LARAVEL Framework - Tutorial 06 - Notas sobre version 8.x de Laravel

Sumario

- 1 Script de instalación de apps y configuración de una máquina Debian en Amazon EC2.
- 2 Configuración de Nginx para el nuevo sitio de Laravel
- 3 Instalación de Laravel para máquina Debian en Amazon EC2
- 4 Instalación de NodeJS y npm
- 5 Configuración de User Interface UI en Laravel 8
- 6 Validación de datos recibidos desde formularios en Laravel 8
- 7 Redirecciones en Laravel
- 8 Layouts en Laravel - Plantillas
- 9 Seeders
- 10 Instalación de CKEditor
- 11 Estilos CSS
- 12 Configuración de Nginx para permitir subidas de archivos hasta 25M
- 13 Localización de mensajes en Español en Laravel 8
- 14 Autenticación en Laravel
- 15 Envío de correos en Laravel
- 16 Construir una API REST y Laravel Passport
    - 16.1 Información e instalación de extensiones recomendables
    - 16.2 Creación del controlador para gestionar las rutas de la API
    - 16.3 Creación de los métodos de ejemplo de una Persona para la API
    - 16.4 Creación de un Trait para implementar función que devuelva el JSON de respuesta
    - 16.5 Proteger el acceso a la API REST mediante autenticación
- 17 Notas varias sobre Laravel 8.x
- 18 Laravel CRUD generator

Script de instalación de apps y configuración de una máquina Debian en Amazon EC2.

- La máquina en Amazon EC2 se crea en 1 minuto.
- Con este script se instala y configura en 5 minutos 20 segundos de reloj.
- Modo de uso:

Desde la shell de nuestra máquina Debian en Amazon EC2 con el usuario **admin**, **copiar y ejecutar el siguiente comando**:

```
wget https://raw.githubusercontent.com/raveiga/aws-install/main/aws_instalacion.sh -O aws_instalacion.sh && sudo bash aws_instalacio
```

Configuración de Nginx para el nuevo sitio de Laravel

Si habéis ejecutado el script de instalación y configuración, ya se han creado las carpetas de los dominios.

Introducción a MVC:

- https://codigofacilito.com/articulos/mvc-model-view-controller-explicado

Creación de nuevo dominio virtual en el servidor AWS:

- Creamos un nuevo dominio en freeddns.org o similar apuntando a la dirección IP de nuestro servidor (yo he creado el dominio laravel.freeddns.org).
- Creamos esta estructura en /var/www: mkdir -p /var/www/laravel.freeddns.org/public
- Copiamos la carpeta default por defecto con otro nombre (yo le he puesto el nombre de mi dominio al fichero de configuración):

```
cp /etc/nginx/sites-available/default laravel.freeddns.org
nano /etc/nginx/sites-available/laravel.freeddns.org
```

Si queremos tener **certificado SSL** en el nuevo dominio ejecutamos el siguiente comando:

```
letsencrypt
```

Podemos indicarle que si haga Redirect de http a https.

Cómo servir páginas con PHP desde la línea de comandos:

```
php -S localhost:3000  (para servir el directorio actual como root): http://localhost:3000
php -S localhost:3000 -t public (para servir el directorio public como root): http://localhost:3000
```

Instalación de Laravel para máquina Debian en Amazon EC2

Vamos a configurar primeramente el servidor Nginx. Aquí se muestra un ejemplo de configuración.

Se incluye también la configuración de Letsencrypt:

Contenido:

```
sudo nano /etc/nginx/sites-enabled/laravel.freeddns.org:

server {
listen 80;
server_name laravel.freeddns.org;
root /var/www/laravel.freeddns.org/public;

add_header X-Frame-Options "SAMEORIGIN";
add_header X-XSS-Protection "1; mode=block";
add_header X-Content-Type-Options "nosniff";

index index.php index.html index.htm;

charset utf-8;

location / {
try_files $uri $uri/ /index.php?$query_string;
}

location = /favicon.ico { access_log off; log_not_found off; }
location = /robots.txt { access_log off; log_not_found off; }

error_page 404 /index.php;

location ~ \.php$ {
fastcgi_pass unix:/var/run/php/php7.4-fpm.sock;
fastcgi_index index.php;
fastcgi_param SCRIPT_FILENAME $realpath_root$fastcgi_script_name;
include fastcgi_params;
}

location ~ /\.(?!well-known).* {
deny all;
}
}

# Reiniciamos el servicio de nginx:

sudo service nginx restart

# Ejecutamos letsencrypt para obtener certificado SSL para nuestro dominio y le indicamos que haga Redirect de peticiones http a htt

sudo certbot

# Reiniciamos el servicio y a continuación pasamos a ejecutar la instalación de Laravel.
```

Para instalar Laravel, primeramente tendremos que **instalar composer**. Realizaremos los siguientes pasos:

Extraído de: https://linuxize.com/post/how-to-install-and-use-composer-on-ubuntu-20-04

```
sudo apt update
sudo apt install wget php-cli php-zip unzip php-xml php-mbstring
```

```
wget -O composer-setup.php https://getcomposer.org/installer
sudo php composer-setup.php --install-dir=/usr/local/bin --filename=composer

# A continuación vamos a instalar laravel en una carpeta dentro de /var/www:
# (sustituir laravel.freeddns.org por vuestra carpeta personal)

# Ponemos propietario y grupo en la carpeta /var/www:
chown admin:www-data /var/www -R

# Accedemos a la carpeta /var/www:
cd /var/www

# Si necesitamos borrar la carpeta antigua:
sudo rm -rf /var/www/laravel.freeddns.org

# Instalamos laravel en la carpeta laravel.freeddns.org:
composer create-project laravel/laravel laravel.freeddns.org

# Por último ponemos los siguientes permisos:
# Cambiamos el usuario y grupo de laravel:
sudo chown admin:www-data laravel.freeddns.org/ -R

# Cambiamos los permisos de las carpetas storage y cache:
sudo chmod -R 775 /var/www/laravel.freeddns.org/storage/
sudo chmod -R 775 /var/www/laravel.freeddns.org/bootstrap/cache

# Probamos a acceder a nuestro dominio, por ejemplo:
https://laravel.freeddns.org

# y comprobamos que se muestra una página de ejemplo de Laravel.
```

Vamos a descargarnos Laravel en nuestro equipo local: En lugar de sincronizar el servidor remoto con el local, vamos a hacerlo de otra forma mucho más rápida. 1.- Nos descargamos el siguiente archivo desde https://veiga.freeddns.org/laravel.zip 2.- Lo descomprimimos en la carpeta que queramos, por ejemplo al lado de veiga.freeddns.org 3.- Renombramos la carpeta laravel a laravel.freeddns.org (por ejemplo) 4.- Abrimos la carpeta laravel.freeddns.org con el VSCode 5.- Configuramos el FTP del VSCode para la nueva carpeta 5.1.- Pulsamos CTRL + MAYUS + P y tecleamos SFTP (seleccionamos SFTP:Config) 5.2.- Configuramos los siguientes parámetros en el fichero .json del FTP:

```
{
    "name": "Servidor Amazon Laravel.freeddns.org",
    "host": "laravel.freeddns.org",
    "protocol": "sftp",
    "port": 22,
    "username": "admin",
    "remotePath": "/var/www/laravel.freeddns.org/",
    "uploadOnSave": true,
    "privateKeyPath": "D:\\Documentos\\_Curso Actual\\_Curso 20-21\\DWCS\\ClavesAmazon\\amazon.ppk",
    "syncOption": {
        "delete": true
    },
    "ignore": [
        ".vscode",
        ".git",
        ".DS_Store"
    ]
}
```

Instalación de NodeJS y npm

Para trabajar correctamente con Laravel necesitamos instalar también NodeJs y su gestor de dependencias npm.

Para ello realizaremos lo siguiente como el usuario **admin**:

```
cd

curl -sL https://deb.nodesource.com/setup_14.x -o nodesource_setup.sh

sudo bash nodesource_setup.sh
```

```
sudo apt install nodejs

Cuando termine de instalar tendremos la última versión 14.x disponible:

node -v

npm -v
```

Configuración de User Interface UI en Laravel 8

- A partir de Laravel 6, éste no decidirá por ti con cuál pre-procesador y framework de Javascript y CSS se va a trabajar en tu aplicación.
- Es por ello que fue movida toda la funcionalidad para el frontend (scaffolding) que venía por defecto (Bootstrap y Vue) en el framework a un nuevo paquete de composer llamado laravel/ui.
- Tendremos que realizar los siguientes cambios.

