Una Guía Completa sobre la Autenticación en Laravel

La autenticación es una de las características más críticas y esenciales de las aplicaciones web. Los marcos de trabajo web como Laravel proporcionan muchas formas para que los usuarios se autentiquen.

Puedes implementar las funciones de autenticación de Laravel de forma rápida y segura. Sin embargo, implementar mal estas funciones de autenticación puede ser arriesgado, ya que los malintencionados pueden aprovecharse de ellas.

Esta guía te enseñará todo lo que necesitas saber para empezar a utilizar los métodos de autenticación de Laravel que elijas.

¡Sigue leyendo!

Introducción a la Autenticación en Laravel

Laravel introduce módulos que se componen de «guards» y «providers» Los guards definen la autenticación del usuario para cada solicitud, y los providers definen la recuperación del usuario desde un almacenamiento persistente (por ejemplo, una base de datos MySQL).

Definimos nuestros parámetros de autenticación en un archivo llamado `config/auth.php`. Incluye varias opciones para ajustar y modificar el comportamiento de autenticación de Laravel.

En primer lugar, tienes que definir los valores predeterminados de autenticación. Esta opción controla las opciones predeterminadas de «guard» de autenticación y restablecimiento de contraseña de tu aplicación. Puedes cambiar estos valores por defecto según necesites, pero son un comienzo perfecto para la mayoría de las aplicaciones.

A continuación, define los guardias de autenticación para tu aplicación. Aquí, nuestra configuración por defecto utiliza el almacenamiento de sesiones y el proveedor de usuarios de Eloquent. Todos los controladores de autenticación tienen un proveedor de usuarios.

```
<?php

return [
    /*
    Defining Authentication Defaults
    */

    'defaults' => [
        'guard' => 'web',
        'passwords' => 'users',
    ],

    /*
    Defining Authentication Guards

    Supported: "session"
    */

    'guards' => [
        'web' => [
            'driver' => 'session',
            'provider' => 'users',
```

```
        ],
    ],

    /*
    Defining User Providers

    Supported: "database", "eloquent"
    */

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

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

    /*
    Defining Password Resetting
    */

    'passwords' => [
        'users' => [
            'provider' => 'users',
            'table' => 'password_resets',
            'expire' => 60,
            'throttle' => 60,
         ],
    ],

    /*
    Defining Password Confirmation Timeout
    */

    'password_timeout' => 10800,
```

```
    ];
```

Más adelante, nos aseguraremos de que todos los controladores de autenticación tengan un proveedor de usuarios. Éste define cómo se recuperan los usuarios de tu base de datos u otros mecanismos de almacenamiento para persistir los datos de tus usuarios. Puedes configurar varias fuentes que representen a cada modelo o tabla si tienes varias tablas o modelos de usuario. Estas fuentes pueden asignarse a cualquier guardia de autenticación adicional que hayas definido.

Los usuarios también pueden querer restablecer sus contraseñas. Para ello, puedes especificar múltiples configuraciones de restablecimiento de contraseña si tienes más de una tabla o modelo de usuario en la aplicación y quieres configuraciones separadas basadas en los tipos de usuario específicos. El tiempo de caducidad es el número de minutos que será válido cada token de restablecimiento. Esta función de seguridad mantiene los tokens de corta duración, de modo que tienen menos tiempo para ser adivinados. Puedes cambiarlo según necesites.

Por último, debes definir el tiempo que transcurrirá antes de que se agote el tiempo de confirmación de una contraseña, y se pida al usuario que vuelva a introducir su contraseña a través de la pantalla de confirmación. Por defecto, el tiempo de espera dura tres horas.

Tipos de Métodos de Autenticación de Laravel

No existe un método perfecto de autenticación para cada escenario, pero conocerlos te ayudará a tomar mejores decisiones. Esto y cómo Laravel está evolucionando con las nuevas características de Laravel 9. Esto hace que nuestro trabajo como desarrolladores sea mucho más fácil a la hora de cambiar de modo de autenticación.

Autenticación basada en contraseña

Como forma rudimentaria de autenticar a un usuario, todavía es utilizada por miles de organizaciones, pero teniendo en cuenta el desarrollo actual, se está quedando claramente obsoleta.

Los proveedores deben imponer implementaciones de contraseñas complejas, garantizando al mismo tiempo una fricción mínima para el usuario final.

