리프레시 토큰 저장

Status	완료
날짜	@2025년 1월 6일 → 2025년 1월 6일
백-속성	DB 인증/인가
태그	백
작성자	 이주은

What I do?

- jwt 기반 토큰 인증 로그인을 구현하기 위해서 `리프레시 토큰` 을 어떻게 저장할 것인지 공부해봅시다 !

1. AccessToken과 RefreshToken의 발급 시나리오

https://inpa.tistory.com/entry/WEB-%F0%9F%93%9A-Access-Token-Refresh-Token-%EC%9B%90%EB%A6%AC-feat-JWT

1. 기본적으로 로그인 같은 과정을 하면 Access Token과 Refresh Token을 모두 발급한다.

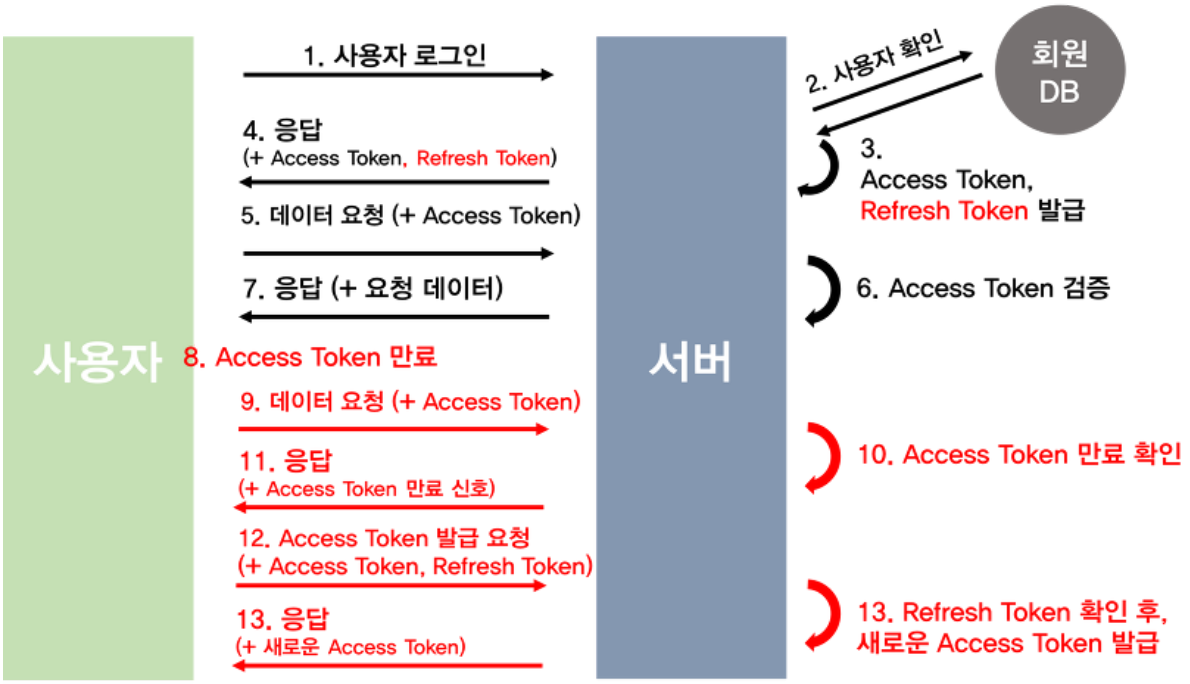
이때, Refresh Token만 서버측의 DB에 저장하며, Refresh Token과 Access Token을 쿠키 혹은 웹스토리지에 저장한다.

2. 사용자가 인증이 필요한 API에 접근하고자 하면, 가장 먼저 토큰을 검사한다.

이때, 토큰을 검사함과 동시에 각 경우에 대해서 토큰의 유효기간을 확인하여 재발급 여부를 결정한다.