Entramos a la carpeta de nuestra instalación de Laravel:

```
cd /var/www/laravel.freeddns.org

# Ejecutar el comando: composer require laravel/ui --dev

# Con el comando anterior ya tenemos disponible los comandos Artisan para agregar Bootstrap, Vue o React.

# En nuestro caso vamos a decirle a Boostrap que queremos trabajar con Vue y Bootstrap en el fronted y para ello lo haremos con:

cd /var/www/laravel.freeddns.org

php artisan ui vue

# Otros comandos disponibles, pero que no vamos a instalar ahora:

php artisan ui bootstrap

php artisan ui react

# Ahora solamente nos falta descargar los comandos de

# A continuación tendremos que ejecutar los paquetes necesarios (del frontend) con el gestor de paquetes NPM. Este gestor de paquete

# Instalamos las dependencias indicadas en package.json:

npm install

# Compilamos los archivos descargados que generarán los compilados en las carpetas dentro de public: css y js correspondientes.

npm run dev
```

Resumiendo:

- Con composer instalamos las dependencias del servidor - Con npm las dependencias del cliente.

Para usar los estilos de Bootstrap y las librerías de Vue y Bootstrap, las incluiremos en nuestras vistas con:

```
<link rel="stylesheet" href="{{ asset('css/app.css') }}">
<script src="{{ asset('js/app.js') }}"></script>
```

Validación de datos recibidos desde formularios en Laravel 8

- Documentación Validación: **https://laravel.com/docs/8.x/validation**
- Reglas de validación disponibles: **https://laravel.com/docs/8.x/validation#available-validation-rules**
- Cuando se valida si hay errores vuelve automáticamente al formulario del que proviene.
- Los datos procedentes del formulario: **https://laravel.com/docs/8.x/requests#input**

```
# Este comando para y muestra el contenido de la variable que se pasa como parámetro:
dd(  $request->all() )
```

```
# Para el acceso a los campos recibidos en el formulario:

$request->input("apellidos")  (para GET y POST)
$request->query("campo") (solamente para GET)
$request->input('objeto.nombre')  (para acceder a datos recibidos por JSON)
$request->campo (para acceder directamente al campo)

# También hay otra opción que sería el método: request("campo") y no se necesitaría incluir la clase Request al principio del archiv
# Cuando se valida y hay errores regresa al formulario automáticamente pasando los Input enviados.
# En el controlador para probar: return back()->withInput() para probar el {{ old('apellidos') }}
```

Las validaciones se pueden gestionar de 2 formas:

- En el propio método del controlador:

Ejemplo:

```
public function store(Request $request)
{
    $validated = $request->validate([
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ]);

    // The blog post is valid...
}


# En la vista mostraríamos los errores:


<h1>Create Post</h1>

@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif



# Para mostrar un error determinado debajo de un campo:

@error('apellidos')
  <small class="text-danger">{{ $message }}</small>
@enderror

# Para mensaje Flash:
  @if (session('estado'))
        <div class="alert alert-success">
            {{ session('estado') }}
        </div>
   @endif

# O también creando una Request (para escenarios más complejos de validaciones):

php artisan make:request MétodoModeloTipopeticion  (método store, create, etc... ) tipo peticion Post Get, etc..

# Ejemplo:
php artisan make:request StorePersonaPost

# Los Request aparecerán en app/Http/Requests
# Dentro del método authorize del Request, lo cambiaremos a return true; (temporalmente por que no estamos gestionando las autorizac
# Nos quedará inyectar la validación en el método Store, por ejemplo:
...function store(StorePersonaPost $request)

#Los datos validados por este método los tenemos en $request->validated().
```

Redirecciones en Laravel

- **https://laravel.com/docs/8.x/responses#redirects**

```
return back(); nos manda a la página anterior.
return back()->withInput(); nos manda a la página anterior con los datos de entrada.
return redirect(route('personas.create'))->with('estado', 'Dado de alta correctamente');
return back()->with('estado', 'Dado de alta correctamente');
```

Layouts en Laravel - Plantillas

- **https://laravel.com/docs/8.x/blade#building-layouts**
- Ejemplo de Plantilla Master: layouts.master

```
<html>
    <head>
        <title>App Name - @yield('title')</title>
    </head>
    <body>
        @section('sidebar')
            This is the master sidebar.
        @show

        <div class="container">
            @yield('content')
        </div>
    </body>
</html>
```

- En la plantilla **Master** - con **@yield** se indica para mostrar contenido de una sección determinada.
- En la plantilla **Hija**- con **@section** se indica el contenido que se pasará a la zona dónde se puso @yield

- Ejemplo de Plantilla Hija:

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

@section('title', 'Page Title')

@section('sidebar')
    @parent

    <p>This is appended to the master sidebar.</p>
@endsection

@section('content')
    <p>This is my body content.</p>
@endsection
```

Seeders

- Con los seeders podemos rellenar las tablas con datos de ejemplo.

```
php artisan make:seeder PersonaSeeder
php artisan migrate:fresh --seed (reinicia las migraciones y ejecuta los seeders)

# En database/seeders/DatabaseSeeder.php se programa el orden de ejecución de los Seeders:

        $this->call([
            PersonaSeeder::class,
            HospitalesSeeder::class
        ]);

# Ejemplo de un seeder (PersonaSeeder.php)

    public function run()
    {
        for ($i = 0; $i < 100; $i++) {
```

```
            DB::table('personas')->insert([
                'nombre' => Str::random(3),
                'apellidos' => Str::random(15),
                'dni' => '12312345K',
                'telefono' => '9494949494',
                'fechanacimiento' => '2020-1-1'
            ]);
        }
    }
```

- Si queremos que los datos de ejemplo sean más reales, tendríamos que utilizar un componente llamado Faker que ya está instalado en Laravel:
- Información sobre diferentes tipos de datos generados en faker: https://fakerphp.github.io/
- Ejemplo del mismo Seeder anterior:

```
<?php

namespace Database\Seeders;

use Faker\Generator as Faker;
use Illuminate\Support\Str;
use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\DB;

class PersonaSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run(Faker $faker)
    {
        // Inyectamos una instancia de Faker.
        // Información de formateadores en: https://fakerphp.github.io/formatters

        for ($i = 0; $i < 100; $i++) {
            DB::table('personas')->insert([
                'nombre' => $faker->name(),
                'apellidos' => $faker->lastName(),
                'dni' => $faker->text(9),
                'telefono' => $faker->randomNumber(9, true),
                'fechanacimiento' => $faker->date('Y_m_d')
            ]);
        }
    }
}

php artisan migrate:fresh --seed   (para resetear las migraciones y ejecutar los seeders).
```

Instalación de CKEditor

- Editor CKEditor aqui: https://ckeditor.com/ckeditor-5/
- Para instalarlo en Laravel lo podemos instalar con npm (gestor de paquetes de NodeJS)
- Accedemos a la carpeta de nuestra instalación de Laravel:

```
cd /var/www/laravel.freeddns.org

# Agregamos la dependencia (instala físicamente el editor en la carpeta node_modules):

npm install --save @ckeditor/ckeditor5-build-classic

# Una vez instalado en los módulos de Node, tendremos que compilar el recurso para que lo incluyan en js/app.js

# Pero antes tenemos que incluir ese módulo en /resources/js/app.js el módulo

require('./bootstrap');

window.ClassicEditor = require('@ckeditor/ckeditor5-build-classic');
```

```
# Compilamos para que genere el archivo final en /public/js/app.js
npm run dev
```

- Ya podemos integrar el editor clásico con la siguiente información en la vista que nos interese:

https://ckeditor.com/docs/ckeditor5/latest/builds/guides/integration/basic-api.html

Código que iría en una de las vistas:

```
@section('scripts')
    <script>
        $(document).ready(function() {
            ClassicEditor.create($('#descripcion').get()[0]);
        });
    </script>
@endsection
```

Estilos CSS

- Para añadir estilos CSS propios:
- En el fichero **webpack.mix.js** (descargar del servidor) se indican las rutas dónde buscará nuevo css para que cuando ejecutemos "npm run dev" lo incluya en nuestra carpeta public/css.
- En este caso está puesto lo siguiente:

```
mix.js('resources/js/app.js', 'public/js')
    .vue()
    .sass('resources/sass/app.scss', 'public/css');
```

- Con lo que buscará en la carpeta **resources/sass/app.scss** y los estilos que tengamos ahí dentro los compilará a **public/css/app.css**
- En este caso voy a incluir un estilo para separar los botones del menú:

```
// Estilos para el menú
.dropdown {
    margin-right:15px;
}

# Acordarse de ejecutar npm run dev una vez incluídos los estilos propios.
```

Configuración de Nginx para permitir subidas de archivos hasta 25M

- En el script de instalación están ya estos parámetros configurados. Si queréis hacerlo de forma manual tendríamos que:

```
# Modificar en /etc/nginx/nginx.conf para permitir subir archivos grandes

sudo nano /etc/nginx/nginx.conf

# Añadir debajo de server_tokens off;
client_max_body_size 25M;

# Reiniciar sudo service nginx restart
Editar sudo nano etc/php/7.4/fpm/php.ini

# Modificar upload_max_filesize = 2M
upload_max_filesize = 25M

# Reiniciar PHP
sudo service php7.4-fpm restart
```

Localización de mensajes en Español en Laravel 8

- Si queremos tener los mensajes de error en lenguaje español podremos descargarnos la carpeta es que se encuentra en el siguiente repositorio **https://github.com/raveiga/laravel-lang** y colocarla dentro de

resources/lang:

```
# Luego tendremos que ir al fichero '''/config/app.php'''
# Y poner la siguiente propiedad a español:
'locale' => 'es'

# También podemos aprovechar y poner la zona horaria a Madrid, editando la propiedad:
'timezone' => 'Europe/Madrid',

# Para Faker:
'faker_locale' => 'es_ES',

# Si queremos personalizar los mensajes de validación de errores que tenemos en los Requests:
# Abrimos el fichero Request que queremos personalizar en app/Http/Requests:

# Creamos una función llamada messages()
# Y luego metemos la siguientes reglas específicas en un array, por ejemplo:

    /**
     * Localización de mensajes específicos de error.
     */
    public function messages()
    {
        return [
            'fechanacimiento.required' => 'La fecha de nacimiento es obligatoria para dar de alta una persona.',
        ];
    }

# Otra forma sería agrupar los mensajes en un fichero específico para compartirlos entre diferentes formulario, por ejemplo en el fi

# Le ponemos el siguiente contenido:

<?php

/**
 * Recompilación de mensajes personalizados en un fichero mensajes.php específico para así poder compartirlos entre formularios.
 */

return [
    'fechanacimiento' => 'La fecha de nacimiento es obligatoria para dar de alta una persona.',
];

# Y a continuación modificamos nuestro Request para que cada campo coja el mensaje específico de nuestro fichero mensajes.php:
# Para ello se llama a ese fichero usando __('fichero.campo')
# Desde Blade se puede usar la función lang
# Más información en:  https://laravel.com/docs/8.x/validation#specifying-custom-messages-in-language-files

    /**
     * Localización de mensajes específicos de error usando un fichero adicional recopilatorio de mensajes.
     */
    public function messages()
    {
        return [
            'fechanacimiento.required' => __('mensajes.fechanacimiento')
        ];
    }

# Si queremos personalizar el nombre del campo :attribute tendríamos que hacer una nueva función llamada attributes() con un array y
# Deberíamos comentar la función messages() para que así solamente nos traduzca el nombre del campo:

    /**
     * Localización al español de un campo del formulario en este caso 'fechanacimiento':
     */
    public function attributes()
    {
        return [
```

```
            'fechanacimiento' => 'Fecha de nacimiento'
        ];
    }
```

Autenticación en Laravel

- Cómo proteger el acceso a las páginas de nuestra aplicación mediante la autenticación
- Queremos proteger la aplicación para que solamente los usuarios registrados en la misma puedan gestionar las vacunaciones: **https://laravel.com/docs/8.x/authentication**
- Tenemos que indicar que queremos usar el módulo de autenticación con Vue:

```
php artisan ui vue --auth

# El comando anterior nos generó una carpeta nueva /resources/views/auth que contiene varias vistas (sincroniza del remoto al local
# A parte de eso se han creado nuevas rutas: sincronizar fichero /routes/web.php:
Auth::routes();

# Con las rutas: /login /logout /register y rutas para gestionar las password que se han creado para gestionar la autenticación.

# Para añadir un nuevo campo al registro con los apellidos:

# Abrir el fichero /app/Http/Controllers/Auth/RegisterController.php
# Añadimos el campo surname en el controlador  /app/Http/Controllers/Auth/RegisterController.php , en el modelo  User , en la vista
# Probamos a ver si nos registra, chequeamos la base de datos a ver si se creó el registro y si todo ok...
# Modificamos la dirección a la cuál nos tiene que conectar cuando un usuario se registra. En lugar de mandarnos a la vista /home qu
protected $redirectTo ='/';

# Para detectar en Blade si el usuario está o no autenticado:

            @auth
                El usuario está autenticado
            @endauth

            @guest
                El usuario no está autenticado
            @endguest

# A nivel de rutas podemos proteger el acceso a través del middleware auth.
# El middleware auth se puede activar a nivel de constructor del controlador, o a nivel de ruta:
# En app/Http/Kernel.php aparecen los middleware asignados en las rutas de web o api.
# Podemos asignar el middleware en el constructor del controlardor:

    public function __construct()
    {
        $this->middleware('auth');
        // $this->middleware('auth')->only('create');
        // $this->middleware('auth')->except('index');
    }

# O también lo podemos asignar en la ruta, aunque se recomienda asignarlos en el controlador:
Route::resource('hospitales', HospitalesController::class)->middleware('auth');

# Para crear roles de usuarios:
# Crear una migración: php artisan make:migration create_rols_table

    public function up()
    {
        Schema::create('rols', function (Blueprint $table) {
            $table->id();

            // Para indicar el tipo de rol, por ejemplo:
            $table->string('key', 10);

            // opcionales
            $table->string('name', 100);
            $table->string('descripcion', 500);

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

```
    }

# Creamos el modelo Rol: php artisan make:model rol
# Añadimos el campo rol en la migración de create_users_table.php:   $table->string('rol_id');
# Indicamos en el modelo User que un usuario tiene un rol creando una function rol()

    public function rol()
    {
        return $this->belongsTo(Rol::class);
    }

# Creamos un seeder para añadir 2 tipos de usuario: 1-> admin y 2->regular:  php artisan make:seeder

    public function run()
    {
        DB::table('rols')->insert([
            'key' => 'admin',
            'name' => 'Administrador de la web',
            'descripcion' => 'Información adicional sobre el administrador'
        ]);

        DB::table('rols')->insert([
            'key' => 'regular',
            'name' => 'Usuarios regulares de la web',
            'descripcion' => 'Información adicional sobre los usuarios regulares'
        ]);
    }

# Añadimos el seeder anterior a DatabaseSeeder.php
# Refrescamos las migraciones: php artisan migrate:refresh --seed
# En el RegisterController.php tendremos que añadir el campo rol_id:

        return User::create([
            'name' => $data['name'],
            'rol_id' => 2, // Le asignamos el 2 que es el segundo registro de tipo de usuario (regular) por defecto.
            'surname' => $data['surname'],
            'email' => $data['email'],
            'password' => Hash::make($data['password']),
        ]);

# En el modelo User.php añadiremos en $fillable el campo rol_id.
# Podemos registrar de nuevo un usuario y le cambiamos en el phpmyadmin el rol_id a 1 para que sea admin.
# Y registramos un segundo usuario y que sea de tipo regular.
# Podemos crearnos un middleware para gestionar que tipo de usuario se está logueando: https://laravel.com/docs/8.x/middleware
# Se crea con php artisan make:middleware CheckRolAdmin

    public function handle(Request $request, Closure $next)
    {
        if (auth()->user()->rol->key == 'admin') {
            return $next($request);
        }

        // Si no es admin lo manda a la raíz.
        return redirect('/');
    }

# Una vez creado el Middleware lo registramos en Kernel.php en la sección de routeMiddleware:
        'rol.admin' => \App\Http\Middleware\CheckRolAdmin::class,

# A partir de ahora podemos referenciar a nuestro middleware en nuestros controladores en el constructor (por ejemplo el de Personas

    public function __construct()
    {
        $this->middleware('auth');
        $this->middleware('rol.admin');
        // También se pueden cargar todos en un array.
        // $this->middleware(['auth','rol.admin']);

# De esta manera solamente los usuarios autenticados y que sean admin podrían entrar en cualquiera de los métodos del controlador de
# Si quisiéramos ocultar el botón de Personas y que no se muestre a los usuarios regulares podríamos hacer algo en el master.blade.p

@if (auth()->user()->rol->key == 'admin')
        <div class="dropdown">
```

```
            <button class="btn btn-secondary dropdown-toggle" type="button" id="dropdownMenuButton"
                data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
                Personas
            </button>
            <div class="dropdown-menu" aria-labelledby="dropdownMenuButton">
                <a class="dropdown-item" href="{{ route('personas.create') }}">Altas</a>
                <a class="dropdown-item" href="{{ route('personas.index') }}">Listado</a>
            </div>
        </div>
@endif
```

