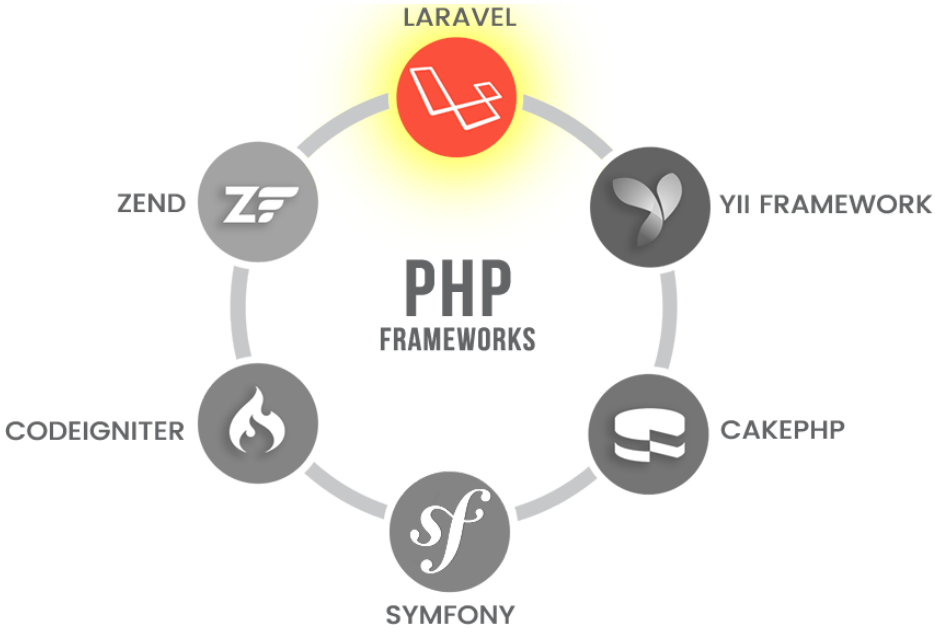
unidad didáctica 7

# Laravel - autenticación de usuarios



# 1. autenticación basada en sesiones

En esta sesión veremos cómo incluir mecanismos de autenticación en nuestros proyectos Laravel. Partiremos de la base de un proyecto ya creado (como el ejemplo de la biblioteca que venimos haciendo de sesiones anteriores) e incorporaremos paso a paso los elementos necesarios para autenticar usuarios.

## 1.1. 1. Configuración general de la autenticación

En el archivo `config/auth.php` se dispone de algunas opciones de configuración generales de autenticación. Esta autenticación en Laravel se apoya en dos elementos: los *guards* y los *providers*.

- Los ***guards*** son mecanismos que definen cómo se van a autenticar los usuarios para cada petición. El mecanismo más habitual es mediante sesiones, donde se guarda la información del usuario autenticado en la sesión, aunque por defecto también se habilita la autenticación mediante tokens.
- Los ***providers*** indican cómo se van a obtener los usuarios de la base de datos para comprobar la autenticación. Las opciones habilitadas por defecto son mediante Eloquent (y el modelo de usuarios que tengamos definido), o mediante *query builder*, consultando directamente la tabla correspondiente de usuarios.

Deberemos modificar en el archivo la referencia a la tabla donde almacenaremos los usuarios (por defecto se hace referencia a una tabla llamada `users`) y/o al modelo asociado (por defecto, la clase `User`). Así que convendrá modificar los nombres de estos dos elementos en la sección `providers`, así como la ubicación (*namespace*) del modelo de usuario, si procede. Por ejemplo:

```
// ...

'providers' => [
    'users' => [
        'driver' => 'eloquent',
        'model' => App\Models\Usuario::class,
    ],

    // 'users' => [
    //     'driver' => 'database',
    //     'table' => 'usuarios',
    // ],
],
```

Notar que la sección `providers` dispone de dos proveedores de autenticación: uno (el que está habilitado) está basado en Eloquent, y hace uso del modelo de usuarios que hayamos definido. El otro (que aparece comentado) no utiliza Eloquent, sino el *query builder* contra la propia base de datos. Si preferimos esta segunda opción, deberemos comentar la primera y dejar habilitada la segunda. También es posible dejar habilitados múltiples *providers*, cada uno con un nombre diferente, y asignarlo a múltiples *guards*.

### 1.1.1. 1.1. El modelo o la tabla de usuarios

Si elegimos el *provider* basado en Eloquent, deberemos tener un modelo de usuarios al que acceder. En el caso de nuestra aplicación de biblioteca (o el ejercicio del blog), disponemos ya de un modelo creado en `App\Models\Usuario`, por lo que el ejemplo anterior nos serviría para establecer este modelo como el modelo de usuarios por defecto.

Si optamos por utilizar el *query builder* en lugar de Eloquent, deberemos tener una tabla en la base de datos donde estén los datos de los usuarios. En nuestro caso, también disponemos de esa tabla *usuarios*, por lo que podríamos emplear esta otra opción para autenticar usuarios si quisiéramos. No obstante, nos valdremos de Eloquent para la autenticación.