Funciona de forma bastante sencilla, el usuario introduce el nombre y la contraseña, y si en la Base de Datos hay una coincidencia entre esos dos, el servidor decide autenticar la solicitud y dejar que el usuario acceda a los recursos durante un tiempo predefinido.

Autenticación basada en token

Esta metodología se utiliza cuando se emite al usuario un token único tras la verificación.

Con este token, el usuario puede acceder a los recursos pertinentes. El privilegio está activo hasta que caduca el token.

Mientras el token está activo, el usuario no tiene que utilizar ningún nombre de usuario ni contraseña, pero al recuperar un nuevo token, se requieren esos dos.

Hoy en día, los tokens se utilizan ampliamente en múltiples escenarios, ya que son entidades sin estado que contienen todos los datos de autenticación.

Proporcionar una forma de separar la generación de tokens de su verificación da a los vendedores mucha flexibilidad.

Autenticación multifactor

Como su nombre indica, implica utilizar al menos dos factores de autenticación, lo que eleva la seguridad que proporciona.

A diferencia de la autenticación de dos factores, que sólo implica dos factores, este método puede implicar dos, tres, cuatro y más..

La implementación típica de este método implica el uso de una contraseña, tras lo cual se envía al usuario un código de verificación a su smartphone. Los proveedores que implementen este método deben tener cuidado con los falsos positivos y las interrupciones de la red, que pueden convertirse en grandes problemas al escalar rápidamente.

Cómo Implementar la Autenticación en Laravel

Esta sección te enseñará múltiples formas de autenticar a los usuarios de tu aplicación. Algunas librerías como Jetstream, Breeze y Socialite tienen tutoriales gratuitos sobre cómo utilizarlas.

Autenticación manual

Comenzando con el registro de usuarios y la creación de las rutas necesarias en `routes/web.php`.

Crearemos dos rutas, una para ver el formulario y otra para registrarse:

```
use AppHttpContrllersAuthRegisterController;
use IlluminateSupportFacadesRoute;

/*
Web Routes

Register web routes for your app's RouteServiceProvider
in a group containing the "web" middleware
*/

Route::get('/register', [RegisterController::class], 'create']);

Route::post('/register', [RegisterController::class], 'store']);
```

Y crearemos el controlador necesario para ellas:

```
php artisan make controller Auth/RegisterController -r
```

Ahora actualiza el código como sigue:

```
namespace AppHttpControllersAuth;

use AppHttpControllersController;
use illuminateHtppRequest;

class RegisterController extends Controller
{
    public function create()
    {
        return view('auth.register');
    }

    public function store(Request $request)
    {
    }
}
```

El controlador está vacío ahora y devuelve una vista para registrar. Hagamos esa vista en `resources/views/auth` y llamémosla `register.blade.php`.

```blade
<h2> Register </h2>

@if ($errors->any())
    <div>
        <div>Something went wrong!</div>

        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif

<form action="/register" method="POST">
    @csrf

    <div>
        <label for="name">Name</label>
        <input type="text" id="name" name="name" value="{{ old('name') }}" autofocus>
    </div>
    <div>
        <label for="email">Email</label>
        <input type="email" id="email" name="email" value="{{ old('email') }}">
    </div>
    <div>
        <label for="password">Password</label>
        <input type="password" id="password" name="password" value="{{ old('password') }}">
    </div>
    <div>
        <label for="password_confirmation">Password Confirmation</label>
        <input type="password_confirmation" id="password_confirmation" name="password_confirmation"
               value="{{ old('password_confirmation') }}">
    </div>

    <duv>
        <button>Register</button>
    </duv>
</form>
```

— Vista de la hoja de Laravel para registrar usuarios.

Ahora, con todo en su sitio, deberíamos visitar nuestra ruta `/register` y ver el siguiente formulario:

— Formulario de registro para la autenticación manual.

Ahora que podemos mostrar un formulario que un usuario puede completar y obtener los datos para ello, deberíamos obtener los datos de los usuarios, validarlos y luego almacenarlos en la base de datos si todo está bien. Aquí deberías utilizar una transacción de base de datos para asegurarte de que los datos que insertas están completos.