Envío de correos en Laravel

1. Documentación de Laravel sobre el correo: **https://laravel.com/docs/8.x/mail**
2. Veamos cómo podemos configurar Laravel para poder enviar correos y para que funcione la opción de recuperación de la contraseña:
3. Para configurar el correo por ejemplo para Gmail, deberíamos activar la doble autenticación en nuestra cuenta de correo y crear una contraseña de aplicación específica para nuestra web:
4. Editaremos el fichero config/mail.php o el fichero .env y lo editaremos con datos similares a éstos:

```
MAIL_MAILER=smtp
MAIL_HOST=smtp.gmail.com
MAIL_PORT=587
MAIL_USERNAME=veiga@iessanclemente.net
MAIL_PASSWORD=zoxxxxxgxxxxgxxxxt
MAIL_ENCRYPTION=tls
MAIL_FROM_ADDRESS=veiga@iessanclemente.net
MAIL_FROM_NAME="${APP_NAME}"

# Cuando cambiamos los datos en el fichero .env,  es recomendable ejecutar: php artisan config:clear

# Si queremos enviar un correo electrónico por ejemplo desde un formulario de contacto, tendremos que hacer lo siguiente:
# 1.- Crearemos un Mailable (clase  que nos ayudará a enviar correos electrónicos).

php artisan make:mail NotificacionContacto

# 2.- Se creará una carpeta /app/Mail con el fichero NotificacionContacto.php
# 3.- En el constructor recibiremos las variables que les pasaremos a la vista con el formulario de contacto.

    use Queueable, SerializesModels;
    protected $nombre, $apellidos, $contenido;

    /**
     * Create a new message instance.
     *
     * @return void
     */
    public function __construct($nombre, $apellidos, $email, $contenido)
    {
        $this->nombre = $nombre;
        $this->apellidos = $apellidos;
        $this->email = $email;
        $this->contenido = $contenido;
    }

# 4.- En el método build llamaremos a la vista que se enviará en el correo, pasándole los datos que necesita con el método with.

    public function build()
    {
        return $this->view('correos.contenidomail')
            ->subject("Solicitud de contacto desde Formulario web")
            ->with([
                "nombre" => $this->nombre,
                "apellidos" => $this->apellidos,
                "email" => $this->email,
                "contenido" => $this->contenido
            ]);
    }
```

5.- Crearemos la vista que se utilizará para formatear los datos que se enviarán por correo. Le llamaremos /resources/views/correo

```
<h1>Formulario de contacto</h1>
<p>Datos recibidos desde el formulario:</p>
<ul>
    <li>Nombre: {{ $nombre }}</li>
    <li>Apellidos: {{ $apellidos }}</li>
    <li>Email de contacto: {{ $email }}</li>
    <li>Texto de contacto: {{ $contenido }}</li>
</ul>
```

6.- Creamos la vista del formulario de contacto /resources/views/correos/create.blade.php:

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

@section('titulopagina', 'Formulario de contacto')

@section('textocabecera', 'Formulario de contacto')

@section('central')
    <form action={{ route('contactar') }} method="POST">
        @csrf
        <div class="form-group">
            <label for="nombre">Nombre:</label>
            <input type="text" class="form-control" id="nombre" name="nombre" value="" />
        </div>
        <div class="form-group">
            <label for="apellidos">Apellidos:</label>
            <input type="text" class="form-control" id="apellidos" name="apellidos" value="" />
        </div>
        <div class="form-group">
            <label for="email">E-mail:</label>
            <input type="email" class="form-control" id="email" name="email" value="" />
        </div>
        <div class="form-group">
            <label for="name">Contenido</label>
            <textarea class="form-control" name="contenido"></textarea>
        </div>
        <div class="form-group">
            <button type="submit" class="btn btn-primary">Enviar correo</button>
        </div>
    </form>
@endsection
```

7.- Vamos a crear un controlador para gestionar las rutas de contactar:

```
php artisan make:controller ContactarController
```

8.- Vamos a crear 2 rutas para el correo. Por un lado una para mostrar formulario de contacto y por otro otra que recibirá los dat

```
Route::get('contactar', [App\Http\Controllers\ContactarController::class, 'create'])->name('contactar');
Route::post('contactar', [App\Http\Controllers\ContactarController::class, 'enviar'])->name('contactar');
```

9.- Por último nos falta programar el código de ContactarController:

```
    public function create()
    {
        return view('correos.create');
    }

    /**
     * Método para enviar los correos.
     */
    public function enviar(Request $request)
    {
        // Creamos un objeto de la clase NotificacionContacto
        $mailable = new NotificacionContacto($request->nombre, $request->apellidos, $request->email, $request->contenido);

        // Enviamos el correo al destinatario
        Mail::to('veiga@iessanclemente.net')->send($mailable);
        return back()->with('estado', 'Se han enviado los datos del formulario correctamente. Contactaremos con usted lo antes posib
```

```
    }

# 10.- Si quisiéramos enviar un correo de una manera más sencilla sin usar formato HTML, ni hacer un maillable, podríamos hacerlo co

Mail::raw("Alguien con la IP " . $_SERVER['REMOTE_ADDR'] . " esta accediendo a tu web", function ($message) {

 $message->subject("Acceso al index de tu web")->to("veiga@iessanclemente.net");

 });
```

Construir una API REST y Laravel Passport

Información e instalación de extensiones recomendables

- Vamos a ver una forma de compartir todos los datos de los que disponemos a través de una API REST, de tal forma que otras webs o servicios externos puedan utilizar nuestros datos para hacer estadísticas, mapas, gráficos, etc..
- La API REST se podría utilizar por ejemplo para que Vue se pueda comunicar con nuestra aplicación de servidor en Laravel, o bien para exportar/compartir datos con otros servicios, dispositivos móviles, etc.
- Básicamente una API REST es un conjunto de funciones que devuelven datos (generalmente en XML o JSON, que es el más recomendado) o permiten crear también elementos.
- Verbos HTTP disponibles en una API REST: **https://developer.mozilla.org/es/docs/Web/HTTP/Methods**
- Códigos de estado que se pueden devolver en una API REST(se usan para dar información de cómo fue la petición a la API): **https://developer.amazon.com/es/docs/amazon-drive/ad-restful-api-response-codes.html**
- Extensiones recomendadas para trabajar:

1. Chrome: JSON Viewer Pro
2. Chrome: Postman o Advance Rest Client

Creación del controlador para gestionar las rutas de la API

```
# 1.- Crearemos un controlador para nuestra API, vamos a comenzar haciendo la API REST para personas, pero dentro de la carpeta Cont

php artisan make:controller api/PersonasController --resource

# 2.- Crearemos la ruta en routes/api.php que apunte a ese controlador.

<?php

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Route;
use App\Http\Controllers\api\PersonasController;

/*
|--------------------------------------------------------------------------
| API Routes
|--------------------------------------------------------------------------
|
| Here is where you can register API routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| is assigned the "api" middleware group. Enjoy building your API!
|
*/

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

Route::resource('personas', PersonasController::class);

# 3.- Si comprobamos las rutas con php artisan route:list veremos que nos aparecen las siguientes rutas dentro de /api:

| | GET|HEAD | api/personas | personas.index | App\Http\Controllers\api\PersonasController@index | api |
| | POST | api/personas | personas.store | App\Http\Controllers\api\PersonasController@store | api |
| | GET|HEAD | api/personas/create | personas.create | App\Http\Controllers\api\PersonasController@create | api |
| | GET|HEAD | api/personas/{persona} | personas.show | App\Http\Controllers\api\PersonasController@show | api |
```

```
| | PUT|PATCH | api/personas/{persona} | personas.update | App\Http\Controllers\api\PersonasController@update | api |
| | DELETE | api/personas/{persona} | personas.destroy | App\Http\Controllers\api\PersonasController@destroy | api |
| | GET|HEAD | api/personas/{persona}/edit | personas.edit | App\Http\Controllers\api\PersonasController@edit | api

# 4.- En app/Providers/RouteServiceProvider.php vemos por qué se ha creado el /api/ para las rutas api que se programen en /routes/a
```

Creación de los métodos de ejemplo de una Persona para la API

```
#5.- Editaremos el controlador /app/Http/Controllers/api/PersonasController.php para los métodos index y show.

