

- Lorsque le refresh token est expiré qu'est ce que se passe ?

* **Gestion du refreshToken** : Le refreshToken est stocké dans un cookie HTTP sécurisé (HttpOnly, Secure, SameSite=Strict, Path=/api/v1/auth, durée de vie de 7 jours).

L'endpoint `/api/v1/auth/refresh` est utilisé pour générer un nouvel `accessToken` et un nouveau `refreshToken` lorsque l'`accessToken` expire, en lisant le refreshToken depuis le cookie.

* **Validation du refreshToken** : Dans `JwtService`, la méthode `isTokenValid` vérifie si le refreshToken est valide en : Vérifiant que le username extrait correspond à celui de l'utilisateur. Vérifiant que le token n'est pas expiré (via `isTokenExpired`, qui compare la date d'expiration avec la date actuelle).

* **Comportement du frontend** : Ton `authInterceptor` envoie une requête à `/api/v1/auth/refresh` avec `withCredentials: true` lorsque l'`accessToken` provoque une erreur 401.

→ Si la requête à `/refresh` échoue (par exemple, à cause d'un refreshToken invalide ou expiré), l'intercepteur supprime les données locales (token et user) et redirige vers la page de connexion.

Que se passe-t-il lorsque le refreshToken expire ? -> Lorsque le refreshToken expire (après 7 jours), voici la séquence des événements :

1. Déclenchement : L'`accessToken` expire après 1 minute, ce qui est fréquent pendant tes tests. Ton frontend détecte une erreur 401 lors d'une requête protégée et envoie une requête POST à `/api/v1/auth/refresh` avec le cookie refreshToken (via `withCredentials: true`).
2. Traitement dans `AuthenticationController` : L'endpoint `/refresh` lit le refreshToken depuis le cookie :

```
String refreshToken = null;
Cookie[] cookies = request.getCookies();
if (cookies != null) {
    for (Cookie cookie : cookies) {
        if ("refreshToken".equals(cookie.getName())) {
            refreshToken = cookie.getValue();
            break;
        }
    }
}
if (refreshToken == null) {
    throw new ResponseStatusException(HttpStatus.UNAUTHORIZED, "No refresh token provided");
}
```

Si le cookie existe (ce qui est probable, car il a une durée de vie de 7 jours), le refreshToken est extrait.

3. Validation du refreshToken : L'endpoint extrait le matricule (username) et récupère l'utilisateur :

```
String matricule = jwtService.extractUsername(refreshToken);
User user = userRepository.findByMatricule(matricule)
    .orElseThrow(() -> new RuntimeException(HttpStatus.NOT_FOUND,
    "Utilisateur non trouvé"));
```

Ensuite, il valide le refreshToken avec jwtService.isTokenValid :

```
if (!jwtService.isTokenValid(refreshToken, user)) {
    throw new RuntimeException(HttpStatus.UNAUTHORIZED, "Invalid refresh
    token");
}
```

Dans **JwtService**, **isTokenValid** vérifie :

- Si le username correspond.
- Si le token n'est pas expiré :

```
private boolean isTokenExpired(String token) {
    return extractExpiration(token).before(new Date());
}
```

Puisque le refreshToken est expiré (après 7 jours), isTokenExpired retourne true, donc isTokenValid retourne false.

4. Résultat de l'expiration : Comme isTokenValid retourne false, l'endpoint /refresh lève une exception : `throw new RuntimeException(HttpStatus.UNAUTHORIZED, "Invalid refresh token");`

5. Comportement du frontend :

- Ton `authInterceptor` gère l'erreur 401 de /refresh dans le bloc `catchError` :

typescript

... Copier

```
catchError((refreshErr) => {
    console.error('Refresh token failed:', refreshErr);
    localStorage.removeItem('token');
    localStorage.removeItem('user');
    router.navigate(['/']);
    return throwError(() => refreshErr);
})
```

Conséquences : Les données locales (token et user) sont supprimées de localStorage.

- ➔ L'utilisateur est redirigé vers la page de connexion (/).
- ➔ L'utilisateur doit se reconnecter pour obtenir un nouvel accessToken et un nouveau refreshToken.
- ➔ Cookie refreshToken : Le cookie refreshToken reste dans le navigateur jusqu'à ce qu'il atteigne sa durée de vie maximale (7 jours, définie par `setMaxAge(7 * 24 * 60 * 60)`).

- Faut-il supprimer le cookie refreshToken expiré ?

