Authentification JWT Angular / Symfony

Table des matières

1 Présentation

1.1 Rappel des types d’authentification

Une application web fournit des fonctionnalités en s’appuyant sur un serveur et des clients. Les clients envoient des demandes au serveur qui leur répond. Vus du serveur, les échanges peuvent être :

- **stateless**, c’est à dire indépendants les uns des autres (une demande – une réponse). Le serveur ne conserve alors pas d’état ou de contexte des échanges avec le client.
- ou **stateful**, c’est à dire liés dans une session (plusieurs demandes et réponses avec une logique d’ensemble). Le serveur conserve alors l’état ou le contexte des échanges avec le client.

Une architecture classique d’application web se compose d’un serveur en mode API (Application Programming Interface) et une partie client. Le serveur est alors géré en **stateless**, et c’est au client de s’occuper de la session applicative.

Dans les échanges entre client et serveur, l’authentification est une problématique centrale. Elle permet au serveur de s’assurer lors de chaque échange que la demande du client est valide.

Dans un modèle **stateful**, le serveur gère des sessions. Lorsqu'un client s'authentifie pour démarrer une nouvelle session, un identifiant de session est créé et stocké côté serveur, et envoyé au client. Le client doit ensuite transmettre cet identifiant avec chaque requête afin que le serveur puisse retrouver l'état et les informations associés à la session.

À l'inverse, dans une approche **stateless**, le serveur ne conserve aucun état des sessions client. Un premier échange permet au client de demander au serveur un jeton d'authentification qu'il stocke coté client, puis qu'il transmet au serveur à chaque requête. Le serveur se contente de vérifier la validité de ce jeton. Les **JWT (JSON Web Tokens)** s'inscrivent dans ce modèle.

1.2 Mécanisme d'authentification par JWT

1.2.1 Création initiale du JWT

L'authentification par JWT permet de sécuriser l'accès aux ressources sans gestion de session côté serveur. La session client commence par une authentification réussie, pendant laquelle le serveur génère un jeton d'authentification JWT qui a une durée de vie limitée.

Le cycle de vie d'un JWT comprend trois étapes :

1. **Création**
   Le client fait une demande d'authentification au serveur en lui fournissant ses identifiants de connexion (login et mot de passe). Le serveur génère et lui renvoie un JWT signé, avec une durée de vie courte (généralement entre 5 minutes et 1 heure).
2. **Stockage**
   Coté client, le jeton JWT est stocké dans le navigateur, le plus souvent dans un **cookie sécurisé** avec les attributs :
   - **HttpOnly** : empêche l'accès via JavaScript
   - **Secure** : transmission uniquement via HTTPS
3. **Utilisation**
   Lors de l'accès à une ressource protégée, le navigateur du client envoie automatiquement le JWT au serveur.
   Le serveur :
   - vérifie l'intégrité et la validité du jeton
   - contrôle les autorisations de l'utilisateur
   - autorise ou refuse l'accès à la ressource

Si le JWT est expiré, l'accès est refusé et un mécanisme de renouvellement peut être déclenché.

1.2.2 Renouvellement du JWT

Le **refresh token** est un second jeton généré et transmis au client lors de l'authentification initiale. Contrairement au JWT, il possède une durée de vie plus longue (plusieurs semaines ou mois) et sert uniquement à obtenir de nouveaux JWT.

Tant que le navigateur dispose :

- d’un refresh token valide
- et d’un JWT expiré ou proche de l’expiration

l’utilisateur peut être réauthentifié automatiquement, sans ressaisie de ses identifiants.

Lorsque le client reçoit une réponse refusée par le serveur pour cause de JWT expiré :

1. Le client envoie au serveur une requête de demande de renouvellement du JWT, contenant le refresh token
2. Le refresh token est vérifié par le serveur
3. Si le refresh token est valide, un nouveau JWT est généré et renvoyé au client
4. Le client remplace l’ancien JWT par le nouveau
5. Le client renvoie la requête initiale avec le JWT renouvelé

Ce mécanisme permet d’allier **sécurité** (JWT courts) et **confort utilisateur** (connexion persistante).

