Laravel. De esta manera Visual Studio Code se podrá conectar a un contenedor local y da igual que se esté ejecutando en Linux, Windows o MacOS. La ventaja de este sistema es que vamos a tener acceso al intérprete de PHP para poder ayudarnos durante el desarrollo.

Hacer uso de esta funcionalidad es muy útil ya que al estar dentro del contenedor, estamos dentro del

entorno de desarrollo de manera “inmersiva”, pudiendo instalar componentes o ejecutar órdenes dentro del contenedor.

Para realizar la conexión, deberemos ver los contenedores en la extensión “Remote Explorer”, en el apartado “**Dev Containers**”:



Al acceder al contenedor, en este caso “example-app_laravel.test_1”, nos aseguramos entrar al directorio principal donde está situada la aplicación Laravel, `/var/www/html`.

Funcionalidad básica

1. Introducción

Ahora que ya tenemos el entorno creado, es momento de empezar a añadir funcionalidad básica a nuestra aplicación y comenzar a crear nuestra aplicación. Para estos ejemplos se ha decidido crear una pequeña aplicación a modo de blog, con posts y comentarios.

2. Artisan

Artisan es la interfaz de línea de comandos que vamos a utilizar para realizar todo tipo de interacción entre el proyecto y el propio *framework* Laravel. Esta interfaz nos va a permitir, entre otras cosas:

- Crear modelos y controladores.
- Crear una sesión a la base de datos.
- Controlar el estado de los "migrations".
- Hacer uso de los "seeds" en la base de datos.
- Limpiar la caché de objetos.

Cada comando contará con su ayuda, por lo que es recomendable ir mirando la ayuda y así conocer las distintas opciones para cada uno de ellos.

3. Crear modelo

Un blog tiene una serie de "Posts", que son las entradas que los usuarios introducen en el blog. De momento vamos a ignorar el apartado de usuarios, para simplificarlo. Una entrada del blog contará con los siguientes atributos:

- Título
- Texto
- Si está publicada o no

Para crear el modelo, ejecutaremos el siguiente comando. Este comando lo debemos ejecutar dentro del contenedor Docker y dentro de la ruta donde se encuentra el proyecto Laravel, que es `/var/www/html`:

Crear Modelo

```
root@1b29e46c10ae:/var/www/html# php artisan make:model Post -crms
```

Este comando nos va a crear el modelo Post junto con:

- **Controlador** de tipo "resource", lo que va a permitir realizar acciones "**CRUD**" (*create*, *read*, *update* y *delete*), necesarias en cualquier aplicación web.
- ***Migration***: Un fichero para realizar la migración del modelo en la base de datos.

- ***Seed***: Un fichero de tipo "semilla" para introducir datos en la base de datos.

4. Entendiendo las "*migrations*" de base datos

Hoy en día son muchos los *frameworks* que hacen uso de sistemas de tipo ***migration*** a la hora de interactuar en el tiempo con la base de datos. Podríamos definirlo como un **sistema de control de versiones para el esquema de base de datos**.

Este sistema permite ir evolucionando el esquema de base de datos (tablas, columnas de las tablas, funciones...) a medida que el propio código fuente de la aplicación va evolucionando. De esta manera, si tenemos el código en un punto concreto, con el sistema **migrations** nos va a crear la base de datos tal como se necesita en ese punto.

Al crear nuestro proyecto Laravel, ya contamos con una serie de ficheros de migraciones para la base de datos. Estos ficheros se encuentran en `app/database/migrations/`, teniendo cada fichero un formato similar a `YYYY\_mm\_dd\_HHMMSS\_comentario.php` siendo:

- **YYYY**: el año que se ha creado el fichero.
- **mm**: el mes que se ha creado el fichero.
- **dd**: el día que se ha creado el fichero.
- **HHMMSS**: la hora, minuto y segundo.
- **comentario**: un pequeño comentario sobre el contenido del fichero.

De esta manera, los migrations se van a poder ejecutar en orden de fecha de creación, donde normalmente suele ser:

- **De más antiguo a más nuevo**: Se van creando las tablas, y se van añadiendo modificaciones. Es el ciclo normal de desarrollo, y este es el sistema de uso habitual.
- **De más nuevo a más antiguo**: Se vuelve atrás en el proyecto, eliminando modificaciones. Utilizado para ir a una versión antigua del proyecto.

Vamos a utilizar como ejemplo el primer fichero que existe en el directorio, que es para hacer uso de la tabla de usuarios del sistema de autenticación de Laravel. El fichero tiene una clase que extiende de la clase **Migration** con dos funciones:

Fichero Migration

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

```php
return new class extends Migration {
    public function up(): void {
        Schema::create('users', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('email')->unique();
            $table->timestamp('email_verified_at')->nullable();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });
    }

    public function down(): void {
        Schema::dropIfExists('users');
    }
};
```

La función `up()` se ejecutará cuando realizamos la migración, mientra que la función `down()` se usará cuando realicemos un "***rollback***" (echar para atrás una migración).

¡Atención!

Por convenio, el nombre de los modelos suelen ser en singular, mientras que las tablas se deben crear en plural. Pero se puede cambiar el nombre de la tabla.

4.1. Opciones de las migraciones

En la documentación oficial se explican cómo funcionan los *migrations*, las funcionalidades básicas y avanzadas que tienen, así como los distintos tipos de columnas que podemos utilizar.

Teniendo en cuenta lo visto en el punto anterior, podemos visualizar que las acciones del *migration* contiene varias líneas, y vamos a destacar lo siguiente para el fichero `2014\_10\_12\_000000\_create\_users\_table.php`:

- Crea una tabla llamada "**users**", que contiene varias columnas
- **id**: es un alias al método **bigIncrements**. Va a generar una columna de tipo "*big integer*" sin signo, que se va a ir incrementando y que va a ser **clave primaria**.
- **string**: existen varias columnas de tipo "string", que son "name", "email" y "password". Es lo equivalente a "varchar", sin indicar en este caso el número de longitud. Se le puede indicar como segundo parámetro.

- **unique()**: el contenido de este campo (en el ejemplo el **email**) debe ser único en la tabla.
- **timestamp**: crea un campo de tipo TIMESTAMP.
- **nullable**: permite que este campo sea **null**.
- **timestamps()**: Este es un método especial que crea dos campos en la base de datos: "**created_at**" y"**updated_at**". De esta manera sabemos cuándo se ha creado y modificado el registro en la base de datos.

Ejercicio

Añade a la migración del modelo Post, la generación de los campos: "título", "texto" y "publicado". Recuerda mirar la documentación oficial.

4.2. Uso de las migraciones

Una vez tenemos distintos ficheros de migraciones, hay que saber cómo aplicarlos y qué sucede con ellos. De nuevo, en la documentación aparecen distintos ejemplos, de los cuales se van a destacar sólo unos a continuación:

4.2.1. Desplegar migraciones

Para realizar el despliegue de todas las migraciones debemos ejecutar el siguiente comando:

Ejecutar migraciones

```
root@1b29e46c10ae:/var/www/html# php artisan migrate
   INFO  Preparing database.
Creating migration table ............................ 52ms DONE

   INFO  Running migrations.
2014_10_12_000000_create_users_table ..............   108ms DONE
2014_10_12_100000_create_password_reset_tokens_table  127ms DONE
2019_08_19_000000_create_failed_jobs_table .........   88ms DONE
2019_12_14_000001_create_personal_access_tokens_table 140ms DONE
2023_09_26_094514_create_posts_table ...............   74ms DONE
```

4.2.2. Comprobar estado de las migraciones

Para comprobar el estado de las migraciones podemos realizarlo de la siguiente manera:

Estado de las migraciones

```
root@1b29e46c10ae:/var/www/html# php artisan migrate:status
```

```
Migration name ................................ Batch / Status
2014_10_12_000000_create_users_table ................. [1] Ran
2014_10_12_100000_create_password_reset_tokens_table . [1] Ran
2019_08_19_000000_create_failed_jobs_table ........... [1] Ran
2019_12_14_000001_create_personal_access_tokens_table  [1] Ran
2023_09_26_094514_create_posts_table ................. [1] Ran
```

Si queremos ver a nivel de base de datos qué ha pasado, podemos ejecutar una sesión y visualizar la propia base de datos. Veremos cómo se ha creado la base de datos, las tablas, y una tabla especial llamada **migrations**, que contiene qué ficheros se han desplegado.

Ejecutar migraciones

```
root@1b29e46c10ae:/var/www/html# php artisan db

mysql> use example_app;
Database changed

mysql> show tables;
+------------------------+
| Tables_in_example_app  |
+------------------------+
| failed_jobs            |
| migrations             |
| password_reset_tokens  |
| personal_access_tokens |
| posts                  |
| users                  |
+------------------------+
6 rows in set (0.00 sec)

mysql> select * from migrations;
+----+-----------------------------------------------------+-------+
| id | migration                                           | batch |
+----+-----------------------------------------------------+-------+
|  1 | 2014_10_12_000000_create_users_table                |     1 |
|  2 | 2014_10_12_100000_create_password_reset_tokens_table |     1 |
|  3 | 2019_08_19_000000_create_failed_jobs_table          |     1 |
|  4 | 2019_12_14_000001_create_personal_access_tokens_table |     1 |
```

```
|  5 | 2023_09_26_094514_create_posts_table                |     1 |
+----+-----------------------------------------------------+-------+
5 rows in set (0.00 sec)
```

4.2.3. *Rollback* la última migración

En un momento dado nos puede interesar echar atrás la última migración, y para ello contamos con la opción ***rollback***. Este sistema puede que deshaga las migraciones de varios ficheros.

Ejecutar migraciones

```
root@1b29e46c10ae:/var/www/html# php artisan migrate:rollback

INFO  Rolling back migrations.
2023_09_26_094514_create_posts_table ..............  27ms DONE
2019_12_14_000001_create_personal_access_tokens_table  26ms DONE
2019_08_19_000000_create_failed_jobs_table ..........  25ms DONE
2014_10_12_100000_create_password_reset_tokens_table   24ms DONE
2014_10_12_000000_create_users_table ..............  27ms DONE
```

En este caso, como el migrate hizo todos los ficheros, el *rollback* se ha ejecutado de todos los ficheros pero **en orden inverso al de creación**.

4.2.4. Limpiar, *reset* y recarga de migraciones

Para asegurar que el sistema de migraciones está funcionando bien, para hacer pruebas, o para realizar despliegues limpios quizá nos interese borrar todas las migraciones de la aplicación o realizar una recarga de las mismas.

- **db:wipe**: borra todas las tablas, vistas y tipos.
- **migrate:fresh**: borra todas las tablas de la base de datos y aplica de nuevo todas migraciones.
- **migrate:reset**: deshace todas las migraciones de la aplicación. Básicamente es dejar la base de datos limpia. **En este caso no se borra la tabla "*migration*"**.
- **migrate:refresh**: deshace todas las migraciones de la aplicación y las vuelve a aplicar en orden.