- Puisque le cookie a une durée de vie alignée sur l'expiration du refreshToken (7 jours), il disparaîtra automatiquement peu après l'expiration. Supprimer le cookie manuellement n'apporte qu'un bénéfice marginal si l'expiration est imminente.
- Le refreshToken expiré est déjà inoffensif, car JwtService.isTokenValid le rejette.

Comportement du frontend inchangé :

- Ton frontend redirige déjà vers la page de connexion après un 401 de /refresh, que le cookie soit supprimé ou non. L'utilisateur doit se reconnecter, et un nouveau cookie refreshToken sera créé, remplaçant l'ancien.

Conclusion : Tu ne dois pas nécessairement supprimer le cookie **refreshToken** lorsqu'il est expiré, car ton implémentation actuelle est fonctionnelle et sécurisée : le **refreshToken** expiré est rejeté, et l'utilisateur est déconnecté.

- **si le local storage n'est pas sécurisé pourquoi on stocke l'access token dans le local storage? pq on le stocke pas comme le refreshToken dans le cookies ?**

1. Pourquoi le localStorage n'est pas sécurisé ?

Le localStorage (et le sessionStorage) est accessible via JavaScript dans le navigateur, ce qui le rend vulnérable à certaines attaques, notamment :

- **XSS (Cross-Site Scripting) :**
 - Si un attaquant parvient à injecter du code JavaScript malveillant sur ton site (par exemple, via un champ de formulaire non sécurisé ou une dépendance compromise), ce code peut lire le contenu de localStorage et voler l'accessToken.
 - Une fois volé, l'accessToken peut être utilisé pour effectuer des requêtes authentifiées jusqu'à son expiration (1 minute dans ton cas, ce qui limite les dégâts).
- **Pas de contrôle d'accès intégré :**
 - Contrairement aux cookies avec l'attribut HttpOnly, qui ne sont pas accessibles via JavaScript, localStorage est accessible à tout script exécuté dans le contexte de ton domaine.
- **Persistence :**
 - Les données dans localStorage persistent jusqu'à ce qu'elles soient explicitement supprimées, ce qui peut augmenter le risque si un attaquant accède au navigateur de l'utilisateur.

2. Pourquoi stocke-t-on souvent l'accessToken dans localStorage ?

Malgré ces faiblesses, le localStorage est couramment utilisé pour stocker l'accessToken dans les applications web modernes (comme ton frontend Angular) **pour plusieurs raisons** :

a. Facilité d'accès pour les requêtes HTTP

- Dans une application frontend comme Angular, l'accessToken est nécessaire pour ajouter l'en-tête Authorization: Bearer <token> à chaque requête HTTP vers les endpoints protégés de ton API.
- Avec localStorage, ton authInterceptor peut facilement récupérer l'accessToken et l'ajouter aux requêtes :

```
const token = localStorage.getItem('token');
if (token) {
  req = req.clone({
    setHeaders: { Authorization: `Bearer ${token}` }
  });
}
```

→ Si l'accessToken était dans un cookie, il serait plus difficile (voire impossible, avec HttpOnly) pour le frontend de le lire et de ***l'ajouter à l'en-tête Authorization***.

b. Conventions des API REST

- Les API REST modernes, comme ton backend Spring Boot, s'attendent généralement à ce que l'accessToken soit envoyé dans l'en-tête **Authorization: Bearer <token>** rather than as a cookie.
- Les cookies sont souvent utilisés pour des sessions basées sur des identifiants opaques (comme dans les applications traditionnelles avec sessions côté serveur), mais les JWT (accessToken) sont conçus pour être inclus dans les en-têtes HTTP, ce qui est plus aligné avec **les standards OAuth 2.0 et OpenID Connect**.

c. Contrôle par le frontend

- Stocker l'accessToken dans localStorage donne au frontend un contrôle total sur son utilisation, son ajout aux requêtes, et sa suppression (par exemple, lors de la déconnexion avec localStorage.removeItem('token')).
- **Avec un cookie**, le frontend **a moins de contrôle**, car les cookies sont automatiquement envoyés par le navigateur pour toutes les requêtes correspondant au domaine et au chemin du cookie.

d. Durée de vie courte de l'accessToken

- Dans ton cas, l'accessToken expire après **1 minute**, ce qui réduit considérablement la fenêtre d'opportunité pour un attaquant en cas de vol via XSS.
- Cette courte durée de vie atténue les risques associés au stockage dans localStorage, car un token volé devient rapidement inutilisable.

