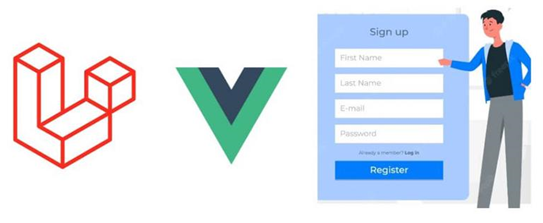
Atelier 5 : Authentification

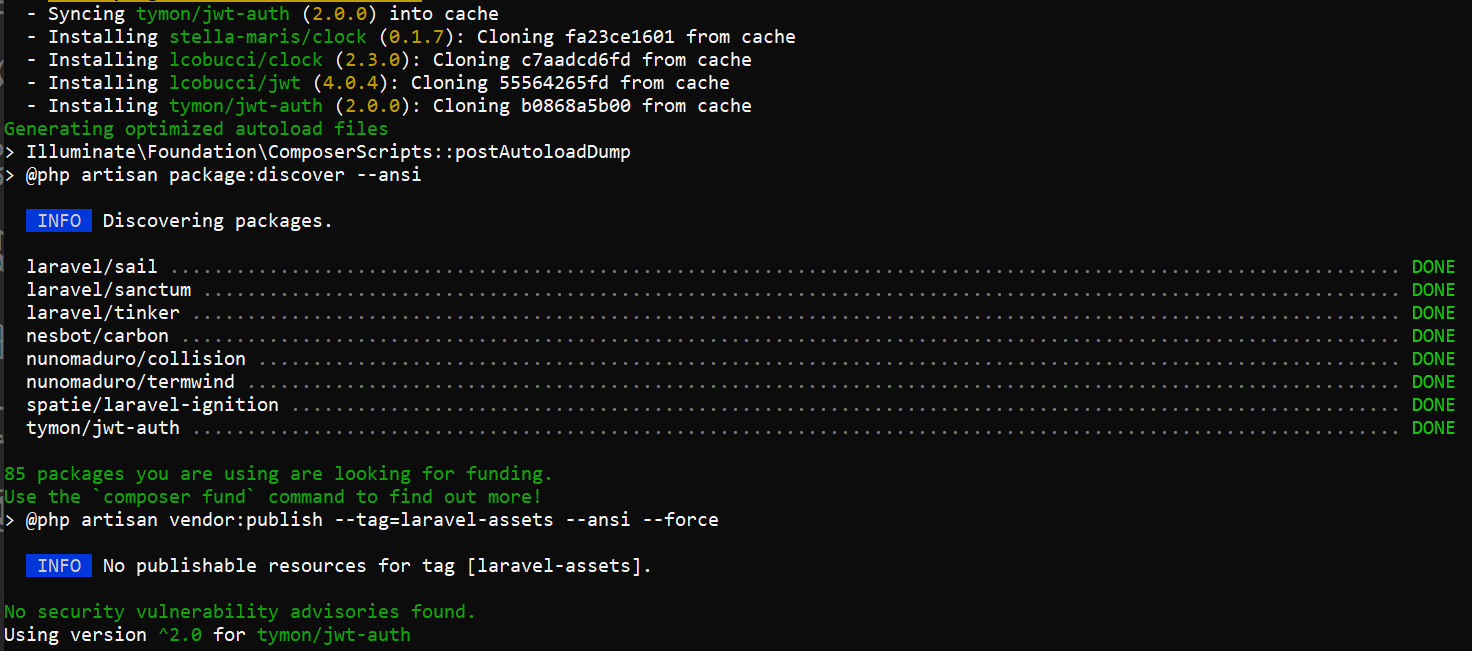


Dans cet atelier, nous allons apprendre à créer des API REST sécurisées dans Laravel à l'aide de JSON Web Token (JWT). Les API utilisent des jetons (tokens) pour valider les utilisateurs et ne conservent pas l'état de la session entre les requêtes.

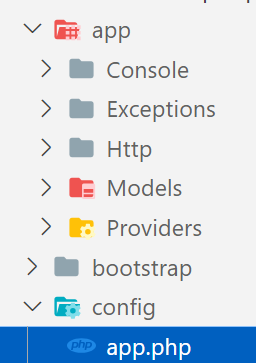
On suppose que le projet est déjà créé et les configurations sont faites.

# Installation du package

Exécutez la commande suivante pour installer **jwt-auth**, il s'agit d'un package JWT tiers et permet l'authentification des utilisateurs à l'aide du jeton Web JSON dans Laravel en toute sécurité.



Maintenant, nous devons aller dans le fichier config/app.php et inclure le fournisseur de services Laravel dans le tableau des fournisseurs. Incluez également les façades JWTAuth et JWTFactory dans le tableau d'alias.



**'providers'** => [

....

....

  Tymon\JWTAuth\Providers\LaravelServiceProvider::class,

],

**'aliases'** => [

....

  'JWTAuth' => Tymon\JWTAuth\Facades\JWTAuth::class,

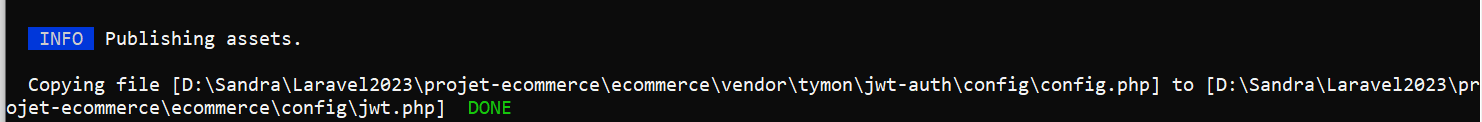
'JWTFactory' => Tymon\JWTAuth\Facades\JWTFactory::class,

....

],

A l'étape suivante, nous devons publier la configuration du package. La commande suivante, copie les fichiers JWT Auth du dossier du fournisseur vers le fichier config/jwt.php.

php artisan vendor:publish --provider="Tymon\JWTAuth\Providers\LaravelServiceProvider"



Pour gérer le chiffrement du token, générez une clé secrète en exécutant la commande suivante.

php artisan jwt:secret



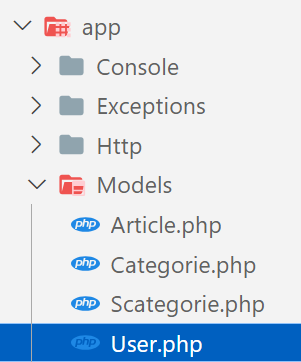
Nous avons généré avec succès la clé secrète JWT et vous pouvez vérifier cette clé dans le fichier **.env**



# Le model User

Laravel est livré avec un modèle utilisateur prédéfini ; nous pouvons utiliser le modèle **User** pour le processus d'authentification. Dans cette étape, nous allons apprendre à implémenter le package jwt-auth dans un modèle utilisateur.

Par défaut, Laravel inclut un modèle App\Models\User Eloquent dans votre répertoire app/Models.



Ce modèle peut être utilisé avec le pilote d'authentification Eloquent par défaut.

Ouvrez le fichier app/Models/User.php et remplacez le code suivant le code existant.