1.3 Contexte du présent document

Le présent document décrit les détails de mise en œuvre du mécanisme JWT pour une application stateless avec un client Angular et une API Symfony coté serveur :

- coté serveur Symfony : installation et configuration du bundle Lexik JWT et du bundle gesdinet_jwt_refresh_token
- coté client Angular : envoi du formulaire de connexion initiale, récupération et stockage du JWT et du refresh_token, interception et contrôle des requêtes pour traiter les JWT expirés et les renouveler automatiquement

En plus du mécanisme JWT, la mise en œuvre décrite ici profite aussi de la transaction initiale d’authentification pour permettre au serveur d’envoyer au client certaines données importantes de l’utilisateur connecté (par exemple son adresse email et ses roles applicatifs). Le client peut alors stocker ces données pour gérer les droits d'accès de l'utilisateur aux différents modules de l'application en fonction de ses roles, sans avoir à les demander au serveur par la suite.

En plus du mécanisme JWT, le présent document décrit donc aussi la récupération des roles de l'utilisateur et leur utilisation pour controler ses droits applicatifs.

Remarque : dans la suite du document, les termes serveur et API sont synonymes, de même que rôles et droits.

2 Mise en œuvre coté API Symfony

2.1 Générer un token JWT pour API Symfony

Pour la génération du token JWT, on utilise le bundle Lexik JWT :
https://symfony.com/bundles/LexikJWTAuthenticationBundle/current/index.html

Pour installer le bundle, dans le terminal symfony :

composer require lexik/jwt-authentication-bundle

Puis il faut générer les clés SSL :

php/bin console lexik:jwt:generate-keypair

Un dossier *config/jwt* doit contenir les clés dans les fichiers *private.pem* et *public.rem*.

Un fichier *config/packages/lexik_jwt_authentication.yaml* permet de paramétrer le bundle.
Par défaut il doit contenir les lignes suivantes :

```
lexik_jwt_authentication:

    secret_key: '%env(resolve:JWT_SECRET_KEY)%'
    public_key: '%env(resolve:JWT_PUBLIC_KEY)%'
    pass_phrase: '%env(JWT_PASSPHRASE)%'
```

On ajoute une ligne qui correspond à la durée de validité du token JWT. On met une durée courte de 5 minutes par sécurité:

```
    token_ttl: 300 # in seconds, default is 3600
```

Dans le fichier *config/packages/security.yaml*, il faut ajouter un firewall pour le **login** et un pour l'**api**.
Attention à bien placer **login** AVANT **api** :

```
    firewalls:
    login:
      pattern: ^/api/login_check
      stateless: true
      json_login:
        check_path: /api/login_check
        username_path: email
        password_path: password
        success_handler: lexik_jwt_authentication.handler.authentication_success
        failure_handler: lexik_jwt_authentication.handler.authentication_failure
      api:
      pattern:   ^/api
      stateless: true
      jwt: ~
      entry_point: jwt
      refresh_jwt:
        check_path: api_refresh_token # or, you may use the `api_refresh_token`
route name
        # or if you have more than one user provider
        # provider: user_provider_name
      logout:
        path: api_logout
        clear_site_data:
          - cookies
          - storage
        invalidate_session: true
```

On doit également définir les routes dans access_control :

```
    access_control:
    - { path: ^/api/(login|token/refresh), roles: PUBLIC_ACCESS }
```

Puis, dans le fichier *config/routes.yaml*, on doit définir la route vers le login de l'api :

```
api_login_check:
  path: /api/login_check
  methods: [POST]
```

Pour vérifier que le token est bien généré, dans Postman on fait un post à l'url *http://localhost:8000/api/login_check* en précisant dans le body de la requête `{"username":"nom ou email de l'utilisateur","password":"mot de passe de l'utilisateur"}` :

L'api doit renvoyer un token, comme par exemple :

```
{
  "token":
"eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJpYXQiOjE3MTY4ODU0MTMsImV4cCI6MTcxNjg4OTAxMywicm9sZXMiOlsiUk9MRV9BRE1JTiIsIlJPTEVfVNFUiJdLCJ1c2VybmFtZSI6ImRldkBzaW1wbG9uLm9yZyJ9.di5OA_hrly5JkBN6Cm0-
zJ402QOTU5tu0gMDFYMryB6DBVYzXrA2Oape6HU0SQbmgwYM62YQMiHMGu5_unAK46RdVX5IUblROnlmfsd1FMfNLH4GZiYHZ_oF9-DzpcPNflHQe80k0h39aROn_rs8a1uJWVo7sncrS-
Zp4irh_RKxDJfBjJ8QLTiUX6gMNvA1UsZrloEPrWb9pJgEXVcP-
984MRcKRf0D5uBX0tWEz0Y4U66GFRut4zLVWVf1YMFwsMweMRvypnRBHTA2J0W4h0-Ls5hC3dB_8Q_ajSb-GkC_bkXA2mwIWnimCuzi4TenNWsSv1exubaKIXPfiQ06fw"
}
```

À ce stade, l'API vérifie les données fournies par l'utilisateur (identifiant et mot de passe) et renvoie un token JWT si les données correspondent à un utilisateur valide.