3. Pourquoi le refreshToken est stocké dans un cookie sécurisé ?

Ton refreshToken est stocké dans un cookie avec les attributs HttpOnly, Secure, SameSite=Strict, et Path=/api/v1/auth, ce qui est une bonne pratique pour plusieurs raisons :

a. Protection contre XSS

- L'attribut HttpOnly empêche tout script JavaScript (y compris les scripts malveillants injectés via XSS) d'accéder au refreshToken. Cela le rend beaucoup plus sécurisé que le stockage dans localStorage.

b. Envoi automatique par le navigateur

- Les cookies sont automatiquement envoyés par le navigateur pour les requêtes correspondant au domaine et au chemin du cookie (dans ton cas, /api/v1/auth).
- Cela est idéal pour l'endpoint /api/v1/auth/refresh, qui lit le refreshToken depuis le cookie sans que le frontend ait besoin de l'envoyer manuellement :

```
String refreshToken = null;
Cookie[] cookies = request.getCookies();
if (cookies != null) {
    for (Cookie cookie : cookies) {
        if ("refreshToken".equals(cookie.getName())) {
            refreshToken = cookie.getValue();
            break;
        }
    }
}
```

- Ton authInterceptor envoie la requête à /refresh avec withCredentials: true, ce qui inclut automatiquement le cookie.

c. Durée de vie longue du refreshToken

- Ton refreshToken a une durée de vie de **7 jours**, ce qui est beaucoup plus long que l'accessToken. Stocker un token de longue durée dans localStorage serait risqué, car un attaquant pourrait le voler via XSS et l'utiliser pour générer de nouveaux accessToken pendant 7 jours.
- Un cookie HttpOnly réduit ce risque en rendant le refreshToken inaccessible aux scripts.

d. Protection contre CSRF

- L'attribut SameSite=Strict protège contre les attaques CSRF en s'assurant que le cookie refreshToken n'est envoyé que pour les requêtes provenant de ton domaine.
- De plus, ton endpoint /refresh est un POST, ce qui ajoute une couche de protection contre CSRF (les requêtes POST nécessitent un préflight pour les requêtes cross-origin).

4. Pourquoi ne pas stocker l'accessToken dans un cookie comme le refreshToken ?

-> **Avantages de stocker l'accessToken dans un cookie**

1. Protection contre XSS :

- Avec HttpOnly, l'accessToken serait inaccessible aux scripts JavaScript, réduisant le risque de vol via XSS.

2. Envoi automatique :

- Le navigateur enverrait automatiquement l'accessToken dans les requêtes, ce qui pourrait simplifier certaines parties du frontend (pas besoin de gérer l'en-tête Authorization).

Inconvénients et défis

1. Incompatibilité avec les conventions REST : Ton backend s'attend à recevoir l'accessToken dans l'en-tête Authorization: Bearer <token>

- Si l'accessToken est dans un cookie, tu devrais modifier JwtAuthenticationFilter pour le lire depuis le cookie, ce qui va à l'encontre des standards REST et JWT. Les cookies sont généralement utilisés pour des identifiants de session, pas pour des JWT.

2. Gestion des cookies pour toutes les requêtes :

- Si l'accessToken est dans un cookie, il serait envoyé automatiquement à **toutes** les requêtes correspondant au domaine et au chemin du cookie (par exemple, Path=/). *Cela pourrait inclure des requêtes non protégées ou des endpoints tiers*, augmentant le risque d'exposition.
- Avec localStorage, le frontend contrôle précisément quelles requêtes incluent l'accessToken via l'en-tête Authorization.

3. Complexité pour le frontend :

- Ton authInterceptor ne pourrait pas lire l'accessToken d'un cookie HttpOnly pour l'ajouter à l'en-tête Authorization. Tu devrais soit :
 - Supprimer HttpOnly (ce qui rendrait l'accessToken vulnérable à XSS, annulant l'avantage du cookie).
 - Modifier le backend pour accepter l'accessToken depuis un cookie, ce qui nécessiterait des changements dans JwtAuthenticationFilter et potentiellement d'autres composants de sécurité.

4. Durée de vie courte de l'accessToken :

- Avec une expiration de 1 minute, l'accessToken doit être fréquemment rafraîchi via /refresh. Si l'accessToken est dans un cookie, le frontend ne peut pas facilement le mettre à jour ou le supprimer sans intervention du serveur.
- Avec localStorage, le frontend peut simplement écraser localStorage.setItem('token', newToken) après un appel à /refresh.

