Un guide complet de l’authentification Laravel

kinsta

L’authentification est l’une des fonctions les plus critiques et les plus essentielles des applications web. Les frameworks web tels que Laravel offrent de nombreuses possibilités d’authentification pour les utilisateurs.

Vous pouvez mettre en œuvre les fonctions d’authentification de Laravel rapidement et en toute sécurité. Cependant, une mauvaise implémentation de ces fonctions d’authentification peut s’avérer risquée, car des parties malveillantes peuvent les exploiter.

Ce guide vous apprendra tout ce que vous devez savoir pour commencer à utiliser les méthodes d’authentification Laravel de votre choix.

Poursuivez votre lecture !

Introduction à l’authentification Laravel

Laravel introduit des modules composés de « guards (ou gardes) » et de « providers (ou fournisseurs) ».Les guards définissent l’authentification de l’utilisateur pour chaque requête, et les providers définissent la récupération de l’utilisateur à partir d’un stockage persistant (par exemple, une base de données MySQL).

Nous définissons nos paramètres d'authentification dans un fichier nommé `config/auth.php`. Il comprend plusieurs options permettant d'ajuster et de modifier le comportement de Laravel en matière d'authentification.

Tout d'abord, vous devez définir les paramètres d'authentification par défaut. Cette option contrôle les options par défaut d'authentification « guard » et de réinitialisation du mot de passe de votre application. Vous pouvez modifier ces valeurs par défaut si nécessaire, mais elles constituent un bon point de départ pour la plupart des applications.

Ensuite, vous définissez les guards d'authentification pour votre application. Ici, notre configuration par défaut utilise le stockage de session et le fournisseur d'utilisateurs Eloquent. Tous les pilotes d'authentification ont un fournisseur d'utilisateurs.

```
<?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,
```

```
    ];
```

Plus loin, nous nous assurerons que tous les pilotes d'authentification disposent d'un fournisseur d'utilisateurs. Celui-ci définit la manière dont les utilisateurs sont extraits de votre base de données ou d'autres mécanismes de stockage afin de conserver les données de l'utilisateur. Vous pouvez configurer plusieurs sources représentant chaque modèle ou table si vous avez plusieurs tables ou modèles d'utilisateurs. Ces sources peuvent être affectées à toutes les gardes d'authentification supplémentaires que vous avez définies.

Les utilisateurs peuvent également souhaiter réinitialiser leur mot de passe. Pour cela, vous pouvez spécifier plusieurs configurations de réinitialisation de mot de passe si vous avez plusieurs tables ou modèles d'utilisateurs dans l'application et que vous souhaitez des paramètres distincts en fonction des types d'utilisateurs spécifiques. Le délai d'expiration correspond au nombre de minutes pendant lesquelles chaque jeton de réinitialisation sera valide. Cette fonction de sécurité permet aux jetons d'avoir une durée de vie courte, de sorte qu'ils ont moins de temps pour être devinés. Vous pouvez modifier ce paramètre selon vos besoins.

Enfin, vous devez définir le délai d'attente avant qu'une confirmation de mot de passe ne se termine et que l'utilisateur soit invité à saisir à nouveau son mot de passe dans l'écran de confirmation. Par défaut, le délai d'attente est de trois heures.

Types de méthodes d'authentification Laravel

Il n'existe pas de méthode d'authentification parfaite pour tous les scénarios, mais le fait de les connaître vous aidera à prendre de meilleures décisions. C'est ainsi que Laravel évolue avec les nouvelles fonctionnalités de Laravel 9. Cela rend notre travail de développeur beaucoup plus facile lorsque nous changeons de mode d'authentification.

Authentification par mot de passe

Cette méthode rudimentaire d'authentification d'un utilisateur est encore utilisée par des milliers d'organisations, mais compte tenu du développement actuel, elle est clairement en

train de devenir obsolète.

Les fournisseurs doivent mettre en œuvre des mots de passe complexes tout en garantissant une friction minimale pour l'utilisateur final.

Le fonctionnement est assez simple : l'utilisateur saisit son nom et son mot de passe, et si la base de données contient une correspondance entre ces deux éléments, le serveur décide d'authentifier la demande et de laisser l'utilisateur accéder aux ressources pendant une durée prédéfinie.

Authentification par jeton

Cette méthode est utilisée lorsque l'utilisateur reçoit un jeton unique après vérification.