<?php

namespace App\Http\Controllers\api;

use App\Models\Persona;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\DB;
use App\Http\Controllers\Controller;
//use App\Http\Controllers\api\ApiResponseController;

//class PersonasController extends Controller
class PersonasController extends ApiResponseController
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        // Aquí dentro vamos a mostrar un listado en JSON de todas las personas.
        // Devolveremos un array en JSON con el contenido.
        // Ejemplo: return response()->json(["titulo" => 'HOla mundo Laravel']);

        // Dentro de una API REST deberíamos dar siempre una respuesta acorde a la petición.
        // A la hora de devolver datos en una API para cumplir correctamente el estandard se debería devolver el código de respuesta
        // https://developer.amazon.com/es/docs/amazon-drive/ad-restful-api-response-codes.html
        $personas = Persona::orderBy("nombre")->paginate(10);

        // En este estado devolvemos un status 200
        //return response()->json(['status' => 'ok', 'data' => $personas], 200);
        return $this->respuestaExito($personas);

        // // Si queremos devolver todos los datos de esa persona junto con los hospitales y vacunas que ha realizado tendríamos que
        // $todoslosdatos = DB::table('personas')
        //     ->join('hospital_persona', 'personas.id', '=', 'hospital_persona.persona_id')
        //     ->join('hospitales', 'hospital_id', '=', 'hospitales.id')
        //     ->orderBy('personas.nombre')->get();

        // $todoslosdatos = DB::table('personas')
        //     ->join('hospital_persona', 'personas.id', '=', 'hospital_persona.persona_id')
        //     ->join('hospitales', 'hospital_id', '=', 'hospitales.id')
        //     ->orderBy('personas.nombre')
        //     ->select('personas.*', 'hospitales.nombre as HospitalNombre')->paginate(10);

        // $todoslosdatos = DB::table('personas')
        //     ->join('hospital_persona', 'personas.id', '=', 'hospital_persona.persona_id')
        //     ->join('hospitales', 'hospital_id', '=', 'hospitales.id')
        //     ->orderBy('personas.nombre')
        //     ->select('personas.nombre', 'personas.apellidos', 'personas.telefono', 'fecha_vacunacion', 'hospitales.nombre as Hosp

        // // return response()->json(['data' => $todoslosdatos,'code' => 200, 'status' => 'ok' ], 200);
        // return $this->respuestaExito($todoslosdatos);
        // // return $this->respuestaExito($todoslosdatos, 500);

    }


    /**
     * Display the specified resource.
     *
     * @param  int  $id
```

```
     * @return \Illuminate\Http\Response
     */
    public function show(Persona $persona)
    {
        // return response()->json(['status' => 'ok', 'code' => 200, 'data' => $persona], 200);

        return $this->respuestaExito($persona);
    }
}

# 6.- También tendremos que limitar las operaciones disponibles dentro del controlador resource.
Route::resource('personas', PersonasController::class)->only('index', 'show');
```

Creación de un Trait para implementar función que devuelva el JSON de respuesta

```
# 7.- Como vamos a repetir muchas veces el formato de respuesta, lo mejor es crear un Trait ya que estamos repetiendo la misma estru
return response()->json(['data' => $datos,'code' => 200, 'status' => 'ok' ], 200);

# Con los Traits podemos usar herencia múltiple en nuestras aplicaciones, de tal manera que podemos compartir una función que nos de

# 8.- Para crear un Trait, crear carpeta /app/Traits.

# 9.- Crear el archivo /app/Traits/ApiResponse.php con el siguiente código:

<?php

namespace App\Traits;

trait ApiResponse
{
    // Los parámetros opcionales se ponen al final.
    public function respuestaExito($data, $code = 200, $status = 'ok')
    {
        return response()->json(['data' => $data, 'code' => $code, 'status' => $status], $code);
    }

    public function respuestaError($data, $code = 500, $status = 'ok')
    {
        return response()->json(['data' => $data, 'code' => $code, 'status' => $status], $code);
    }
}

#10.- Creamos un fichero /app/Http/Controllers/api/ApiResponseController.php

<?php

namespace App\Http\Controllers\api;

use App\Http\Controllers\Controller;
use App\Traits\ApiResponse;

class ApiResponseController extends Controller
{
    use ApiResponse;
}

# 11.- Ahora tendremos que decirle a nuestro PersonasController que extienda a a ApiResponseController:
class PersonasController extends ApiResponseController

# 12.- De esta manera podemos llamar a la respuesta de la siguiente forma dentro del controlador:
return $this->respuestaExito($datos);

# 13.- Para manejar las excepciones (rutas no encontradas por ejemplo) de la API REST las vamos a devolver también con un JSON. Para
Dentro de la carpeta /app/Exceptions/ tenemos un archivo Handler.php que se encarga de gestionar todas las excepciones y dónde podem

# 14.- Editamos ese fichero y ponemos el siguiente contenido importando la clase NotFoundHttpException:

use App\Traits\ApiResponse;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;
....
class Handler extends ExceptionHandler
{
```

```
    use ApiResponse;
....
    public function register()
    {
        $this->reportable(function (Throwable $e) {
            //
        });

        $this->renderable(function (NotFoundHttpException $e, $request) {
            if ($request->is('api/*'))
                return $this->respuestaError("Pagina no encontrada", $code = 404, $status = 'Error');
        });
    }
```

Proteger el acceso a la API REST mediante autenticación

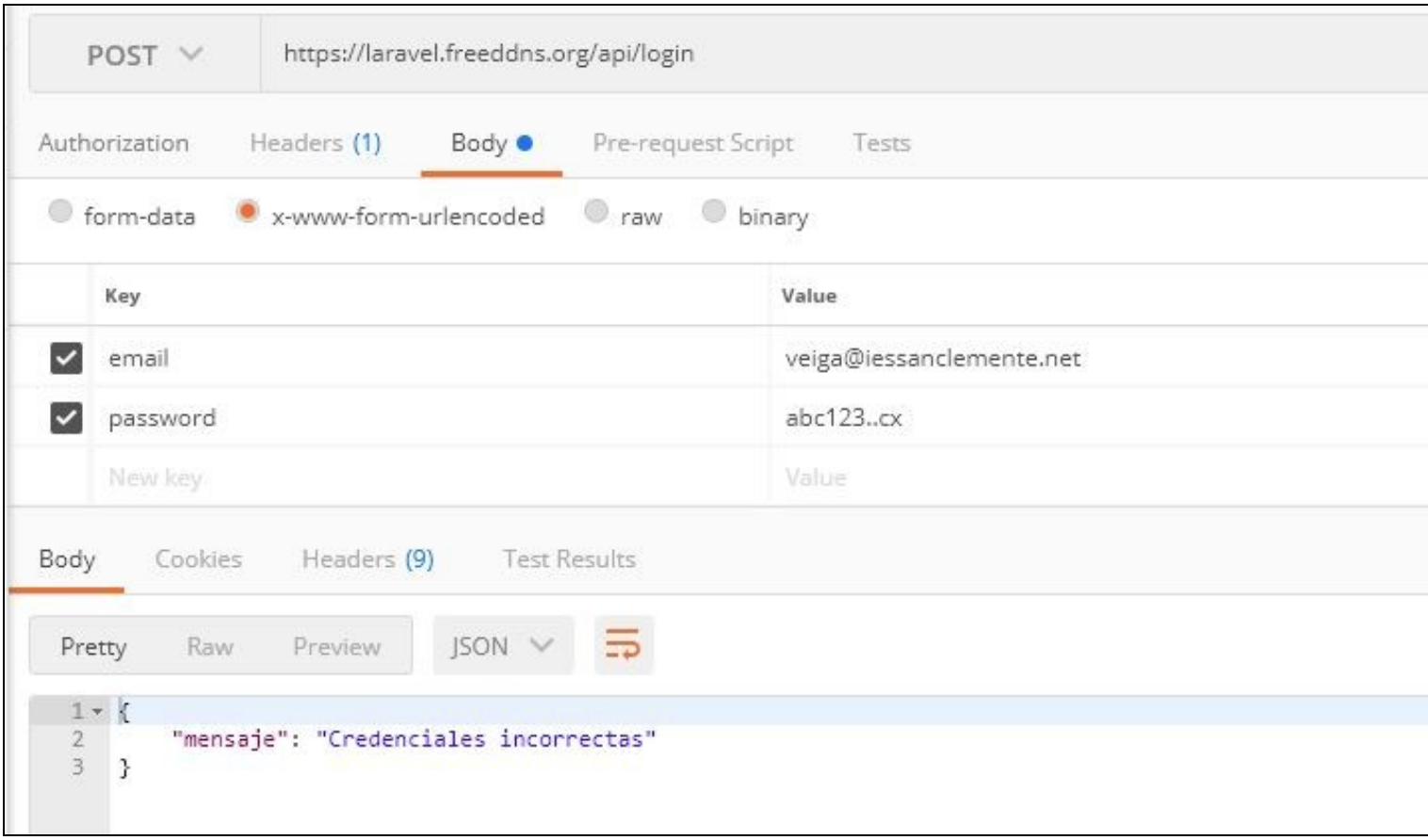
```
# 15.- Para ello utilizaremos Laravel Passport: https://laravel.com/docs/8.x/passport