5. Conflit avec le refreshToken :

- Avoir deux cookies (refreshToken et accessToken) pourrait compliquer la gestion, surtout si leurs chemins ou attributs diffèrent. Par exemple, tu devrais définir un Path spécifique pour l'accessToken (par exemple, /api) et gérer les conflits avec le cookie refreshToken (limité à /api/v1/auth).

6. Risque de CSRF :

- Les cookies sont envoyés automatiquement, ce qui peut augmenter le risque d'attaques CSRF si l'accessToken est utilisé pour des endpoints sensibles (par exemple, des POST ou DELETE). Bien que `SameSite=Strict` atténue ce risque, l'en-tête Authorization est intrinsèquement plus sûr pour les requêtes authentifiées, car il n'est pas envoyé automatiquement.

5. Ton contexte spécifique

Dans ton application :

- L'accessToken (nommé token) est stocké dans localStorage et envoyé dans l'en-tête Authorization par authInterceptor.
- Le refreshToken est dans un cookie sécurisé, envoyé automatiquement à /api/v1/auth/refresh avec withCredentials: true.
- L'accessToken expire après 1 minute, ce qui limite fortement les risques en cas de vol via XSS.
- Le refreshToken expire après 7 jours, et sa protection via HttpOnly est essentielle pour éviter un vol prolongé.

Pourquoi cette approche est raisonnable :

- **Sécurité équilibrée :**
 - L'accessToken dans localStorage est acceptable car sa durée de vie est très courte (1 minute). Même en cas de vol, l'attaquant a peu de temps pour l'exploiter.
 - Le refreshToken dans un cookie HttpOnly est bien protégé contre XSS, ce qui est crucial pour un token de longue durée.
- **Conformité aux standards :**
 - Ton backend utilise l'en-tête Authorization: Bearer <token>, ce qui est standard pour les JWT. Stocker l'accessToken dans localStorage permet à authInterceptor de l'ajouter facilement aux requêtes.
- **Simplicité :**
 - Ton authInterceptor gère bien le rafraîchissement des tokens et la déconnexion. Passer l'accessToken à un cookie nécessiterait des changements importants dans le

backend (notamment JwtAuthenticationFilter) et pourrait compliquer la gestion des tokens.

- Tu as déjà une expiration de 1 minute pour l'accessToken, ce qui est très sécurisé. Maintiens cette pratique pour limiter l'impact d'un vol.

1. Clarification : Le refreshToken ne devient pas l'accessToken

Tu sembles penser que, après un appel à /refresh, le refreshToken devient l'accessToken. Ce n'est pas le cas. Voici ce qui se passe réellement dans ton endpoint /refresh (dans AuthenticationController) :

```
// Générer un nouvel accessToken et refreshToken
String newAccessToken = jwtService.generateToken(user);
String newRefreshToken = jwtService.generateRefreshToken(user);

// Ajouter le nouveau refreshToken dans un cookie sécurisé
Cookie refreshTokenCookie = new Cookie("refreshToken", newRefreshToken);
refreshTokenCookie.setHttpOnly(true);
refreshTokenCookie.setSecure(true);
refreshTokenCookie.setPath("/api/v1/auth");
refreshTokenCookie.setMaxAge(7 * 24 * 60 * 60);
refreshTokenCookie.setAttribute("SameSite", "Strict");
response.addCookie(refreshTokenCookie);

// Retourner la réponse avec le nouvel accessToken
return ResponseEntity.ok(AuthenticationResponse.builder()
    .token(newAccessToken)
    .user(user)
    .groupe(user.getGroupe() != null ? user.getGroupe().getNom() : "Aucun")
    .build());
```

Le refreshToken ne devient pas l'accessToken. Au lieu de cela, deux nouveaux tokens sont générés :

- Un accessToken pour authentifier les requêtes protégées.
- Un refreshToken pour permettre de futurs rafraîchissements.
- Ces tokens sont distincts, avec des revendications différentes (token_type: "access" vs token_type: "refresh") et des durées de vie différentes (1 minute vs 7 jours).