Utilizaremos la función de validación de solicitudes de Laravel para asegurarnos de que se requieren las tres credenciales. Tenemos que asegurarnos de que el correo electrónico tiene formato de email y es único en la tabla `users` y de que la contraseña está confirmada y tiene un mínimo de 8 caracteres:

```
namespace AppHttpControllersAuth;

use AppHttpControllersController;
use IlluminateFoundationAuthUser;
use IlluminateHttpRequest;
use IlluminateSupportFacadesHash;

class RegisterController extends Controller
{
    public function store(Request $request)
    {
```

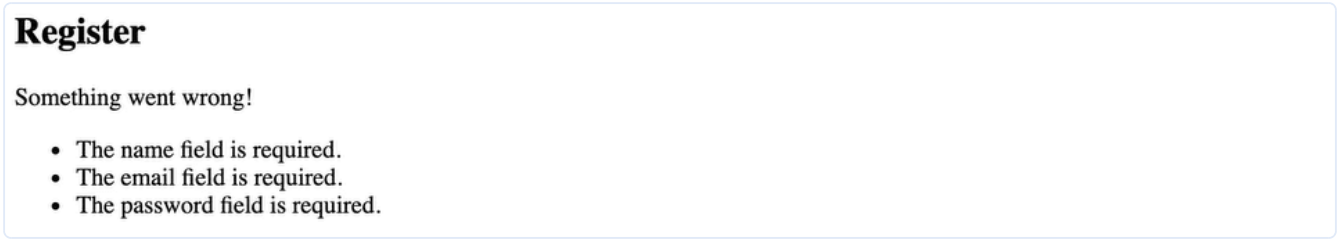
```
            /*
            Validation
            */
            $request->validate([
                'name' => 'required',
                'email' => 'required|email|unique:users',
                'password' => 'required|confirmed|min:8',
            ]);

            /*
            Database Insert
            */
            $user = User:;create([
                'name' => $request->name,
                'email' => $request->email,
                'password' => Hash::make($request->password),
            ]);

            return back();
        }

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

Ahora que nuestra entrada está validada, cualquier cosa que vaya en contra de nuestra validación arrojará un error que se mostrará en el formulario:



— Ejemplo de entrada no válida para registrarse

Suponiendo que hemos creado una cuenta de usuario en el método `store`, también queremos registrar al usuario. Podemos hacerlo de dos formas. Podemos hacerlo manualmente o utilizar la **fachada Auth**.

Después de que el usuario inicie sesión, no debemos devolverlo a la pantalla de Registro, sino a una nueva página, como un panel de control o una página de inicio. Eso es lo que vamos a hacer aquí:

```
namespace AppHttpControllersAuth;

use AppHttpControllersController;
use AppProvidersRouteServiceProvider;
use IlluminateFoundationAuthUser;
use IlluminateHttpRequest;
use IlluminateSupportFacadesAuth;
use IlluminateSupportFacadesHash;

class RegisterController extends Controller
{
    public function store(Request $request)
    {
        /*
        Validation
        */
        $request->validate([
```

```
            'name' => 'required',
            'email' => 'required|email|unique:users',
            'password' => 'required|confirmed|min:8',
        ]);

        /*
        Database Insert
        */
        $user = User:;create([
            'name' => $request->name,
            'email' => $request->email,
            'password' => Hash::make($request->password),
        ]);

        Auth::login($user):

        return redirect(RouteServiceProvider::HOME);
    }

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

Y ahora que tenemos un usuario registrado y conectado `-n`, debemos asegurarnos de que puede desconectarse de forma segura.

Laravel sugiere que invalidemos la sesión y regeneremos el token por seguridad después de un cierre de sesión. Y esto es precisamente lo que vamos a hacer. Empezaremos creando una nueva ruta `/logout` utilizando el métodod`estroy` del **LogoutController**:

```
use AppHttpControllersAuthRegisterController;
use AppHttpControllersAuthLogoutController;
use IlluminateSupportFacadesRoute;

/*
Web Routes

Here is where you can register web routes for your application. These
routes are loaded by the RrouteServiceProvider with a group which
contains the "web" middleware group. Now create something great!
*/

Route::get('/register', [RegisterController::class, 'create']);

Route::post('/register', [RegisterController::class, 'store']);

Route::post('/logout', [Logoutcontroller::class, 'destroy'])
    ->middleware('auth');
```

Pasar el cierre de sesión a través del middleware `auth` es muy importante. Los usuarios no deben poder acceder a la ruta si no han iniciado sesión.