<?php

namespace App\Models;

// use Illuminate\Contracts\Auth\MustVerifyEmail;

use Illuminate\Database\Eloquent\Factories\HasFactory;

use Illuminate\Foundation\Auth\User as Authenticatable;

use Illuminate\Notifications\Notifiable;

use Laravel\Sanctum\HasApiTokens;

use Tymon\JWTAuth\Contracts\JWTSubject;

class User extends Authenticatable implements JWTSubject

{

    use HasApiTokens, HasFactory, Notifiable;

    /\*\*

     \* The attributes that are mass assignable.

     \*

     \* @var array<int, string>

     \*/

    protected $fillable = [

        'name',

        'email',

        'password',

    ];

    /\*\*

     \* The attributes that should be hidden for serialization.

     \*

     \* @var array<int, string>

     \*/

    protected $hidden = [

        'password',

        'remember\_token',

    ];

    /\*\*

     \* The attributes that should be cast.

     \*

     \* @var array<string, string>

     \*/

    protected $casts = [

        'email\_verified\_at' => 'datetime',

        'password' => 'hashed',

    ];

    /\*\*

     \* Get the identifier that will be stored in the subject claim of the JWT.

     \*

     \* @return mixed

     \*/

    public function getJWTIdentifier() {

        return $this->getKey();

    }

    /\*\*

     \* Return a key value array, containing any custom claims to be added to the JWT.

     \*

     \* @return array

     \*/

    public function getJWTCustomClaims() {

        return [];

    }

}

La charge utile du JSON Web Token (payload) est la partie qui contient les informations qui doivent être transmises à l’application. C’est là que sont définis certains standards qui déterminent quelles données doivent être transmises. Les informations sont fournies en paire clé/valeur, les clés sont appelées « claims » dans les JWT.

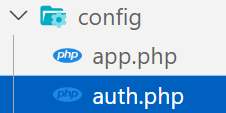
* **getJWTIdentifier**() : pour obtenir l'identifiant qui sera stocké dans le subject claim du JWT. « sub » (subject) claim identifie le mandant qui fait l'objet du JWT. Les claims dans un JWT sont normalement des déclarations sur le sujet. La valeur du sujet doit être étendue pour être localement unique dans le contexte de l'émetteur ou être globalement unique.
* **getJWTCustomClaims**() : renvoie un tableau de valeurs de clé, contenant tous les custom claims à ajouter au JWT. Les custom claims désignent à la fois les claims privés et les claims publics. Vous pouvez choisir le nom de votre choix, mais comme les jetons JWT doivent être aussi compacts que possible, la valeur maximale recommandée d'un nom de claims est de 8 caractères. De plus, chaque nom de claims que vous définissez doit être unique.

Cette étape vous aidera à faciliter la gestion et l'utilisation des relations et prend en charge un certain nombre de types de relations.

# Auth Guard

Maintenant, nous devons configurer JWT Auth Guard pour sécuriser le processus d'authentification de l'application Laravel. Laravel guard utilise le pilote de session pour protéger les gardes. Cependant, nous définissons la garde par défaut sur api, et les gardes api doivent utiliser le pilote jwt.

Ouvrez le fichier config/auth.php



Placez le code suivant dans le fichier config/auth.php.

'guards' => [

        'web' => [

            'driver' => 'session',

            'provider' => 'users',

        ],

        'api' => [

            'driver' => 'jwt',

            'provider' => 'users',

            'hash' => false,

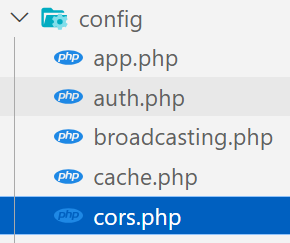
        ],

    ],

Les jetons d'API sont stockés dans votre base de données sous forme de texte brut. Si vous souhaitez hacher vos jetons d'API à l'aide du hachage SHA-256, vous pouvez définir l'option de hachage de votre configuration api guard sur true.

# Cors

Dans config/cors.php



Modifiez :

'exposed\_headers' => ['\*'],

« exposed\_headers » définit l'en-tête de réponse Access-Control-Expose-Headers.

« Access-Control-Expose-Headers » permet à un serveur d'indiquer quels en-têtes de réponse doivent être mis à la disposition des scripts.

La valeur « \* » compte uniquement comme une valeur générique spéciale pour les demandes sans informations d'identification (demandes sans cookies HTTP ou informations d'authentification HTTP). Dans les demandes avec informations d'identification, il est traité comme le nom d'en-tête littéral « \* » sans sémantique spéciale.

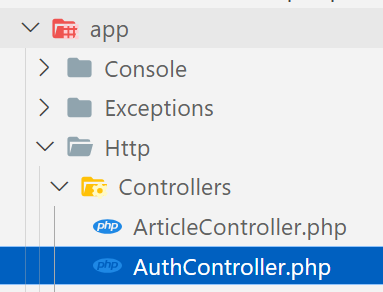
# AuthController

Dans cette étape, nous allons créer le contrôleur d'authentification JWT, et dans ce contrôleur d'authentification, nous allons définir la logique de base pour le processus d'authentification sécurisé dans Laravel.

Définissons le contrôleur d'authentification en utilisant la commande ci-dessous pour gérer les demandes d'authentification via les routes que nous avons créées à l'étape précédente.

php artisan make:controller AuthController

Ouvrez le fichier : app/Http/Controllers/AuthController.php



Placez le code suivant dans le fichier app/Http/Controllers/AuthController.php

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use Illuminate\Support\Facades\Auth;

use App\Models\User;

use Validator;

use JWTAuth;

use Tymon\JWTAuth\Exceptions\JWTException;

class AuthController extends Controller

{

    /\*\*

     \* Create a new AuthController instance.

     \*

     \* @return void

     \*/

    public function \_\_construct() {

        $this->middleware('auth:api', ['except' => ['login', 'register']]);

    }

    /\*\*

     \* Get a JWT via given credentials.

     \*

     \* @return \Illuminate\Http\JsonResponse

     \*/

    public function login(Request $request){

        $input = $request->only('email', 'password');

        $jwt\_token = null;

        if (!$jwt\_token = JWTAuth::attempt($input)) {

            return response()->json([

                'success' => false,

                'message' => 'Invalid Email or Password',

            ], Response::HTTP\_UNAUTHORIZED);

        }

        return response()->json([

            'success' => true,

            'token' => $jwt\_token,

        ]);

    }

    /\*\*

     \* Register a User.

     \*

     \* @return \Illuminate\Http\JsonResponse

     \*/

    public function register(Request $request) {

        $validator = Validator::make($request->all(), [

            'name' => 'required|string|between:2,100',

            'email' => 'required|string|email|max:100|unique:users',

            'password' => 'required|string|confirmed|min:6',

        ]);

        if($validator->fails()){

            return response()->json($validator->errors()->toJson(), 400);

        }

        $user = User::create(array\_merge(

                    $validator->validated(),

                    ['password' => bcrypt($request->password)]

                ));

        return response()->json([

            'message' => 'User successfully registered',

            'user' => $user

        ], 201);

    }

    /\*\*

     \* Log the user out (Invalidate the token).

     \*

     \* @return \Illuminate\Http\JsonResponse

     \*/

    public function logout() {

        $this->guard()->logout();

        return response()->json([

            'status' => 'success',

            'msg' => 'Logged out Successfully.'