4.3. Uso de las semillas

A la hora de crear una aplicación es posible que nos interese que tras realizar un primer despliegue existan datos en la base de datos. Ya sea porque estos datos son necesarios para el correcto funcionamiento de la aplicación o para darle una funcionalidad básica.

Para poblar de datos la base de datos existe el sistema de semillas, o ***seeds***. Este sistema funciona a través de sus propios ficheros, que se pueden crear por modelo (tal como hemos hecho en este capítulo), o de

manera general en una semilla propia.

Con la generación del modelo se ha creado también el fichero al que vamos a añadirle el código necesario para que cree un primer post de pruebas: `app/database/seeders/PostSeeder.php`.

Seed del PostSeeder.php

```php
<?php
namespace Database\Seeders;

use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\DB;

class PostSeeder extends Seeder {
    public function run(): void {
        DB::table('posts')->insert([
            "titulo"=>"Primer post",
            "texto"=>"Este es el texto del primer post",
            "publicado"=>true,
            "created_at"=>now(),
        ]);
    }
}
```

Para poder hacer uso del modelo "**DB**" es necesario hacer uso de la librería "**Illuminate\Support\Facades\DB**". Ahora sólo queda ejecutar el *seed* tal como se explica en la documentación:

Ejecutar el seed

```
root@1b29e46c10ae:/var/www/html# php artisan db:seed PostSeeder
INFO  Seeding database.
```

Si se comprueba la base de datos, se verá cómo en la tabla aparecen los datos del *seed*.

5. Rutas de la aplicación

Aunque ya tenemos un controlador y datos en la aplicación, hasta ahora son inaccesibles. Lo único que vemos en la aplicación es la página de bienvenida al proyecto y si ponemos cualquier ruta en la URL nos aparece un error "404 Not Found".

Esto es debido al sistema de enrutado de la aplicación, que sólo permite acceder al *path* "/" que nos muestra

la plantilla de bienvenida. Esta configuración se puede ver en el fichero routes/web.php.

Rutas de la aplicación web de Laravel

```
<?php
use Illuminate\Support\Facades\Route;

Route::get('/', function () {
    return view('welcome');
});
```

Cualquier intento de acceso a algo que no sea esa ruta dará un error 404. Este es un sistema de seguridad para controlar a qué se tiene acceso en la aplicación, y por eso que debemos modificar este fichero para poder acceder a nuestro nuevo controlador.

Añadiendo rutas para el nuevo controlador

```
<?php
// ...
use App\Http\Controllers\PostController;
Route::controller(PostController::class)->group(function () {
    Route::get('/posts', 'index')->name('posts.index');
    Route::get('/posts/{post}', 'show')->name('posts.show');
});
```

Este código indica que se va a utilizar la clase "PostController" para el grupo de las rutas que aparecen en ese trozo de código. Si vamos al fichero App/Http/Controllers/PostController.php veremos que por defecto todas las funciones están vacías, y es por eso que no nos devuelve ningún dato.

Por lo tanto, la idea es:

- **/posts**: irá a la función "index" del controlador. Esta función normalmente lista el contenido de la tabla de base de datos que hace referencia al modelo. En nuestro caso, mostrará todos los posts del blog (normalmente en formato paginado).
- **/posts/{post}**: esta ruta será la utilizada cuando queramos ir a ver un registro del modelo concreto. En este caso "{post}" indicará el "id" dentro de la base de datos que se le pasará a la función "show".

Existe un comando para visualizar qué rutas están configuradas en nuestra aplicación:

Añadiendo rutas para el nuevo controlador

```
root@1b29e46c10ae:/var/www/html# php artisan route:list

  GET|HEAD        / ..........................................................
```

```
GET|HEAD       posts ........... posts.index › PostController@index
GET|HEAD       posts/{post} ...... posts.show › PostController@show
GET|HEAD       storage/{path} ....................... storage.local
GET|HEAD       up ..................................................
                                                  Showing [5] routes
```

En el siguiente apartado, cuando modifiquemos el controlador quedará más claro.

5.1. Tipos de rutas

Hay que entender que las rutas funcionan en base al protocolo HTTP. Esto quiere decir que existen distintas maneras de acceder a la misma URL dependiendo del tipo de petición que se realice en base a lo que realicemos con el navegador.

Normalmente, cuando navegamos por internet, las peticiones que se realizan son de tipo **GET**, ya que estamos pidiendo información al servidor web. En cambio, cuando rellenamos un formulario y le damos a enviar, se hace uso del "verbo" **POST**, ya que se envían datos al servidor.

Las peticiones HTTP que se pueden utilizar son:

- **GET**: Se realiza una petición a la ruta especificada. Estas peticiones sólo obtienen información.
- **POST**: Se envían datos al servidor, que van incluidos dentro del cuerpo de la petición. Lo habitual cuando utilizamos formularios. Se utiliza para crear nuevos recursos.
- **PUT**: Similar a POST, pero en este caso suele estar orientado a modificar datos previamente creados.
- **PATCH**: Como PUT, sobreescribe completamente un recurso existente.
- **DELETE**: Borra el recurso especificado.

Es conveniente mirar la documentación cuando queramos realizar algún tipo de petición distinto de GET, ya que nos ayudará a comprender mejor qué es lo que está sucediendo.

6. Controladores y Vistas

Ahora que ya tenemos las rutas creadas, es momento de que los datos se visualicen en la aplicación. Para ello es necesario entender cómo funciona el sistema de plantillas utilizado por Laravel, llamado **Blade**, que junto con el sistema de **enrutado** visto previamente, relaciona la URL a la que se llama con el controlador y la vista correspondientes.

6.1. Obtener datos en el controlador

El ejemplo va a consistir en obtener todos los posts de la base de datos y hacer un listado con ellos. Por otro lado, al seleccionar un post concreto, se mostrará dicho post. Para ello vamos a modificar el controlador para modificar las dos funciones que se utilizan en las rutas:

Funciones modificadas en el controlador Post

```php
<?php
// ...
use App\Models\Post;
// ...
class PostController extends Controller{
    public function index(){
        $posts = Post::orderBy('created_at')->get();
        return view('posts.index',['posts' => $posts]);
    }
    //...
    public function show(Post $post){
        return view('posts.show',['post'=>$post]);
    }
```

El problema de este código es que estamos llamando a unas vistas que todavía no hemos creado, y les estamos pasando como variables a la vista los datos obtenidos dentro de un array. Podremos pasar tantas variables como queramos.

6.2. Generar vista

El sistema de plantillas y vistas Blade se guardan en la ruta `resources/views`, y en el primer caso lo que estamos diciendo es que haga uso de "posts.index", que quiere decir el fichero "index.blade.php" del directorio "posts". Por lo tanto, deberemos crear un fichero en la ruta `resources/views/posts/index.blade.php`, que corresponde a la vista que estamos llamando.

Información

Es recomendable para cada Modelo/Controlador crear un directorio de vistas

Ahora es momento de visualizar los datos en la vista. Para ello, recorreremos el listado obtenido y lo visualizaremos, todo ello en la vista. El sistema de plantillas Blade permite introducir funcionalidad similar a PHP en la vista mezclado con HTML. También permite incrustar código PHP directamente, pero intentaremos evitarlo.

El sistema de plantillas tiene una serie de palabras reservadas similar a la de los lenguajes de programación más habituales. En este ejemplo se va a recorrer con un bucle for la lista, se crea una variable de indexación, y así poder visualizar los atributos:

Vista "index.blade.php"

```
<ul>
  {{--esto es un comentario: recorremos el listado de posts--}}
  @foreach ($posts as $post)
    {{-- visualizamos los atributos del objeto --}}
    <li>
      <a href="{{route('posts.show',$post)}}"> {{$post->titulo}}</a>.
      Escrito el {{$post->created_at}}
    </li>
  @endforeach
</ul>
```

Si ahora visualizamos la ruta "/posts" obtendremos el listado. Es importante destacar que para el enlace que nos lleva a visualizar un post concreto **se ha hecho uso del sistema ed rutas** al que se le pasa como parámetro el post.

Ejercicio

Crea la vista para visualizar toda la información del post en "show.blade.php".

7. Soft Deleting

Laravel, a través de su ORM Eloquent, nos permite hacer uso del sistema "*soft deleting*", que en lugar de borrar los registros de la base de datos, lo que hace es marcarlo como borrado. Esto lo hace a través de una columna en la base de datos, indicando con una fecha cuándo se ha borrado.

Es habitual hacer uso de estos sistemas, por si el borrado ha sido erróneo, y de esta manera poder recuperar registros (ya que realmente no se han borrado).

Para hacer uso de este sistema debemos indicarlo en el modelo, para ello le indicaremos que se va a usar "**SoftDeletes**":

Indicar en el modelo el uso de Softdeletes

```
<?php
//...
use Illuminate\Database\Eloquent\SoftDeletes;
class Post extends Model{
    use SoftDeletes;
    //...
}
```

Y también debemos indicarlo en la generación de la base de datos (o en un nuevo *migration*). De esta manera, se creará la columna correspondiente que es necesaria.

Indicar en el "migration" el uso de Softdeletes

```php
<?php
//...
public function up(): void {
    Schema::create('posts', function (Blueprint $table) {
        $table->id();
        $table->string("titulo",128);
        $table->string("texto",5000);
        $table->boolean("publicado");
        $table->softDeletes();
        $table->timestamps();
    });
}
```

Si ejecutamos los *migrations*, veremos que la tabla tiene un campo "**deleted_at**", que por defecto estará a NULL. Si ahora borramos un registro, se actualizará esa columna con la fecha del momento en el que se ha realizado la acción de borrado. **Estos registros pueden ser recuperados**.

8. Debug

Durante el desarrollo es habitual hacer uso de sistemas de *debug*, por ejemplo para poder ver el contenido de variables y parar la ejecución del algoritmo que estamos programando.

Laravel cuenta con una función llamada `dd()` que podemos utilizar en cualquier momento. Por ejemplo, si lo usamos en el controlador creado previamente:

Añadida llamada al debug

```php
<?php
 public function index(){
    $posts = Post::all();
    dd($posts);
    return view('posts.index',['posts' => $posts]);
}
```

En este caso, se ejecutará la petición de obtener todos los *posts*, y acto seguido la función `dd($posts)` lo que hará será mostrar por pantalla el contenido de la variable y terminará la ejecución del código.

9. Consola Tinker

Hoy en día muchos *frameworks* tienen algún sistema de consola interactiva con la que poder utilizar las funcionalidades del mismo. De esta manera, podemos realizar comprobaciones, interactuar con los modelos, objetos... pero sin tener que hacerlo desde el código de la web.

En el caso de Laravel la consola se llama Tinker. Para asegurarnos que cuenta con toda la información actualizada, debemos ejecutar un comando previo.