En cualquier caso, como veremos a continuación, será conveniente que los passwords de los usuarios estén **encriptados** mediante *bcrypt*, que es el mecanismo de encriptación por defecto que utiliza Laravel. Vamos a crear un nuevo *seeder* en nuestra aplicación de *biblioteca* para crear un usuario con un login y password predefinidos. Llamamos al *seeder* `UsuariosSeeder`:

```
1 php artisan make:seeder UsuariosSeeder
```

En el método `run` del nuevo *seeder* añadiremos un nuevo usuario con login *admin* y password *admin* (encriptado usando *bcrypt*):

```
1 public function run()
2 {
3   $usuario = new Usuario();
4   $usuario->login = 'admin';
5   $usuario->password = bcrypt('admin');
6   $usuario->save();
7 }
```

Finalmente, cargamos este nuevo *seeder* en la base de datos, o bien con una migración completa nueva (`php artisan migrate:fresh --seed`), o bien ejecutando sólo el seeder con esto:

```
1 php artisan db:seed --class=UsuariosSeeder
```

## 1.2. 2. Añadir autenticación a un proyecto

Para añadir autenticación a un proyecto Laravel ya existente que no disponga de estos mecanismos, seguiremos estos pasos:

1. Definiremos un formulario de login.
2. Definiremos un nuevo controlador que se encargue de gestionar el login: tanto de mostrar el formulario cuando el usuario no esté autenticado como de validar sus credenciales cuando las envíe.
3. Añadiremos las rutas pertinentes en el archivo `routes/web.php` tanto para el formulario de login como para la autenticación posterior.
4. Protegeremos las rutas que sean de acceso restringido.
5. Opcionalmente, podemos añadir también una opción de *logout*.

### 1.2.1. 2.1. El formulario de login

Vamos a definir un formulario de login en la vista `resources/views/auth/login.blade.php`, para que el usuario especifique su *login* y su *password*. También dejaremos una zona para mostrar un posible mensaje de error si la autenticación no ha sido exitosa:

```
@extends('plantilla')

@section('titulo', 'Login')

@section('contenido')

  <h1>Login</h1>

  @if (!empty($error))
  <div class="text-danger">
     {{ $error }}
  </div>
  @endif

  <form action="{{ route('login') }}" method="POST">
     @csrf

     <div class="form-group">
        <label for="login">Login:</label>
        <input type="text" class="form-control"
               name="login" id="login" />
      </div>

      <div class="form-group">
        <label for="password">Password:</label>
        <input type="password" class="form-control"
               name="password" id="password" />
      </div>

      <input type="submit" class="btn btn-dark btn-block"
             name="enviar" value="Enviar" />

  </form>

@endsection
```

### 1.2.2. 2.2. El controlador de login

Crearemos un nuevo controlador que se encargue de gestionar toda la autenticación:

```
php artisan make:controller LoginController
```

Dentro, definimos una función que se encargará de mostrar el formulario anterior:

```
public function loginForm()
{
  return view('auth.login');
}
```

Y añadiremos una segunda función que se encargue de validar las credenciales enviadas por el usuario. Para esto, haremos uso del *facade* de autenticación, existente en `Illuminate\Support\Facades\Auth`. Recuerda de sesiones previas que un *facade* es básicamente un elemento que proporciona acceso a una serie de métodos estáticos de utilidad, en este caso para autenticar usuarios.

```php
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

class LoginController extends Controller
{
  // ...

  public function login(Request $request)
  {
     $credenciales = $request->only('login', 'password');

     if (Auth::attempt($credenciales))
     {
        // Autenticación exitosa
        return redirect()->intended(route('libros.index'));
     } else {
        $error = 'Usuario incorrecto';
        return view('auth.login', compact('error'));
     }
  }
}
```

El método `attempt` acepta una serie de pares *clave-valor* como primer parámetro. En este caso, le pasamos un sólo par formado por el login (o el e-mail, dependiendo del campo que usemos para autenticar) y el password recibidos en la petición. Esto servirá para localizar al usuario por la clave (login), y comprobar si tiene el valor asociado (el password). En el caso de los passwords, Laravel automáticamente los encripta en formato *bcrypt*, por lo que debemos cerciorarnos de que el password está encriptado en ese formato en la base de datos.

Por otra parte, el método `intended` trata de enviar al usuario a la ruta a la que intentaba acceder antes de que se le solicitara autenticación. Le pasamos como parámetro una ruta por defecto en el caso de que el destino previsto no esté disponible.

### 1.2.3. 2.3. Las rutas asociadas

Finalmente, debemos definir las rutas tanto para mostrar el formulario (por *get*) como para recoger las credenciales y validar al usuario (por *post*).

```php
Route::get('login', [LoginController::class, 'loginForm'])->name('login');
Route::post('login', [LoginController::class, 'login']);
```

#### 1.2.3.1. 2.3.1. Redirección en caso de error

Cuando se detecta que un usuario no autenticado intenta acceder a una ruta protegida, automáticamente se le redirige a la ruta nombrada como *login* (como la que hemos definido previamente), donde verá el formulario de acceso. Si queremos cambiar el nombre de la ruta a la que redirigir (en el caso de que no queramos que sea *login*), debemos modificar el método `redirectTo` en el *middleware* de autenticación `app/Http/Middleware/Authenticate.php`:

```
protected function redirectTo($request)
{
  // ...
  return route('login');
}
```

### 1.2.4. 2.4. Proteger las rutas de acceso restringido

Ahora que ya tenemos definido el mecanismo de login (controlador con método de autenticación, formulario de login y ruta asociada), podemos proteger aquellas rutas o enlaces que queramos que sean de acceso restringido. Por ejemplo, podemos hacer que las operaciones de creación, borrado y edición de libros (funciones `create`, `store`, `edit`, `update` y `destroy`) sólo estén disponibles para usuarios autenticados. Esto puede hacerse de varias formas.