Grâce à ce jeton, l'utilisateur peut accéder aux ressources pertinentes. Le privilège est actif jusqu'à l'expiration du jeton.

Tant que le jeton est actif, l'utilisateur n'a pas besoin d'utiliser de nom d'utilisateur ou de mot de passe, mais lorsqu'il récupère un nouveau jeton, ces deux éléments sont nécessaires.

Les jetons sont aujourd'hui largement utilisés dans de nombreux scénarios, car ce sont des entités sans état qui contiennent toutes les données d'authentification.

Le fait de pouvoir séparer la génération et la vérification des jetons offre aux fournisseurs une grande flexibilité.

Authentification multifactorielle

Comme son nom l'indique, elle implique l'utilisation d'au moins deux facteurs d'authentification, ce qui accroît la sécurité qu'elle offre.

Contrairement à l'authentification à deux facteurs qui n'implique que deux facteurs, cette méthode peut impliquer deux, trois, quatre, et plus..

La mise en œuvre typique de cette méthode implique l'utilisation d'un mot de passe, après quoi l'utilisateur reçoit un code de vérification sur son smartphone. Les fournisseurs qui

mettent en œuvre cette méthode doivent se méfier des faux positifs et des pannes de réseau, qui peuvent devenir de gros problèmes lors d'une mise à l'échelle rapide.

Comment implémenter l'authentification Laravel

Cette section vous apprendra plusieurs façons d'authentifier les utilisateurs de votre application. Certaines bibliothèques comme Jetstream, Breeze, et Socialite ont des tutoriels gratuits sur la façon de les utiliser.

Authentification manuelle

En commençant par l'enregistrement des utilisateurs et la création des routes nécessaires dans `routes/web.php`.

Nous allons créer deux routes, l'une pour visualiser le formulaire et l'autre pour s'enregistrer :

```
use App\Http\Controllers\Auth\RegisterController;
use Illuminate\Support\Facades\Route;

/*
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']);
```

Et nous créerons le contrôleur nécessaire pour ces routes :

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

Mettez maintenant le code à jour comme suit :

```
namespace App\Http\Controllers\Auth;

use App\Http\Controllers\Controller;
use illuminate\Htpp\Request;

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

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

Le contrôleur est maintenant vide et renvoie une vue à enregistrer. Créons cette vue dans `resources/views/auth` et appelons-la `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>
```

— Vue Laravel blade pour l’enregistrement des utilisateurs.

Maintenant que tout est en place, nous devrions visiter notre route `/register` et voir le formulaire suivant :

— Formulaire d’inscription pour l’authentification manuelle.

Maintenant que nous pouvons afficher un formulaire que l’utilisateur peut remplir et obtenir les données correspondantes, nous devons récupérer les données de l’utilisateur, les valider, puis les stocker dans la base de données si tout va bien. Dans ce cas, vous devez utiliser une transaction de base de données pour vous assurer que les données que vous insérez sont complètes.

Nous utiliserons la fonction de validation des requêtes de Laravel pour nous assurer que les trois informations d’identification sont nécessaires. Nous devons nous assurer que l’e-mail a un format e-mail et est unique dans la table `users` et que le mot de passe est confirmé et a un minimum de 8 caractères :

```
namespace App\Http\Controllers\Auth;

use App\Http\Controllers\Controller;
use Illuminate\Foundation\Auth\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;

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

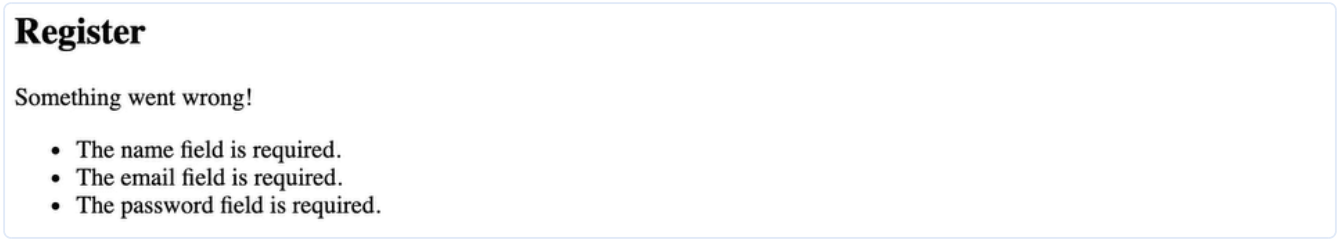
```
        {
            /*
            Validation
            */
            $request->validate([
                'name' => 'required',
                'email' => 'required|email|unique:users',
                'password' => 'required|confirmed|min:8',
            ]);