        ], 200);

    }

    /\*\*

    \* Return auth guard

    \*/

    private function guard()

    {

        return Auth::guard();

    }

    /\*\*

     \* Refresh a token.

     \*

     \* @return \Illuminate\Http\JsonResponse

     \*/

    public function refresh() {

        return $this->createNewToken(auth()->refresh());

    }

    /\*\*

     \* Get the authenticated User.

     \*

     \* @return \Illuminate\Http\JsonResponse

     \*/

    public function userProfile() {

        return response()->json(auth()->user());

    }

    /\*\*

     \* Get the token array structure.

     \*

     \* @param  string $token

     \*

     \* @return \Illuminate\Http\JsonResponse

     \*/

    protected function createNewToken($token){

        return response()->json([

            'access\_token' => $token,

            'token\_type' => 'bearer',

            'expires\_in' => auth('api')->factory()->getTTL() \* 60,

            'user' => auth()->user()

        ]);

    }

}

- Le middleware **auth:api** est utilisé dans le constructeur. Les fonctions à l'intérieur du contrôleur d'authentification ne sont pas accessibles sans avoir le jeton valide. De plus, nous pouvons passer le nom des fonctions à l'intérieur du middleware que nous voulons exclure de l'obligation du jeton.

- La méthode de connexion (**login**) est utilisée pour fournir un accès à l'utilisateur et est déclenchée lorsque /api/auth/login API est appelé. Il authentifie l'e-mail et le mot de passe saisis par l'utilisateur dans un champ e-mail et mot de passe. En réponse, il génère un jeton d'autorisation s'il trouve un utilisateur dans la base de données. Vice versa, il affiche une erreur si l'utilisateur n'est pas trouvé dans la base de données.

La méthode « attempt » accepte un tableau de paires clé /valeur comme premier argument. Les valeurs du tableau seront utilisées pour trouver l'utilisateur dans votre table de base de données. Ainsi, dans l'exemple ci-dessus, l'utilisateur sera récupéré par la valeur de la colonne email. Si l'utilisateur est trouvé, le mot de passe haché stocké dans la base de données sera comparé à la valeur password transmise à la méthode via le tableau. Vous ne devez pas hacher le mot de passe spécifié comme valeur password, car le Framework hachera automatiquement la valeur avant de la comparer au mot de passe haché dans la base de données. Si les deux mots de passe hachés correspondent, une session authentifiée sera lancée pour l'utilisateur. La méthode « attempt » renverra « true » si l'authentification a réussi. Sinon, « false » sera renvoyé.

- La méthode **register** est utilisée pour créer un utilisateur lorsque /api/auth/register route est appelé. Tout d'abord, les valeurs utilisateur telles que le nom, l'adresse e-mail et le mot de passe sont validées via le processus de validation, puis l'utilisateur est enregistré si les informations d'identification de l'utilisateur sont valides. Ensuite, il génère le jeton Web JSON pour fournir un accès valide à l'utilisateur.

Laravel est livré avec une fonction simple et pratique pour valider les données et récupérer les messages d'erreur de validation via la classe Validator.

Le premier argument passé à la méthode « make » correspond aux données en cours de validation. Le deuxième argument concerne les règles de validation qui doivent être appliquées aux données.

- La méthode de déconnexion (**logout**) est appelée lorsque l'API /api/auth/logout est demandée et efface le jeton d'accès JWT transmis.

Cette méthode fait appel à la méthode guard qui retourne l'utilisateur actuellement authentifié (Auth::guard()).

- La méthode d'actualisation (**refresh**) crée un nouveau jeton Web JSON sur une période plus courte, et il est considéré comme une bonne pratique de générer un nouveau jeton pour le système d'authentification utilisateur sécurisé dans Laravel. Il bloque l'utilisateur actuellement connecté si le jeton JWT n'est pas nouveau.

La fonction « createNewToken », appelée par la méthode « refresh », crée le nouveau jeton d'authentification JWT après une période de temps spécifiée, nous avons défini l'expiration du jeton et enregistré les données utilisateur dans cette fonction.

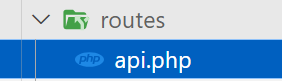
- La méthode **userProfile** restitue les données de l'utilisateur connecté. Cela fonctionne lorsque nous plaçons le jeton d'authentification dans les en-têtes pour authentifier la demande d'authentification effectuée via l'API /api/auth/user-profile.

# API Rest

Nous devons définir les routes d'authentification de l'API REST pour le processus d'authentification dans l'application d'authentification Laravel JWT. Les routes qui sont servies via routes/api.php sont fixées à api/ et les routes d'authentification sont désignées par auth/.

Cela devient donc /api/auth/login, et il en va de même pour chaque route que nous avons créée pour l'authentification.

Nous devons ajouter des routes d'authentification dans routes/api.php :



use App\Http\Controllers\AuthController;

Route::group([

    'middleware' => 'api',

    'prefix' => 'auth'

], function ($router) {

    Route::post('/login', [AuthController::class, 'login']);

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

    Route::post('/logout', [AuthController::class, 'logout']);

    Route::post('/refresh', [AuthController::class, 'refresh']);

    Route::get('/user-profile', [AuthController::class, 'userProfile']);

});

Le préfixe de chemin est utile lorsque nous voulons fournir une structure d'URL commune. Nous créons des fonctionnalités pour le module d'authentification, ce qui signifie que nous pouvons créer un préfixe auth commun pour toutes nos routes sous la fonctionnalité d'authentification.

Pour ajouter un préfixe de chemin, on ajoute un élément supplémentaire dans le paramètre de tableau associatif du groupe de routes appelé « prefix » en tant que clé et « auth » en tant que valeur.

Le middleware agit comme un pont entre une demande et une réponse. C'est un type de mécanisme de filtrage.

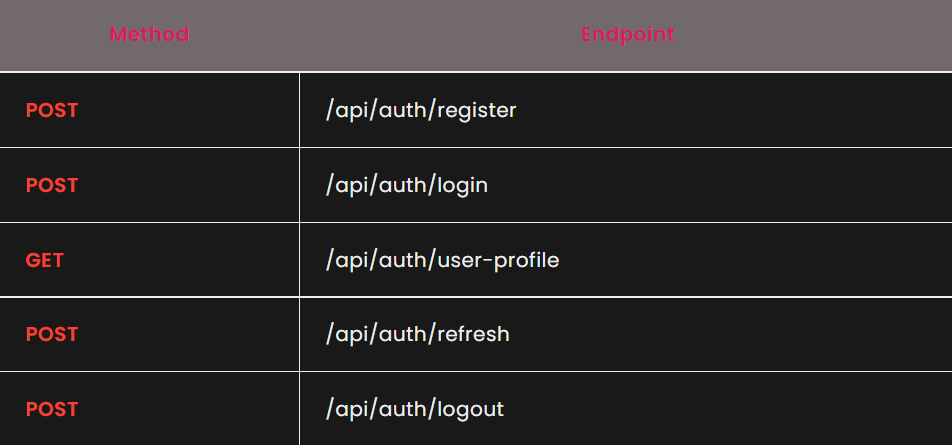
Si l'utilisateur n'est pas authentifié, le middleware redirigera l'utilisateur vers l'écran de connexion de votre application. Cependant, si l'utilisateur est authentifié, le middleware permettra à la demande d'aller plus loin dans l'application.