Du coté du client, le browser devra stocker ce token de manière sécurisée, dans un cookie HttpOnly.

2.2 Générer un refresh token pour API Symfony

Pour générer un refresh token lors de l'authentification, au même moment que l'on génère le JWT donc, on utilise le bundle gesdinet_jwt_refresh_token :

https://github.com/markitosgv/JWTRefreshTokenBundle

Installation depuis un terminal :

composer require doctrine/orm doctrine/doctrine-bundle gesdinet/jwt-refresh-token-bundle

Dans le fichier *config/packages/gesdinet_jwt_refresh_token.yaml* (à créer si besoin), ajouter (ou mettre à jour) :

```
gesdinet_jwt_refresh_token:
  refresh_token_class: App\Entity\RefreshToken
  ttl: 2592000 #define the refresh token TTL, this value is set in seconds and defaults to 1 month
```

```
    ttl_update: true #configure the bundle to refresh the TTL on a refresh token when it is used
    token_parameter_name: refresh_token #define the parameter name for the refresh token when it is read from the request, the default value is refresh_token
    return_expiration: false #expiration Unix timestamp will be added to the response
    return_expiration_parameter_name: refresh_token_expiration
    single_use: true #configure the refresh token so it can only be consumed once. If set to true and the refresh token is consumed, a new refresh token will be provided
    cookie:
      enabled: true
      same_site: lax               # default value
      path: /                      # default value
      domain: null                 # default value
      http_only: true              # default value
      secure: true                 # default value
      partitioned: false           # default value
      remove_token_from_body: true # default value
```

Dans *config/packages/security.yaml*, dans `firewalls:`

juste après la section `login:` et avant la section `api:` , on crée une nouvelle section :

```
        api_token_refresh:
          pattern: api_refresh_token
          stateless: true
          refresh_jwt:
            check_path: gesdinet_jwt_refresh_token
```

2.3 Extraire le token JWT des requêtes envoyées du client vers l'api

Côté API, dans le fichier *config/packages/lexik_jwt_authentication.yaml*,

ajouter les lignes suivantes :

```
    token_extractors:
      cookie:
        enabled: true
        name:    BEARER
    set_cookies:
      BEARER: ~
    blocklist_token:
      enabled: true
      cache: cache.app
```

Puis dans *config/services.yaml*, il faut déclarer l'EventListener :

```
    # écoute l'événement avant que le token ne soit envoyé et ajoute les informations sur l'utilisateur (email, roles)
    App\EventListener\AuthenticationSuccessListener:
```

```
    tags:
      - { name: kernel.event_listener, event:
lexik_jwt_authentication.on_authentication_success }
```

2.4 Ajouter les données d'utilisateur à la réponse de l'api

Une fois l'authentification via Lexik JWT en place (la requête à l'api qui renvoie un token JWT fonctionne) on peut mettre en place côté serveur un EventListener qui se déclenche lors de l'authentification, afin de transmettre au client les informations sur l'utilisateur (par exemple id, email, roles).

Dans la partie api de l'appli, on crée un dossier *src/EventListeners* dans lequel on crée un fichier *AuthenticationSuccessListener.php*.

Dans ce fichier, mettre :

```php
<?php

namespace App\EventListener;

use Lexik\Bundle\JWTAuthenticationBundle\Event\AuthenticationSuccessEvent;
use Symfony\Component\Security\Core\User\UserInterface;

class AuthenticationSuccessListener
{
  public function __invoke(AuthenticationSuccessEvent $event): void
  {
    $data = $event->getData();

    /** @var \App\Entity\User $user */
    $user = $event->getUser();

    if (!$user instanceof UserInterface) {
      return;
    }

    $data['id']    = $user->getId();
    $data['email'] = $user->getUserIdentifier();
    $data['roles'] = $user->getRoles();

    $event->setData($data);
  }
}
```

Ce code intercepte la réponse de l'api vers le client, et en plusieurs étapes :

1. récupère les données de l'event et les stocke dans la variable $data
2. vérifie que l'utilisateur est légitime
3. ajoute les informations de l'utilisateur (par exemple id, email, roles) à $data

4. met à jour l'event avec $data enrichi

3 Mise en œuvre coté client Angular

3.1 Stocker le token JWT côté client en cookie HttpOnly

Pour stocker le token dans le browser dans un cookie HttpOnly, il faut pour chaque requête passer l'option withCredentials: true.