Ahora, crea un controlador como hicimos antes:

```
php artisan make:controller Auth/LogoutController -r
```

Podemos asegurarnos de que obtenemos la solicitud como parámetro en el método `destroy`. Cerramos la sesión del usuario a través de la fachada Auth, invalidamos la sesión y, regeneramos el token, luego redirigimos al usuario a la página de inicio:

```
namespace AppHttpControllersAuth;

use AppHttpControllersController;
use IlluminateHttpRequest;
use IlluminateSupportFacadesAuth;

class LogoutController extends Controller
{
    public function destroy(Request $request)
    {
        Auth::logout();

        $request->session()->invalidate();

        $request->session()->regenerateToken();

        return redirect('/');
     }
}
```

Recordar a los usuarios

La mayoría, si no todas, las aplicaciones web modernas proporcionan una casilla de verificación «recuérdame» en su formulario de inicio de sesión.

Si queremos proporcionar una funcionalidad «recuérdame», podemos pasar un valor booleano como segundo argumento al método attempt.

Cuando sea válido, Laravel mantendrá al usuario autenticado indefinidamente o hasta que se desconecte manualmente. La tabla de usuarios debe incluir la columna string `remember_token` (por eso regeneramos los tokens), donde almacenaremos nuestro token «recuérdame».

La migración por defecto de los usuarios ya la incluye.

Lo primero es lo primero, tienes que añadir el campo » **Remember Me** » a tu formulario:

```
<div>
    <label for="remember">
        <input id="remember" type="checkbox" name="remember">
        <span class="">Remember Me</span>
    </label>
</div>
```

— Añadir campo Remember Me.

Y después de esto, obtén las credenciales de la solicitud y utilízalas en el método attempt de la fachada Auth.

Si el usuario debe ser recordado, iniciaremos sesión con él y le redirigiremos a nuestra página de inicio. En caso contrario, lanzaremos un error:

```
public function store(Request $request)
{
    $credentials = $request->only('email', 'password');

    if (Auth::attempt($credentials, $request->filled('remember'))) {
        $request->session()->regenerate();
```

```
            return redirect()->intended('/');
        }

        return back()->withErrors([
            'email' => 'The provided credentials do not match our records.',
        ]);
    }
```

Restablecer contraseñas

La mayoría de las aplicaciones web actuales ofrecen a los usuarios la posibilidad de restablecer sus contraseñas.

Haremos otra ruta para la contraseña olvidada y crearemos el controlador como hicimos. Además, añadiremos una ruta para el enlace de restablecimiento de contraseña que contenga el token para todo el proceso:

```
Route::post('/forgot-password', [ForgotPasswordLinkController::class, 'st

Route::post('/forgot-password/{token}', [ForgotPasswordController::class,
```

Dentro del método store, tomaremos el email de la petición y lo validaremos como hicimos.

Después de esto, podemos utilizar el método `sendResetLink` de la fachada de contraseña.

Y luego, como respuesta, queremos devolver el estado si se ha conseguido enviar el enlace o errores en caso contrario:

```
namespace AppHttpControllersAuth;

use AppHttpControllersController;
use IlluminateHttpRequest;
use IlluminateSupportFacadesPassword;

class ForgotPasswordLinkController extends Controller
{
    public function reset(Request $request)
    {
        $request->validate([
             'email' => 'required|email',
        ]);

        $status = Password::sendResetLink(
            $request->only('email');
        );

        return $status === Password::RESET_LINK_SENT
            ? back()->with('status', __($status))
            : back()->withInput($request->only('email'))->withErrors(['ema
     }
}
```

Ahora que el enlace de restablecimiento se ha enviado al correo electrónico del usuario, debemos ocuparnos de la lógica de lo que ocurre después.

Obtendremos el token, el correo electrónico y la nueva contraseña en la solicitud y los validaremos.

Después de esto, podemos utilizar el método de restablecimiento de la fachada de contraseña para dejar que Laravel se encargue de todo lo demás entre bastidores.

Siempre haremos un hash de la contraseña para mantenerla segura.

Al final, comprobaremos si se ha restablecido la contraseña y, en caso afirmativo, redirigiremos al usuario a la pantalla de inicio de sesión con un mensaje de éxito. En caso contrario, mostraremos un error indicando que no se ha podido restablecer:

```
namespace AppHttpControllersAuth;

use AppHttpControllersController;
use IlluminateHttpRequest;
use IlluminateSupportFacadesHash;
use IlluminateSupportFacadesPassword;
use IlluminateSupportStr;

class ForgotPasswordController extends Controller
{
    public function store(Request $request)
    {
        $request->validate([
            'token' => 'required',
            'email' => 'required|email',
            'password' => 'required|string|confirmed|min:8',
        ]);

        $status = Password::reset(
            $request->only('email', 'password', 'password_confirmation',
            function ($user) use ($request) {
                $user->forceFill(
                    'password' => Hash::make($request->password),
                    'remember_token' => Str::random(60)
                ])->save();
            }
        );

        return $status == Password::PASSWORD_RESET
            ? redirect()->route('login')->with('status', __($status))
            : back()->withInput($request->only('email'))->withErrors(['ema
    }
```

```
}
```