# Test des APIs

Démarrez l'application laravel avec la commande suivante :

php artisan serve

<http://localhost:8000>



POST <http://localhost:8000/api/auth/register>

{

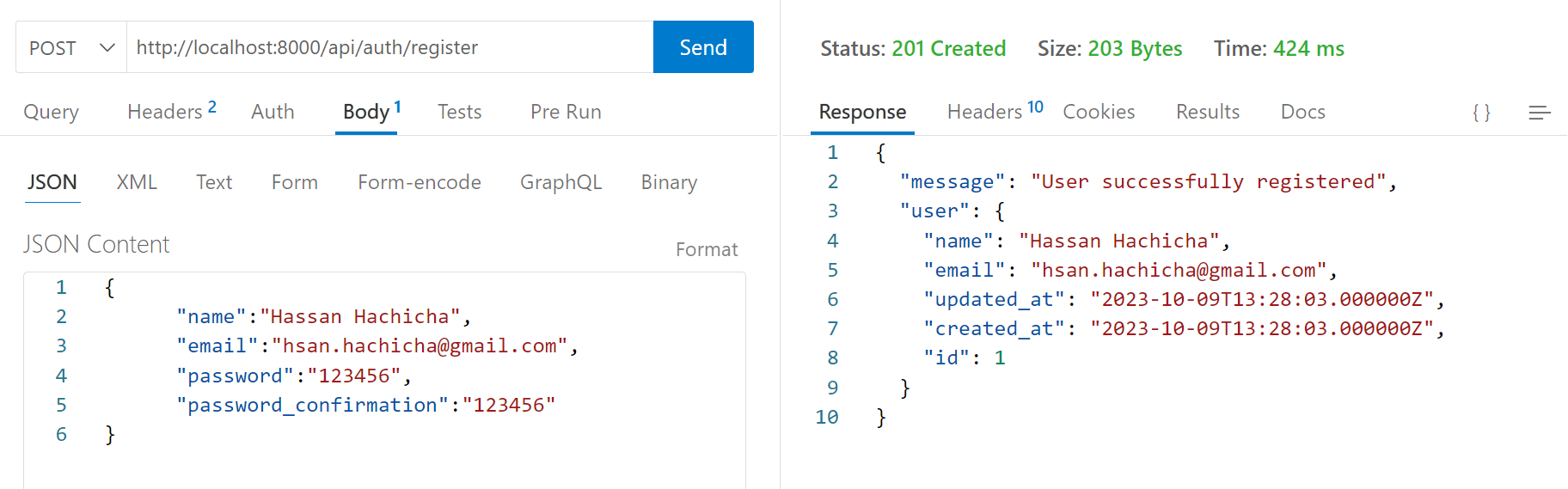
"name":"Hassan Hachicha",

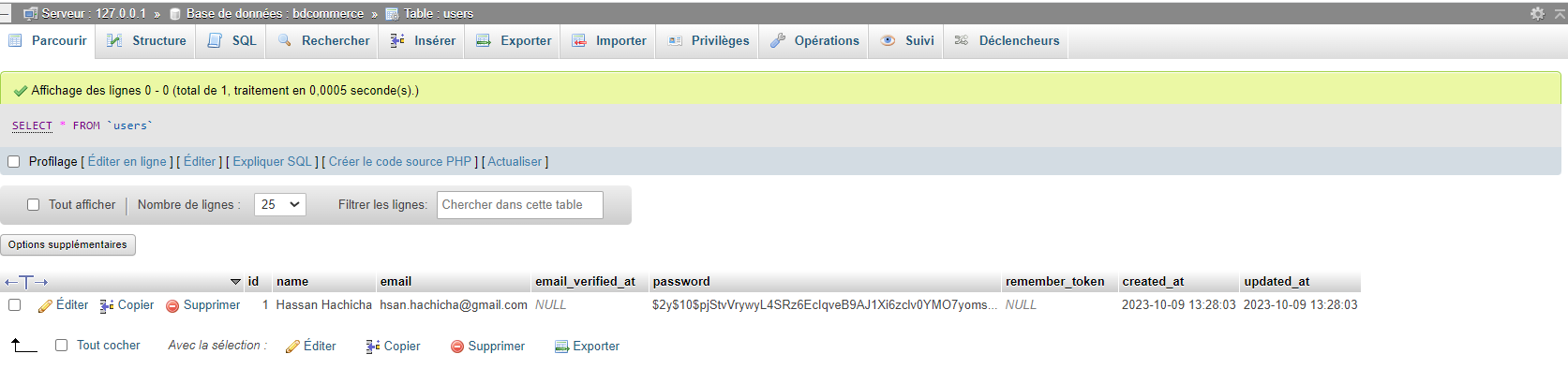
"email":"hsan.hachicha@gmail.com",

"password":"123456",

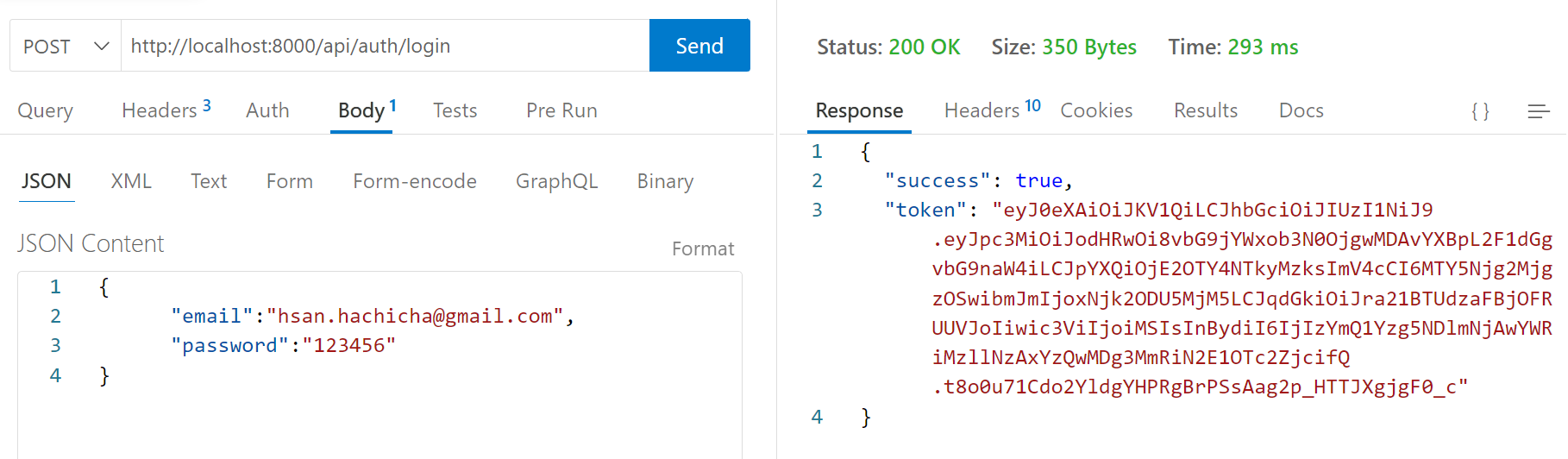
"password\_confirmation":"123456"