- Si tenemos una ruta de recursos (`Route::resource`) en el archivo `routes/web.php`, entonces la opción más cómoda es definir un constructor en el controlador asociado (en este caso, `LibroController`), y especificar qué funciones queremos proteger, bien con `only` o con `except` (en este último caso, se protegerán todas las rutas salvo las indicadas en la lista):

```
class LibroController extends Controller
{
  public function __construct()
  {
     $this->middleware('auth',
        ['only' => ['create', 'store', 'edit', 'update', 'destroy']]);
 }

  // ...
```

- Si definimos las rutas sueltas, podemos emplear el método `middleware` para indicar en cada una si queremos que se aplique el *middleware* de autenticación:

```
Route::get('prueba', [PruebaController::class, 'create'])->middleware('auth');
```

### 1.2.5. 2.5. Detectar en las vistas al usuario autenticado

Puede ser muy necesario detectar en una vista si el usuario se ha autenticado o no, bien para mostrar ciertos controles (por ejemplo, enlaces para crear libros), o para cargar información propia del usuario (por ejemplo, posts creados por el usuario que ha entrado en un blog).

Por ejemplo, de este modo podemos modificar el menú de navegación (`resources/views/partials/nav.blade.php`) para que muestre el enlace de crear nuevo libro sólo si el usuario se ha autenticado:

```
@if(auth()->check())
  <li class="{{ setActivo('libros.create') }} nav-item">
      <a class="nav-link" href="{{ route('libros.create') }}">Nuevo libro</a>
  </li>
@endif
```

Podemos emplear el método `auth()->guest()` si queremos comprobar si el usuario aún NO se ha autenticado (por ejemplo, para mostrarle el enlace a *login*), y el método `auth()->check()` para comprobar si SÍ está autenticado (para mostrarle, por ejemplo, las opciones restringidas). De forma análoga, el método `auth()->user()` obtiene el objeto del usuario autenticado, con lo que podemos acceder a sus atributos:

```
Bienvenido/a {{ auth()->user()->login }}
```

### 1.2.6. 2.6. Implementación del *logout*

Para implementar el *logout*, basta con llamar al método `logout` del *facade* `Auth` utilizado anteriormente, en el método que se vaya a encargar de esa tarea. Lo podemos añadir en el mismo controlador anterior:

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

class LoginController extends Controller
{
  // ...

  public function logout()
  {
     Auth::logout();
     // ... renderizar la vista deseada
  }
```

También hará falta definir la ruta asociada en `routes/web.php`:

```
Route::get('logout', [LoginController::class, 'logout'])->name('logout');
```

Obviamente, también será necesario añadir un enlace para hacer *logout* en alguna parte. Podemos ponerlo en el menú de navegación (archivo `resources/views/partials/nav.blade.php`, cuando detectemos que el usuario está autenticado):

```
@if(auth()->check())
  <li class="{{ setActivo('libros.create') }} nav-item">
     <a class="nav-link" href="{{ route('libros.create') }}">Nuevo libro</a>
  </li>
  <li class="nav-item">
     <a class="nav-link" href="{{ route('logout') }}">Logout</a>
  </li>
@endif
```

## 1.3. 3. Definir roles. Uso de *middleware*

Para poder definir roles para los distintos usuarios de nuestra aplicación, obviamente debemos comenzar por definir un nuevo campo en la tabla de usuarios para almacenar dicho rol. Deberemos crear la migración correspondiente y lanzarla.

Después, para proteger ciertas rutas en función de los roles, podemos ocultar el enlace en las vistas con una simple comprobación. Por ejemplo, asumiendo que el campo de los roles se llama `rol`:

```
@if (auth()->user()->rol === 'admin')
  // mostrar contenido
@endif
```

Sin embargo, si accedemos a la URL sin pasar por el enlace, podremos ver el contenido. Debemos nuevamente incorporar el *middleware* `auth` al controlador que corresponda (si no lo está ya), para proteger el acceso general para usuarios autenticados.

Además, debemos definir un *middleware* propio que verifique el rol del usuario logueado. Podemos crearlo con este comando:

```
php artisan make:middleware RolCheck
```

En este caso hemos llamado al *middleware* `RolCheck`, pero el nombre puede ser el que queramos. Este *middleware* se creará en la carpeta `App\Http\Middleware`. Debemos editar su método `handle` para verificar que los usuarios son de tipo “admin”:

```
public function handle($request, Closure $next, $rol)
{
  if (auth()->user()->rol === $rol)
     return $next($request);
  else
     return redirect('/');
}
```

Tras definir el *middleware*, lo registramos en el archivo `App/Http/Kernel.php` (en el apartado de *routeMiddleware*):

```
protected $routeMiddleware = [
  // ...
  'roles' => \App\Http\Middleware\RolCheck::class
```

Finalmente, lo cargamos en el constructor de nuestro controlador. Podemos incluir con `except` y `only` restricciones sobre qué métodos del controlador se verán afectados o no por el *middleware*.

