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**.

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. 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.

2.1. 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 sólo la migración de la tabla de usuarios. Sobre ella, 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
{
    use HasFactory, Notifiable;

    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = [
        'name',
        'password',
    ];

    /**
     * The attributes that should be hidden for arrays.
     *
     * @var array
     */
    protected $hidden = [
        'password',
    ];
}
```

Y hacemos lo mismo con el *factory* y el *seeder* correspondiente:

```php
namespace Database\Factories;

use App\Models\Usuario;
use Illuminate\Database\Eloquent\Factories\Factory;

class UsuarioFactory extends Factory
{
    /**
     * The name of the factory's corresponding model.
     *
     * @var string
     */
    protected $model = Usuario::class;

    /**
     * Define the model's default state.
     *
     * @return array
     */
    public function definition()
    {
        return [
            'login' => $this->faker->word,
            'password' => bcrypt('1234')
        ];
    }
}
```

```php
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.

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.

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',
    ...
],
```

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, que emplea la autenticación nativa de Laravel:

```
// Eliminar esta ruta:
Route::middleware('auth:api')->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 4.5 para ver cómo probarlo todo desde Postman.

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...

4.4.1. Configuración de Sanctum

En primer lugar, debemos incorporar Laravel Sanctum a nuestro proyecto, escribiendo este comando desde la raíz del mismo:

```
composer require laravel/sanctum
```

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 modificar el modelo de usuarios (`App\Models\Usuario`) para añadir 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;
    ...
```

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, que emplea la autenticación nativa de Laravel:

```
// Eliminar esta ruta:
Route::middleware('auth:api')->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']]);
    }
    ...
```

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 de Prueba de servicios REST para ver cómo probarlo todo desde Postman.