}



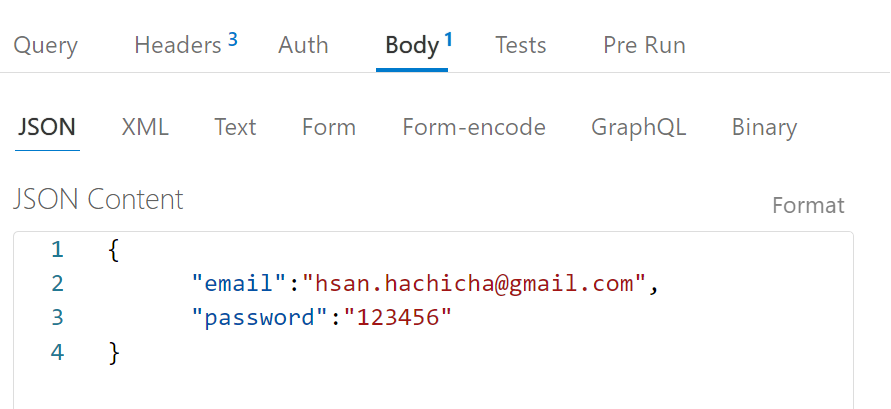


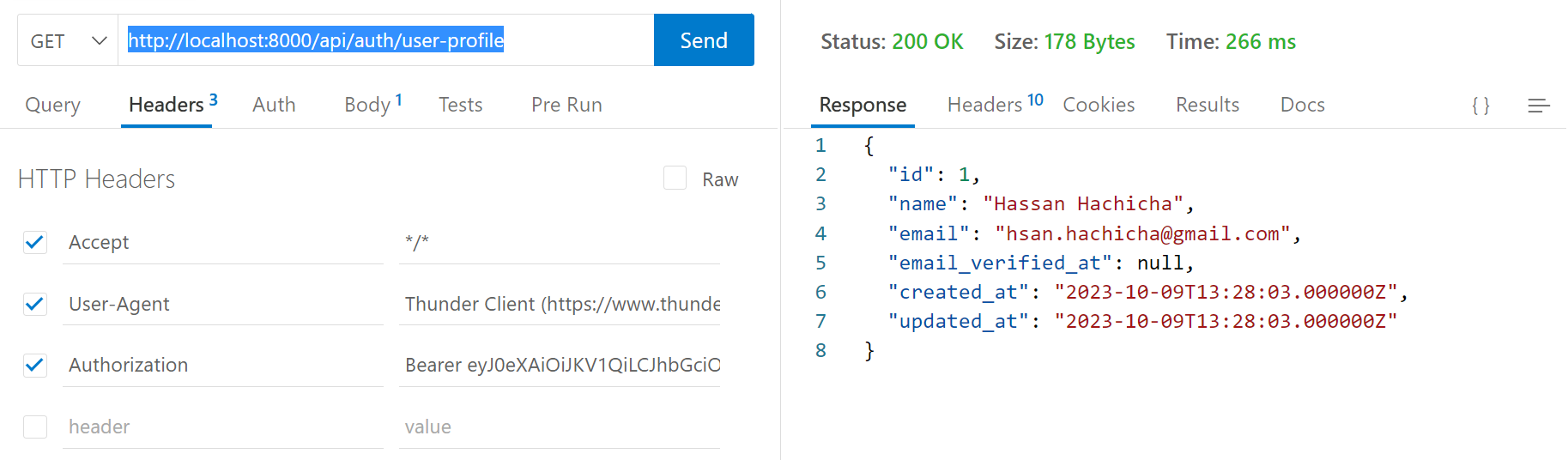
POST <http://localhost:8000/api/auth/login>



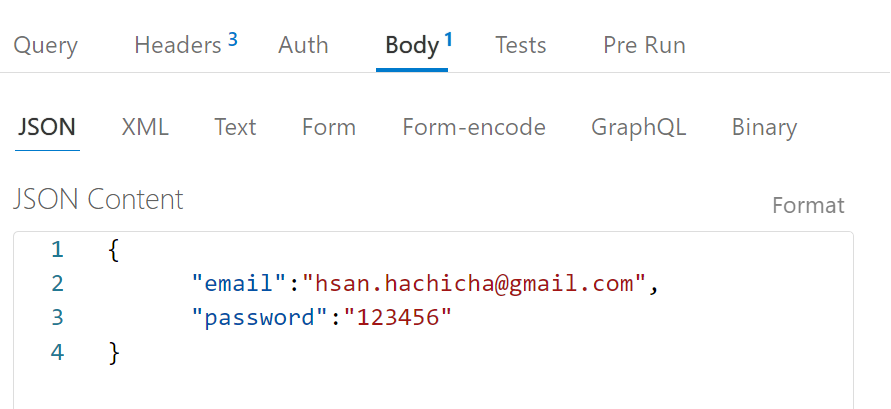
Assurez-vous que vous devez définir le jeton d'accès en tant que champ d'en-tête « Authorization : Bearer valeur-du-Token-copié » pour les API REST de profil utilisateur, d'actualisation de jeton et de déconnexion.

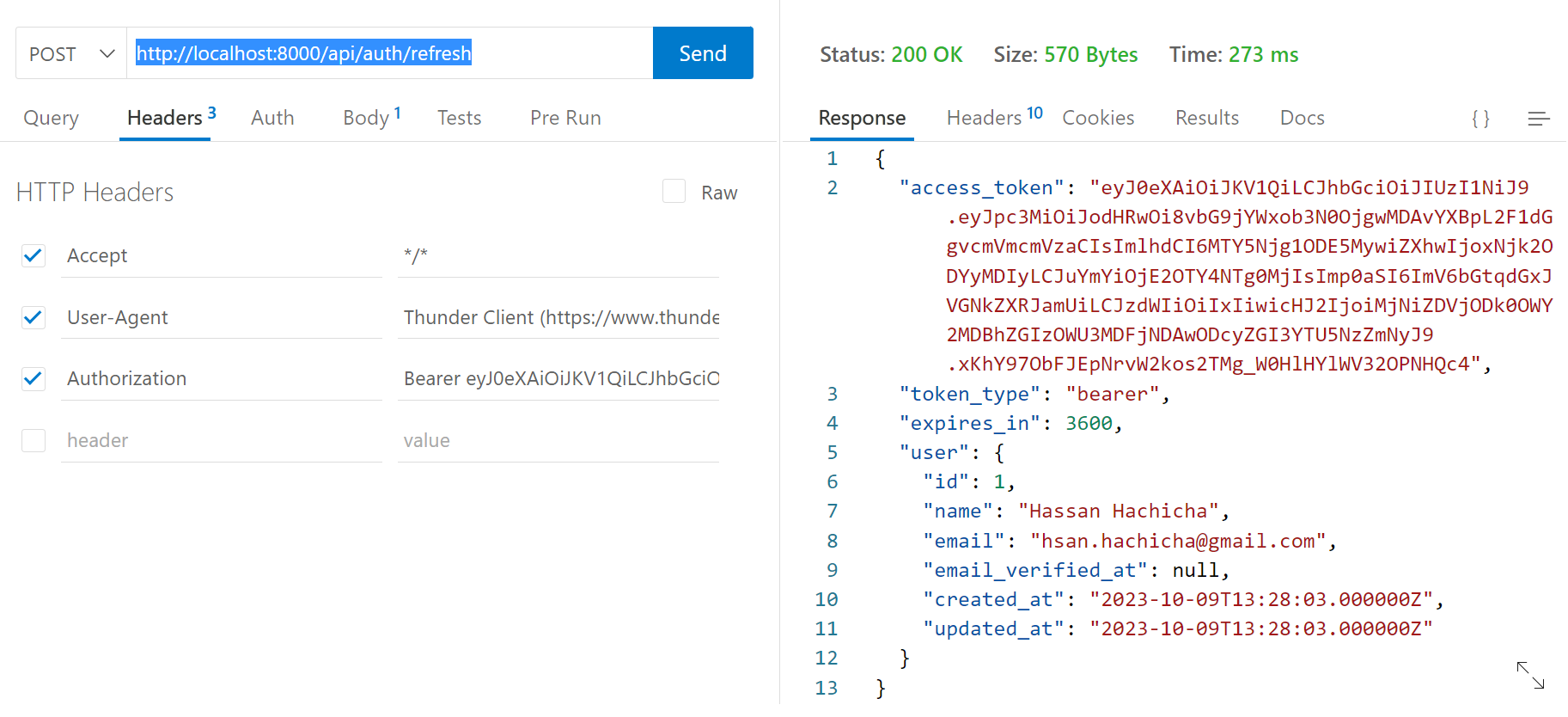
GET <http://localhost:8000/api/auth/user-profile>