```
public function __construct()
{
  $this->middleware(['auth', 'roles:admin']);
}
```

En este ejemplo, hemos mapeado el *middleware* con el alias *roles* en el archivo `Kernel.php`, y lo que hay tras los dos puntos es el parámetro extra que tiene el método `handle` del *middleware* (el rol a comprobar).

### 1.3.1. 3.1. Middleware con más de un parámetro

En el caso de querer poder definir más de un rol en nuestro método del *middleware*, necesitamos definir la función con un número de parámetros variable, de este modo:

```
public function handle($request, Closure $next, ...$roles)
{
  if (in_array(auth()->user()->rol, $roles))
     return $next($request);
  else
     return redirect('/');
}
```

Notar que, en este caso, lo que obtenemos en el parámetro `$roles` es un array, y debemos comprobar si el rol del usuario es uno de los que hay en el array. Ahora podemos cargar un *middleware* basado en múltiples roles, separados por comas, o incluso varios *middleware* para funciones diferentes, en el constructor.

Por ejemplo, este constructor permitiría listar (*index*) a usuarios con rol editor, y permitiría ver la ficha (*show*) a editores y administradores.

```
public function __construct()
{
  $this->middleware(['auth', 'roles:editor'], ['only' => ['index']]);
  $this->middleware(['auth', 'roles:editor,admin'], ['only' => ['show']]);
}
```

### 1.3.2. 3.2. Sobre el concepto de *middleware*

Hemos comentado brevemente el concepto de *middleware* asociado tanto al mecanismo de autenticación como a la clase "extra" que podemos crear para comprobar roles. En general, un **middleware** es un fragmento de código (normalmente una función) que se ejecuta en medio de un proceso. En este caso, se ejecuta desde que se recibe la petición hasta que se emite la respuesta, y permite alterar ese flujo normal, haciendo ciertas comprobaciones sobre la petición. Por ejemplo, como es el caso, verificar que el usuario tiene los permisos adecuados antes de emitir una respuesta u otra.

## 1.4. 4. Más información

Los mecanismos de autenticación de Laravel son muy variados y flexibles. Aquí hemos pretendido ofrecer sólo una parte, quizá la más habitual o estándar. Para más información, podéis consultar la documentación oficial:

- Autenticación con Laravel
- Uso de middleware

# 2. autenticación basada en tokens

En una API REST también puede ser necesario proteger ciertos servicios, de forma que sólo puedan acceder a ellos los usuarios autenticados. Sin embargo, en este caso no tenemos disponible el mecanismo de autenticación basado en sesiones que vimos antes, ya que la parte cliente que consulta la API REST no tiene por qué estar basada en un navegador. Podríamos acceder desde una aplicación de escritorio hecha en Java, por ejemplo, o desde una aplicación móvil, y en estos casos no podríamos disponer de las sesiones, propias de clientes web o navegadores. En su lugar, emplearemos un mecanismo de autenticación basado en **tokens**.

## 2.1. 1. Fundamentos de la autenticación basada en tokens

La autenticación basada en tokens es un mecanismo de validación de usuarios en aplicaciones cliente-servidor que podríamos decir que es más universal que la autenticación basada en sesiones, ya que permite autenticar usuarios provenientes de distintos tipos de clientes. Lo que se hace es lo siguiente:

- El usuario necesita enviar sus credenciales (*login* y *password*), de forma similar a como se hace en una aplicación web normal, aunque esta vez los datos se envían normalmente en formato JSON.
- El servidor valida esas credenciales y, si son correctas, genera una cadena de texto llamada *token*, de una cierta longitud, y que servirá para identificar unívocamente al usuario a partir de ese momento. Dicho *token* debe ser enviado de vuelta (también en formato JSON) al cliente que se validó
- A partir de este punto, el cliente debe adjuntar el *token* como parte de la información en cada petición que realiza a una zona de acceso restringido, de forma que el servidor pueda consultar el token y comprobar si corresponde con el de algún usuario autorizado. Este token normalmente se envía en una cabecera de la petición llamada *Authorization*, como veremos después, y el servidor puede consultar el valor de dicha cabecera para verificar el acceso del cliente.

## 2.2. 2. Alternativas para la implementación de la autenticación basada en tokens

Podemos emplear distintas alternativas para la autenticación basada en tokens bajo Laravel. Comentaremos en esta sesión dos de ellas.

- Por un lado, podemos emplear el mecanismo nativo de Laravel para autenticación basada en tokens. Como ventajas principales, no se necesita instalar ninguna dependencia adicional, y es relativamente sencillo de utilizar. Como inconvenientes, requiere añadir un campo más a la tabla de usuarios, para almacenar el token generado para cada usuario, y requiere también de una gestión manual del token, aunque es sencilla.
- Por otro lado podemos valernos de la librería Laravel Sanctum, que proporciona mecanismos de autenticación para APIs y para SPAs (*Single Page Applications*, aplicaciones de página única). Entre sus ventajas podemos destacar que es sencilla de integrar en la aplicación y automatiza algunos aspectos de la gestión de tokens, además de contar con el soporte oficial de Laravel. Como inconvenientes, es una librería más intrusiva que la anterior, ya que requiere crear una tabla adicional donde almacenar los tokens.

En los siguientes apartados veremos cómo proteger mediante tokens un proyecto sencillo en Laravel empleando cada uno de estos mecanismos. Como ejercicio de este apartado se pide que elijáis cualquiera de ellos y sigáis paso a paso el ejemplo para configurar la protección mediante tokens en él.

### 2.2.1. 2.1. Preparando el ejemplo base

Partiremos de un mismo proyecto base, que luego adaptaremos en función del mecanismo de autenticación elegido. Comenzaremos creando un proyecto llamado `pruebaToken`, en nuestra carpeta de proyectos:

```
laravel new pruebaToken
```

Después, eliminaremos las migraciones que no vamos a utilizar de la carpeta `database/migrations`: en concreto, eliminaremos los archivos sobre *create_password_resets_table* y *create_failed_jobs_table*, y dejaremos el resto. Sobre la migración de usuarios, editaremos los métodos `up` y `down` para dejar sólo los campos que nos interesen, y renombrar la tabla a *usuarios*, de este modo:

```
public function up()
{
    Schema::create('usuarios', function (Blueprint $table) {
        $table->id();
        $table->string('login')->unique;
        $table->string('password');
        $table->timestamps();
    });
}
...
public function down()
{
    Schema::dropIfExists('usuarios');
}
```

A continuación, renombramos el modelo `App\Models\User.php` a `App\Models\Usuario.php`, cambiando también el nombre de la clase interior:

```
class Usuario extends Authenticatable
{
    ...
```

Y hacemos lo mismo con el *factory* y el *seeder* correspondiente (modificamos directamente el `DatabaseSeeder` para no crear un *seeder* específico, en este caso):

```
namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;

/**
 * @extends
\Illuminate\Database\Eloquent\Factories\Factory<\App\Models\Usuario>
 */
class UsuarioFactory extends Factory
```

```
{
    /**
     * Define the model's default state.
     *
     * @return array
     */
    public function definition()
    {
        return [
            'login' => $this->faker->word,
            'password' => bcrypt('1234')
        ];
    }
}
class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
         \App\Models\Usuario::factory(2)->create();
    }
}
```

Vamos a modificar también el archivo `.env` del proyecto para acceder a una base de datos llamada `pruebaToken`, que deberemos crear a través de *phpMyAdmin*:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=pruebaToken
DB_USERNAME=root
DB_PASSWORD=
```

Necesitamos también editar el archivo `App\Http\Middleware\Authenticate.php` para indicar en el método `redirectTo` que sólo queremos redirigir al formulario de login cuando la petición no espere una respuesta en formato JSON. En caso contrario, no hay que mostrar dicho formulario, sino enviar una respuesta JSON adecuada. De hecho, si la aplicación sólo va a tener servicios REST podríamos eliminar o dejar comentado el código de este método para que no trate de redirigir a ningún formulario.

```
class Authenticate extends Middleware
{
    protected function redirectTo($request)
    {
        /*
        if (! $request->expectsJson()) {
            return route('login');
        }
        */
    }
}
```

Por otra parte, debemos editar el archivo `App\Exceptions\Handler.php`, en concreto su método `register` para definir los diferentes errores que pueden producirse y los mensajes que hay que devolver en cada caso:

```
use Illuminate\Auth\AuthenticationException;
use Illuminate\Foundation\Exceptions\Handler as ExceptionHandler;
use Illuminate\Database\Eloquent\ModelNotFoundException;
use Illuminate\Validation\ValidationException;
use Throwable;

class Handler extends ExceptionHandler
{
    ...
    public function register()
    {
        $this->renderable(function (Throwable $exception) {
            if (request()->is('api*'))
            {
                if ($exception instanceof ModelNotFoundException)
                    return response()->json(
                        ['error' => 'Elemento no encontrado'], 404);
                else if ($exception instanceof AuthenticationException)
                    return response()->json(
                        ['error' => 'Usuario no autenticado'], 401);
                else if ($exception instanceof ValidationException)
                    return response()->json(
                        ['error' => 'Datos no válidos'], 400);
                else if (isset($exception))
                    return response()->json(
                        ['error' => 'Error en la aplicación (' .
                        get_class($exception) . '):' .
                        $exception->getMessage()], 500);
            }
        });
    }
}
```

Finalmente, vamos a definir un controlador de API con una serie de métodos de prueba. No lo vamos a vincular a ningún modelo, porque generaremos unos datos a mano en cada método para simplificar el código. Escribimos este comando:

```
php artisan make:controller Api/PruebaController --api
```

Rellenamos el código de los métodos del controlador con alguna respuesta sencilla para cada caso:

```
class PruebaController extends Controller
{
    public function index()
    {
        return response()->json(['mensaje' => 'Accediendo a index']);
    }

    public function store(Request $request)
    {
        return response()->json(['mensaje' => 'Insertando'], 201);
    }

    public function show($id)
    {
        return response()->json(['mensaje' => 'Ficha de ' . $id]);
    }

    public function update(Request $request, $id)
    {
        return response()->json(['mensaje' => 'Actualizando elemento']);
    }

    public function destroy($id)
    {
        return response()->json(['mensaje' => 'Borrando elemento']);
    }
}
```

Y añadimos las rutas correspondientes en el archivo `routes/api.php`:

```
Route::apiResource('prueba', PruebaController::class);
```

A partir de este punto, vamos a proteger el acceso a alguno de estos métodos. Escoge uno de los siguientes apartados (3 o 4) para definir el mecanismo de autenticación basado en tokens correspondiente. También puedes intentar hacerlos todos; en este caso, copia y pega otra vez el proyecto Laravel, para trabajar por separado en cada carpeta con un mecanismo diferente.

## 2.3. 3. Autenticación basada en tokens nativa

Vamos a emplear en esta sección la autenticación nativa por tokens que ofrece Laravel. Los pasos a seguir los indicamos a continuación.