# 16.- Para instalar Laravel Passport lo hacemos con un paquete de Composer y luego ejecutaremos las migraciones necesarias que gene
composer require laravel/passport
php artisan migrate



#17.- Ahora tenemos que generar ciertas claves para la autenticación que además se guardan en la tabla oauth_clients:

php artisan passport:install
Encryption keys generated successfully.
Personal access client created successfully.
Client ID: 1
Client secret: jg9H1wNFd4mDR59PE8raacG07Yp1tVyz0q0VxdeY
Password grant client created successfully.
Client ID: 2
Client secret: XlenCdWqEFjIJystPqrlR48S0kmv52GAg8OJMkht

# 18.- Tenemos que modificar nuestro modelo app/Models/User.php para incluir un nuevo Trait (sincronizar remoto->local carpeta /vend

use HasApiTokens, HasFactory, Notifiable;

#19.- Abrimos el fichero /app/Providers/AuthServiceProvider.php e incorporamos en el método boot() la siguiente instrucción:

    public function boot()
    {
        $this->registerPolicies();
        Passport::routes();
    }

# 20.- Ejecutamos php artisan r:l para ver las nuevas rutas y nos aparecerán un montón de rutas nuevas de oauth/xxxx

# 21.- El último paso de configuración sería el de abrir el fichero /config/auth.php e indicar que en api vamos a usar passport:

        'api' => [
            // 'driver' => 'token',
            // 'provider' => 'users',
            // 'hash' => false,
            'driver' => 'passport',
            'provider' => 'users',
        ],

# 22.- A partir de ahora podemos comenzar a trabajar en la autenticación de nuestras rutas API REST. Lo primero que necesitamos es c
php artisan make:controller api/AuthController

# 23.- Creamos una ruta en /routes/api.php que apunte a este nuevo controlador:
// Nueva ruta para loguearnos y obtener el Token para hacer peticiones a la API.
Route::post('login', [AuthController::class, 'login']);

# 24.- Ahora en el controlador /app/Http/Controllers/api/AuthController en el método login validaremos el usuario y crearemos su tok

<?php

namespace App\Http\Controllers\api;
```

```
use Illuminate\Http\Request;
use Illuminate\Support\Carbon;
use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\Auth;
use Illuminate\Validation\Validator;

class AuthController extends Controller
{
    public function login(Request $request)
    {
        // Validar datos
        $request->validate([
            'email' => 'required|string',
            'password' => 'required|string'
        ]);

        $credenciales = request(['email', 'password']);

        // Comprobamos las credenciales manualmente usando el facade Auth.
        if (!Auth::attempt($credenciales)) {
            return response()->json(['mensaje' => 'Credenciales incorrectas'], 401);
        }

        // Si todo fue correcto, tenemos que generar un personal access token para que nuestro usuario pueda hacer la conexión a la
        // Ese token será el que usará posteriormente para hacer peticiones a la API REST.
        $user = $request->user();

        $tokenAuth = $user->createToken('Token personal de acceso');

        // Una vez creado el token podemos acceder a él de esta manera
        // Si se muestra es que está todo funcionando correctamente.
        // dd($tokenAuth);

        // Lo que tenemos que hacer ahora es devolver el token al usuario para que pueda utilizarlo en las siguientes peticiones a l
        // Con el tipo de token y la fecha de caducidad del token.
        return response()->json(
            [
                'access_token' => $tokenAuth->accessToken,
                'token_type' => 'Bearer ',
                'expires_at' => Carbon::parse($tokenAuth->token->expires_at)->toDateTimeString()
            ]
        );

        // En la tabla oauth_access_tokens aparecerán todos los tokens generados.

    }
}

# 25.- La vigencia del token por defecto es 1 año, podríamos modificarlo de la siguiente forma:
#Este código lo agregaríamos justo antes de enviar el token con el response()->json:

      // Modificamos la fecha de caducidad en 1 semana en lugar de 1 año por defecto.
        $token = $tokenAuth->token;
        $token->expires_at = Carbon::now()->addWeeks(1);
        $token->save();

# El archivo final /app/Http/Controllers/api/AuthController quedaría así:

<?php

namespace App\Http\Controllers\api;

use Illuminate\Http\Request;
use Illuminate\Support\Carbon;
use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\Auth;
use Illuminate\Validation\Validator;

class AuthController extends Controller
{
    public function login(Request $request)
    {
        // Validar datos
```

```
        $request->validate([
            'email' => 'required|string',
            'password' => 'required|string'
        ]);

        $credenciales = request(['email', 'password']);

        // Comprobamos las credenciales manualmente usando el facade Auth.
        if (!Auth::attempt($credenciales)) {
            return response()->json(['mensaje' => 'Credenciales incorrectas'], 401);
        }

        // Si todo fue correcto, tenemos que generar un personal access token para que nuestro usuario pueda hacer la conexión a la
        // Ese token será el que usará posteriormente para hacer peticiones a la API REST.
        $user = $request->user();

        $tokenAuth = $user->createToken('Token personal de acceso');

        // Una vez creado el token podemos acceder a él de esta manera
        // Si se muestra es que está todo funcionando correctamente.
        // dd($tokenAuth);

        // Modificamos la fecha de caducidad en 1 semana en lugar de 1 año por defecto.
        $token = $tokenAuth->token;
        $token->expires_at = Carbon::now()->addWeeks(1);
        $token->save();

        // Lo que tenemos que hacer ahora es devolver el token al usuario para que pueda utilizarlo en las siguientes peticiones a l
        // Con el tipo de token y la fecha de caducidad del token.
        return response()->json(
            [
                'access_token' => $tokenAuth->accessToken,
                'token_type' => 'Bearer ',
                'expires_at' => Carbon::parse($tokenAuth->token->expires_at)->toDateTimeString()
            ]
        );

        // En la tabla oauth_access_tokens aparecerán todos los tokens generados.

    }


    public function logout(Request $request)
    {
        // Aquí podemos acceder al usuario autenticado
        //dd($request->user());

        // Para revocar el token hacemos:
        $request->user()->token()->revoke();

        // Devolvemos el mensaje de sesión terminada.
        return response()->json(
            [
                'mensaje' => 'Sesión terminada con éxito',

            ]
        );
    }
}

# 26.- Nos faltaría a partir de ahora el proteger las rutas de la API para que hagan uso de la autenticación que hemos configurado.