            /*
            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');
        }
    }
```

Maintenant que notre entrée est validée, tout ce qui va à l’encontre de notre validation va générer une erreur qui sera affichée dans le formulaire :



— Exemple d’entrée non valide pour l’enregistrement

En supposant que nous ayons créé un compte d’utilisateur dans la méthode `store`, nous voulons également nous connecter à l’utilisateur. Nous pouvons le faire de deux manières. Nous pouvons le faire manuellement ou utiliser la **façade Auth**.

Une fois que l’utilisateur s’est connecté, nous ne devons pas le renvoyer à l’écran d’inscription, mais plutôt à une nouvelle page, comme un tableau de bord ou une page d’accueil. C’est ce que nous allons faire ici :

```
namespace App\Http\Controllers\Auth;

use App\Http\Controllers\Controller;
use App\Providers\RouteServiceProvider;
use Illuminate\Foundation\Auth\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;
use Illuminate\Support\Facades\Hash;

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');
    }
}
```

Maintenant qu'un utilisateur est enregistré et connecté `-n`, nous devons nous assurer qu'ils peuvent se déconnecter en toute sécurité.

Laravel suggère d'invalider la session et de régénérer le jeton de sécurité après une déconnexion. Et c'est précisément ce que nous allons faire. Nous commençons par créer une nouvelle route `/logout` en utilisant la méthode `destroy` du **LogoutController** :

```
use App\Http\Controllers\Auth\RegisterController;
use App\Http\Controllers\Auth\LogoutController;
use Illuminate\Support\Facades\Route;

/*
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');
```

Le passage de la déconnexion par l'intergiciel `auth` est très important. Les utilisateurs doivent être incapables d'accéder à la route s'ils ne sont pas connectés.

Maintenant, créez un contrôleur comme nous l'avons fait précédemment :

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

Nous pouvons nous assurer que nous recevons la requête en tant que paramètre dans la méthode `destroy`. Nous déconnectons l'utilisateur via la façade Auth, invalidons la session et régénérons le jeton, puis redirigeons l'utilisateur vers la page d'accueil :

```
namespace App\Http\Controllers\Auth;

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

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

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

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

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

Se souvenir des utilisateurs

La plupart des applications web modernes, si ce n'est toutes, proposent une case à cocher « se souvenir de moi » dans leur formulaire de connexion.

Si nous voulons fournir une fonctionnalité « se souvenir de moi », nous pouvons passer une valeur booléenne comme deuxième argument de la méthode attempt.

Si cette valeur est valide, Laravel conservera l'utilisateur authentifié indéfiniment ou jusqu'à ce qu'il soit déconnecté manuellement. La table des utilisateurs doit inclure la colonne `remember_token` (c'est pourquoi nous régénérons les jetons), dans laquelle nous stockerons notre jeton « remember me ».

La migration par défaut des utilisateurs l'inclut déjà.

Tout d’abord, vous devez ajouter le champ **Remember Me** à votre formulaire :

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

— Ajouter le champ Remember Me à partir de.

Ensuite, récupérez les informations d’identification de la requête et utilisez-les dans la méthode attempt de la façade Auth.

Si l’utilisateur doit être mémorisé, nous le connecterons et le redirigerons vers notre page d’accueil. Dans le cas contraire, nous enverrons une erreur :

```
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.',
        ]);
    }
```

Réinitialisation des mots de passe

La plupart des applications web actuelles permettent aux utilisateurs de réinitialiser leur mot de passe.

Nous allons créer une autre route pour le mot de passe oublié et créer le contrôleur comme nous l'avons fait. En outre, nous ajouterons une route pour le lien de réinitialisation du mot de passe qui contient le jeton pour l'ensemble du processus :

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

Dans la méthode store, nous prendrons l'e-mail de la requête et le validerons comme nous l'avons fait.

Après cela, nous pouvons utiliser la méthode `sendResetLink` de la façade password.

Enfin, en guise de réponse, nous voulons renvoyer le statut si l'envoi du lien a réussi ou les erreurs dans le cas contraire :

```
namespace App\Http\Controllers\Auth;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Password;

class ForgotPasswordLinkController extends Controller
{
    public function store(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
     }
}
```

Maintenant que le lien de réinitialisation a été envoyé à l'e-mail de l'utilisateur, nous devons nous occuper de la logique de ce qui se passe ensuite.