Llamar a Tinker con Artisan

```
root@1b29e46c10ae:/var/www/html# composer dump-autoload
Generating optimized autoload files
> Illuminate\Foundation\ComposerScripts::postAutoloadDump
> @php artisan package:discover --ansi

   INFO  Discovering packages.
...
root@1b29e46c10ae:/var/www/html# php artisan tinker
Psy Shell v0.11.21 (PHP 8.2.10 — cli) by Justin Hileman
```

Una vez dentro, podremos hacer uso de los modelos, por ejemplo, para ver los datos que tenemos en la base de datos.

Obtenemos los Posts en Tinker

```
>>> Post::all();
[!] Aliasing 'Post' to 'App\Models\Post' for this Tinker session.
= Illuminate\Database\Eloquent\Collection {#7247
    all: [
    App\Models\Post {#7249
        id: 1,
        titulo: "Primer post",
        texto: "Este es el texto del primer post",
        publicado: 1,
        created_at: "2023-10-01 16:57:30",
        updated_at: null,
    },
    ],
}
```

IV

Usar Bootstrap en Laravel

1. Instalar dependencias

Laravel tenía soporte nativo de Bootstrap, pero decidió sustituirlo por Tailwind. Eso no quita que podamos usar Bootstrap, pero necesitaremos realizar la instalación de dependencias.

Para usar Bootstrap con Laravel

```
root@1b29e46c10ae:/var/www/html# composer require laravel/ui --dev
root@1b29e46c10ae:/var/www/html# php artisan ui bootstrap --auth
root@1b29e46c10ae:/var/www/html# npm install
root@1b29e46c10ae:/var/www/html# npm run build
```

A continuación se detalla qué hace cada comando:

- `composer require laravel/ui --dev`: Composer es el gestor de dependencias utilizado por PHP. Lo que se está indicando es que se necesita como dependencia el paquete "laravel/ui" durante el desarrollo.
- `php artisan ui bootstrap --auth`: Se indica qué *framework* para el interfaz se va a utilizar. Aparte, con el parámetro "–auth" se le indica que genera las plantillas para la autenticación.
- `npm install`: instala las dependencias indicadas en el primer comando.
- `npm run build`: ejecuta la acción "build" indicada en el fichero `package.json`. En este caso "compila" los javascripts y los css que se van a utilizar. Más adelante hablamos de ello.

De esta manera no sólo hemos instalado las dependencias necesarias para hacer uso de Bootstrap, si no que también nos ha generado unas vistas para el sistema de autenticación en `resources/views/auth` y una plantilla general para la aplicación.

2. Plantilla general

Anteriormente se ha mencionado que Blade es un sistema de plantillas para Laravel. Esto significa que es capaz de generar unos componentes de vistas que a su vez incorporan otras vistas, de esta manera generando plantillas que se pueden reutilizar ahorrando código y simplificando la aplicación.

Con lo realizado previamente se ha generado una plantilla general en el fichero `resources/views/layouts/app.blade.php`, que se puede dividir en dos apartados:

- **Cabecera "nav"**: es la cabecera de la aplicación. Aparece el nombre de la aplicación y a la derecha tiene enlaces para hacer login o registrarse en la aplicación con el sistema de autenticación.
- `(`content')`: este apartado será sustituido por la vista desde la que se llame a esta plantilla.

De esta manera, en todas las vistas de la aplicación que llamemos a la plantilla, no tendremos que escribir el código de la cabecera. Lógicamente, **es posible añadir nuevos apartados a esta vista** para cumplir con el objetivo final de la aplicación.

2.1. Cómo usar la plantilla

Para poder hacer uso de la plantilla, debemos indicarlo en las correspondientes vistas. Como ejemplo, se va a utilizar la vista creada en el capítulo anterior, la que muestra por pantalla los *posts* del blog.

La vista modificada quedaría:

Vista "show.blade.php" modificada usando la plantilla

```
@extends('layouts.app')

@section('content')
<div class="container">
  <h1>{{$post->titulo}}</h1>
  <p>Creado el {{$post->created_at}}</p>
  <p>{{$post->texto}}</p>
</div>
@endsection
```

Tal como se puede ver, lo primero que se indica es que esta vista "extiende" de una plantilla concreta. Después se indica la sección de la plantilla que va a ser sustituida por el contenido que aparece entre "@section" y "@endsection", que en este caso es la que corresponde al `(`content')` que hemos visto previamente.

3. Modificación de rutas

Aparte de lo visto hasta ahora, por haber activado el sistema de autenticación, se nos han generado nuevas rutas. Estas rutas las podemos ver a través del siguiente comando, o mirando el fichero de rutas:

Mirando todas las rutas creadas

```
root@1b29e46c10ae:/var/www/html# php artisan route:list
```

4. Añadir CSS o Javascript propio

Si miramos la plantilla general `resources/views/layouts/app.blade.php` que se nos ha creado, podemos ver que en la cabecera hay un apartado de scripts:

Plantilla "resources/views/layouts/app.blade.php"

```
<!doctype html>
<html lang="{{ str_replace('_', '-', app()->getLocale()) }}">
  <head>
```

```
    <!-- ... -->
    <!-- Scripts -->
    @vite(['resources/sass/app.scss', 'resources/js/app.js'])
</head>
```

En las nuevas versiones de Laravel se hace uso de Vite, que es un sistema que "compila" el css y el javascript para minimizarlo. Esta configuración de la plantilla en el HTML generado queda como unos links de CSS y Javascript al directorio `public/`.

Información

Vite es un sistema que "minimiza" el CSS y Javascript que necesitamos.

El problema es que si realizamos cualquier modificación del CSS o Javascript, tendríamos que "compilar" para poder ver el efecto. Para evitar esto, Vite puede levantar un pequeño servicio que nos mostrará los cambios en tiempo real.

4.1. Configuración de Vite

La configuración de Vite por defecto hace uso del sistema Sass, que es un metalenguaje para escribir CSS pero que nos permite el uso de variables, anidamientos estilo herencia, ... Dado que vamos a querer usar CSS, podemos hacer un "include" del fichero `resources/css/app.css` en la configuración `resources/sass/app.sass`:

Configurando "resources/sass/app.sass"

```
// Fonts
@import url('https://fonts.bunny.net/css?family=Nunito');
// Variables
@import 'variables';
// Bootstrap
@import 'bootstrap/scss/bootstrap';
// Custom CSS
@import '../css/app.css';
```

4.2. Estilos propios sobre Bootstrap

Los estilos de Bootstrap están creados para que por defecto siempre prevalezcan sobre el resto de estilos, y por cómo funciona el sistema en cascada de CSS, lo debemos tener en cuenta para añadir nuestros estilos.

No es buena idea hacer uso del sistema "!important" en CSS, por lo que para añadir más puntuación a nuestros estilos, la idea es añadir al "<body>" de nuestro HTML un ID y después en nuestra hoja de estilos

hacer uso de ese ID para cada elemento que queramos modificar el estilo.

Por lo tanto, la plantilla general `resources/views/layouts/app.blade.php` podría quedar tal que:

Plantilla "resources/views/layouts/app.blade.php"

```
...
<body id="mis-estilos">
    ...
</body>
...
```

Y en el fichero `resources/css/app.css`, donde pondremos nuestros estilos, deberían ir precedidos del ID. Por ejemplo, para tener un color propio en los elementos "h1":

Fichero "resources/css/app.css"

```
#mis-estilos h1{
    color: red;
}
```

¡Atención!

En este fichero pueden aparecer includes de **tailwind**. Es recomendable quitarlos para que no conflicten con Bootstrap.

Ahora sólo queda levantar el servicio Vite o realizar la compilación para que surja efecto el cambio.

4.3. Cómo usar Vite

Para hacer funcionar los CSS o Javascript propios que hemos podido añadir en los ficheros `resources/css/app.css` o `resources/js/app.js` se puede hacer de dos maneras:

- **Levantando un servicio**: los cambios se verán automáticamente a medida que los realizamos.

  ¡Cuidado!

  Este sistema sólo se debería usar durante el desarrollo de la aplicación.

- **Generando** los ficheros finales: el método que minimiza y "compila" los ficheros y los deja en el directorio `public/build`.

Por defecto, haremos uso de la primera opción durante el desarrollo, mientras que el segundo método es para la puesta en producción, por lo que es importante recordar ejecutar el comando que veremos más adelante.

4.3.1. Servicio Vite

Durante el desarrollo el mejor método es hacer uso del servicio Vite, que podemos levantar dentro del contenedor. Este método levantará el servicio en el puerto 5173 y modificará la plantilla para que haga uso de ello.

Para levantar el servicio, desde la raiz de la aplicación, se ejecutará:

Levantar servicio Vite

```
root@1b29e46c10ae:/var/www/html# npm run dev
> dev
> vite

VITE v5.4.10  ready in 178 ms
->  Local:   http://localhost:5173/
->  Network: http://192.168.144.2:5173/
->  press h to show help

LARAVEL v11.29.0  plugin v1.0.5
->  APP_URL: http://localhost
```

Si miramos el código del HTML generado obtenido del **navegador web**, veremos cómo se hace referencia a ese puerto:

Código HTML de la web

```
<!-- Scripts -->
<script type="module" src="http://localhost:5173/@vite/client"></script>
<link rel="stylesheet" href="http://localhost:5173/resources/sass/app.scss" />
<script type="module" src="http://localhost:5173/resources/js/app.js">
</script>
```

4.3.2. Generar ficheros para producción

El comando que vamos a especificar a continuación ya lo hemos ejecutado durante la puesta en marcha de Bootstrap, al inicio de este capítulo:

Levantar servicio Vite

```
root@1b29e46c10ae:/var/www/html# npm run build
> build
> vite build
```

```
vite v5.4.10 building for production...
✓ 111 modules transformed.
public/build/manifest.json                0.26 kB │ gzip:  0.14 kB
public/build/assets/app-48669401.css    225.72 kB │ gzip: 30.77 kB
public/build/assets/app-c75e0372.js     111.35 kB │ gzip: 36.20 kB
✓ built in 3.58s
```

Tal como se puede ver, este comando ha realizado la generación de los assets dentro del directorio `public/build` que son utilizados en la plantilla, que se puede ver al inspeccionar el código HTML generado.

V

Métodos *create*, *update*, *delete*

1. Crear rutas necesarias

Una aplicación web normalmente nos va a permitir crear datos, no sólo visualizarlos. Por lo tanto vamos a tener que crear la vista de un formulario que el usuario pueda utilizar para crear datos a través del controlador.

Tal como hemos dicho, las funcionalidades de la aplicación empiezan por crear una ruta a la que el usuario puede acceder. En este caso, se podrían crear las rutas necesarias para visualizar el formulario de creación, obtener los datos para la creación, edición y actualización de datos...