# Tendríamos que decirle a cada ruta que use el middleware auth:api

# Contenido del archivo /routes/api.php:

Route::resource('personas', PersonasController::class)->only('index', 'show')->middleware('auth:api');
Route::resource('hospitales', HospitalesController::class)->only('index', 'show')->middleware('auth:api');

// Para proteger la ruta para autenticación con token:
// Route::resource('personas', PersonasController::class)->only('index', 'show')->middleware('auth:api');

// Nueva ruta para loguearnos y obtener el Token para hacer peticiones a la API.
```

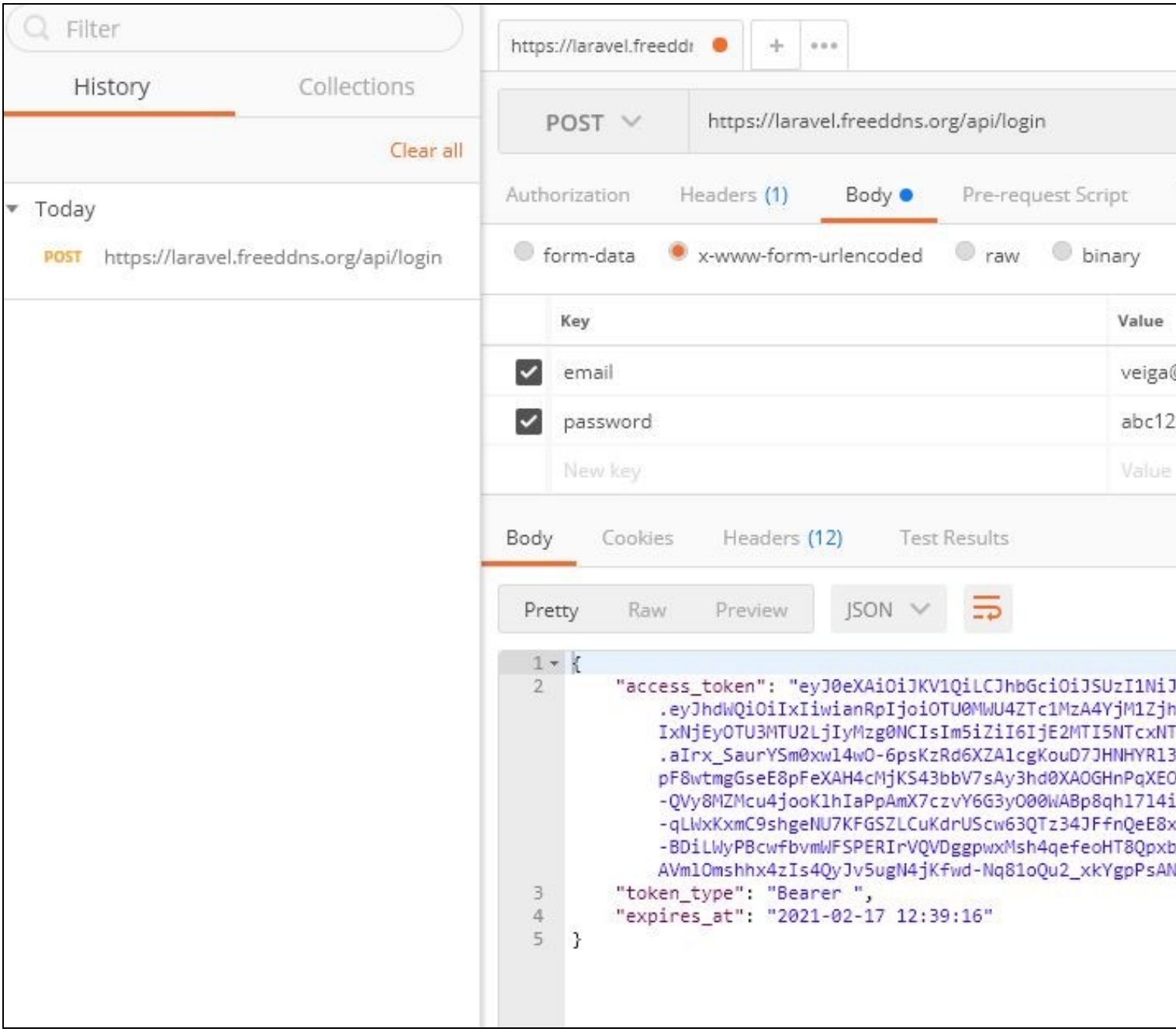
```
Route::post('login', [AuthController::class, 'login']);

// Ruta para cerrar la sesión con la API REST.
Route::post('logout', [AuthController::class, 'logout'])->middleware('auth:api');

== Ejemplo de acceso a la API REST mediante la autenticación ==

<source lang="bash">

# 27.- Todo configurado ! pasemos a utilizar la API con la nueva autenticación.
# Lo primero es obtener el access token (lo haremos con Postman):

Método: POST
Ruta: https://laravel.freeddns.org/api/login
Body: x-www-form-urlencoded
Key: email ---- Value: veiga@iessanclemente.net
Key: password  --- Value: xxxxxxxxxx

Pulsar botón SEND.
```

- Ver Foto de **acceso a la API con credenciales incorrectas**:



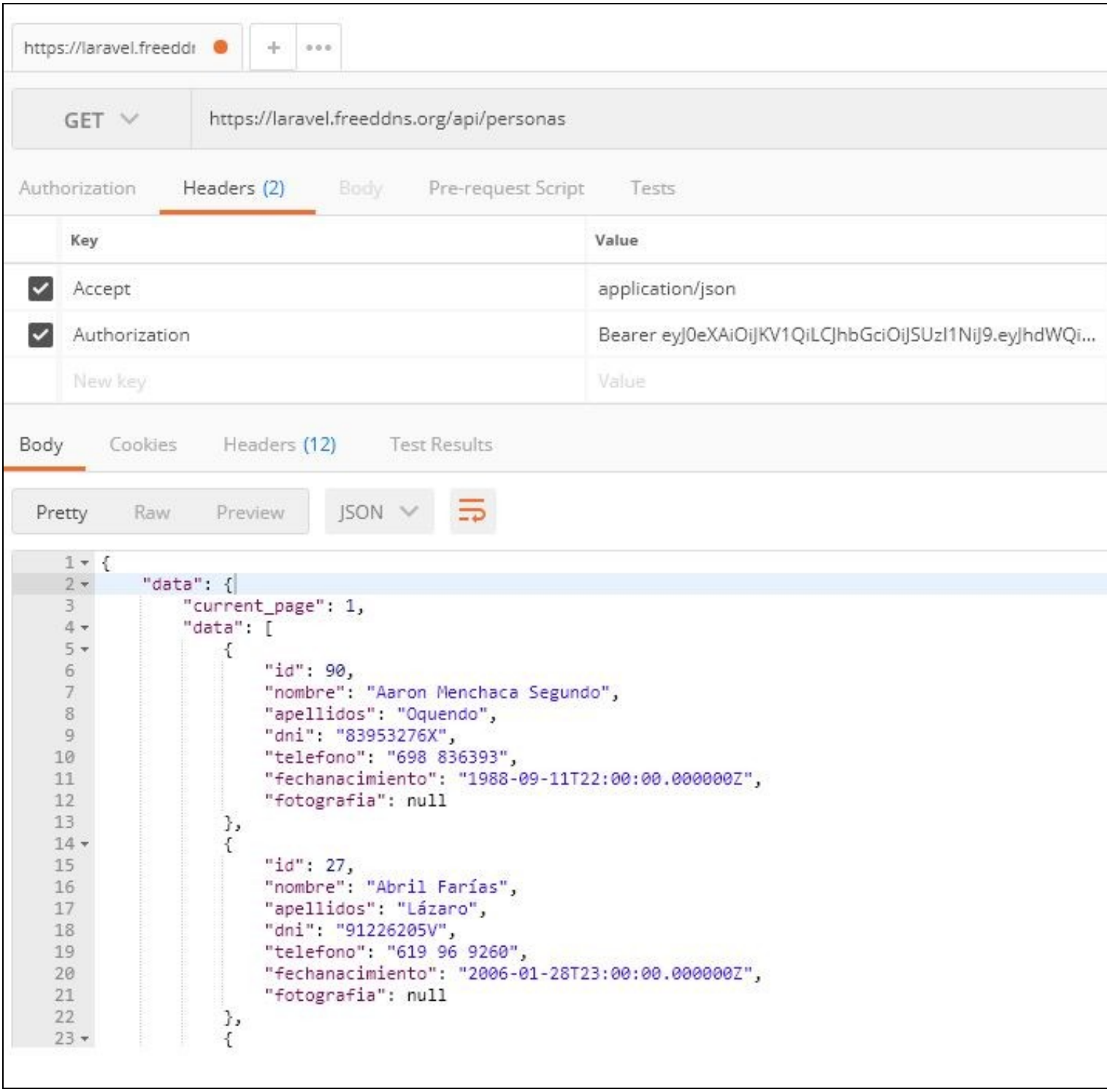
- Ver Foto de **acceso al login de la API para conseguir el Token de acceso**:

1. Ese token de acceso lo copiaremos para usarlo luego en todas las peticiones a la API.