Laravel Breeze

Laravel Breeze es una implementación sencilla de las funciones de autenticación de Laravel: inicio de sesión, registro, restablecimiento de contraseña, verificación por correo electrónico y confirmación de contraseña. Puedes utilizarla para implementar la autenticación en tu nueva aplicación Laravel.

Instalación y configuración

Después de crear tu aplicación Laravel, todo lo que tienes que hacer es configurar tu base de datos, ejecutar tus migraciones e instalar el paquete laravel/breeze a través de composer:

```
composer require laravel/breeze –dev
```

Después de esto, ejecuta lo siguiente:

```
php artisan breeze:install
```

Que publicará tus vistas de autenticación, rutas, controladores y otros recursos que utilice. Después de este paso, tendrás el control total de todo lo que Breeze proporciona.

Ahora tenemos que renderizar nuestra aplicación en el frontend, así que instalaremos nuestras dependencias JS (que utilizarán @vite):

```
npm install
```

:

```
npm run dev
```

Después de esto, los enlaces de inicio de sesión y registro deberían aparecer en tu página de inicio, y todo debería funcionar sin problemas.

Laravel Jetstream

Laravel Jetstream extiende las funciones de Laravel Breeze con funciones útiles y otras pilas de frontend.

Proporciona inicio de sesión, registro, verificación de correo electrónico, autenticación de dos factores, gestión de sesiones, compatibilidad con API a través de Sanctum y gestión de equipos opcional.

Debes elegir entre Livewire e Inertia en el frontend al instalar Jetstream. En el backend, utiliza Laravel Fortify, que es un backend de autenticación «headless» para Laravel, agnóstico al frontend.

Instalación y configuración

Lo instalaremos a través de Composer en nuestro Proyecto Laravel:

```
composer require laravel/jetstream
```

A continuación, ejecutaremos el comando `php artisan jetstream:install [stack]`, que acepta los argumentos `[stack]` `Livewire` o `Inertia`. Puedes pasar la opción `–team` para activar la función de equipos.

Esto también instalará Pest PHP para las pruebas.

Y por último, tenemos que renderizar el frontend de nuestra aplicación utilizando lo siguiente:

```
npm install
npm run dev
```

Laravel Fortify

Laravel Fortify es una implementación de autenticación backend que es agnóstica al frontend. No tienes que usar Laravel Fortify para implementar las características de autenticación de Laravel.

También se utiliza en kits de inicio como Breeze y Jetstream. También puedes utilizar Fortify de forma independiente, que no es más que una implementación de backend. Si lo utilizas de forma independiente, tu frontend debe llamar a las rutas de Fortify.

Instalación y configuración

Podemos instalar Fortify a través de Composer:

```
composer require laravel/fortify
```

Ahora tenemos que publicar los recursos de Fortify:

```
php artisan vendor:publish –provider="LaravelFortifyFortifyServiceProvide
```

Después de esto, crearemos un nuevo directorio **app/Actions**, además del nuevo **FortifyServiceProvider**, el archivo de configuración y las migraciones de base de datos.

Por último, ejecuta:

```
php artisan migrate
```

O

```
php artisan migrate:fresh
```

Y tu Fortify está listo para usar.

Laravel Socialite

Laravel incluye una sencilla función de autenticación de usuarios basada en OAuth. Acepta inicios de sesión de redes sociales a través de Facebook, Twitter, LinkedIn, Google, Bitbucket, GitHub y GitLab.

Instalación

Podemos instalarlo a través de Composer:

```
composer require laravel/socialite
```

Instalación y uso

Después de instalarlo, tenemos que añadir las credenciales para el proveedor OAuth que utiliza nuestra aplicación. Las añadiremos en **config/services.php** para cada servicio.