En lugar de eso el sistema de rutas de Laravel nos permite simplificarlo, y si tenemos un modelo que sabemos que es de tipo "resource", nos permite crear todas las rutas necesarias para la gestión de los datos. Por lo tanto, las rutas quedarían de la siguiente manera:

Rutas simplificadas para un modelo de tipo "resource"

```php
<?php
//...
Route::resources([
    'posts' => PostController::class,
]);
```

Si miramos las rutas generadas, veremos que nos ha creado todas las rutas necesarias para interactuar con los posts:

Mirando todas las rutas creadas

```
root@1b29e46c10ae:/var/www/html# php artisan route:list
GET|HEAD   posts ............ posts.index › PostController@index
POST       posts ............ posts.store › PostController@store
GET|HEAD   posts/create ..... posts.create › PostController@create
GET|HEAD   posts/{post} ..... posts.show › PostController@show
PUT|PATCH  posts/{post} ..... posts.update › PostController@update
DELETE     posts/{post} ..... posts.destroy › PostController@destroy
GET|HEAD   posts/{post}/edit. posts.edit › PostController@edit
```

Tal como se puede ver, por haber indicado la ruta anterior, automáticamente nos ha creado las rutas para listar, crear, visualizar, actualizar, editar y borrar el recurso. Con una única línea nos evita tener que escribir todas las líneas que supondrían de configuración.

2. Crear registro

A la hora de crear un registro, Laravel por defecto hace uso de la ruta “create”, por lo que deberemos crear un botón en la vista principal que nos mande a la URL “/posts/create”, por lo que la vista de creación será `create.blade.php`.

En esta vista debemos crear un formulario, en el que se tendrá en cuenta los siguientes apartados:

- **Ruta del *action* a ejecutar**: se debe indicar la ruta donde se mandarán los datos, que debe coincidir con la función que guardará los datos en la base de datos. Por defecto, la ruta es “store”.
- **Método**: el método para enviar datos al servidor será POST.
- `@csrf`: es un atributo especial de Blade que genera un atributo oculto “input” en el formulario donde se guarda el token CSRF, para posteriormente comprobar que es correcto.

Con todo ello, la vista quedaría de la siguiente forma:

Vista del formulario

```
@extends('layouts.app')
@section('content')
  <div class="container">
    <form class="mt-2" name="create_platform"
       action="{{route('posts.store')}}" method="POST" enctype="multipart/form-data">
      @csrf
      <div class="form-group mb-3">
        <label for="titulo" class="form-label">Titulo</label>
        <input type="text" class="form-control" id="titulo" name="titulo" required/>
      </div>
      <div class="form-group mb-3">
        <label for="texto" class="form-label">Texto</label>
        <textarea type="textarea" rows="5" class="form-control" id="texto" name="texto">
        </textarea>
      </div>
      <div class="form-check">
        <input class="form-check-input" type="checkbox" id="publicado"
          name="publicado">
        <label class="form-check-label" for="publicado">
        ¿Publicar?
      </label>
      </div>
      <button type="submit" class="btn btn-primary" name="">Crear</button>
    </form>
  </div>
@endsection
```

Al pulsar el botón “Crear” se realizará una petición “**POST**” a la ruta “**posts.store**”, por lo tanto es la función que debemos modificar ahora en el controlador, que junto con la función “create”, tendrá la siguiente forma:

Añadiendo funcionalidad al controlador

```php
<?php
//...
public function create(){
    return view('posts.create');
}

public function store(Request $request){
    $post = new Post();
    $post->titulo = $request->titulo;
    $post->texto = $request->texto;
    $post->publicado = $request->has('publicado');
    $post->save();
    return redirect()->route('posts.index');
}
```

3. Editar registro

Una vez creados los registros nos va a interesar poder editarlos. Para ello, tendremos que añadir a las vistas correspondientes (el listado general y/o desde la vista del post) un botón que nos lleve a la ruta para editar, que es: `/posts/\{id\}/edit`.

Para poder visualizar los datos, deberemos obtener desde el controlador los datos referentes a ese "id" para poder visualizarlo en el formulario que crearemos en la vista `posts/edit.blade.php`. Después, en la función update deberemos realizar el guardado de las modificaciones.

Añadiendo funcionalidad al controlador

```php
<?php
//...
public function edit(Post $post){
    return view('posts.edit',['post'=>$post]);
}

public function update(Request $request, Post $post){
    $post->titulo = $request->titulo;
    $post->texto = $request->texto;
    $post->publicado = $request->has('publicado');
    $post->save();
    return view('posts.show',['post'=>$post]);
```

```
}
```

Tal como se puede ver, la función de actualizar lo que hace es recibir los datos del formulario y el registro a actualizar. Existe la posibilidad de conocer si alguno de los campos ha sido modificado antes de realizar la actualización. Después, debemos sustituir los campos y para finalizar guardar los cambios del registro. Por último, volvemos a la vista para visualizar los cambios.

La vista para editar el registro quedaría:

Vista del formulario

```
@extends('layouts.app')
@section('content')
<div class="container">
  <form class="mt-2" name="create_platform" action="{{route('posts.update',$post)}}"
    method="POST" enctype="multipart/form-data">
    @csrf
    @method('PUT')
    <div class="form-group mb-3">
      <label for="titulo" class="form-label">Titulo</label>
      <input type="text" class="form-control" id="titulo" name="titulo" required
        value="{{$post->titulo}}"/>
    </div>
    <div class="form-group mb-3">
      <label for="texto" class="form-label">Texto</label>
      <textarea type="textarea" rows="5" class="form-control" id="texto" name="texto">
        {{$post->texto}}
      </textarea>
    </div>
    <div class="form-check">
      <input class="form-check-input" type="checkbox" id="publicado" name="publicado"
        @checked($post->publicado)>
      <label class="form-check-label" for="publicado">
      ¿Publicar?
      </label>
    </div>

    <button type="submit" class="btn btn-primary" name="">Actualizar</button>
  </form>
</div>
@endsection
```

Dado que es el formulario de edición, deben existir datos, de ahí que para cada apartado haya que añadir el parámetro “value” en los *inputs*, el valor dentro del *textarea*, o darle el valor correspondiente al *checkbox*.

También hay que tener en cuenta que debido a cómo funciona el protocolo HTTP con los formularios, el método PUT no se puede utilizar en formularios diréctamente, por lo que debemos añadir `(`PUT')`

para que lo genere oculto en el formulario.

Ejercicio

Dado que el formulario de crear y actualizar es prácticamente igual, sería interesante crear una única vista que sirva para ambos métodos.

4. Borrar registro

Por último, tenemos que poder eliminar registros, por lo que deberemos añadir un botón que ejecute la acción de borrado que se recibirá en el controlador. Este botón lo vamos a añadir a la lista de posts, que junto con el botón editar del apartado anterior, quedaría:

[Vista del index.blade.php]

```
@foreach ($posts as $post)
  {{-- visualizamos los atributos del objeto --}}
  <li class="pt-1">
    <div class="d-flex flex-row">
      <a href="{{route('posts.show',$post)}}"> {{$post->titulo}}</a>.
      Escrito el {{$post->created_at}}
      <a class="btn btn-warning btn-sm" href="{{route('posts.edit',$post)}}"
        role="button">Editar</a>

      <form action="{{route('posts.destroy',$post)}}" method="POST">
        @csrf
        @method('DELETE')
        <button class="btn btn-sm btn-danger" type="submit"
          onclick="return confirm('Are you sure?')">Delete
        </button>
      </form>
    </div>
  </li>
@endforeach
```

Y por último el controlador debe borrar el objeto cuando se llama a la función **delete**:

Añadiendo funcionalidad al controlador

```
<?php
//....
public function destroy(Post $post) {
    $post->delete();
    return redirect()->route('posts.index');
}
```

VI

Middlewares y autenticación

1. *Middlewares*

Un *middleware* en Laravel es un mecanismo que inspecciona y filtra las peticiones HTTP que llegan a la aplicación. El ejemplo más claro, y que veremos después, es comprobar si un usuario está autenticado mientras usa la aplicación. En caso de no estar autenticado, le mandará a la página de login/registro.

Se pueden crear otros *middlewares* que nuestra aplicación necesite, como por ejemplo registrar todas las peticiones que llegan a la aplicación, validaciones CSRF de formularios, validación de cabeceras...

Los *middlewares* se sitúan en la ruta `app/Http/Middleware/`, donde ya existen varios tras realizar la instalación del *framework* Laravel. Para que entren en funcionamiento, se debe realizar la configuración en el fichero de rutas, ya que se activarán dependiendo de las rutas en las que lo indiquemos.

Aparte de estar creados en la ruta especificada, también deben de estar configurados dentro del fichero `app/Http/kernel.php`, donde ya están configurados para su apartado correspondiente, y donde se han creado unos *alias* que hacen referencia a ellos.

2. Configurando el *middleware* de autenticación

El sistema de autenticación de Laravel es el ejemplo más claro de *middleware* que podemos utilizar, ya que por defecto viene instalado, pero no está configurado. En pasos anteriores hemos creado el interfaz para poder registrar usuarios y realizar el login en la aplicación.

Ahora es el momento de realizar la activación del sistema de autenticación, y que si no se ha hecho el login, no se pueda ver la aplicación y nos envíe a la página de registro.

Para ello, debemos realizar la modificación de rutas, en la que debemos indicar qué rutas queremos que estén dentro del *middleware* de autenticación. En este caso vamos a elegir que para toda la aplicación sea necesario estar autenticado:

Rutas bajo el middleware de autenticación

```php
<?php
//...
Route::middleware(['auth'])->group(function () {
    Route::resources([
      'posts' => PostController::class,
    ]);
});
```

2.1. Comprobar rutas bajo *middlewares*

Si queremos comprobar qué rutas están bajo la influencia de un *middleware*, necesitaremos mirar las rutas en modo *verbose*:

Vista de las rutas en modo *verbose

```
root@1b29e46c10ae:/var/www/html# php artisan route:list -v
...
GET|HEAD  posts ............. posts.index › PostController@index
  → web
  → App\Http\Middleware\Authenticate
POST      posts ............. posts.store › PostController@store
  → web
  → App\Http\Middleware\Authenticate
DELETE    posts/{post} ...... posts.destroy › PostController@destroy
  → web
  → App\Http\Middleware\Authenticate
...
```

Se puede comprobar que estas rutas se aplican para la parte “web” de nuestra aplicación, y que antes de ser ejecutadas pasarán por el *middleware* “**Authenticate**”.

3. Realizar excepciones

No siempre vamos a querer que toda la aplicación esté bajo el sistema de autenticación, ya que lo habitual es que sólo sea necesario para las acciones que puedan suponer un riesgo de seguridad (edición de datos, borrado de datos, apartados sensibles,...), por lo tanto es interesante que haya rutas que no requieran de estar autenticado.

Para ver cómo funciona, vamos a añadir excepciones al listado de todos los posts y a la visualización de cada post por separado. Para ello, al fichero de rutas añadiremos:

Rutas que están exentas del middleware de autenticación

```php
<?php
//...
Route::controller(PostController::class)->group(function () {
    Route::get('/posts', 'index')->name('posts.index');
    Route::get('/posts/{post}', 'show')->name('posts.show');
})->withoutMiddleware([Auth::class]);
```

Tal como se puede ver, hemos creado dos rutas del controlador “PostController” que se les indica “`->withoutMiddleware`”, para que no se aplique, en este caso, la comprobación de autenticación.

