Sécuriser une API avec JWT

JSON Web Token

Table des matières

JWT, c'est quoi ?

JWT, ou JSON Web Token, est un moyen pour deux applications de s'échanger des informations sous la forme d'un objet JSON invisible, ce qui présente donc un avantage considérable pour la sécurité puisque ce token sera effectivement caché pour les utilisateurs. C'est en réalité le strict minimum lorsque l'on cherche à échanger entre une API et une appli qui demande de s'authentifier.

Comment ça marche ?

1. Le JWT doit d'abord être **généré** du côté de l'API
2. On va ensuite devoir faire une requête préliminaire pour **récupérer** le JWT
3. On va **stocker** ce token côté front (dans des cookies de session par exemple)
4. On va **lire** ce token et l'utiliser à chaque appel de l'API cible
5. L'API cible va **accepter** ou **rejeter** l'authentification.

1. Générer le JWT

Pour générer ce token, nous allons faire appel à une librairie prévue à cet effet. En effet, nous ne génèrons pas le token à la main : ce n'est pas sécurisé.

Nous avons besoin d'installer cette librairie PHP-JWT dans un premier temps grâce à composer. Dans notre dossier exercice_api, nous pouvons donc faire ceci :

```
composer require firebase/php-jwt
```

Puis nous allons nous créer un fichier jwt-login.php dans lequel nous allons copier ceci :

```php
<?php
// Headers CORS
header('Content-Type: text/html; charset=utf-8');
header('Access-Control-Allow-Origin: *');
header('Access-Control-Allow-Methods: GET, OPTIONS');

// Traiter la preflight request en PREMIER
if ($_SERVER['REQUEST_METHOD'] === 'OPTIONS') {
    http_response_code(200);
    exit();
}

require 'vendor/autoload.php';
use Firebase\JWT\JWT;
use Firebase\JWT\Key;

// Le token doit être long d'au moins 30 caractères

$secret = "tw8e3GT9ZcttMz93tOlE7SUqE2gIlH6bDLGQnAuJEp4ncfnUSL";
```

```php
$payload = [
  'iss' => 'mon-angular-app', // Qui appelle
  'aud' => 'exercice-api', // Qui reçoit
  'iat' => time(),
  'exp' => time() + (3600 * 24 * 180) // Expiration au bout de 6 mois
];
$jwt = JWT::encode($payload, $secret, 'HS256');

echo json_encode(["token" => $jwt]);
```

? Mais le token est toujours visible dans le code, ce n'est pas un problème ?

La différence est subtile, mais elle est capitale : Certes le token « classique », le « mot de passe », est toujours en clair dans le code… Mais cette fois il est côté back, il n'y a donc pas de risques que ce code soit visible pour le public. On pourrait aller encore plus loin en déclarant ce code dans un fichier de config caché inaccessible au public, puis en appelant la constante qui contient ce code dans notre fichier, mais pour l'instant, faisons simple.

2 Et 3. Récupérer le JWT et Stocker le JWT

Nous allons maintenant faire en sorte d'interroger ce fichier jwt-login.php afin d'en récupérer le JWT. Pour cela, nous allons créer un service dedié dans notre projet Angular, mais retenez qu'on pourrait faire la même chose dans un projet full PHP : de manière générale on va **isoler** la partie authentification d'un site. On ne mélange pas les torchons et les serviettes ☺

On va donc créer un service qui gère la requête d'**authentification** vers jwt-login.php, mais également un **intercepteur** qui va nous permettre d'alléger les appels à l'API : Cet intercepteur va se placer entre Angular et l'API pour ajouter le token à chaque requête sortante qu'il va détecter. On appelle cela un **middleware**.