En la configuración, debemos hacer coincidir la clave con los servicios anteriores. Algunas de esas claves son

- facebook
- twitter (Para OAuth 1.0)
- twitter-oauth-2 (Para OAuth 2.0)

- linkedin
- google
- github
- gitlab
- bitbucket

La configuración de un servicio puede tener este aspecto

```
'google' => [
    'client_id' => env("GOOGLE_CLIENT_ID"),
    'client_secret' => env("GOOGLE_CLIENT_SECRET"),
    'redirect' => "http://example.com/callback-url",
],
```

Autenticar usuarios

Para esta acción, necesitaremos dos rutas, una para redirigir al usuario al proveedor de OAuth:

```
use LaravelSocialiteFacadesSociliate;

Route::get('/auth/redirect', function () {
    return Socialite:;driver('google')->redirect();
});
```

Y otra para la llamada de retorno del proveedor tras la autenticación:

```
use LaravelSocialiteFacadesSocialite;

Route:;get('/auht/callback', function () {
    $user = Socialite:;driver('google')->user();

    // Getting the user data
    $user->token;
});
```

Socialite proporciona el método de redirección, y la fachada redirige al usuario al proveedor de OAuth, mientras que el método de usuario examina la solicitud entrante y recupera la información del usuario.

Después de recibir nuestro usuario, tenemos que comprobar si existe en nuestra base de datos y autenticarlo. Si no existe, crearemos un nuevo registro para representar al usuario:

```
use AppModelsUser;
use IlluminateSupportFacadesAuth;
use LaravelSocialiteFacadesSocialite;

Route::get('/auth/callback', function () {
   /*
   Get the user
   */
   $googleUser = Socialite::driver('google')->user();

   /*
   Create the user if it does not exist
   Update the user if it exists

   Check for google_id in database
```

```
    */
    $user = User::updateOrCreate([
        'google_id' => $googleUser->id,
    ], [
        'name' => $googleUser->name,
        'email' => $googleUser->email,
        'google_token' => $googleUser->token,
        'google_refresh_token' => $googleUser->refreshToken,
    ]);

    /*
    Authenticates the user using the Auth facade
    */
     Auth::login($user);

     return redirect('/dashboard');
});
```

Si queremos limitar los ámbitos de acceso del usuario, podemos utilizar el método `scopes`, que incluiremos con la solicitud de autenticación. Esto fusionará todos los ámbitos especificados anteriormente con los especificados.

Una alternativa a esto es utilizar el método `setScopes`, que sobrescribe cualquier otro ámbito existente:

```
use LaravelSocialiteFacadesSocialite;

return Socialite::driver('google')
    ->scopes(['read:user', 'write:user', 'public_repo'])
    ->redirect();

return Socialite::driver('google')
    ->setScopes(['read:user', 'public_repo')
```

```
        ->redirect();
```

Ahora que lo sabemos todo y cómo obtener un usuario tras la llamada de retorno, veamos algunos de los datos que podemos obtener de él.

Usuario OAuth1 tiene `token` y `tokenSecret`:

```
$user = Socialite::driver('google')->user();

$token = $user->token;
$tokenSecret = $user->tokenSecret;
```

OAuth2 proporciona `token`, `refreshToken`, y `expiresIn`:

```
$user = Socialite::driver('google')->user();

$token = $user->token;
$refreshToken = $user->refreshToken;
$expiresIn = $user->expiresIn;
```

Tanto OAuth1 como OAuth2 proporcionan `getId`, `getNickname`, `getName`, `getEmail`, y `getAvatar`:

```
$user = Socialite::driver('google')->user();

$user->getId();
$user->getNickName();
$user->getName();
$user->getEmail();
$user->getAvatar();
```

Y si queremos obtener detalles del usuario a partir de un token (OAuth 2) o de un token y un secreto (OAuth 1), sanctum proporciona dos métodos para ello: `userFromToken` y `userFromTokenAndSecret`:

```
use LaravelSocialiteFacadesSocialite;

$user = Socialite:;driver('google')->userFromToken($token);
$user = Socialite::driver('twitter')->userFromTokenAndSecret($token, $sec
```

Laravel Sanctum

Laravel Sanctum es un sistema de autenticación ligero para SPA (Single Page Applications) y aplicaciones móviles. Permite a los usuarios generar múltiples tokens de API con ámbitos específicos. Estos ámbitos especifican las acciones permitidas por un token.

Usos

Sanctum puede utilizarse para emitir tokens de API al usuario sin las complejidades de OAuth. Esos tokens suelen tener tiempos de caducidad largos, como años, pero pueden ser

revocados y regenerados por el usuario en cualquier momento.