Pour cela, on peut ajouter l'option manuellement dans toutes les requêtes, ou automatiser le processus à l'aide d'un interceptor.

Côté client, on crée un fichier *auth.interceptor.ts* dans le dossier *src/app/core/auth*, qui contient la fonction :

```
export function authInterceptor(req: HttpRequest<unknown>, next: HttpHandlerFn): Observable<HttpEvent<unknown>> {

 const authGateway = inject(AuthGateway);
 const authState = inject(AuthState);
 const newReq = req.clone({
  withCredentials: true
 });

 return next(newReq).pipe(
  catchError((error: HttpErrorResponse) => {

   if (
    error.status === HttpStatusCode.Unauthorized &&
    !req.url.endsWith('/login') &&
    !req.url.endsWith('/register') &&
    !req.url.endsWith('/token/refresh') &&
    !req.url.endsWith('/verify/resend')
   ) {

    return authGateway.refreshToken().pipe(
     switchMap(() => {
      return next(newReq);
     }),
     catchError(err => {
      if (authState.isLoggedIn()) {
       authGateway.logout().subscribe();
      }
      return throwError(() => err);
     })
    );
   }

   return throwError(() => error);
```

```
  })
  );
}
```

Cette fonction intercepte toutes les requêtes http avant qu'elles ne soient envoyées.

Quand une requête est interceptée, elle est clonée, et on ajoute au clone l'option qui nous intéresse, withCredentials. Puis on return le clone de la requête, qui est envoyé à l'api.

Cette option indique à l'api de nous répondre en incluant les éventuels cookies, et au client d'inclure les cookies dans les requêtes vers l'api.

3.2 Passerelle d'authentification côté client.

Coté client, on récupére et stocke les données de l'utilisateur qui se connecte. On gère les requêtes d'authentification (login, logout, register, etc) du client vers l'api dans le fichier *src/app/core/auth/auth-gateway.ts* qui contient une fonction pour le login :

```
  public login(payload: LoginPayload): Observable<HttpResponse<AuthUser>> {
    return this.http.post<AuthUser>(
      `${this.apiUrl}/login_check`,
      payload,
      {
        withCredentials: true,
        observe: 'response'
      }
      ).pipe(
      tap({
        next: (response) => {
          if (response.status === HttpStatusCode.Ok) {
            this.authState.setLogin(response.body!)
          }
        },
        error: (err) => {
          if (err.status === HttpStatusCode.Forbidden) {
            this.authState.setLogin(err.body)
          }
          throwError(() => new Error(err))
        },
      })
    );
  }
```

Quand l'authentification est acceptée, l'application récupère certaines données de l'utilisateur dans response.body et les stocke dans la classe AuthState (dans *src/app/core/auth/auth-state.ts*) :

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

```
  readonly isLoggedIn: WritableSignal<boolean | null> = signal(null);
  readonly currentUser: WritableSignal<AuthUser | null> = signal(null);

  setLogin(user: AuthUser) {
    this.isLoggedIn.set(true);
    this.currentUser.set(user);
  }

  setLogout() {
    this.isLoggedIn.set(false);
    this.currentUser.set(null);
  }

  clear() {
    this.isLoggedIn.set(null);
    this.currentUser.set(null);
  }
}
```

Le signal `isLoggedIn` de la classe vaut alors `true`

3.3 Gestion des droits de l'utilisateur côté client.

Pour limiter l'accès à certaines pages côté client, en fonction des droits de l'utilisateur, dans *app.route.ts* on ajoute une option canMatch, par exemple :

```
path: 'admin', component: AdminLayout, canMatch: [AuthGuard], data: {roles: ['ROLE_ADMIN']},
```

Ici, canMatch pointe vers la classe AuthGuard (dans *src/app/core/auth/auth-guard.ts*) :

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

  constructor(private authState: AuthState, private router: Router) {}

  canMatch: CanMatchFn = (route: Route) => {

    if (!this.authState.isLoggedIn() || !this.authState.currentUser()) {
      // pas connecté = redirection login
      return this.router.parseUrl('/login');
    }