4. Comprobar si el usuario está autenticado

Por último, debemos asegurar que los botones de edición o borrado sólo aparezcan cuando el usuario esté logueado. Para ello tenemos el sistema `...`. Si modificamos el fichero `posts/index.blade.php` para evitar que aparezcan los botones de edición y borrado de un post, quedaría:

Comprobar si se está autenticado

```
@auth
  <a class="btn btn-warning btn-sm" href="{{route('posts.edit',$post)}}"
   role="button">Editar</a>

  <form action="{{route('posts.destroy',$post)}}" method="POST"
    enctype="multipart/form-data">
    @csrf
    @method('DELETE')
    <button class="btn btn-sm btn-danger" type="submit"
      onclick="return confirm('Are you sure?')">Delete
    </button>
  </form>
@endauth
```

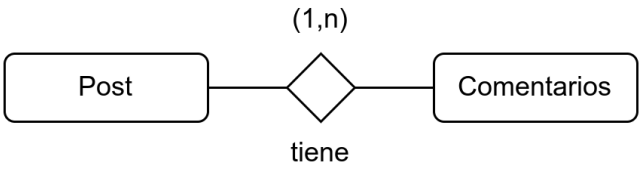
VII

Relacionar modelos

1. Crear modelo relacionado

Siguiendo con nuestro blog, donde ya tenemos una aplicación donde crear posts sólo si estamos logueado, es buen momento de añadir nuevas características. Vamos a incluir la opción de tener comentarios, al menos a nivel relacional.

Sin entrar en los atributos que tiene cada entidad/modelo, la relación que tienen los comentarios respecto a un *post* sería la siguiente:



Es decir, un *post* puede tener muchos comentarios. Un comentario sólo puede pertenecer a un *post*. En principio el único atributo que vamos a permitir es el propio comentario, aparte de la fecha de creación. Para crear el modelo haríamos:

Crear Modelo

```
root@1b29e46c10ae:/var/www/html# php artisan make:model Comentario -crms
```

2. Crear migración

Al igual que vimos al inicio, este comando nos ha creado el modelo, el controlador de *resource*, el sistema de migración y el fichero para añadir la semilla a la base de datos. A la hora de generar la tabla, tenemos que hacer referencia a qué *post* pertenece el comentario, por lo tanto el *migration* queda:

Crear migration

```php
<?php
//...
public function up(): void{
    Schema::create('comentarios', function (Blueprint $table) {
        $table->id();
        $table->string('texto');
        $table->unsignedBigInteger('post_id');
        $table->foreign('post_id')->references('id')->on('posts');
        $table->timestamps();
    });
}
```

Tal como se puede ver, a la hora de crear la tabla en el *migration* se ha creado un campo llamado "**post_id**"

que después se le ha indicado que es de tipo "clave foránea". En la documentación se explican distintas opciones para este tipo de casos.

Ejercicio

Crea un "seed" para añadir un comentario al primer post

3. Crear relación de modelos

Hasta ahora la relación se ha creado a nivel de base de datos, pero es necesario que Laravel a nivel de *framework*, mientras programamos, sea consciente de que los modelos están relacionados entre sí. Para ello, una vez más en la documentación se puede ver cómo Eloquent hace uso de los distintos tipos de relaciones.

Para ello, deberemos modificar ambos ficheros de los modelos que entran en juego en esta relación:

- Relación "**uno a muchos**", donde un *post* puede tener muchos comentarios. Modificaremos el modelo `App/Models/Post.php` para que contenga:

Añadir relación "uno a muchos" en Post

```php
<?php
//...
use Illuminate\Database\Eloquent\Relations\HasMany;
class Post extends Model{
    public function comentarios(): HasMany {
        return $this->hasMany(Comentario::class);
    }
}
```

- Relación inversa "***BelongsTo***", donde un comentario pertenece a un *post*. En este caso, modificaremos el modelo `App/Models/Comentario.php`.

Añadir relación inversa en Comentario

```php
<?php
//...
use Illuminate\Database\Eloquent\Relations\BelongsTo;

class Comentario extends Model{
    public function post(): BelongsTo{
        return $this->belongsTo(Post::class);
    }
```

```
}
```

Tras esto, ya sea a través de una acción o desde Tinker, podremos obtener los comentarios de un *post* específico, perfecto para dibujarlos en la vista donde se visualiza el *post*. Y al revés, dado un comentario, obtener a qué *post* pertenece.

3.1. Modificaciones en vistas, controllers y rutas

Una vez hecha la relación, es momento de que se puedan crear los comentarios. Vamos a empezar realizando modificaciones en las vistas. La idea es que desde la vista de un Post, se pueda añadir un comentario, por lo tanto, en el fichero `resources/views/posts/show.blade.php` añadimos:

Modificada la vista para añadir comentarios

```
@auth
  <div>
    <h3>Añadir comentario</h3>
    <form class="mt-2" name="create_platform"
      action="{{route('comentarios.store')}}" method="POST"
      enctype="multipart/form-data">

      @csrf
      <input type="text" class="form-control" id="post" name="post"
        required hidden value="{{$post->id}}"/>

      <div class="form-group mb-3">
        <textarea type="textarea" rows="5" class="form-control"
            id="texto" name="texto"></textarea>
      </div>
      <button type="submit" class="btn btn-primary" name="">
        Comenta!
      </button>
    </form>
  </div>
@endauth
```

A continuación añadimos una ruta para poder llamar a la función **store** de comentarios. Tal como se puede ver, para poder añadir un comentario debemos estar logueados, ya que sea ha incluído la ruta dentro del *middleware* de autenticación.

Añadir ruta para guardar comentarios

```php
<?php
Route::middleware(['auth'])->group(function () {
    Route::resources([
      'posts' => PostController::class,
    ]);
    Route::post('/comentarios', [ComentarioController::class, 'store'])
        ->name('comentarios.store');
});
```

Por último, dentro del controlador **ComentarioController** se debe modificar la función **store** para guardar la información enviada desde el formulario. Las modificaciones se realizan en app/Htpp/posts/ComentarioController.php

Función store del ComentarioController

```php
<?php
//...
public function store(Request $request) {
    $comentario = new Comentario();
    $comentario->texto=$request->texto;
    $post = Post::find($request->post);
    $post->comentarios()->save($comentario);
    return redirect()->route('posts.show',['post'=>$post]);
}
```

VIII

Cómo crear una API

1. Introducción

Las **API** (en inglés "*application programming interface*") son hoy en día una parte fundamental de servicios y aplicaciones. Nos permite obtener datos, comunicar aplicaciones entre sí, y realizar una separación entre la parte lógica de la aplicación y la parte visual, pudiendo ser esta última una aplicación web, una de móvil, de televisión...

Es por eso que aprender y entender cómo crear una API es una parte fundamental que todo programador debe conocer, ya que de esta manera vamos a entender de mejor manera cómo funcionan. Esto nos será muy útil también, incluso, para utilizarlas.

Por todo ello, a continuación se va a explicar cómo hacer que nuestra aplicación cuente con API, para poder ser utilizada desde otra aplicación o para ser utilizada para obtener datos desde otro tipo de interfaz. Para ello, cabe recordar que las peticiones y los resultados deben ir en formato JSON.

2. Rutas para la API

Hasta ahora hemos hecho uso del fichero `routes/web.php` para añadir rutas a nuestra aplicación. En versiones anteriores de Laravel existía un fichero `routes/api.php`, pero han decidido que por defecto no es necesario.

Para que nuestra aplicación cuente con el sistema de API debemos ejecutar:

Crear sistema API

```
root@5cff1feaf785:/var/www/html# php artisan install:api
```

Si observamos el fichero, vemos ya existe una ruta creada:

Contenido del fichero api.php

```php
<?php
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Route;

Route::get('/user', function (Request $request) {
    return $request->user();
})->middleware('auth:sanctum');
```

Si obtenemos el listado de rutas, veremos que ya existe una ruta para conocer el estado del usuario. A continuación vamos a añadir las rutas correspondientes para toda la gestión de los "posts" de nuestra aplicación.

Debemos recordar que para poder añadir las nuevas rutas, hay que incluir el controlador correspondiente, en este caso el **PostController**.

¡Atención!

Para sólo generar las rutas de la API se llama a **Route::apiresources**

Añadir nuevas rutas al fichero api.php

```
<?php
...
use App\Http\Controllers\PostController;
Route::apiresources([
    'posts' => PostController::class,
]);
```

Si ahora visualizamos el listado de rutas completo veremos las nuevas rutas. Si sólo nos interesan las rutas específicas a la API, podemos añadir un parámetro indicando sólo parte de la ruta, como se muestra a continuación:

Listar nuevas rutas

```
root@5cff1feaf785:/var/www/html# php artisan route:list --path=api

GET|HEAD        api/posts ............. posts.index › PostController@index
POST            api/posts ............. posts.store › PostController@store
GET|HEAD        api/posts/{post} ..... posts.show › PostController@show
PUT|PATCH       api/posts/{post} ..... posts.update › PostController@update
DELETE          api/posts/{post} ..... posts.destroy › PostController@destroy
GET|HEAD        api/user ..............................................
```

3. Uso de controladores para la API

Para que la modificación previa funcione es necesario modificar el controlador, ya que actualmente sólo devuelve la vista en formato código HTML. Por tanto, si queremos utilizar el mismo controlador, deberemos modificar las funciones. En el caso de la función “index” del PostController queda:

Modificar la función de PostController

```
<?php
...
public function index(Request $request) {
    $posts = Post::orderBy('created_at')->get();
    if ($request->expectsJson()) {
        return response()->json($posts);
```

```
    } else {
        return view('posts.index',['posts' => $posts]);
    }
}
```

Tal como se puede ver, la función recibe dos modificaciones:

- **Añadir parámetro "Request"**: De esta manera, podremos conocer si la petición viene desde la web, o si por el contrario se espera la respuesta en formato JSON.
- **Comprobar qué se espera**: Tal como se puede ver, se ha añadido un "if" donde se mira si la petición se espera en formato JSON ("expectsJson()"). En caso afirmativo, se devuelve la respuesta correspondiente en formato JSON.

Para comprobar que recibimos un JSON a la petición deseada, podemos ejecutar el siguiente comando:

Modificar api.php para el nuevo controlador

```
ruben@vega:~$ curl -s  http://localhost/posts
{"posts":[{"id":1,"titulo":"Primer post","texto":"Este es..."}]}
```

¡Cuidado!

Es recomendable hacer uso de controladores específicos para la API

3.1. Crear controladores específicos

Debido a que las API se suelen versionar, es recomendable mantener los controladores de la web y de la API separados. Esto permite seguir el principio ***KISS*** (*Keep It Simple, Stupid!*). De esta manera se va a poder realizar modificaciones en un apartado de nuestra aplicación sin temer que podemos romper otra parte.