Nous allons récupérer le jeton, l'email et le nouveau mot de passe dans la requête et les valider.

Après cela, nous pouvons utiliser la méthode reset de la façade du mot de passe pour laisser Laravel s'occuper de tout le reste en coulisses.

Nous allons toujours hacher le mot de passe pour le sécuriser.

À la fin, nous vérifierons si le mot de passe a été réinitialisé, et si c'est le cas, nous redirigerons l'utilisateur vers l'écran de connexion avec un message de succès. Dans le cas contraire, nous affichons une erreur indiquant que le mot de passe n'a pas pu être réinitialisé :

```
namespace App\Http\Controllers\Auth;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Facades\Password;
use Illuminate\Support\Str;

class ForgotPasswordController extends Controller
{
    public function reset(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 est une implémentation simple des fonctions d'authentification de Laravel : connexion, enregistrement, réinitialisation du mot de passe, vérification de l'e-mail et confirmation du mot de passe. Vous pouvez l'utiliser pour implémenter l'authentification dans votre nouvelle application Laravel.

Installation et configuration

Après avoir créé votre application Laravel, tout ce que vous avez à faire est de configurer votre base de données, d'exécuter vos migrations et d'installer le paquetage laravel/breeze via composer :

```
composer require laravel/breeze --dev
```

Après cela, exécutez ce qui suit :

```
php artisan breeze:install
```

Ce dernier publiera vos vues d'authentification, routes, contrôleurs et autres ressources qu'il utilise. Après cette étape, vous avez un contrôle total sur tout ce que Breeze fournit.

Maintenant, nous devons rendre notre application au frontend, donc nous allons installer nos dépendances JS (qui utiliseront @vite) :

```
npm install
```

:

```
npm run dev
```

Après cela, les liens de connexion et d'enregistrement devraient figurer sur votre page d'accueil, et tout devrait fonctionner sans problème.

Laravel Jetstream

Laravel Jetstream étend Laravel Breeze avec des fonctionnalités utiles et d'autres piles frontales.

Il permet la connexion, l'inscription, la vérification des e-mails, l'authentification à deux facteurs, la gestion des sessions, la prise en charge des API via Sanctum, et la gestion optionnelle des équipes.

Lors de l'installation de Jetstream, vous devez choisir entre Livewire et Inertia pour le front-end. Au niveau du backend, il utilise Laravel Fortify, qui est un backend d'authentification « headless » agnostique pour Laravel.

Installation et configuration

Nous allons l'installer via composer dans notre projet Laravel :

```
composer require laravel/jetstream
```

Ensuite, nous lancerons la commande `php artisan jetstream:install [stack]`, qui accepte les arguments `[stack]`, `Livewire` ou `Inertia`. Vous pouvez passer l'option `--teams` pour activer la fonctionnalité teams.

Cela installera également Pest PHP pour les tests.

Enfin, nous devons rendre le frontend de notre application en utilisant ce qui suit :

```
npm install
npm run dev
```

Laravel Fortify

Laravel Fortify est une implémentation d'authentification backend qui est agnostique au frontend. Vous n'avez pas besoin d'utiliser Laravel Fortify pour implémenter les fonctionnalités d'authentification de Laravel.

Il est également utilisé dans des kits de démarrage comme Breeze et Jetstream. Vous pouvez également utiliser Fortify de manière autonome, ce qui n'est qu'une implémentation backend. Si vous l'utilisez seul, votre frontend doit appeler les routes Fortify.

Installation et configuration

Nous pouvons installer Fortify via composer :

```
composer require laravel/fortify
```

Nous devons maintenant publier les ressources de Fortify :

```
php artisan vendor:publish --provider="Laravel\Fortify\FortifyServiceProv
```

Après cela, nous allons créer un nouveau répertoire **app/Actions** en plus du nouveau **FortifyServiceProvider**, du fichier de configuration et des migrations de base de données.

Enfin, exécutez :

```
php artisan migrate
```

Ou :

```
php artisan migrate:fresh
```

Et votre Fortify est prêt à être utilisé.

Laravel Socialite

Laravel inclut une fonction d'authentification des utilisateurs basée sur OAuth. Elle prend en charge les connexions sociales via Facebook, Twitter, LinkedIn, Google, Bitbucket, GitHub et GitLab.

Installation

Nous pouvons l'installer via composer :