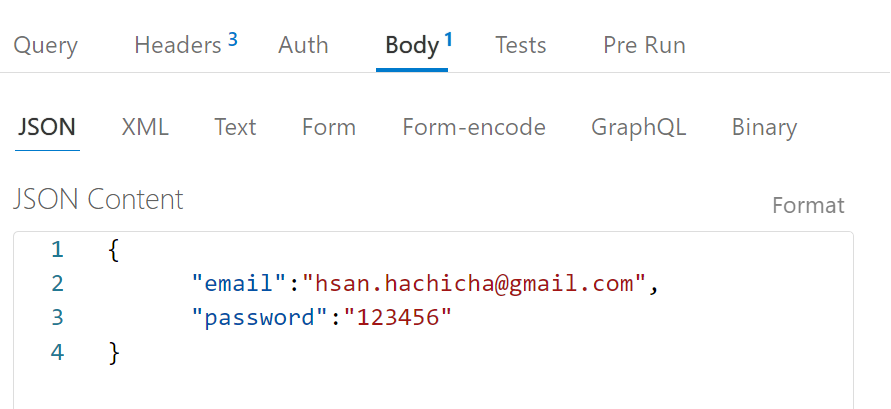
POST <http://localhost:8000/api/auth/refresh>

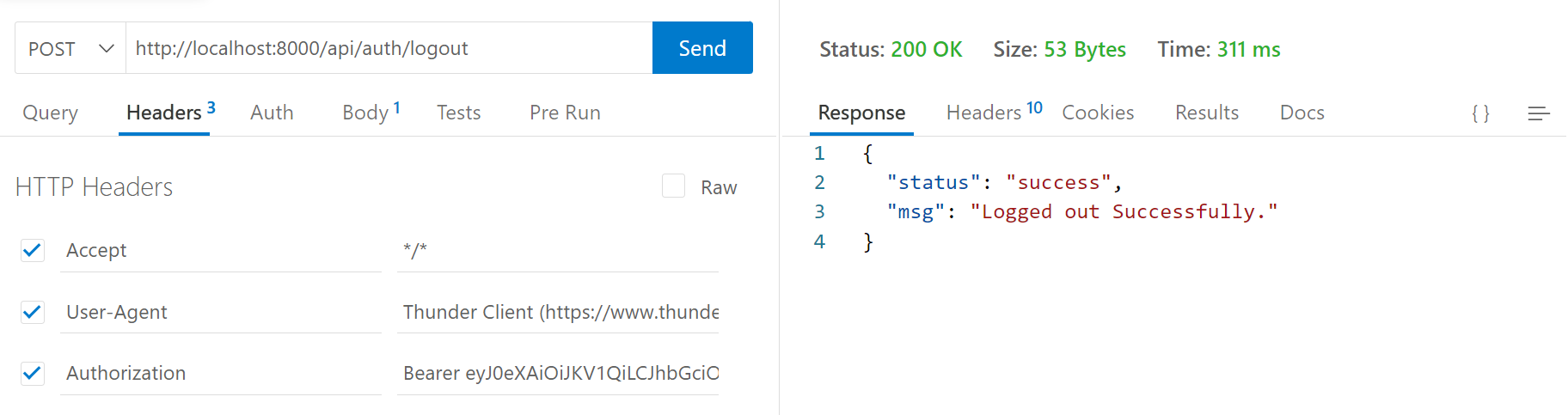




Copier le token généré dans access\_token. On devrait préciser le jeton dans l’entête Authorization (si on a fait le refresh il faut mettre le jeton nouvellement généré).

POST <http://localhost:8000/api/auth/logout>





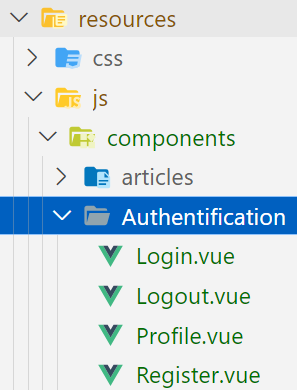
# UI VueJS

Dans ces étapes, nous allons créer du côté de Vue 3 les interfaces de Register, Login et Profile ainsi qu’un composant Logout.

Créez le dossier **Authentification** sous resources/js/components.

Puis créez y les fichiers :

* Register.vue
* Login.vue
* Profile.vue
* Logout.vue



# Routes.js



Ajoutez dans routes.js

import Register from './components/Authentification/Register.vue';

import Login from './components/Authentification/Login.vue';

import Profile from './components/Authentification/Profile.vue';

 {

        name: 'register',

        path: '/register',

        component: Register

    },

    {

        name: 'login',

        path: '/login',

        component: Login

    },

    {

        name: 'profile',

        path: '/profile',

        component: Profile

    },

# App.vue



Ajoutez dans App.vue

<router-link to="/register" class="nav-item nav-link">  Authentification </router-link>

# Register.vue



<template>

    <div class="col-md-12">

      <div class="card card-container">

          <h3> Si vous disposez déjà d'un compte passez à la connexion

               <router-link to="/login" class="btn btn-warning">  Login </router-link>

          </h3>

     <h2>Enregistrement</h2>

        <form @submit.prevent="handleRegister">

            <div class="form-group">

              <input placeholder="name" type="text" class="form-control" v-model="user.name"/>

            </div>

            <div class="form-group">

              <input placeholder="email" type="email" class="form-control" v-model="user.email"/>

            </div>

            <div class="form-group">

              <input placeholder="password" type="password" class="form-control" v-model="user.password"/>

            </div>

              <div class="form-group">

              <input placeholder="password\_confirmation" type="password" class="form-control" v-model="user.password\_confirmation"/>

            </div>

            <div class="form-group">

              <button class="btn btn-primary btn-block">

                  S'enregistrer

              </button>

          </div>

        </form>

      </div>

    </div>

  </template>

  <script setup>

  import axios from "../config/axios.js"

  import { useRouter } from 'vue-router';

  const router = useRouter()

      let  user= {}

    const  handleRegister=async()=> {

        await axios.post('/api/auth/register', user)

