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. 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. 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`:

```
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*):

```
public function run()
{
    $usuario = new Usuario();
    $usuario->login = 'admin';
    $usuario->password = bcrypt('admin');
    $usuario->save();
}
```

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:

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

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*.

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" name="enviar" value="Enviar"
         class="btn btn-dark btn-block">

    </form>

@endsection
```

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.

```
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.

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*).

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

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');
}
```

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');
```

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 }}
```

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
```

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). En el caso de querer pasar más parámetros, se puede hacer separados por comas.

3.1. 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.

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