2. Pourquoi le refreshToken fonctionne bien dans un cookie ?

Le refreshToken est stocké dans un cookie avec les attributs **HttpOnly**, **Secure**, **SameSite=Strict**, et **Path=/api/v1/auth**. Voici pourquoi cela fonctionne bien et ne pose pas les mêmes problèmes que l'accessToken dans un cookie :

a. Usage limité du refreshToken

- Le refreshToken est uniquement utilisé par l'endpoint **/api/v1/auth/refresh**, qui est un endpoint spécifique avec un chemin bien défini (/api/v1/auth).
- Le cookie est configuré avec Path=/api/v1/auth, ce qui signifie qu'il n'est envoyé que pour les requêtes vers cet endpoint, réduisant l'exposition à d'autres parties de l'application.

b. Protection contre XSS

- L'attribut `HttpOnly` garantit que le `refreshToken` n'est pas accessible via JavaScript, ce qui le protège contre les attaques XSS. Un script malveillant ne peut pas voler le `refreshToken` depuis le navigateur.

c. Envoi automatique par le navigateur

- Le cookie est envoyé automatiquement par le navigateur pour les requêtes à `/api/v1/auth/refresh` (avec `withCredentials: true` dans le frontend), ce qui simplifie la logique
 - Le frontend n'a pas besoin de lire ou de manipuler le `refreshToken`.
 - Le serveur extrait le `refreshToken` directement depuis le cookie.

d. Longue durée de vie

- Le `refreshToken` a une durée de vie de 7 jours, ce qui justifie son stockage dans un endroit sécurisé comme un cookie `HttpOnly`. Une longue durée dans `localStorage` serait risquée, car il pourrait être volé via XSS et utilisé pour générer des `accessToken` pendant 7 jours.

e. Protection contre CSRF

- L'attribut `SameSite=Strict` empêche le cookie d'être envoyé dans des requêtes cross-site, réduisant le risque d'attaques CSRF.
- L'endpoint `/refresh` est un POST, ce qui ajoute une protection supplémentaire contre CSRF (les requêtes POST cross-origin nécessitent un préflight).

4. Pourquoi pas de problèmes similaires pour le refreshToken ?

Les problèmes liés à l'`accessToken` dans un cookie ne se posent pas pour le `refreshToken` car :

- **Rôle différent :**
 - L'`accessToken` est utilisé pour authentifier toutes les requêtes protégées, nécessitant un accès fréquent et un ajout dans l'en-tête `Authorization`. Cela le rend plus adapté à `localStorage`, où le frontend peut le manipuler.
 - Le `refreshToken` est utilisé uniquement pour rafraîchir l'`accessToken` via un endpoint spécifique (`/refresh`). Son envoi automatique via un cookie est idéal, car le frontend n'a pas besoin de le lire.
- **Exposition limitée :**
 - Le `refreshToken` est confiné à un chemin spécifique (`/api/v1/auth`) et protégé par `HttpOnly`, ce qui limite son exposition et le protège contre XSS.
 - Un `accessToken` dans un cookie serait envoyé à de nombreux endpoints, augmentant les risques.

Comment ça marche après un /refresh ?

Après un appel à /refresh :

1. Backend :

- Le serveur valide le refreshToken depuis le cookie.
- Il génère un **nouvel accessToken** (exp. 1 minute) et un **nouveau refreshToken** (exp. 7 jours).
- Le nouvel accessToken est renvoyé dans la réponse JSON (AuthenticationResponse.token).
- Le nouveau refreshToken est stocké dans un cookie, remplaçant l'ancien.

2. Frontend :

- L'authInterceptor reçoit la réponse de /refresh et met à jour localStorage avec le nouvel accessToken :

```
localStorage.setItem('token', res.token);
```

- La requête originale (qui avait échoué avec 401) est réessayée avec le nouvel accessToken dans l'en-tête Authorization.
- Le nouveau cookie refreshToken est automatiquement géré par le navigateur et sera utilisé pour le prochain appel à /refresh.

7. Si tu veux vraiment l'accessToken dans un cookie

Si tu insistes pour stocker l'accessToken dans un cookie malgré les inconvénients, voici un résumé des changements nécessaires (détaillés dans ma réponse précédente) :

- Modifie JwtAuthenticationFilter pour lire l'accessToken depuis un cookie nommé accessToken.
- Mets à jour AuthenticationService et AuthenticationController (/refresh) pour ajouter l'accessToken dans un cookie avec Path=/api et MaxAge=60 (1 minute).
- Supprime la gestion de localStorage dans le frontend et configure authInterceptor pour ne pas ajouter l'en-tête Authorization.
- **Problèmes :**
 - Non-standard (pas d'en-tête Authorization).
 - Envoi automatique à tous les endpoints /api, augmentant l'exposition.
 - Gestion complexe des mises à jour fréquentes du cookie (toutes les minutes).