                  .then(response => (

                          localStorage.setItem('user', JSON.stringify(response.data)),

                          router.push({ name: 'login' })

                      ))

                      .catch(err => {console.log(err);alert("Erreur !") })

                   }

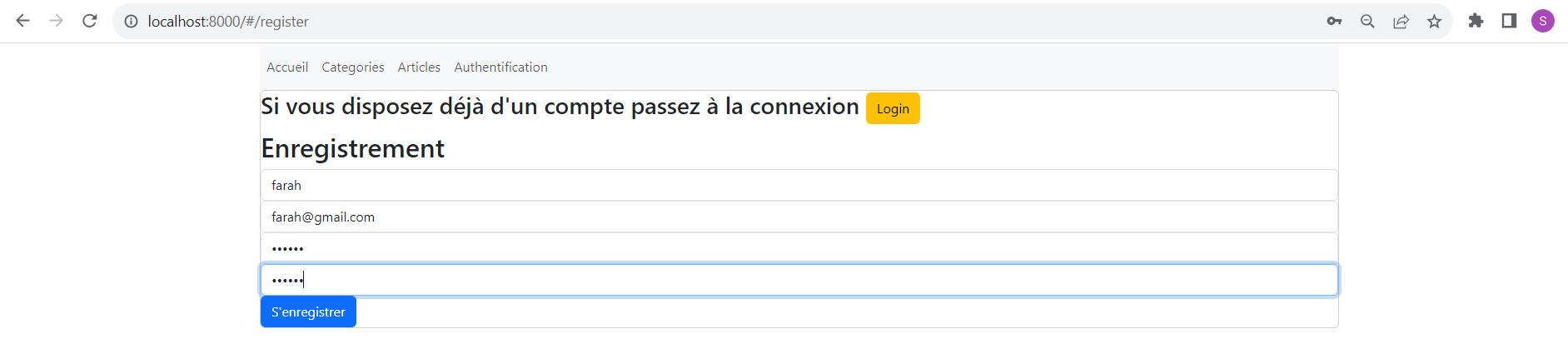
  </script>

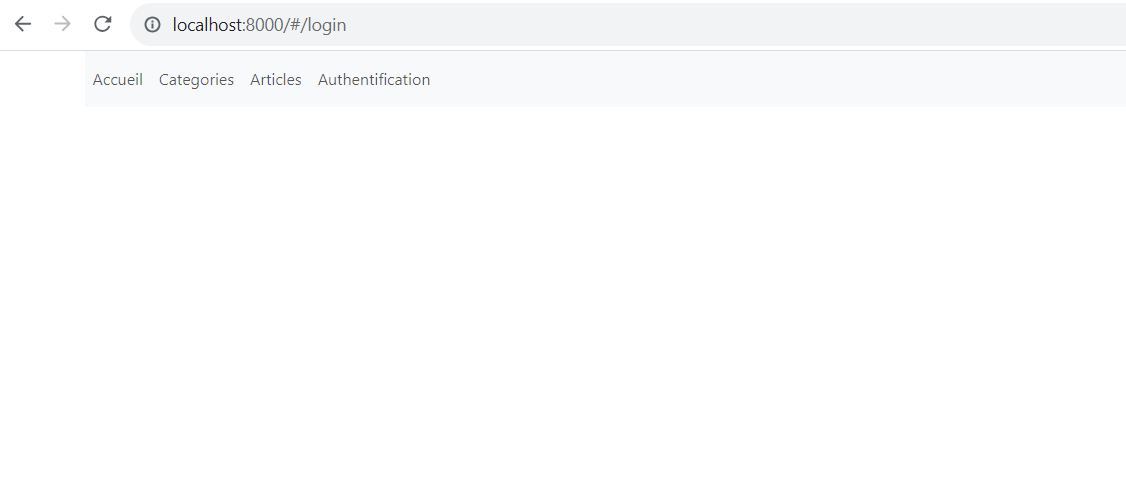
Exécuter dans 2 terminaux :

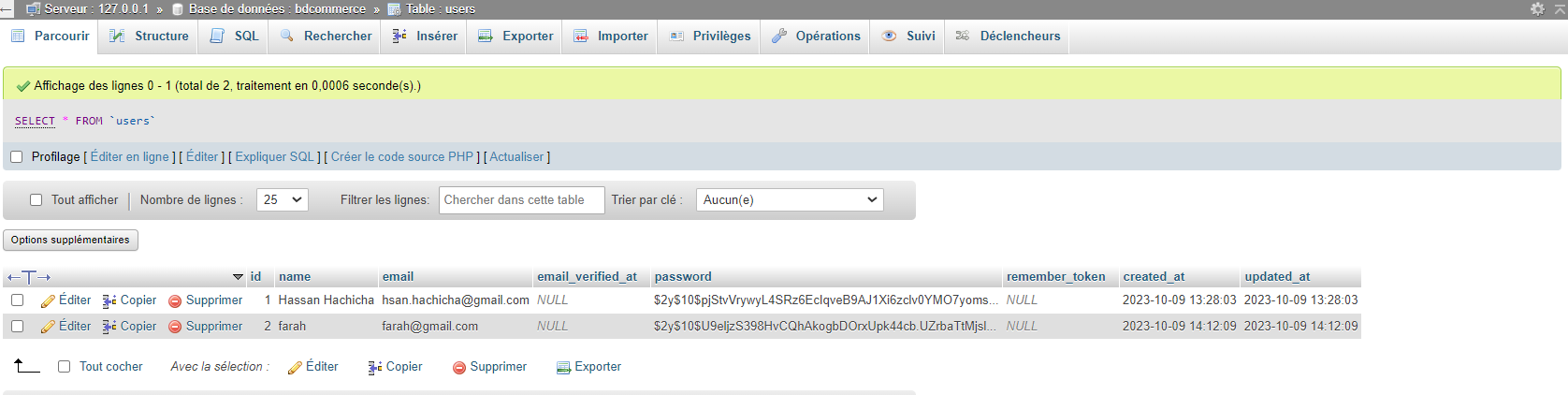
php artisan serve

npm run dev









# Login.vue



<template>

    <div class="col-md-8">

      <div class="card card-container">

       <h2>Connexion</h2>

        <form @submit.prevent="handleLogin">

            <div class="form-group">

              <input placeholder="email" type="email" class="form-control" v-model="user.email"/>

            </div>

            <div class="form-group">

              <input placeholder="password" type="password" class="form-control" v-model="user.password"/>

            </div>

            <div class="form-group">

              <button class="btn btn-primary btn-block">

                     Valider

              </button>

          </div>

        </form>

      </div>

    </div>

  </template>

  <script setup>

  import axios from "../config/axios.js"

  import { useRouter } from 'vue-router';

  const router = useRouter()

      let  user= {}

    const   handleLogin=async()=> {

                 await axios

                      .post('/api/auth/login', user)

                      .then(response => (

                          localStorage.setItem('user', JSON.stringify(response.data)),

                         router.push({ name: 'profile' })