Instalación y configuración

Podemos instalarlo mediante Composer:

```
composer require laravel/sanctum
```

Y tenemos que publicar los archivos de configuración y migración:

```
php artisan vendor:publish –provider="LaravelSanctumSanctumServiceProvide
```

Ahora que hemos generado nuevos archivos de migración, tenemos que migrarlos:

```
php artisan migrate </code> or <code> php artisan migrate:fresh
```

Cómo emitir tokens de API

Antes de emitir tokens, nuestro modelo de Usuario debe utilizar el rasgo **Laravel\Sanctum\HasApiTokens**:

```
use LaravelSanctumHasApiTokens;

class User extends Authenticable
{
    use HasApiTokens;
}
```

Cuando tengamos el usuario, podemos emitir un token llamando al método `createToken`, que devuelve una instancia **Laravel\Sanctum\NewAccessToken**.

Podemos llamar al método `plainTextToken` en la instancia **NewAccessToken** para ver el valor en texto plano **SHA-256** del token.

Consejos y Buenas Prácticas para la Autenticación en Laravel

Invalidar sesiones en otros dispositivos

Como hemos comentado anteriormente, invalidar la sesión es crucial cuando el usuario cierra la sesión, pero eso también debería estar disponible como opción para todos los dispositivos propios.

Esta función se suele utilizar cuando el usuario cambia o actualiza su contraseña, y queremos invalidar su sesión desde cualquier otro dispositivo.

Con la fachada Auth, esto es una tarea fácil de conseguir. Teniendo en cuenta que la ruta que estamos utilizando tiene los métodos `auth` y `auth.session middleware`, podemos utilizar el método estático `logoutOtherDevices` de la fachada:

```
Route::get('/logout', [LogoutController::class, 'invoke'])
    ->middleware(['auth', 'auth.session']);
```

```
use IlluminateSupportFacadesAuth;

Auth::logoutOtherDevices($password);
```

Configuración con Auth::routes()

El método routes de la fachada Auth es sólo un ayudante para generar todas las rutas necesarias para la autenticación de usuarios.

Las rutas incluyen Iniciar sesión (Get, Post), Cerrar sesión (Post), Registrarse (Get, Post) y Restablecer contraseña/correo electrónico (Get, Post).

Cuando llamas al método de la fachada, ésta hace lo siguiente:

```
public static fucntion routes(array $options = [])
{
    if (!static::$app->providerIsLoaded(UiServiceProvider::class)) {
        throw new RuntimeException('In order to use the Auth:;routes() me
    }

    static::$app->make('router')->auth($options);
}
```

Nos interesa saber qué ocurre cuando se llama al método estático en el router. Esto puede ser complicado debido al hecho de cómo funcionan las fachadas, pero el siguiente método llamado es así:

```
/**
Register the typical authentication routes for an application.

@param array $options
@return void
*/
public function auth(array $options = [])
{
    // Authentication Routes...
    $this->get('login', 'AuthLoginController@showLoginForm')->name('login
    $this->post('login', 'AuthLoginController@login');
    $this->post('logout', 'AuthLoginController@logout')->name('logout');

    // Registration Routes...
    if ($options['register'] ?? true) {
        $this->get('register', 'AuthRegisterController@showRegistrationFo
        $this->post('register', 'AuthRegisterController@register');
    }

    // Password Reset Routes...
    if ($options['reset'] ?? true) {
        $this->resetPassword();
    }

    // Email Verification Routes...
    if ($options['verify'] ?? false) {
        $this->emailVerification();
    }
}
```

Por defecto, genera todas las rutas excepto la de verificación de correo electrónico. Siempre tendremos las rutas Login y Logout, pero las demás las podemos controlar a través del array de opciones.

Si queremos tener sólo login/logout y registro, podemos pasar el siguiente array de opciones:

```
$options = ["register" => true, "reset" => false, "verify" => false];
```

Proteger rutas y guardias personalizadas

Queremos asegurarnos de que sólo los usuarios autenticados pueden acceder a algunas rutas y esto se puede hacer rápidamente añadiendo una llamada al método middleware en la fachada Ruta o encadenando el método middleware en ella:

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

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

Esta guardia garantiza que las peticiones entrantes estén autenticadas.

Confirmación de contraseña

Para mayor seguridad del sitio web, a menudo querrás confirmar la contraseña de un usuario antes de continuar con cualquier otra tarea.