### 2.3.1. 3.1. Configuración básica

En primer lugar, modificamos la migración de la tabla de usuarios para añadir un nuevo campo donde almacenar el token. Dicho campo basta con que tenga 60 caracteres de longitud, y será necesario también que sea único para cada usuario:

```
public function up()
{
    Schema::create('usuarios', function (Blueprint $table) {
        $table->id();
        $table->string('login')->unique;
        $table->string('password');
        $table->string('api_token', 60)->unique()->nullable();
        $table->timestamps();
    });
}
```

Podemos lanzar ya la migración para que se cree la tabla y se rellene con los usuarios que hayamos indicado en el *seeder*.

```
php artisan migrate:fresh --seed
```

También debemos modificar el archivo `config/auth.php` para indicar cuál es el modelo de usuarios que vamos a utilizar:

```
'providers' => [
    'users' => [
        'driver' => 'eloquent',
        'model' => App\Models\Usuario::class,
    ],
```

En este mismo fichero, también podemos modificar el *guard* por defecto, que es *web*, para que sea *api*, si nuestra aplicación no va a tener autenticación web:

```
'defaults' => [
    'guard' => 'api',
    ...
],
```

En la sección de `guards`, añadimos el nuevo *guard* `api`, si no está ya definido:

```
'guards' => [
    'web' => [
        ...
    ],
    'api' => [
        'driver' => 'token',
        'provider' => 'users'
    ]
]
```

## 2.3.2. 3.2. Protección de rutas

Para proteger las rutas de acceso restringido, primero crearemos un controlador que se encargue de validar las credenciales del usuario:

```
php artisan make:controller Api/LoginController
```

Definimos un método `login`, por ejemplo, que validará las credenciales que le lleguen (login y password). Si son correctas, generará una cadena de texto aleatoria de 60 caracteres y la almacenará en el campo `api_token` del usuario validado. También devolverá dicho token como respuesta en formato JSON. En caso de que haya un error en la autenticación, enviará de vuelta un mensaje de error, con el código 401 de acceso no autorizado.

```
use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use Illuminate\Support\Str;
use Illuminate\Support\Facades\Hash;
use App\Models\Usuario;

class LoginController extends Controller
{
    public function login(Request $request)
    {
        $usuario = Usuario::where('login', $request->login)->first();

        if (!$usuario ||
            !Hash::check($request->password, $usuario->password))
        {
            return response()->json(
                ['error' => 'Credenciales no válidas'], 401);
        }
        else
        {
            $usuario->api_token = Str::random(60);
            $usuario->save();
            return response()->json(['token' => $usuario->api_token]);
        }
    }
}
```

Definimos en el archivo `routes/api.php` una ruta que redirija a este método, para cuando el usuario quiera autenticarse (recuerda añadir con `use` la correspondiente clase):

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

También podemos eliminar en este caso la ruta predefinida de este archivo:

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

Para proteger las rutas que necesitemos en los controladores API, las especificamos en el constructor del controlador. Por ejemplo, así protegeríamos todas las rutas de nuestro controlador `PruebaController`, salvo `index` y `show`:

```
class PruebaController extends Controller
{
    public function __construct()
    {
        $this->middleware('auth:api',
            ['except' => ['index', 'show']]);
    }
    ...
```

Alternativamente, también podemos emplear el modificador `only` en lugar de `except` para indicar las rutas concretas que queremos proteger.

Con esto ya tenemos el mecanismo de autenticación por token establecido, y las rutas protegidas. Echa un vistazo al apartado 5 para ver cómo probarlo todo desde *Thunder Client* o Postman.

## 2.4. 4. Autenticación basada en tokens usando Laravel Sanctum

Como hemos comentado anteriormente, Laravel Sanctum es una librería que proporciona mecanismos de autenticación para SPAs (*Single Page Applications*, aplicaciones de página única), y APIs. En nuestro caso, la emplearemos para autenticarnos mediante tokens en nuestras APIs. Los pasos a seguir para la configuración son los siguientes...

### 2.4.1. 4.4.1. Configuración de Sanctum

En primer lugar, debemos incorporar Laravel Sanctum a nuestro proyecto. **En las últimas versiones de Laravel aparece incorporado por defecto** (lo podemos comprobar en el archivo `composer.json`, en la sección de *require*). Si no es así, podemos instalarlo escribiendo este comando desde la raíz del proyecto:

```
composer require laravel/sanctum
```

Si nos diera algún problema de incompatibilidad con la versión de proyecto que tengamos, podemos probar a ejecutar `composer require laravel/sanctum:*` para solucionarlo, e instalar la última versión compatible.

Después, debemos publicar la configuración de Sanctum y su fichero de migración, que generará una tabla adicional donde almacenar los tokens. Escribimos el siguiente comando (todo en una línea, aunque aquí se divide en dos para poderlo ver completo):

```
php artisan vendor:publish
--provider="Laravel\Sanctum\SanctumServiceProvider"
```

Al finalizar este paso, tendremos la migración creada y un archivo de configuración `config/sanctum.php` disponible, para editar la configuración por defecto de la librería. Por ejemplo, podemos editarlo para especificar el tiempo de vida (TTL) de los tokens. El siguiente ejemplo establece un tiempo de vida de 5 minutos, por ejemplo, aunque en el caso de aplicaciones basadas en tokens es habitual dejar tiempos mucho mayores (o indefinidos, según el caso, dejando esta propiedad a `null`):

```
'expiration' => 5,
```

Después, debemos lanzar la migración que se ha creado, junto con las que tengamos pendientes (la de la tabla de usuarios, por ejemplo). Se añadirá una tabla llamada *personal_access_tokens* a nuestra base de datos.

```
php artisan migrate:fresh --seed
```