Es por ello, que lo ideal es crear controladores específicos para las funcionalidades que va a tener la API, y que se encuentren separados. Para ello realizaremos lo siguiente:

- **Deshacer los cambios** de la función "index" vistos en el apartado anterior.
- **Crear nuevo controlador** que será específico para la API:

  Crear nuevo PostController exclusivo para la API

  ```
  # php artisan make:controller API/PostController --api --model=Post
  ```

  Es necesario explicar lo siguiente:

  - **"API/PostController"**: Esto indica cuál es la ruta donde se creará el fichero, que en este caso es `app/Http/Controllers/API/PostController.php`.

- **`--api`**: Este parámetro va a generar un controlador que carece de las funciones "create" y "edit", ya que no son necesarias en una API, dado que son exclusivas a visualizar los formularios en un interfaz web.
- **`--model=Post`**: Para que el nuevo fichero del controlador ya tenga el include del modelo necesario.

- **Modificar la ruta para la API** y que de esta manera haga uso del nuevo controlador exclusivo. El cambio es el siguiente:

Modificar api.php para el nuevo controlador

```
<?php
...
use App\Http\Controllers\API\PostController;
```

- Modificar nuevo controlador, para que devuelva los datos correspondientes:

Modificar el nuevo controlador PostController

```
<?php
...
use Illuminate\Http\Response;
public function index(){
    $posts = Post::orderBy('created_at')->get();
    return response()->json(['posts'=>$posts])
        ->setStatusCode(Response::HTTP_OK);
}
```

En este caso no hemos realizado ninguna comprobación, pero en una API de verdad debemos comprobar si se han encontrado resultados, y dependiendo de ellos devolver un **estado de respuesta distinto**.

La librería contiene una gran cantidad de variables para hacer referencia a los posibles códigos que podemos devolver, así como el texto que acompañan al código.

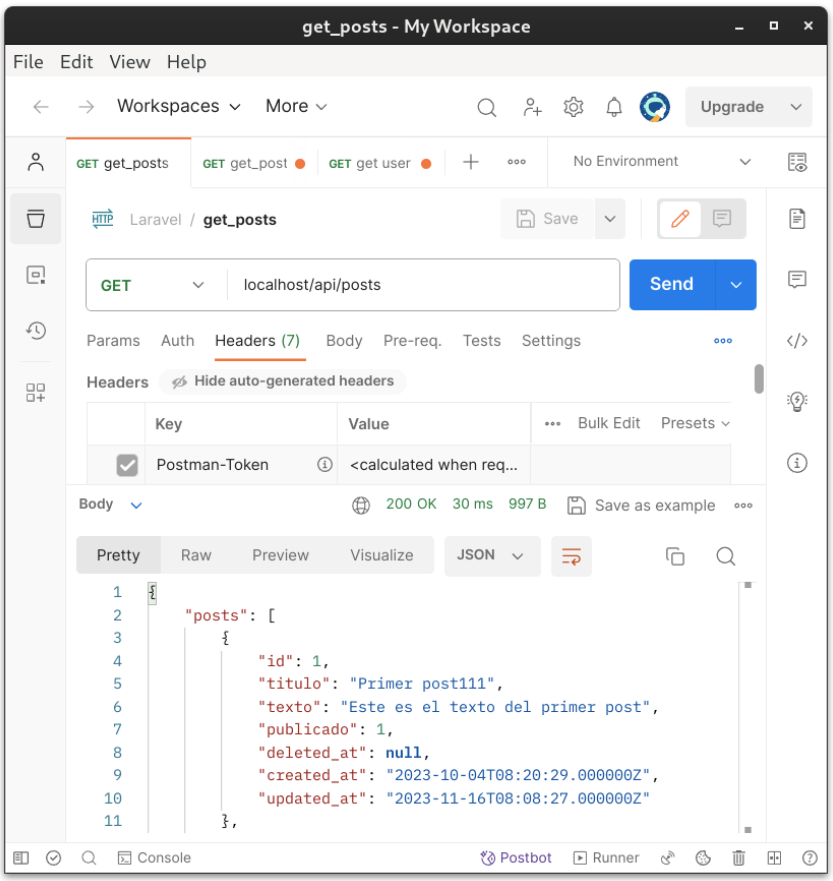
4. Comprobar funcionamiento

Es momento de comprobar que todo funciona de manera correcta, y para ello debemos realizar una petición a la URL http://localhost/api/posts teniendo en cuenta el ejemplo que hemos estado realizando.

Para realizar la prueba podemos hacerlo de distintas formas, cada una de ellas dependiendo de la motivación que tengamos:

- Utilizando un interfaz gráfico como Postman o Firecamp (versión web o escritorio), que nos va a

facilitar hacer peticiones complejas a la API.



- Desde el propio **navegador web**. Veremos los datos JSON devueltos en formato texto directamente. Para comprobar que funciona, puede ser más que suficiente.
- Desde una **consola Linux**, haciendo uso del comando **curl**, podemos también comprobar de manera rápida si el *endpoint* está funcionando:

Modificar api.php para el nuevo controlador

```
ruben@vega:~$ curl -s  http://localhost/api/posts
{"posts":[{"id":1,"titulo":"Primer post111","texto":"Este es..."}]}
```

Si queremos tener un resultado más visual, podremos hacer uso del comando "**jq**", que deberemos instalarlo. De esta manera, podremos hacer:

Modificar api.php para el nuevo controlador

```
ruben@vega:~$ curl -s  http://localhost/api/posts | jq
{
  "posts": [
    {
      "id": 1,
      "titulo": "Primer post",
```

```
        "texto": "Este es el texto del primer post",
        "publicado": 1,
        "deleted_at": null,
        "created_at": "2024-10-29T17:09:56.000000Z",
        "updated_at": "2024-10-29T18:09:56.000000Z"
    }
  ]
}
```

5. Gestionar excepciones

Laravel gestiona las excepciones generando por defecto un *stacktrace* con la excepción y un error 404, por lo que en caso de existir alguna excepción en la API, el errorcode será correcto, pero devolverá información interna del *framework*.

Vamos a utilizar como ejemplo la función **show** del nuevo controlador creado. El código para esta función es:

Modificar la función show de API/PostController

```php
<?php
...
public function show(Post $post) {
    return response()->json($post)->setStatusCode(Response::HTTP_OK);
}
```

Si ahora realizamos la petición a la API de un **id** existente, nos devolverá el JSON con el contenido. En cambio, con un **id** no existente **obtendremos un json con el *stacktrace* de la excepción**.

Ejercicio

Realizar petición a la API de un ID de un post inexistente y ver el resultado.

Para evitar eso, debemos modificar el fichero `bootstrap/app.php` y asegurar que le añadimos lo siguiente:

Modificar bootstrap/app.php

```php
<?php
...
use Illuminate\Http\Request;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;
```

```
...

->withExceptions(function (Exceptions $exceptions) {
  $exceptions->render(function (NotFoundHttpException $e, Request $request) {
      if ($request->is('api/*')) {
          return response()->json([
              'message' => 'Record not found.'
          ], 404);
      }
  });
})->create();
```

6. Autenticación

Para poder realizar ciertas acciones a través de la API es lógico pensar que también deberemos estar autenticados, y para eso es necesario asegurar que al acceder a las rutas lo estemos. Para todo ello, vamos a crear un controlador propio donde tener en cuenta los datos que se envían al acceder a la API.

La idea general es que una aplicación al hacer uso de una API debe tener en cuenta:

- Si el usuario no está registrado, poder registrarse.
- Si el usuario está registrado, se podrá loguear con lo que recibirá un ***token***.
- A partir de este momento, cada acción que quiera realizar, deberá ir acompañado del token para demostrar que está autenticado.
- Los *tokens* pueden tener una vida útil. Por lo tanto, si el token expira, deberá volver a loguearse.

6.1. Crear controlador de autenticación

Se va a crear un controlador propio para tener el control de las acciones que se pueden realizar a través de la API, y así asegurar cuál es el estado de los tokens y/o del usuario que pide realizar una acción.

Para crear un controlador propio que sólo será usado para la API:

Crear nuevo controlador

```
root@5cff1feaf785:/var/www/html# php artisan make:controller API/AuthController
INFO  Controller [app/Http/Controllers/API/AuthController.php] created successfully.
```

En este controlador vamos a tener tres funciones:

- **register**: para que el usuario se pueda registrar.
- **login**: para que el usuario se pueda loguear y recibir un token.
- **logout**: para que el usuario se pueda desloguear y de esta manera el token se revoque.

El controlador, quedaría de la siguiente manera.

¡Cuidado!

Es importante entender qué hace cada una de las funciones

Nuevo AuthController para la API

```php
<?php
namespace App\Http\Controllers\API;
use App\Http\Controllers\Controller;
use Illuminate\Http\Request;

use App\Models\User;
use Illuminate\Support\Facades\Hash;
use Illuminate\Validation\ValidationException;
use Illuminate\Http\Response;

class AuthController extends Controller {

    public function register(Request $request){
        $validatedData = $request->validate([
            'name' => 'required|string|max:255',
            'email' => 'required|string|email|max:255|unique:users',
            'password' => 'required|string|min:8',
        ]);

        $user = User::create([
            'name' => $validatedData['name'],
            'email' => $validatedData['email'],
            'password' => Hash::make($validatedData['password']),
        ]);

        return response()->json([
            'name' => $user->name,
            'email' => $user->email,
        ])->setStatusCode(Response::HTTP_CREATED);
    }

    public function login(Request $request){
        $request->validate([
            'email' => 'required|email',
            'password' => 'required',
            'device_name' => 'required',
        ]);

        $user = User::where('email',  $request->email)->first();
```

```
        if (! $user || !Hash::check($request->password, $user->password)){
            return response()->json([
                'message' => ['Username or password incorrect'],
            ])->setStatusCode(Response::HTTP_UNAUTHORIZED);
        }
        // FIXME: queremos dejar más dispositivos?
        // $user->tokens()->delete();

        return response()->json([
            'status' => 'success',
            'message' => 'User logged in successfully',
            'name' => $user->name,
            'token' => $user->createToken($request->device_name)->plainTextToken,
        ]);
    }

    public function logout(Request $request){
        $request->user()->currentAccessToken()->delete();
        return response()->json([
            'status' => 'success',
            'message' => 'User logged out successfully'
        ])->setStatusCode(Response::HTTP_OK);
    }
}
```

Es importante notar un comentario que se ha dejado en la función "**login**". Dependiendo de si queremos que la API permita tener varios tokens para un mismo usuario o no (posibles logins desde distintos dispositivos), deberemos dejar comentado o descomentar la línea indicada.