services/auth-service.ts

```typescript
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { tap } from 'rxjs/operators';

@Injectable({
  providedIn: 'root'
})
export class AuthService {

  constructor(private http: HttpClient) {}

  // On appelle l'URL de login
  fetchJwtToken() {
```

```
    // On type la réponse attendue de la requête GET
    // Ici on utilise pipe plutôt que subscribe car on ne lance pas une opération à la réception du token, on décrit simplement ce qui va se passer
    // Et tap() se déclenche normalement sans permettre la modification de response, ce qui le différencie de next()
    return this.http.get<{ token: string }>('http://exercice-api.loc/jwt-login.php').pipe(
      tap(response => {
        localStorage.setItem('api_jwt', response.token);
      })
    );
  }

  getToken(): string | null {
    return localStorage.getItem('api_jwt');
  }

  clearToken(): void {
    localStorage.removeItem('api_jwt');
  }
}
```

On va ensuite appeler cette méthode fetchJwtToken() dans app.ts

```
import { Component, signal } from '@angular/core';
import { Navbar } from "./components/navbar/navbar";
import { RouterOutlet } from '@angular/router';
import { AuthService } from './services/auth-service';

@Component({
  selector: 'app-root',
  imports: [Navbar, RouterOutlet],
  templateUrl: './app.html',
  styleUrl: './app.scss'
})
export class App {
  protected readonly title = signal('tp-form-contact');
  constructor(private authService:AuthService) {

    this.authService.fetchJwtToken().subscribe({
      next: () => {},
      error: (err) => {
        console.error('Impossible de récupérer le JWT');
        console.log(err);
      }
    });
  }
}
```

Et enfin, on va utiliser la commande **ng generate interceptor interceptors/jwt** pour obtenir le fichier jwt-interceptor.ts :

```
import { Injectable } from '@angular/core';
// Import de toutes les classes dont l'intercepteur aura besoin
import {HttpEvent, HttpHandler, HttpInterceptor, HttpRequest} from '@angular/common/http';
import { Observable } from 'rxjs';
import { AuthService } from '../services/auth-service';

@Injectable()
export class JwtInterceptor implements HttpInterceptor {

  constructor(private authService: AuthService) {}
  // Deux propriétés importantes : la requête HTTP en soi et un gestionnaire de requêtes HTTP, le Handler.
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

    // On récupère le token depuis le service d'authentification
    const token = this.authService.getToken();

    if (!token) {
      return next.handle(req);
    }

    // Si on a bien passé le premier test et qu'on a bien un token, alors on rajoute le token par dessus les headers de la requête interceptée
    const authReq = req.clone({
      setHeaders: {
        Authorization: `Bearer ${token}`
      }
    });

    return next.handle(authReq);
  }
}
```

Et ce n'est pas tout, on va également devoir modifier le fichier app.config.ts afin de lui demander de nous fournir les éléments nécessaires pour gérer un intercepteur, un peu comme on aurait dû le faire avec HttpClient.

app.config.ts

```
import { ApplicationConfig, provideBrowserGlobalErrorListeners } from '@angular/core';
import { provideRouter } from '@angular/router';

import { routes } from './app.routes';
```

```
import { provideHttpClient } from '@angular/common/http';
import { withInterceptorsFromDi } from '@angular/common/http';
import { HTTP_INTERCEPTORS } from '@angular/common/http';
import { JwtInterceptor } from './interceptors/jwt-interceptor';
export const appConfig: ApplicationConfig = {
  providers: [
    provideBrowserGlobalErrorListeners(),
    provideRouter(routes),
    provideHttpClient(
      withInterceptorsFromDi()
    ),
    {
      provide: HTTP_INTERCEPTORS,
      useClass: JwtInterceptor,
      multi: true
    }
  ]
};
```

4. Lire le token et l'utiliser à chaque appel

C'est là que notre intercepteur va entrer en jeu : Chacun des appels à l'api que nous avons fait pourra être modifié et ne plus inclure le token, car c'est l'intercepteur qui va retoucher nos requêtes sortantes pour y inclure le token en question.