```
El contenido de access_token lo copiamos y lo guardamos, por que a partir de ahora todas las peticiones que hagamos a las rutas prot

# 28.- Para acceder a una ruta protegida de la api tendremos que pasarle siempre en todas las peticiones (dentro de la cabecera Head

# El campo Accept llevará el valor application/json
# El campo Authorization llevará como contenido: Bearer + espacio + token

# Cómo hacerlo en Postman. En la sección de Headers metemos 2 campos:

Accept:     application/json
Authorization:      Bearer nuestro_token_de_acceso_obtenido_en_api/login


Método: GET
Ruta: https://laravel.freeddns.org/api/personas

Headers:
Key: Accept  ---- Value: application/json
Key: Authorization --- Value: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJhdWQiOiIxIiwianRpIjoiNWMwwic2NvcGVzIj....
```

- Ver FOTO de **petición a una ruta de la API con el token de acceso que va incluído en el Authorization header**:


https://laravel.freeddr
GET
https://laravel.freeddns.org/api/personas
Authorization
Headers (2)
Body
Pre-request Script
Tests
Key
Value
Accept
application/json
Authorization
Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJhdWQi...
New key
Value
Body
Cookies
Headers (12)
Test Results
Pretty
Raw
Preview
JSON
{
"data": {
"current_page": 1,
"data": [
{
"id": 90,
"nombre": "Aaron Menchaca Segundo",
"apellidos": "Oquendo",
"dni": "83953276X",
"telefono": "698 836393",
"fechanacimiento": "1988-09-11T22:00:00.000000Z",
"fotografia": null
},
{
"id": 27,
"nombre": "Abril Farías",
"apellidos": "Lázaro",
"dni": "91226205V",
"telefono": "619 96 9260",
"fechanacimiento": "2006-01-28T23:00:00.000000Z",
"fotografia": null
},
{

```
# 29.- Por último vamos a ver cómo podemos cerrar la sesión que tenemos con la API REST. Para ello comprobamos la ruta de /logout qu

# Creamos una ruta en /routes/api.php:
// Ruta para cerrar la sesión con la API REST.
Route::post('logout', [AuthController::class, 'logout'])->middleware('auth:api');

# 30.-  En el controlador /app/Http/Controllers/api/AuthController añadimos el método de logout():

    public function logout(Request $request)
    {
        // Aquí podemos acceder al usuario autenticado
        //dd($request->user());

        // Para revocar el token hacemos:
        $request->user()->token()->revoke();

        // Devolvemos el mensaje de sesión terminada.
        return response()->json(
            [
                'mensaje' => 'Sesión terminada con éxito',

            ]
        );
    }
```

Notas varias sobre Laravel 8.x

Comandos más utilizados:

```
php artisan route:list ó php artisan r:l  # Muestra las rutas activas en la aplicación.
php artisan make:migration create_NOMBRE_TABLA_EN_PLURAL_table     # Creación de migración


# Directorios importantes en Laravel

/resources/views            # Contiene las vistas con extensión .blade.php
/app/Http/Controllers/      # Contiene los controladores
/app/Models  # Contiene los modelos
/database/migrations       # Contiene las migraciones
/database/seeders          # Contiene los semilleros de datos
/routes      # Contiene las rutas

# Asociación de archivos Blade con html
# Cuando estamos trabajando con Laravel por ejemplo, podemos asociar las plantillas de Blade para que las reconozca como html.

# Pulsamos CTRL + MAYUS + P
# Abrimos el fichero settings.json

# Pegamos al final del archivo esta configuración y la guardamos:

"files.associations": {
"*.blade.php": "html",
"*.tpl": "html"
}

# Formularios:
# Para añadir el token csrf: @csrf
# Para poner una ruta de POST="{{ route("nombreruta") }}". Ejemplo: <form action="{{ route("persona.store") }}">
#Para poner un stylesheet precompilado con webpack: <link rel="stylesheet" href="{{ asset("css/app.css") }}">
#Para poner un script precompilado con webpack:    <script src="{{ asset("js/app.js")}}"></script>

# Datos recibidos del formulario:

# Para procesar datos recibidos de un formulario: https://laravel.com/docs/8.x/requests
$request->all() --> todos los valores recibidos de un formulario
$request->input("campo","o valor por defecto si no recibimos ese campo");

# Podemos imprimir todos los campos recibidos del formulario con: dd($request->all());
# También el contenido del objeto $request: dd ($request);
```

```
# Si el campo recibido es por el método GET se usa: $request->query
# También podemos acceder a un campo en concreto con $request->campo;
# También se puede llamar a un método request("campo") en lugar de usar el objeto $request.
# Validación de datos recibidos del formulario: https://laravel.com/docs/8.x/validation
# Vamos a usar el facadeValidate (es como una clase que maneja métodos estáticos que podremos usar en la aplicación).
# Al usar ese facade, podremos usar el método validate() que recibe un array con las reglas de validación.

# Ejemplo de método de validación que iría en el controlador que recibe los datos. Los valida y si todo está ok, continuaría con el
$request->validate([

'titulo'=>'required|min:5|max:500',

]);

# Reglas de validación disponibles: https://laravel.com/docs/8.x/validation#available-validation-rules
# Para mostrar los errores de validación se usa el sistema de enviarlos a través de una variable de sesión.
# Para ver los mensajes de error en español, véase: https://laravel.com/docs/8.x/localization
# Si hay errores se mostrarán en el formulario origen de los datos:

# Este código de ejemplo lo pondríamos en la vista dónde está el formulario para recorrer todos los errores y mostrarlos en una list

@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif

# Si queremos comprobar el error en un campo específico, podríamos poner algo como lo siguiente y se mostraría el error debajo de ca

        <input type="text" name="apellidos">
        @error('apellidos')
            <small class="text-danger">{{ $message }}</small>
        @enderror


#  Plantillas Layouts en Blade.

#  Sirve para crear una plantilla esqueleto y luego reutilizar en el resto de vistas.
#  https://laravel.com/docs/8.x/blade#building-layouts
# En la plantilla master se usa @yield("contenido") para mostrar el contenido de la sección de la plantilla hija.
# En la plantilla hija se usa @section para indicar el contenido que se incorporará a la plantilla.
# Ver la sección https://laravel.com/docs/8.x/blade#layouts-using-template-inheritance

# Crear modelo

# Se utilizan para hacer todo tipo de consultas a la base de datos.
php artisan make:model NombreModeloSingular , y para la ayuda php artisan make:model -h

# A la hora de grabar los datos se recomienda grabar los datos validados.

# Crear requests
php artisan make:request Método_Validacion_ControladorModeloTipo_peticion  (StoreBlogPost)



# Guardar una foto en el Storage.
En el método storage de nuestro controlador podríamos hacer:
if ($request->hasFile('fotografia')){
  // Almacenará la fotografía en la carpeta storage/app/public/uploads
  $datosguardar['fotografia'] = $request->file("fotografia")->store('uploads','public');
}

# Para mostrar la fotografía en una vista:
Primero tenemos que ejecutar un comando para hacer el link de esa carpeta storage/app/public/uploads:
php artisan storage:link

<img src="{{ asset('storage').'/'.$empleado->Foto }}" alt=""/>

# Instalación de crud generator:
```

```
composer require ibex/crud-generator --dev

php artisan vendor:publish --tag=crud

Para crear un crud:
php artisan make:crud libros

# Añadir el controlador
Route::resource('libros',[App\Http\Controllers\LibrosController::class]);

# Si queremos que solamente puedan entrar en esa ruta usuarios logueados:
Route::resource('libros',[App\Http\Controllers\LibrosController::class])->middleware('auth');

# En una plantilla para chequear si un usuario está logueado:
@if (Auth::check())


@endif
```

Laravel CRUD generator

Podemos generar los modelos, vistas, formularios, etc de forma automática partiendo desde el diseño de las tablas en MySQL:

https://github.com/awais-vteams/laravel-crud-generator

```
# Instalamos el paquete:
composer require ibex/crud-generator --dev
php artisan vendor:publish --tag=crud

# Modo de uso
php artisan make:crud nombre_tabla
```

Veiga (discusión) 00:06 18 ene 2022 (CET)