Authentification JWT Angular / Symfony

Table des matières

1 Générer un token JWT

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 SLL :

php/bin console lexik:jwt:generate-keypair

Si tout s'est bien passé, 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 :

```
    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 en précisant dans le body de la requête `{"username":"nom ou email de l'utilisateur","password":"mot de passe de l'utilisateur"}` :

http://localhost:8000/api/login_check

L'api doit renvoyer un token, comme :

```
{
  "token":
"eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJpYXQiOjE3MTY4ODU0MTMsImV4cCI6MTcxNjg4OTAxMywicm9
sZXMiOlsiUk9MRV9BRE1JTiIsIlJPTEVfVVNFUiJdLCJ1c2VybmFtZSI6ImRldkBzaW1wbG9uLm9yZyJ9.di5OA_h
rly5JkBN6Cm0-
```

```
zJ402QOTU5tu0gMDFYMryB6DBVYzXrA2Oape6HU0SQbmgwYM62YQMiHMGu5_unAK46RdVX5IUblROnlmfsd1FMfNL
H4GZiYHZ_oF9-DzpcPNflHQe80k0h39aROn_rs8a1uJWVo7sncrS-
Zp4irh_RKxDJfBjJ8QLTiUX6gMNvA1UsZrloEPrWb9pJgEXVcP-
984MRcKRf0D5uBX0tWEz0Y4U66GFRut4zLVWVf1YMFwsMweMRvypnRBHTA2J0W4h0-Ls5hC3dB_8Q_ajSb-
GkC_bkXA2mwIWnimCuzi4TenNWsSv1exubaKIXPfiQ06fw"
}
```


À ce stade, l'API vérifie les données de l'utilisateur (email et mot de passe) et renvoie un token JWT si les données correspondent à un utilisateur présent dans la base de données.

La prochaine étape consiste à stocker ce token dans le browser de manière sécurisée, dans un cookie httponly.

2 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 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 }
```

4 Passerelle d'authentification côté client.

4.1 Ajouter l'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 api un EventListener qui se déclenche lors de l'authentification, afin de transmettre au client les informations sur l'utilisateur (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 à $data
4. met à jour l'event avec $data enrichi

4.2 Vérifier, récupérer et stocker l'utilisateur

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'utilisateur est récupéré dans response.body et stocké 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`

5 Vérifier la session côté client.

Pour protéger des pages côté client, dans *app.route.ts* on ajoute une option canMatch, par exemple :

```
            { path: 'user-profile', component: UserProfile, canMatch: [AuthGuard],
data: {roles: ['ROLE_USER']} },
```

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 est sollicité pour charger la page correspondant au UserProfile, l'AuthGuard se déclenche et vérifie l'état de la classe AuthState, puis le rôle de l'utilisateur si la route demande une certain rôle.

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

À ce stade, la navigation de l'utilisateur n'est possible que pendant la durée de validité du token JWT.
Quand le token expire, l'utilisateur n'est plus reconnu par l'API, et doit se reconnecter.
Pour renouveler le JWT automatiquement, on utilise un refresh token.

6 Génération d'un refresh token par l'API

Pour générer un refresh token lors de l'authentication, au même moment que l'on génère le JWT donc, on va utiliser 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
```

Désormais, quand le client effectue une requête à l'API avec un JWT expiré, l'interceptor envoie une requête au endpoint /api/token/refresh via la fonction de la classe AuthGateway :

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

Le endpoint 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 un nouveau JWT et 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.

En revanche, 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(),
      })
    );
  }
```