Tal como se puede ver, a la hora de realizar la acción de ***login***, se llama a `createToken($request->device_name)`, por lo que es necesario que el Modelo tenga acceso a esa función. Por ello, nos aseguramos que `app/Models/User.php` cuente con:

Modificar modelo User

```
<?php
namespace App\Models;
//...
use Laravel\Sanctum\HasApiTokens;

class User extends Authenticatable
{
    /** @use HasFactory<\Database\Factories\UserFactory> */
    use HasApiTokens, HasFactory, Notifiable;
```

```
    //...
}
```

Para que estas funciones entren en juego, debemos modificar el fichero de rutas `api.php`.

Rutas de autenticación para la API

```php
<?php
use App\Http\Controllers\API\AuthController;
//...
Route::post('/register', [AuthController::class, 'register']);
Route::post('/login', [AuthController::class, 'login']);
Route::post('/logout', [AuthController::class, 'logout'])
    ->middleware('auth:sanctum');
```

6.2. Modificar rutas

Tal como hemos hecho anteriormente, para que la aplicación funcione bajo el sistema de autenticación, y que automáticamente nos indique que no estamos autenticados, debemos realizar la modificación del fichero de rutas `api.php`

De manera similar a la aplicación web, deberemos indicar qué rutas queremos que se puedan obtener sin estar autenticado y cuáles no.

Rutas autenticadas para la API

```php
<?php
//...
Route::middleware(['auth:sanctum'])->group(function () {
    Route::apiresources([
        'posts' => PostController::class,
    ]);
});

Route::controller(PostController::class)->group(function () {
    Route::get('/posts', 'index')->name('posts.index');
    Route::get('/posts/{post}', 'show')->name('posts.show');
})->withoutMiddleware(['auth:sanctum']);
```

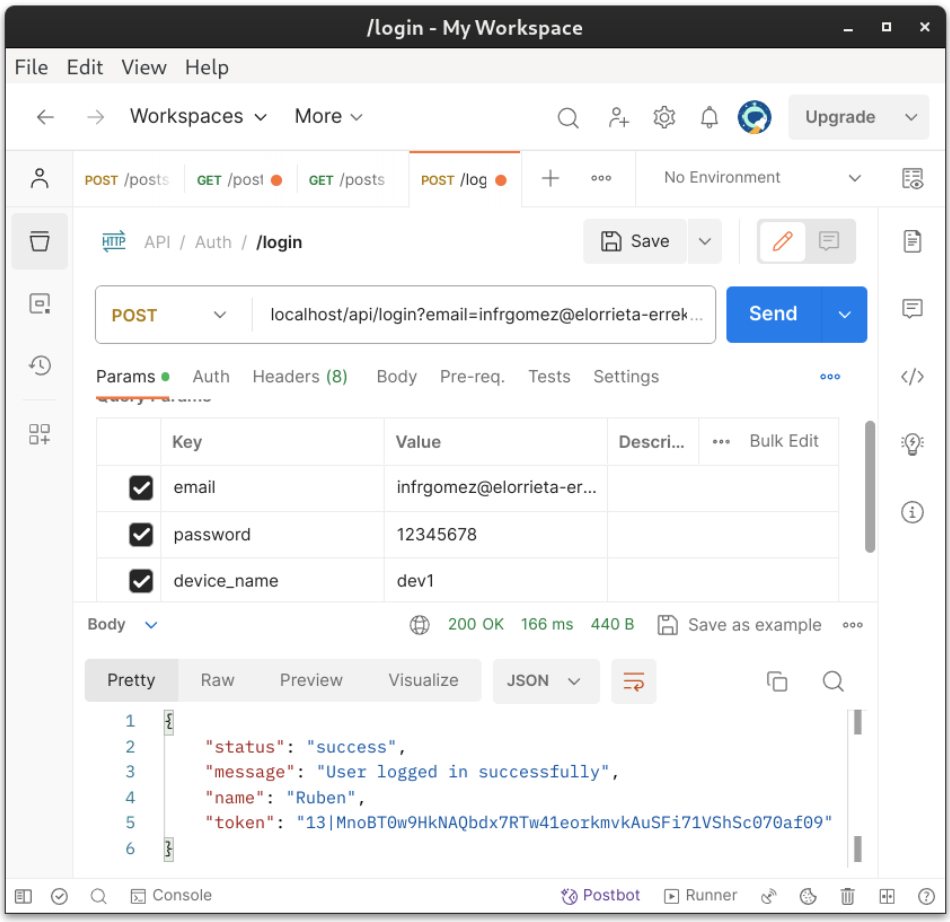
6.3. Pruebas de funcionamiento

Para que la autenticación funcione, podemos realizar pruebas a través de Postman, de manera similar a como hemos hecho anteriormente. Ahora hay que tener en cuenta que la petición va a ser de tipo **POST**, y

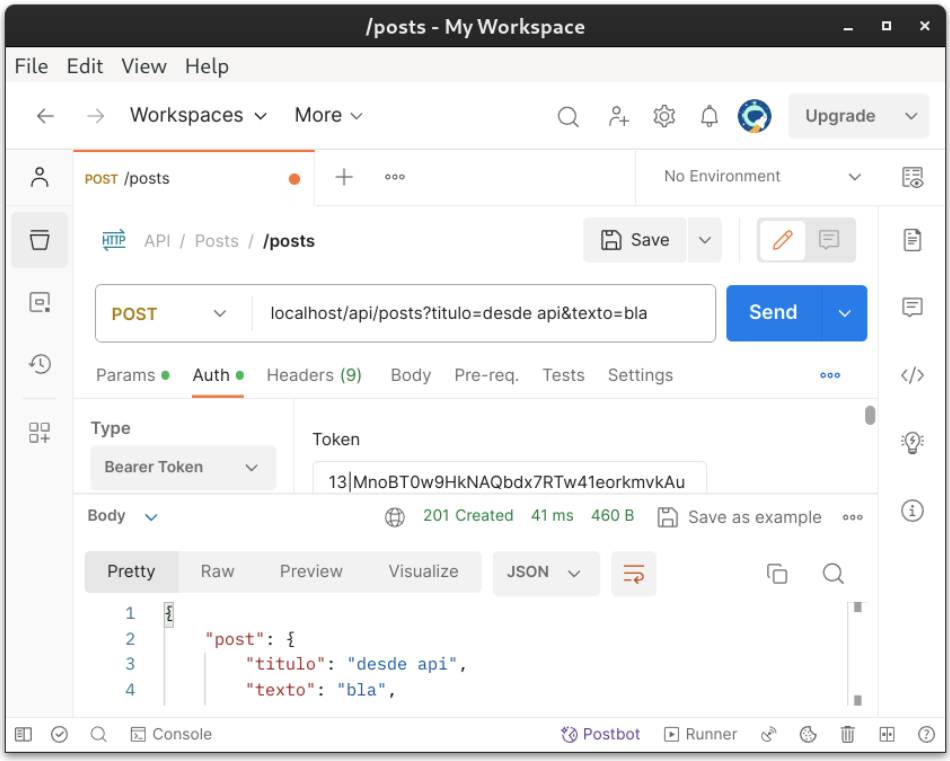
dado que queremos crear un *post*, tendremos que realizar el paso de parámetros.

Por lo tanto, los pasos que debemos realizar son:

- Realizar prueba de login. Debemos pasar los parámetros y obtendremos el token y una serie de datos extra, que podremos utilizar en nuestra aplicación.
    - email
    - password
    - device_name



- Con ese token podremos realizar la prueba de añadir un nuevo *post* a través de la API.



Deberemos indicar:

- Los parámetros necesarios para la creación del *post*: “título”, “texto” y “publicado”.
- El tipo de token de autenticación en la pestaña “Auth” de Postman tiene que ser de tipo “**Bearer Token**”.

7. Visualizar API

Suele ser habitual tener un interfaz donde se muestra el funcionamiento de nuestra API, o cuáles son los *endpoints* de la misma. Es decir, qué URL se pueden consultar, qué método hay que utilizar, si es necesario el paso de parámetros, ...

Hoy en día existe la especificación OpenAPI para la generación de la API, y sobre ella existen distintos interfaces web. Uno de los interfaces más utilizados es Swagger UI que nos muestra un bonito interfaz y a la vez es posible utilizarlo para realizar consultas a la API.

Para poder instalarlo en nuestro proyecto Laravel, necesitamos realizar la instalación de unas dependencias y la posterior instalación en el proyecto.

Instalación de dependencias

```
root@5cff1feaf785:/var/www/html# composer require "darkaonline/l5-swagger"
root@5cff1feaf785:/var/www/html# php artisan vendor:publish \
    --provider "L5Swagger\L5SwaggerServiceProvider"
```

Para poder generar el interfaz de manera correcta añadimos comentarios a las funciones. En uno de los

controladores añadiremos la siguiente cabecera, que nos va a servir para definir el tipo de autenticación:

Cabecera para la API

```php
<?php
//...
/**
* @OA\Info(title="API", version="1.0"),
* @OA\SecurityScheme(
*     in="header",
*     scheme="bearer",
*     bearerFormat="JWT",
*     securityScheme="bearerAuth",
*     type="http",
* ),
*/
```

Para documentar la función **index**, encima de ella añadiremos lo siguiente. Hay que darse cuenta que en este caso sólo hemos documentado la respuesta “200”.

Comentario para /api/posts

```php
<?php
//...
/**
* @OA\Get(
*     path="/api/posts",
*     summary="Mostrar posts",
*     @OA\Response(
*         response=200,
*         description="Mostrar todos los posts."
*     ),
*     @OA\Response(
*         response="default",
*         description="Ha ocurrido un error."
*     )
* )
*/
public function index(){
//...
```

Para documentar la función de guardar un *post* desde la API, usaremos los siguientes comentarios:

Comentario para función POST /api/posts

```php
<?php
//...
    /**
* @OA\Post(
*     path="/api/posts",
*     summary="Create a post",
*     @OA\Parameter(
*         name="titulo",
*         in="query",
*         description="The title of the post",
*         required=true,
*         @OA\Schema(
*             type="string"
*         )
*     ),
*     @OA\Response(
*         response=200,
*         description="successful operation",
*         @OA\JsonContent(
*             type="string"
*         ),
*     ),
*     @OA\Response(
*         response=401,
*         description="Unauthenticated"
*     ),
*     security={
    *         {"bearerAuth": {}}
    *     }
* )
*/
public function store(Request $request){
```

Por último, vamos a poner otro ejemplo, para documentar el obtener un *post* concreto. Para ello los comentarios serán:

Comentario para función GET

```php
<?php
//...
/**
* @OA\Get(
*     path="/api/posts/{id}",
*     summary="Mostrar un post concreto",
*     @OA\Parameter(
*          name="id",
*          description="Project id",
*          required=true,
*          in="path",
*          @OA\Schema(
*              type="integer"
*          )
*     ),
*     @OA\Response(
*          response=200,
*          description="Mostrar el post especificado."
*     ),
*     @OA\Response(
*          response="default",
*          description="Ha ocurrido un error."
*     )
* )
*/
public function show(Post $post){
```

No se han añadido todos los parámetros en todos los casos, ya que resulta redundante y es fácil añadir los que faltan.