Je ne recommande pas cette approche, car elle complique ton système sans apporter de bénéfices significatifs par rapport à `localStorage` avec une expiration de 1 minute.

❖ Qu'est-ce qu'un `refreshToken` ?

- Un `refreshToken` est un token sécurisé utilisé pour obtenir un nouvel `accessToken` lorsque l'`accessToken` expire, sans obliger l'utilisateur à se reconnecter (entrer à nouveau son matricule et mot de passe).
- Contrairement à l'`accessToken`, qui est utilisé pour authentifier les requêtes protégées (par exemple, accéder à `/api/v1/protected`), le `refreshToken` est uniquement utilisé pour appeler l'endpoint `/api/v1/auth/refresh`.

Comment fonctionne-t-il dans ton code ?

- Ton endpoint `/refresh` (dans `AuthenticationController`) utilise le `refreshToken` (extrait d'un cookie) pour valider l'identité de l'utilisateur et générer :
 - Un **nouvel `accessToken`** (valide pour 1 minute actuellement, mais 1 jour en production).
 - Un **nouveau `refreshToken`** (valide pour 7 jours), qui remplace l'ancien dans un nouveau cookie.
 - **Ce que ça signifie :**
 - Le `refreshToken` **est utilisé** pour vérifier que l'utilisateur a une session valide (c'est-à-dire qu'il s'est authentifié récemment).
 - Une fois validé, le serveur génère un nouvel `accessToken` pour permettre à l'utilisateur de continuer à accéder aux ressources protégées.
 - Un nouveau `refreshToken` est aussi généré (rotation des `refreshToken`), ce qui améliore la sécurité en limitant la durée de vie de chaque `refreshToken` individuel.
- **Le `refreshToken` ne devient pas l'`accessToken` :**
 - Tu as peut-être pensé que le `refreshToken` devient l'`accessToken` après `/refresh`. Ce n'est pas le cas. Le `refreshToken` est un token distinct avec un rôle spécifique : il sert de "clé" pour demander un nouvel `accessToken`. Les deux tokens ont des revendications différentes (`token_type: "refresh"` vs `token_type: "access"`) et des durées de vie différentes.

2. Utilité du `refreshToken`

Tu te demandes : *"Pourquoi avoir implémenté un `refreshToken` si on génère juste un nouvel `accessToken` ?"* L'utilité du `refreshToken` est essentielle pour améliorer l'**expérience utilisateur** tout en maintenant la **sécurité**. Voici pourquoi :

a. Éviter des reconnections fréquentes

- Sans refreshToken :
 - Une fois l'accessToken expiré (après 1 minute actuellement, ou 1 jour en production), l'utilisateur devrait se reconnecter manuellement en entrant son matricule et mot de passe.
 - Cela serait frustrant, surtout si l'utilisateur est actif mais que l'accessToken expire après 1 jour. Imagine devoir te reconnecter toutes les 24 heures !
- Avec refreshToken :
 - Lorsque l'accessToken expire, ton authInterceptor appelle /api/v1/auth/refresh automatiquement, utilise le refreshToken (stocké dans le cookie) pour obtenir un nouvel accessToken, et l'utilisateur peut continuer sans interruption.
 - Le refreshToken (valide 7 jours) permet à l'utilisateur de rester connecté pendant 7 jours sans avoir à entrer ses identifiants, tant qu'il utilise l'application régulièrement.

b. Sécurité renforcée

- **Durée de vie courte de l'accessToken :**
 - Une courte durée de vie pour l'accessToken (1 minute pour tes tests, 1 jour en production) réduit la fenêtre d'opportunité pour un attaquant si le token est volé (par exemple, via XSS dans localStorage).
 - Cependant, une courte durée signifie que l'accessToken expire souvent, ce qui serait gênant sans un mécanisme de rafraîchissement.
- **Rôle du refreshToken :**
 - Le refreshToken est plus sécurisé (stocké dans un cookie HttpOnly, inaccessible via JavaScript) et a une durée de vie plus longue (7 jours).
 - Il agit comme une "clé de secours" pour générer de nouveaux accessToken sans compromettre la sécurité (pas besoin de stocker les identifiants de l'utilisateur dans le frontend).