Debemos definir una ruta desde la vista de confirmación de contraseña para gestionar la solicitud. Validará y redirigirá al usuario a su destino. Al mismo tiempo, nos aseguraremos de que nuestra contraseña aparezca confirmada en la sesión. Por defecto, la contraseña debe reconfirmarse cada tres horas, pero esto puede cambiarse en el archivo de configuración en **config/auth.php**:

```
use IlluminateHttpRequest;
use IlluminateSupportFacadesHash;
use IlluminateSupportFacadesRedirect;

Route::post('/confirm-password', function (Request $request) {
    if (!Hash::check($request->password, $request->user()->password)) {
        return back()->withErrors([
            'password' => ['The provided password does not match our reco
        ]);
    }

    $request->session()->passwordConfirmed();

    return redirect()->intended();
})->middleware(['auth']);
```

Contrato Autenticable

El contrato Autenticable situado en IlluminateContractsAuth define un modelo de lo que debe implementar la fachada UserProvider:

```
namespace IlluminateContractsAuth;
```

```
interface Authenticable
{
    public function getAuthIdentifierName();

    public function getAuthIdentifier();

    public function getAuthPassord();

    public function getRememberToken();

    public function setRememberToken($value);

    public function getrememberTokenName();
}
```

La interfaz permite al sistema de autenticación trabajar con cualquier clase de «usuario» que la implemente.

Esto es válido independientemente de qué ORM o capas de almacenamiento se utilicen. Por defecto, Laravel tiene el AppModelsUser que implementa esta interfaz, y esto también se puede ver en el archivo de configuración:

```
return [
    'providers' => [
        'users' => [
            'driver' => 'eloquent',
            'model' => AppModelsUser::class,
        ],
     ],
];
```

Eventos de autenticación

Hay un montón de eventos que se envían durante todo el proceso de autenticación.

Dependiendo de tus objetivos, puedes adjuntar oyentes a esos eventos en tu `EventServiceProvider` .

```php
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Auth\Events\Registered' => [
        'App\Listeners\LogRegisteredUser',
    ],

    'Illuminate\Auth\Events\Attempting' => [
        'App\Listeners\LogAuthenticationAttempt',
    ],

    'Illuminate\Auth\Events\Authenticated' => [
        'App\Listeners\LogAuthenticated',
    ],

    'Illuminate\Auth\Events\Login' => [
        'App\Listeners\LogSuccessfulLogin',
    ],

    'Illuminate\Auth\Events\Failed' => [
        'App\Listeners\LogFailedLogin',
    ],

    'Illuminate\Auth\Events\Validated' => [
        'App\Listeners\LogValidated',
    ],

    'Illuminate\Auth\Events\Verified' => [
        'App\Listeners\LogVerified',
    ],

    'Illuminate\Auth\Events\Logout' => [
        'App\Listeners\LogSuccessfulLogout',
    ],

    'Illuminate\Auth\Events\CurrentDeviceLogout' => [
        'App\Listeners\LogCurrentDeviceLogout',
    ],

    'Illuminate\Auth\Events\OtherDeviceLogout' => [
        'App\Listeners\LogOtherDeviceLogout',
    ],

    'Illuminate\Auth\Events\Lockout' => [
        'App\Listeners\LogLockout',
    ],

    'Illuminate\Auth\Events\PasswordReset' => [
        'App\Listeners\LogPasswordReset',
    ],
];
```

Crear rápidamente nuevos usuarios

Crear un nuevo usuario rápidamente se puede hacer a través del **App\User**:

```
$user = new AppUser();
$user->password = Hash::make('strong_password');
$user->email = 'test-email@user.com';
$user->name = 'Username';
$user->save();
```

O a través del método estático crear de la fachada Usuario:

```
User::create([
 'password' => Hash::make('strong-password'),
 'email' => 'test-email@user.com',
 'name' => 'username'
]);
```

Resumen

El ecosistema Laravel tiene un montón de kits de inicio para poner en marcha tu aplicación con un sistema de autenticación, como Breeze y Jetstream. Son altamente personalizables, ya que el código se genera en nuestro lado, y podemos modificarlo tanto como queramos, usándolo como modelo si es necesario.

Hay muchos problemas de seguridad relacionados con la autenticación y sus entresijos, pero todos ellos pueden resolverse fácilmente mediante las herramientas que proporciona Laravel. Estas herramientas son altamente personalizables y fáciles de usar.

Despliega tus aplicaciones Laravel de forma rápida y eficaz con nuestro rápido servicio de alojamiento Laravel. Ve tu aplicación en acción con una prueba gratuita.