Finalmente, debemos verificar que el modelo de usuarios (`App\Models\Usuario`) incluya el *trait* `HasApiTokens`. De este modo se vincula el modelo de usuario con los tokens que se vayan a generar para los mismos.

```
...
use Laravel\Sanctum\HasApiTokens;

class Usuario extends Authenticatable
{
    use HasApiTokens, HasFactory, Notifiable;
    ...
```

## 2.4.2. 4.2. Protección de rutas

Para proteger las rutas de acceso restringido, primero crearemos un controlador que se encargue de validar las credenciales del usuario:

```
php artisan make:controller Api/LoginController
```

Definimos un método `login`, por ejemplo, que validará las credenciales que le lleguen (login y password). Si son correctas, llamará al método `createToken` de Sanctum (incorporado al usuario a través del *trait* `HasApiTokens`), asociándolo al login del usuario entrante, y le devolverá el token en formato texto plano, como un objeto JSON. En caso de que haya un error en la autenticación, enviará de vuelta un mensaje de error, con el código 401 de acceso no autorizado.

```
use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use App\Models\Usuario;

class LoginController extends Controller
{
    public function login(Request $request)
    {
        $usuario = Usuario::where('login', $request->login)->first();

        if (!$usuario ||
            !Hash::check($request->password, $usuario->password))
        {
            return response()->json(
                ['error' => 'Credenciales no válidas'], 401);
        }
        else
        {
            return response()->json(['token' =>
                $usuario->createToken($usuario->login)->plainTextToken]);
        }
```

```
    }
}
```

Definimos en el archivo `routes/api.php` una ruta que redirija a este método, para cuando el usuario quiera autenticarse (recuerda añadir con `use` la correspondiente clase):

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

También podemos eliminar en este caso la ruta predefinida de este archivo:

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

Para proteger las rutas que necesitemos en los controladores API, las especificamos en el constructor del controlador. Por ejemplo, así protegeríamos todas las rutas de nuestro controlador `PruebaController`, salvo `index` y `show`:

```
class PruebaController extends Controller
{
    public function __construct()
    {
        $this->middleware('auth:sanctum',
            ['except' => ['index', 'show']]);
    }
    ...
```

**NOTA:** observa cómo se invoca al middleware `auth:sanctum`, que es un mecanismo de autenticación por defecto incorporado en Laravel. Esto hace que no tengamos que tocar la configuración de `config/auth.php` para nada en este caso. Al usar *Sanctum* ya se sabe en qué tabla están los *tokens* almacenados, y vinculados a qué *login*.

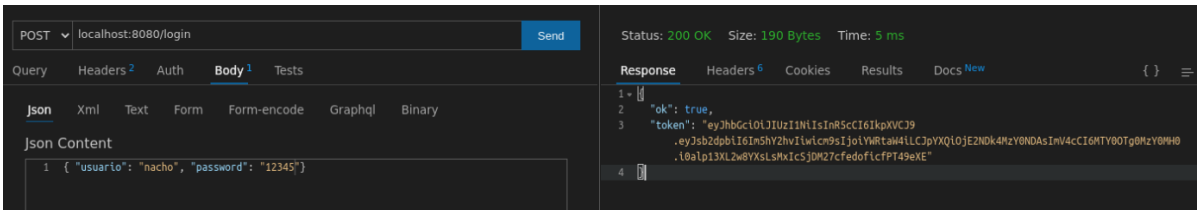
Alternativamente, también podemos emplear el modificador `only` en lugar de `except` para indicar las rutas concretas que queremos proteger.

En el caso de Sanctum, cada vez que validemos un usuario a través de la ruta de *login*, se guardará su *token* (encriptado) en la tabla *personal_access_tokens* de nuestra base de datos MySQL. Podemos comprobarlo con *phpMyAdmin*.

Echa ahora un vistazo al apartado 5 para ver cómo probarlo todo desde *Thunder Client* o Postman.

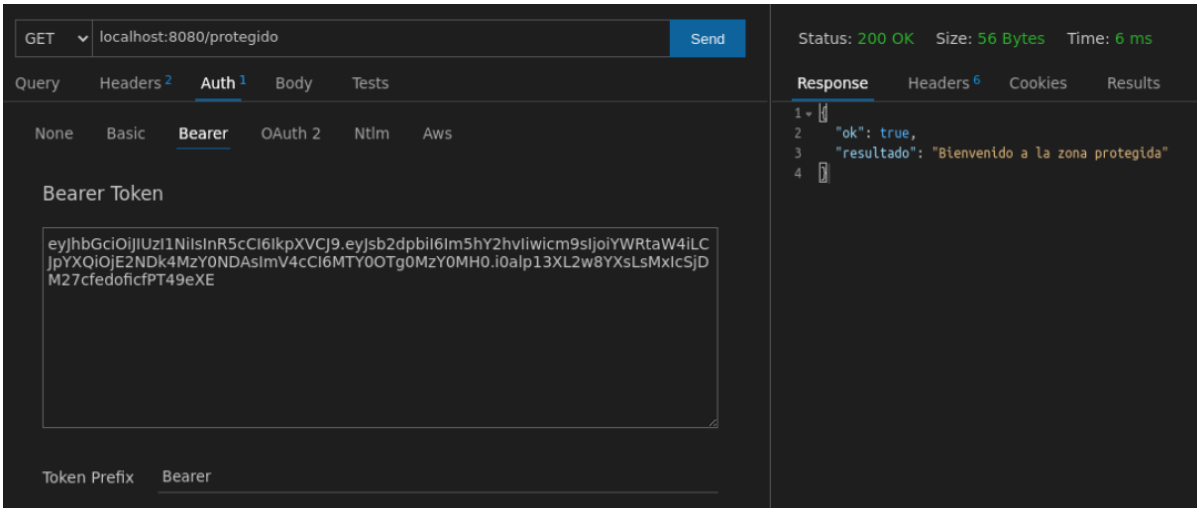
## 2.5. 5. Prueba de autenticación basada en tokens