c. Rotation des refreshToken

- Ton système implémente la **rotation des refreshToken** : à chaque appel à /refresh, un nouveau refreshToken est généré et envoyé dans un cookie, remplaçant l'ancien.
- Cela améliore la sécurité :
 - Si un refreshToken est volé, il ne peut être utilisé que jusqu'au prochain appel à /refresh, car un nouveau refreshToken sera émis, rendant l'ancien invalide pour les futurs rafraîchissements (bien que, sans base de données, l'ancien reste valide jusqu'à son expiration de 7 jours).
 - Cela limite la durée pendant laquelle un refreshToken volé peut être exploité.

b. Maintenir la sécurité

- Une durée de vie de 1 jour pour l'accessToken est raisonnable en production (bien que toujours plus risquée que 1 minute), car elle limite l'impact d'un vol de token.
- Le refreshToken, stocké dans un cookie HttpOnly, est plus sécurisé et peut avoir une durée de vie plus longue (7 jours) sans compromettre la sécurité, car il n'est pas accessible via JavaScript.

c. Standard de l'industrie

- L'utilisation d'un refreshToken pour prolonger les sessions sans reconnexion est une pratique standard dans les systèmes basés sur JWT, conforme à OAuth 2.0 et OpenID Connect.
- Ton implémentation suit ce modèle : un accessToken de courte durée pour les requêtes protégées et un refreshToken de longue durée pour le rafraîchissement.

d. Rotation pour plus de sécurité

- La rotation des refreshToken (générer un nouveau refreshToken à chaque /refresh) ajoute une couche de sécurité. Même si un refreshToken est intercepté, son utilisation est limitée par le fait qu'un nouveau refreshToken sera émis lors du prochain rafraîchissement.

Pourquoi on génère un nouveau refreshToken à chaque appel de l'api /refresh même s'il le refresh token qu'on utilise n'a pas encore expiré ?

Ce qui se passe :

- Le refreshToken actuel (extrait du cookie) est validé via `jwtService.isTokenValid`.
- Si valide, tu génères :
 - Un **nouvel accessToken** (valide 1 jour en production, 1 minute pour les tests).
 - Un **nouveau refreshToken** (valide 7 jours), qui est envoyé dans un nouveau cookie, remplaçant l'ancien.

Cette pratique est appelée **rotation des refreshToken** : à chaque appel à /refresh, un nouveau refreshToken est émis, même si l'ancien est encore valide (par exemple, il reste 6 jours sur ses 7 jours de validité).

Pourquoi faire cela ?

La rotation des refreshToken est une mesure de sécurité courante dans les systèmes d'authentification modernes, et voici les raisons principales :

1. Réduire l'impact d'un vol de refreshToken :

- Si un attaquant vole un refreshToken (par exemple, via une interception réseau ou une faille temporaire), ce token ne peut être utilisé que jusqu'au prochain appel légitime à /refresh.

- Lors du prochain /refresh, un nouveau refreshToken est émis, et l'ancien devient **invalide** pour les futurs appels à /refresh (bien que, dans ton cas, sans base de données, l'ancien reste techniquement valide jusqu'à son expiration de 7 jours).
- Cela limite la fenêtre pendant laquelle un refreshToken volé peut être exploité.

2. Détection des abus :

- La rotation permet de détecter des comportements suspects. Par exemple, si un ancien refreshToken est utilisé après qu'un nouveau a été émis, cela pourrait indiquer une tentative d'attaque (bien que ton implémentation actuelle ne vérifie pas cela explicitement sans base de données).

3. Conformité avec les standards modernes :

- La rotation des refreshToken est recommandée par les standards OAuth 2.0 et OpenID Connect, surtout pour les applications à haute sécurité. Elle est particulièrement courante dans les systèmes où les refreshToken ont une longue durée de vie (comme tes 7 jours).

4. Renouvellement proactif :

- En régénérant le refreshToken, tu réinitialises sa durée de vie à 7 jours à chaque appel à /refresh. Cela garantit que les utilisateurs actifs peuvent rester connectés indéfiniment (tant qu'ils utilisent l'application régulièrement), sans atteindre l'expiration du refreshToken.

5. Sécurité actuelle :

Ton refreshToken est bien protégé dans un cookie HttpOnly, Secure, SameSite=Strict, avec Path=/api/v1/auth. Cela réduit déjà le risque de vol (par exemple, via XSS). La rotation ajoute une couche de sécurité, mais son impact est limité sans invalidation explicite des anciens tokens.

Conformité aux bonnes pratiques :

- La rotation est une pratique standard dans les systèmes modernes, recommandée par OAuth 2.0 pour les applications sensibles.