    // Vérifier si des rôles sont requis
    const requiredRoles = route.data?.['roles'] as string[] | undefined;

    if (requiredRoles && requiredRoles.length > 0) {
      const hasRole = requiredRoles.some(role => this.authState.isGranted(role));
```

```
    if (!hasRole) {
      // pas le bon rôle = redirection ou blocage
      return this.router.navigate(['/not-found']);
    }
  }

  return true;
 };
}
```

Quand le routeur d'Angular est sollicité pour charger la page correspondant au composant `AdminLayout`, l'AuthGuard se déclenche et vérifie l'état de la classe AuthState, puis le rôle de l'utilisateur si la route nécessite un certain rôle.

Si une vérification échoue, on redirige l'utilisateur vers la page de login ou une page d'erreur 404.

3.4 Traitement d'un refresh token fourni par l'API

La navigation de l'utilisateur vers l'API n'est possible que pendant la durée de validité du token JWT. Une requête à l'API avec un JWT expiré retourne une erreur 401 comme décrit ici dans la documentation officielle de Lexik JWT :

> **About token expiration**
>
> Each request after token expiration will result in a 401 response. Redo the authentication process to obtain a new token.

Pour éviter à l'utilisateur d'avoir à se reconnecter en cas d'expiration du JWT, on le renouvelle automatiquement en utilisant le refresh_token.

Le serveur est configuré pour traiter les refresh_token. Quand le client reçoit du serveur un message d'erreur pour JWT expiré, l'interceptor (le même que celui qui gère l'envoi des tokens) envoie une requête au endpoint /api/token/refresh via une fonction de la classe AuthGateway :

```
  public refreshToken(): Observable<void> {
    return this.http.post<void>(
      `${this.apiUrl}/token/refresh`,
      {
      withCredentials: true,
      observe: 'reponse'
      }
    );
  }
```

Le endpoint de l'API vérifie la valeur du refresh token et la validité du JWT, qui bien qu'expiré, a été généré par l'API.
L'API renvoie alors au client un nouveau JWT et le client effectue a nouveau la requête qui avait été

bloquée par l'erreur 401, dûe au fait que le JWT avait expiré.
On peut voir ça comme un processus de re-login automatique, transparent pour l'utilisateur, qui peut se poursuivre aussi longtemps que le refresh token est valide.

Si le refresh token n'est pas valide, l'appli doit déconnecter l'utilisateur via la fonction logout :

```
    public logout(): Observable<void> {
    return this.http.post<void>(
      `${this.apiUrl}/logout`,
      {
        withCredentials: true,
        observe: 'response'
      }
    ).pipe(
      tap({
        next: () => this.authState.setLogout(),
      })
    );
  }
```

4 Fonctionnement du mécanisme JWT

Une session client typique s’appuie sur les transactions suivantes entre client et serveur :

4.1 transaction initiale d’authentification

Cette transaction se fait une seule fois, elle est la première de la session client.

Le client envoie une demande de connexion au serveur, en lui fournissant l'identifiant et le mot de passe de l'utilisateur
Si l’identifiant et le mot de passe ne sont pas valides, le serveur renvoie au client un message d’erreur
Si l’identifiant et le mot de passe sont valides, le serveur renvoie au client un jeton JWT et un refresh_token, accompagnés éventuellement de certaines données de l’utilisateur qui seront utiles pour le client (par exemple son adresse email et ses roles et droits applicatifs). Le client stocke le JWT et les données d’utilisateur connecté.

4.2 transaction applicative

Cette transaction se fait autant de fois que nécessaire au cours de la session client.

Pour certaines transactions d'accès restreint, le client peut vérifier le droit de l'utilisateur
Le client envoie une demande applicative au serveur, en lui fournissant le JWT d'authentification
Le serveur vérifie le JWT

Si le JWT est actif :

Le serveur renvoie au client la réponse applicative demandée

Si le JWT est expiré (ce qui arrive typiquement toutes les 5 minutes) :

Le serveur renvoie au client un message d’erreur JWT expiré (erreur 401 `Unauthorized`)
L’interceptor du client envoie au serveur une demande de renouvellement du JWT en lui fournissant le refresh token
Le serveur renvoie au client un nouveau JWT actif
Le client renvoie la demande applicative au serveur avec le nouveau JWT actif