Para conocer todas las funcionalidades de los comentarios, es recomendable mirar la documentación. Desde aquí nos mostrará el enlace para la documentación oficial de Swagger PHP, o lo que nos puede resultar más interesante, que es un conjunto de ejemplos junto con el resultado en forma de web.

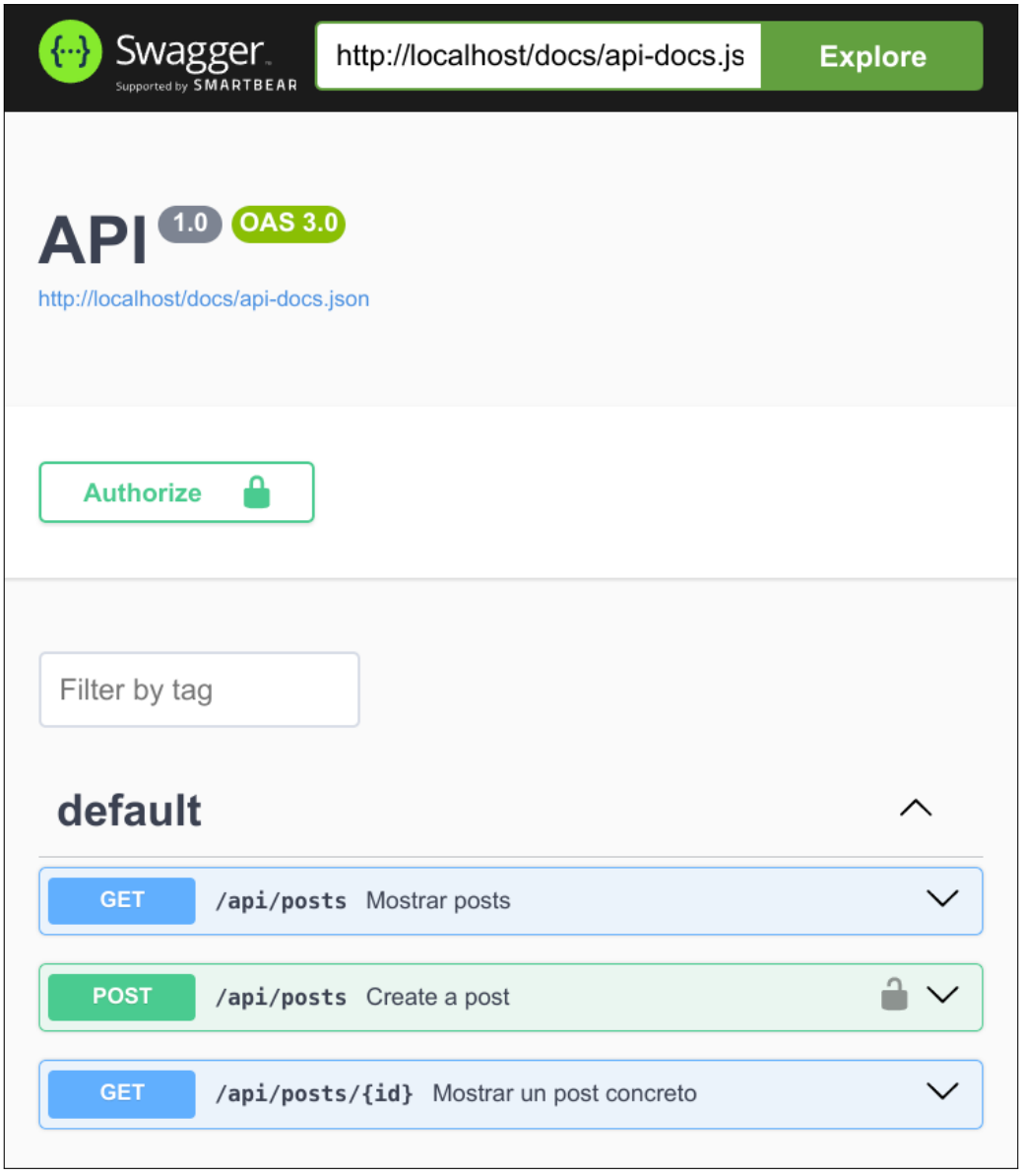
Tras esto, ejecutaremos el comando que recorrerá los controladores para generar el fichero `storage/api-docs/api-docs.json`. Este es el fichero que el interfaz web tendrá en cuenta a la hora de generar la web que podemos ver a continuación.

El comando es el siguiente:

Generamos el fichero json para Swagger

```
root@5cff1feaf785:/var/www/html# php artisan l5-swagger:generate
Regenerating docs default
```

Tras ejecutar el comando anterior, si vamos a la url http://localhost/api/documentation tendremos acceso y veremos el interfaz para nuestra API:



Ejercicio

Completar los comentarios añadiendo los parámetros que faltan a la función de crear y los necesarios para la función de borrado de posts.

IX

Puesta en producción

1. Directorios ignorados del proyecto

Al crear el proyecto, se nos ha creado un fichero `.gitignore` en el que aparecen distintos directorios que están ignorados a la hora de añadir el proyecto a un repositorio Git.

Estos ficheros y directorios no es necesario que estén subidos al repositorio, ya sea porque son ficheros que deben ser generados o porque contienen configuración y contraseñas.

A continuación un listado de algunos ficheros y directorios y la explicación de por qué están ignorados.

- `node\_modules` : Es el directorio donde se guardan los paquetes, y sus dependencias, que se han descargado a través del gestor de paquetes NPM. Esta configuración aparece en el fichero de configuración `package.json` .
- `public/build` : En este directorio se guardan los ficheros (javascript y css, entre otros) que se generan a través del comando `npm run build` . Este comando debe ejecutarse antes de la puesta en producción
- `vendor` : En este caso es el directorio de las librerías que son necesarias a través del gestor de paquetes Composer. Estas librerías, y sus dependencias, son las que el proyecto Laravel necesita y aparecen en el fichero `composer.json` .
- `.env` : Es el fichero de configuración general de la aplicación y donde aparecen los servicios y sus contraseñas. Nunca debería subirse un fichero de configuración con contraseñas al repositorio. Para asegurar que no faltan configuraciones, se podría subir un fichero igual al `.env`, pero sin que aparezcan las contraseñas.
- `.vscode` : No es recomendable subir ficheros de configuración de IDEs, ya que cada desarrollador puede querer configuraciones propias.

2. Poner proyecto en producción

La puesta en producción de un proyecto es un punto crítico, ya que cualquier fallo o problema puede ocasionar que nuestra aplicación no funcione de manera correcta. Es por eso, que siempre debería haber un “guión” para indicar los pasos a realizar y que no se nos olvide ninguno durante la puesta en producción.

Los pasos a seguir dependerán de la aplicación o servicio que vayamos a poner en producción. Para el caso de Laravel, y tal como hemos estado realizando el desarrollo, serán los siguientes:

- Clonar el proyecto.
- Crear contenedores temporales.
- Realizar la instalación de las dependencias necesarias.
- Ejecutar la construcción de los ficheros *assets* necesarios (javascripts y css).

- Crear contenedores finales.
- Ejecutar *migrations* y/o *seeds* necesarios.
- Testear que todo funciona de manera correcta.

Para esta explicación se partirá de una instalación de Ubuntu LTS con Docker y Docker Compose instalado. Hay que tener en cuenta que la puesta en producción utilizando otro sistema de instalación puede variar en alguno de los pasos.

¡Atención!

Los pasos de la puesta en producción pueden variar dependiendo del sistema de instalación utilizado, pero serán parecidos a los explicados aquí.

2.1. Clonar el proyecto

Este paso no tiene mayor dificultad, ya que se presupone que tenemos nuestro proyecto en un repositorio Git en algún tipo de plataforma centralizada (como puede ser Github o GitLab).

El clonado del repositorio lo realizaremos como cualquier otro proyecto, por lo que no se explicará cómo realizarlo. Hay que recordar que en este clonado faltarán ficheros y directorios tal como se ha explicado previamente.

2.2. Crear contenedor temporal

Para poder crear y levantar los contenedores necesarios necesitamos del fichero de configuración donde están las contraseñas. Para este ejemplo, copiaremos el fichero `.env.example` a `.env` y modificaremos los apartados oportunos (normalmente, el apartado de la base de datos).

¡Cuidado!

Es importante asegurar que el fichero .env esté configurado de manera correcta.

Ahora, levantaremos un contenedor intermedio para instalar las dependencias necesarias a través de **composer**. Para ello, debemos ejecutar lo siguiente:

Levantamos contenedor temporal

```
ruben@vega:~/proyecto_produccion$ docker run --rm  -v "$(pwd)":/opt -w /opt \
  -it laravelsail/php83-composer:latest /bin/bash

root@b58c9150c04d:/opt# composer install

root@b58c9150c04d:/opt# php artisan key:generate
```

Los últimos dos comandos ejecutados se realizan dentro del contenedor y sirven para instalar las dependencias y para generar la variable de entorno **APP_KEY** dentro del fichero `.env`.

Tras esto, todos los paquetes necesarios de Laravel deberían estar instalados. Al terminar la ejecución, saldremos de este conendor y automáticamente se borrará, ya que no es necesario volver a usarlo.

2.3. Crear contenedor final

Por último, levantaremos el que será el contenedor de Laravel de igual manera que hemos hecho durante el desarrollo:

Levantamos el contenedor de Laravel

```
ruben@vega:~/proyecto_produccion$ ./vendor/bin/sail up
```

2.3.1. Instalación de las dependencias finales

Y tenemos que entrar en el contenedor para realizar las últimas tareas para asegurar que el proyecto funciona:

Tareas finales

```
root@469e75d4a713:/var/www/html# npm install
added 37 packages, and audited 38 packages in 1s
...
Run `npm audit` for details.
root@469e75d4a713:/var/www/html# npm run build
> build
> vite build
...
✓ built in 3.85s
root@469e75d4a713:/var/www/html# chmod 777 -R storage/
```

A continuación la explicación de los comandos ejecutados, ya que es importante entender qué se ha realizado:

- `npm install` : instalamos las depedencias necesarias a través del gestor de dependencias NPM.
- `npm run build` : es necesario generar los assets de javascript y CSS para que la aplicación funcione.
- `chmod 777 -R storage/` : Modificamos los permisos de ciertos ficheros para que el servidor web pueda escribir datos.

Tras esto, el contenedor debería estar funcionando y ya sólo queda lanzar los *migration* y los *seeder* correspondientes.

X

Anexo

1. Repositorio

Para poder seguir los pasos realizados en este documento, a continuación se incluye un repositorio de GitHub en el que en cada *commit* se ha explicado lo realizado.

GitHub

https://github.com/yuki/laravel-example-app

Se ha tratado de hacer un *commit* por cada paso explicado en este documento, aunque en algunos casos se han podido separar en varios. Es conveniente leer el comentario añadido a cada *commit* y ver los ficheros que han sido modificados, así como las modificaciones realizadas. De esta manera, se entenderá mejor cada paso.