```
composer require laravel/socialite
```

Installation et utilisation

Après l'avoir installé, nous devons ajouter les informations d'identification pour le fournisseur OAuth que notre application utilise. Nous les ajouterons dans **config/services.php** pour chaque service.

Dans la configuration, nous devons faire correspondre la clé avec les services précédents. Voici quelques-unes de ces clés :

- facebook
- twitter (pour OAuth 1.0)

- twitter-oauth-2 (pour OAuth 2.0)
- linkedin
- google
- github
- gitlab
- bitbucket

Une configuration de service peut ressembler à ceci :

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

Authentification des utilisateurs

Pour cette action, nous aurons besoin de deux routes, l'une pour rediriger l'utilisateur vers le fournisseur OAuth :

```
use Laravel\Socialite\Facades\Sociliate;

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

Et une autre pour le rappel du fournisseur après l'authentification :

```
use Laravel\Socialite\Facades\Socialite;

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

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

Socialite fournit la méthode de redirection, et la façade redirige l'utilisateur vers le fournisseur OAuth, tandis que la méthode utilisateur examine la demande entrante et récupère les informations de l'utilisateur.

Après avoir reçu notre utilisateur, nous devons vérifier s'il existe dans notre base de données et l'authentifier. S'il n'existe pas, nous créerons un nouvel enregistrement pour représenter l'utilisateur :

```
use App\Models\User;
use Illuminate\Support\Facades\Auth;
use Laravel\Socialite\Facades\Socialite;

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 nous voulons limiter les possibilités d'accès de l'utilisateur, nous pouvons utiliser la méthode `scopes`, que nous inclurons dans la demande d'authentification. Cette méthode fusionnera tous les champs d'application précédemment spécifiés avec les champs d'application spécifiés.

Une autre solution consiste à utiliser la méthode `setScopes` qui écrase tous les autres champs d'application existants :

```
use Laravel\Socialite\Facades\Socialite;

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

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

Maintenant que nous savons tout et que nous savons comment obtenir un utilisateur après le rappel, examinons certaines des données que nous pouvons obtenir.

L'utilisateur OAuth1 a `token` et `tokenSecret`:

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

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

OAuth2 fournit `token`, `refreshToken`, et `expiresIn`:

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

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

OAuth1 et OAuth2 fournissent tous deux `getId`, `getNickname`, `getName`, `getEmail`, et `getAvatar`:

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

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

Et si nous voulons obtenir les détails de l'utilisateur à partir d'un jeton (OAuth 2) ou d'un jeton et d'un secret (OAuth 1), Socialite fournit deux méthodes pour cela : `userFromToken` et `userFromTokenAndSecret`:

```
use Laravel\Socialite\Facades\Socialite;

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

Laravel Sanctum

Laravel Sanctum est un système d'authentification léger pour les SPA (Single Page Applications) et les applications mobiles. Il permet aux utilisateurs de générer plusieurs jetons API avec des champs d'application spécifiques. Ces champs d'application spécifient les actions autorisées par un jeton.

Utilisations

Sanctum peut être utilisé pour délivrer des jetons d'API à l'utilisateur sans les complexités d'OAuth. Ces jetons ont généralement des durées d'expiration longues, comme des années, mais peuvent être révoqués et régénérés par l'utilisateur à tout moment.

Installation et configuration

Nous pouvons l'installer via composer :

```
composer require laravel/sanctum
```

Et nous devons publier les fichiers de configuration et de migration :

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

Maintenant que nous avons généré de nouveaux fichiers de migration, nous devons les migrer :

```
php artisan migrate:fresh
```

Comment émettre des jetons d’API

Avant d’émettre des jetons, notre modèle User doit utiliser le trait **Laravel\Sanctum\HasApiTokens** :

```
use Laravel\Sanctum\HasApiTokens;

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

Lorsque nous avons l’utilisateur, nous pouvons émettre un jeton en appelant la méthode `createToken`, qui renvoie une instance **Laravel\Sanctum\NewAccessToken**.

Nous pouvons appeler la méthode `plainTextToken` sur l’instance **NewAccessToken** pour voir la valeur en texte clair **SHA-256** du jeton.

Conseils et bonnes pratiques pour l’authentification Laravel

Invalider les sessions sur d’autres appareils

Comme nous l’avons vu précédemment, l’invalidation de la session est cruciale lorsque l’utilisateur se déconnecte, mais cette option devrait également être disponible pour tous les appareils possédés.

Cette fonctionnalité est généralement utilisée lorsque l’utilisateur change ou met à jour son mot de passe, et que nous voulons invalider sa session à partir de n’importe quel autre appareil.

Avec la façade Auth, c'est une tâche facile à réaliser. Étant donné que la route que nous utilisons a les adresses `auth` et `auth.session middleware`, nous pouvons utiliser la méthode statique `logoutOtherDevices` de la façade :

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

```
use Illuminate\Support\Facades\Auth;

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

Configuration avec Auth::routes()

La méthode routes de la façade Auth est juste une aide pour générer toutes les routes nécessaires à l'authentification de l'utilisateur.