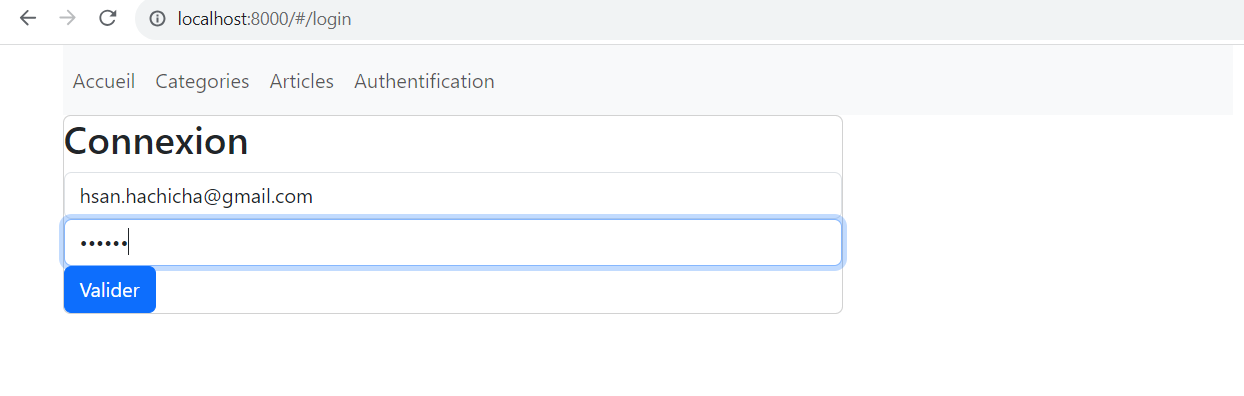
                      ))

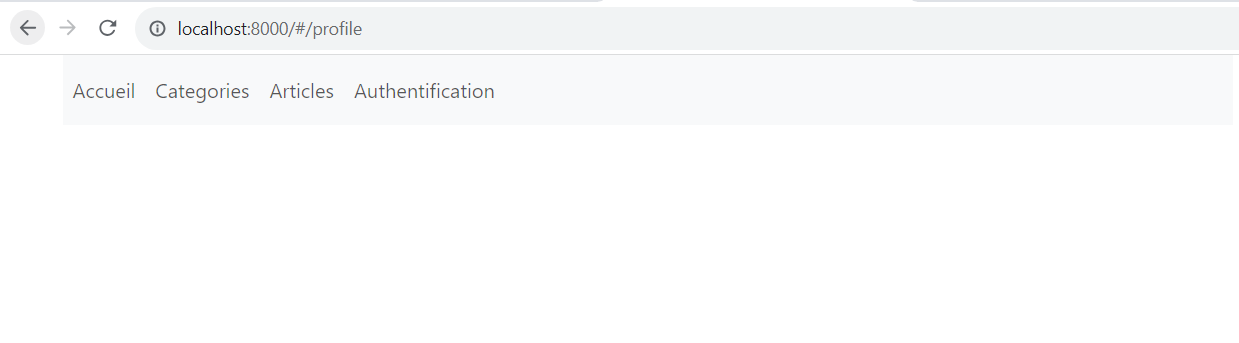
                      .catch(err => {console.log(err);alert("Erreur !") })

                   }

  </script>

### 





# Profile.vue



<template>

    <div class="container">

      <header class="jumbotron">

        <h3>

          <strong>{{ name }}</strong> Profile

        </h3>

      </header>

      <p>

        <strong>Created At:</strong> {{ created\_at }}

      </p>

      <p>

        <strong>Email:</strong> {{ email }}

      </p>

    </div>

  </template>

  <script setup>

import axios from "../config/axios.js";

import { ref, onMounted } from 'vue';

const name = ref({});

const email = ref({});

const created\_at = ref({});

const currentUser=async()=> {

        let user = JSON.parse(localStorage.getItem('user'));

        console.log(user.token)

        await axios.get('/api/auth/user-profile', { headers: { Authorization: 'Bearer ' + user.token }})

                      .then(response =>

                         {

                           // user=response.data;

                           name.value=response.data.name

                           email.value=response.data.email

                           created\_at.value=response.data.created\_at

                         }

                      )

                      .catch(err => {console.log(err);alert("Erreur !")})

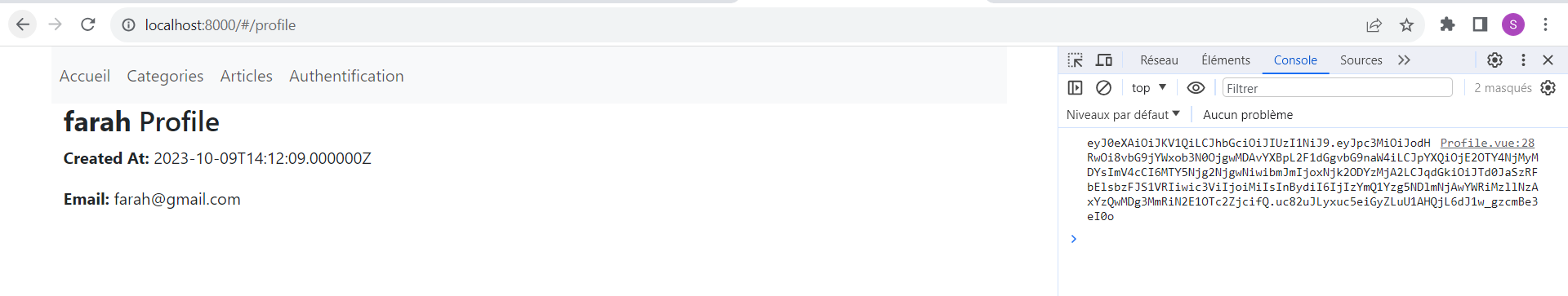
                  }

onMounted(() => {

    currentUser()

});

  </script>



# Logout.vue



<template></template>

<script setup>

  import axios from "../config/axios.js"

  import { onMounted } from 'vue';

  import { useRouter } from 'vue-router';

  const router = useRouter()

   const  handleLogout=async()=> {

               let user = JSON.parse(localStorage.getItem('user'));

               axios.interceptors.request.use(

               (config) => {

               if (user.token) {

                    config.headers['Authorization'] = `Bearer ${user.token}`;

                 }

                return config;

                },

                (error) => {

                return Promise.reject(error);

                }

                );

               await axios

                    .post('/api/auth/logout')

                    .then(response => (

                       localStorage.removeItem('user'),

                       router.push('/login')

                    ))

                    .catch(err => console.log(err))

                 }

onMounted(async () => {

   await handleLogout()

});

</script>

Les intercepteurs Axios (avec this.axios.interceptors.request.use) sont des fonctions qu'Axios appelle pour chaque requête. Vous pouvez utiliser des intercepteurs pour transformer la requête avant qu'Axios ne l'envoie, ou transformer la réponse avant qu'Axios ne renvoie la réponse à votre code.

Dans votre demande de déconnexion, vous allez détruire le jeton qui revient du backend dans le localStorage ou le cookie, cet intercepteur utiliserait cette demande pour définir l'en-tête sur Authorization.

# routes.js



import Logout from './components/Authentification/Logout.vue';

  {

        name: 'logout',

        path: '/logout',

        component: Logout

    },

# Menu.vue



  <router-link to="/logout" class="nav-item nav-link">  Logout </router-link>

Le clic sur Logout nous mène vers la page de connexion.