Dans api-service.ts,

```
sendEmail(formValues: FormData) {
  return this.monClientHttp.post('http://exercice-api.loc/ticketing.php', formValues, {
    headers: {'Authorization' : 'tw8e3GT9ZcttMz93tOlE7SUqE2gIlH6bDLGQnAuJEp4ncfnUSL'},
    responseType: 'text'
  });
}
```

Devient donc :

```
sendEmail(formValues: FormData) {
  return this.monClientHttp.post('http://exercice-api.loc/ticketing.php', formValues, {
    responseType: 'text'
  });
}
```

Objectif atteint : le Token ne figure plus dans le code lisible par l'utilisateur ☺

5. Authentification

C’est la dernière étape, il faut maintenant que dans notre fichier ticketing.php on valide ou que l’on rejete l’authentification. Pour cela on va de nouveau se servir de la classe JWT installée plus tôt avec composer, et on va remplacer nos vérifications de statuts http par ceci :

```php
<?php
// Headers CORS
header('Content-Type: text/html; charset=utf-8');
header('Access-Control-Allow-Origin: *');
header('Access-Control-Allow-Methods: GET, POST, OPTIONS');
header('Access-Control-Allow-Headers: Content-Type, Authorization');

// Traiter la preflight request en PREMIER
if ($_SERVER['REQUEST_METHOD'] === 'OPTIONS') {
    http_response_code(200);
    exit();
}

// inclusion des packages installés avec Composeer
require 'vendor/autoload.php';
use PHPMailer\PHPMailer\PHPMailer;
use Firebase\JWT\JWT;
use Firebase\JWT\Key;

// Le token en lui-même, en dur certes mais côté back
$secret = 'tw8e3GT9ZcttMz93tOlE7SUqE2gIlH6bDLGQnAuJEp4ncfnUSL';

// Si on n'a pas de header "Authorization" alors on coupe court et on renvoie une 401
$headers = getallheaders();
if (!isset($headers['Authorization'])) {
  http_response_code(401);
  exit('Token manquant');
}

// On supprime le mot-clé "Bearer" qui vient avec le Token
$token = str_replace('Bearer ', '', $headers['Authorization']);

// Et on essaie de décoder le JWT pour retomber sur notre clé secrète.
try {
    $decoded = JWT::decode($token, new Key($secret, 'HS256'));
    // Si la cible du JWT décodé n'est pas la bonne, c'est que la clé n'était pas bonne, on renvoie une
erreur
    if ($decoded->aud !== 'exercice-api') {
        throw new Exception('Audience invalide');
    }
```

```php
} catch (Exception $e) {
  http_response_code(401);
  exit('Token invalide');
}
```

Il ne nous reste plus qu'à laisser après ces vérifications le reste du code PHP qui s'exécutera normalement.

6. Un mot pour finir :

Le JWT est beaucoup, beaucoup plus sécurisé que ce que nous avions avant, à savoir une authentification simple, et surtout publique pour quiconque sait obtenir les fichier JS de notre projet, ce qui présentait une faille de sécurité béante. Nous venons de répondre à cette problématique, cependant attention, le JWT lui-même n'est pas infaillible.

Deux choses à savoir : Premièrement, le JWT lui-même étant stocké en session, ou dans les cookies du navigateurs, il est **obtenable** pour quiconque sait fouiller dans un navigateur. Que ça soit dans les confins de l'onglet Network de la console, ou dans les cookies eux-mêmes, il sera interceptable. Mais crypté ☺

Deuxièmement, ce n'est pas parce qu'il est crypté qu'il sera à jamais indéchiffrable. C'est pour ça que l'on nous force à choisir des tokens longs (plus de 30 caractères), car sinon, n'importe quel logiciel de **brute force** pourra être en mesure de tester des chaînes de caractères aléatoires à un rythme de plusieurs milliers par minutes.

Cependant, une chaîne de caractère composée de 30 caractères alphanumériques sensibles à la casse… Ça nous donne 62 caractères possibles.

62 exposant 30…

59 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000

possibilités, ou 5.9 x10 exposant 53

Imaginez avec 115 caractères…