Les routes comprennent Login (Get, Post), Logout (Post), Register (Get, Post), et Password Reset/Email (Get, Post).

Lorsque vous appelez la méthode de la façade, elle effectue les opérations suivantes :

```
public static function 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);
    }
```

Nous nous intéressons à ce qui se passe lorsque la méthode statique est appelée sur le routeur. Cela peut s'avérer délicat en raison du fonctionnement des façades, mais la méthode suivante est appelée de la manière suivante :

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

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

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

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

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

Par défaut, il génère toutes les routes à part celle de la vérification de l'email. Nous aurons toujours les routes Login et Logout, mais les autres routes peuvent être contrôlées à travers le tableau d'options.

Si nous voulons seulement avoir login/logout et register, nous pouvons passer le tableau d'options suivant :

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

Protéger les routes et les guards personnalisés

Nous voulons nous assurer que certaines routes ne sont accessibles qu'aux utilisateurs authentifiés, ce qui peut être fait rapidement en appelant la méthode middleware sur la façade Route ou en chaînant la méthode middleware sur celle-ci :

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

Route::get('/user', function (Request $request) {
```

```
    return $request->user();
})->middleware('auth');
```

Cette protection garantit que les demandes entrantes sont authentifiées.

Confirmation du mot de passe

Pour renforcer la sécurité de votre site web, vous souhaitez souvent confirmer le mot de passe d'un utilisateur avant de passer à une autre tâche.

Nous devons définir une route à partir de la vue de confirmation du mot de passe pour traiter la demande. Elle validera et redirigera l'utilisateur vers la destination prévue. En même temps, nous nous assurerons que notre mot de passe apparaît confirmé dans la session. Par défaut, le mot de passe doit être reconfirmé toutes les trois heures, mais cela peut être modifié dans le fichier de configuration **config/auth.php**:

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Facades\Redirect;

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']);
```

Contrat Authenticable

Le contrat Authenticable situé à IlluminateContractsAuth définit un modèle de ce que la façade UserProvider doit implémenter :

```
namespace Illuminate\Contracts\Auth;

interface Authenticable
{
    public function getAuthIdentifierName();

    public function getAuthIdentifier();

    public function getAuthPassord();

    public function getRememberToken();

    public function setRememberToken($value);

    public function getrememberTokenName();
}
```

L’interface permet au système d’authentification de travailler avec n’importe quelle classe « user » qui l’implémente.

Cette interface permet au système d’authentification de fonctionner avec n’importe quelle classe « user » qui l’implémente, indépendamment de l’ORM ou des couches de stockage utilisées. Par défaut, Laravel possède la classe AppModelsUser qui implémente cette interface, ce qui est également visible dans le fichier de configuration :

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

Événements d’authentification

Il y a beaucoup d’événements qui sont envoyés pendant l’ensemble du processus d’authentification.

En fonction de vos objectifs, vous pouvez attacher des récepteurs à ces événements dans votre site `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',
    ],
];
```

Liste des listeners Laravel générés pour notre service d’authentification

Créer rapidement de nouveaux utilisateurs

La création rapide d’un nouvel utilisateur peut se faire par l’intermédiaire de la fonction **App\User**:

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

Ou par la méthode statique create sur la façade User :

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

Résumé

L’écosystème Laravel dispose d’un grand nombre de kits de démarrage pour faire fonctionner votre application avec un système d’authentification, comme Breeze et Jetstream. Ils sont hautement personnalisables car le code est généré de notre côté, et nous pouvons le modifier autant que nous le souhaitons, en l’utilisant comme un modèle si nécessaire.

Il existe de nombreux problèmes de sécurité concernant l'authentification et ses subtilités, mais tous peuvent être résolus facilement grâce aux outils fournis par Laravel. Ces outils sont hautement personnalisables et faciles à utiliser.