Veamos ahora cómo probar la autenticación basada en tokens desde una herramienta como *ThunderClient*, para lo que veremos cómo obtener y enviar el token de acceso desde esta herramienta. Lo primero que deberemos hacer es una petición POST para loguearnos. Recibiremos como respuesta el *token* que se haya generado:




**NOTA**: en el ejemplo que hemos estado haciendo, los campos a enviar serían *login* y *password*. Echa un vistazo a *phpMyAdmin* para ver qué *logins* se han generado, y el password, siguiendo el ejemplo, debería ser 1234.

Ahora, sólo nos queda adjuntar este token en la cabecera *Authorization* de las peticiones que lo necesiten. Para ello, vamos a la sección *Authorization* bajo la URL de la petición, y elegimos que queremos enviar un *Bearer token*. En el cuadro inferior nos dejará copiar dicho token:



# 3. ejercicios propuestos

## 3.1. parte I

### 3.1.1. Ejercicio 1

Sobre el proyecto **blog** de la sesión anterior, vamos a añadir estos cambios:

- Modifica el archivo `config/auth.php` para que el *provider* acuda al modelo correcto de usuario.
- Modifica el *factory* de usuarios para que los passwords se encripten con *bcrypt*. Para que sea fácil de recordar, haz que cada usuario tenga como password su mismo login encriptado. Ejecuta después `php artisan migrate:fresh --seed` para actualizar toda la base de datos.
- Crea un formulario de login junto con un controlador asociado, y las rutas pertinentes para mostrar el formulario o autenticar, como hemos hecho en el ejemplo para la biblioteca. Recuerda llamar "login" a la ruta que muestra el formulario de login.
- En el controlador de posts, protege todas las opciones menos las de `index` y `show`.
- Añade una opción de *Login* en el menú de navegación superior, que sólo esté visible si el usuario no se ha autenticado aún
- Haz que sólo se muestren los enlaces y botones de crear, editar o borrar posts cuando el usuario esté autenticado. En ese mismo caso, haz que también se muestre una opción de *logout* en el menú superior, que deberás implementar.
- Finalmente, añade la funcionalidad de que el usuario autenticado sólo puede editar y borrar sus propios posts, pero no los de los demás usuarios.

### 3.1.2. Ejercicio 2

**Opcional**

Continuamos con el proyecto **blog** anterior. Sigue estos pasos para definir una autenticación basada en roles:

- Crea una nueva migración que modifique la tabla de *usuarios* para añadir un nuevo campo llamado *rol*, de tipo *string*. Asegúrate de que la migración sea de modificación, y no de creación de tabla. Después, ejecútala para crear el nuevo campo.
- Haz que alguno de los usuarios de la tabla tenga un rol de *admin* (edítalo a mano desde *phpMyAdmin*), y el resto serán de tipo *editor*.
- Crea un nuevo middleware llamado `RolCheck`, con una función que compruebe si el usuario tiene el rol indicado, como en el ejemplo visto antes en los apuntes. Regístralo adecuadamente en el archivo `App/Http/Kernel.php`, como se ha explicado.
- Modifica las vistas necesarias para que, si el usuario es de tipo *admin* pueda ver los botones de edición y borrado de cualquier post, aunque no sean suyos.
- Modifica los métodos `edit`, `update` y `destroy` de `PostController` para que redirijan a `posts.index` si el usuario no es administrador, o si no es el propietario del post a editar o borrar.

**¿Qué entregar?**

Como entrega de esta sesión deberás comprimir el proyecto **blog** con todos los cambios incorporados, y eliminando las carpetas `vendor` y `node_modules` como se explicó en las sesiones anteriores. Renombra el archivo comprimido a `blog_07.zip`.

## 3.2. parte II

## 3.3. Ejercicio 3

Como ejercicio para practicar la autenticación basada en tokens, se pide que sigáis los pasos indicados en el documento correspondiente, donde se explica cómo proteger aplicaciones basadas en servicios REST mediante tokens. En el punto 2 se indica que existen distintas formas de hacerlo, y en los puntos 3 y 4 se detallan dos de ellas. Lo que se pide como ejercicio es que sigáis los pasos detallados en cualquiera de las dos opciones (3 o 4) para proteger el proyecto de ejemplo que se crea en ese apartado.

Deberéis adjuntar como resultado de este ejercicio el proyecto **pruebaToken** con el código completo para proteger mediante tokens las rutas restringidas del controlador `PruebaController` que habremos definido. También debéis adjuntar la colección de pruebas de Postman que habréis empleado para probar la seguridad de la aplicación.

**¿Qué entregar?**

Como entrega de esta parte deberás comprimir el proyecto `pruebaToken` finalizado. Elimina también la carpeta `vendor`, y añade dentro del ZIP la colección Postman para probar los servicios.

# 4. bibliografia


- Nacho Iborra Baeza.