4. Dois-tu garder la rotation des refreshToken ?

Dans ton cas, la rotation des refreshToken est **utile mais pas pleinement optimisée** en raison de l'absence de base de données pour invalider les anciens tokens. Voici une analyse pour t'aider à décider si tu dois la conserver ou la supprimer :

Pourquoi garder la rotation ?

- **Amélioration de la sécurité :**
 - Même sans base de données, la rotation réduit la durée de vie effective des anciens refreshToken pour les utilisateurs actifs. Par exemple, si un utilisateur appelle

/refresh chaque jour, un refreshToken volé il y a 3 jours ne sera plus dans le cookie actuel (bien qu'il reste valide jusqu'à son expiration).

- **Prolongation des sessions :**

- La rotation garantit que les utilisateurs actifs ne verront jamais leur refreshToken expirer, car un nouveau est émis à chaque /refresh.

- **Préparation pour une future base de données :**

- Si tu ajoutes une base de données plus tard pour stocker et invalider les refreshToken, la rotation deviendra beaucoup plus efficace, car tu pourras marquer les anciens tokens comme invalides immédiatement.

Bonne pratique :

- La rotation est standard et attendue dans les systèmes modernes. La conserver te permet de suivre les recommandations de sécurité.

Recommandation

Dans ton contexte actuel (pas de base de données, accessToken de 1 jour, refreshToken de 7 jours, cookie sécurisé), **je recommande de garder la rotation des refreshToken** pour les raisons suivantes :

- Elle prolonge les sessions des utilisateurs actifs indéfiniment, ce qui est une bonne expérience utilisateur.
- Elle ajoute une couche de sécurité, même si l'impact est limité sans invalidation explicite.
- Elle te prépare à une future implémentation avec une base de données, où la rotation deviendra plus puissante.
- La surcharge (générer un nouveau refreshToken et cookie) est minime, et ton code est déjà configuré pour cela.

Pourquoi séparer (utiliser 2 token au lieu d'un seul) ?

- Si l'accessToken était utilisé pour se rafraîchir lui-même, il devrait être envoyé à /refresh en plus d'être utilisé pour les requêtes protégées. Cela augmente les risques de sécurité, car l'accessToken est plus exposé (stocké dans localStorage, accessible via JavaScript, envoyé dans de nombreuses requêtes).
- Un refreshToken distinct limite l'exposition : il n'est utilisé que pour un endpoint spécifique (/refresh), stocké de manière plus sécurisée (cookie HttpOnly), et conçu pour un usage restreint.

b. Sécurité renforcée

- **Durée de vie courte de l'accessToken :**

- Ton accessToken a une durée de vie de 1 jour (en production). Une durée courte réduit la fenêtre d'opportunité pour un attaquant qui volerait l'accessToken (par exemple, via une attaque XSS sur localStorage).
- Si l'accessToken était utilisé pour se rafraîchir, tu devrais soit :
 - **Prolonger sa durée de vie** (par exemple, 7 jours), ce qui augmenterait les risques en cas de vol, car un token volé serait utilisable plus longtemps pour accéder aux ressources protégées **et** générer de nouveaux tokens.
 - **Garder une durée courte** (1 jour), mais alors l'accessToken expirerait avant de pouvoir être utilisé pour se rafraîchir, rendant le mécanisme inutile.
- **Sans refreshToken, en utilisant l'accessToken :**
 - Si l'accessToken expire (après 1 jour), et que tu l'envoies à /refresh pour en obtenir un nouveau, que se passe-t-il si l'accessToken est déjà expiré ? L'endpoint /refresh ne pourrait pas le valider, et l'utilisateur devrait se reconnecter.
 - Pour éviter cela, tu devrais donner à l'accessToken une durée de vie longue (par exemple, 7 jours), mais cela compromettrait la sécurité, car un token volé serait utilisable pendant 7 jours pour tout : accès aux ressources **et** génération de nouveaux tokens.

d. Conformité aux standards

- L'utilisation d'un refreshToken distinct est une pratique standard dans les protocoles comme **OAuth 2.0** et **OpenID Connect**, qui sont largement adoptés pour l'authentification et l'autorisation.
- Ces standards définissent :
 - Un accessToken pour accéder aux ressources protégées, avec une durée de vie courte.
 - Un refreshToken pour renouveler l'accessToken, avec une durée de vie plus longue et des mesures de sécurité renforcées (comme le stockage dans un cookie HttpOnly ou une base de données).
- ➔ Ton implémentation suit ce modèle, ce qui la rend compatible avec les bonnes pratiques et facilite une future intégration avec des systèmes tiers.