- case1 : access token과 refresh token 모두가 만료된 경우 → 에러 발생 (재 로그인하여 둘다 새로 발급)
- case2 : access token은 만료됐지만, refresh token은 유효한 경우 → refresh token을 검증하여 access token 재발급
- case3 : access token은 유효하지만, refresh token은 만료된 경우 → access token을 검증하여 refresh token 재발급
- case4 : access token과 refresh token 모두가 유효한 경우 → 정상 처리

> Tip
>
> **[refresh token을 검증하여 access token 재발급]**
> 클라이언트(쿠키, 웹스토리지)에 저장되어있는 refresh token과 서버 DB에 저장되어있는 refresh token 일치성을 확인한 뒤 access token 재발급한다.
>
> **[access token을 검증하여 refresh token 재발급]**
> access token이 유효하다라는 것은 이미 인증된 것과 마찬가지니 바로 refresh token 재발급한다.



2. RefreshToken 저장소 비교

▼ 1. 데이터베이스(DB)에 저장

장점

1. **내구성 (Durability):**
   - DB는 디스크 기반 스토리지로 데이터를 영구적으로 저장합니다.
   - 시스템 재시작이나 장애 상황에서도 데이터를 안전하게 복구 가능.
2. **통합 관리 용이성:**
   - 기존 사용자 정보와 리프레시 토큰을 동일 DB에서 관리 가능.
   - 관계형 데이터베이스를 사용하면 사용자 ID와 토큰을 매핑해 쉽게 관리 가능.
3. **복잡한 쿼리 지원:**

단점

1. **속도:**
   - 디스크 기반 스토리지이므로 Redis에 비해 접근 속도가 느림.
   - 고빈도의 읽기/쓰기 요청에서 성능 저하 가능.
2. **추가 오버헤드:**
   - 리프레시 토큰 관련 테이블을 관리해야 하며, 스키마 설계와 인덱싱 추가 필요.
3. **복잡한 환경에선 병목 가능:**
   - 인증 요청이 많아질수록 DB의 I/O 병목 가능성이 증가.

- 만료된 토큰 삭제, 특정 사용자 토큰 조회, 통계 분석 등 다양한 쿼리 작업이 가능.

4. **규모 확장성 (Scalability):**
    - 수평 확장(Sharding)이나 클러스터링 기능을 통해 많은 데이터를 처리 가능.

▼ 2. Redis에 저장

장점

1. **속도 (Performance):**
    - Redis는 메모리 기반 데이터 스토리지로 매우 빠른 읽기/쓰기 속도를 제공합니다.
    - 짧은 시간 내에 다수의 요청을 처리하는 데 적합.
2. **TTL(Time-To-Live) 지원:**
    - Redis는 각 키에 TTL을 설정할 수 있어 리프레시 토큰의 자동 만료 관리가 가능.
    - 추가로 만료된 데이터를 정리하는 작업이 필요 없음.
3. **경량화된 관리:**
    - Redis의 Key-Value 저장 방식은 간단하며, 스키마 설계가 필요 없음.
    - 단일 명령으로 키 삭제, 갱신 등의 작업이 가능.
4. **확장성 (Scalability):**
    - 클러스터링 기능으로 고성능과 대용량 데이터 처리 지원.

단점

1. **내구성 부족 (Persistence):**
    - 기본적으로 메모리 기반이라 시스템 재시작 시 데이터 손실 위험이 있음.
    - AOF(Append Only File)나 RDB(Snapshot) 설정을 통해 내구성을 보완할 수 있지만 성능이 약간 감소.
2. **운영 비용:**
    - 메모리 기반 스토리지는 저장 용량 대비 비용이 더 높음.
    - 데이터 크기가 큰 경우 확장이 비용적으로 부담될 수 있음.
3. **복잡성 증가:**
    - 기존 DB 외에 별도의 Redis 인프라를 구축, 운영해야 함.
    - Redis 관리 경험이 필요.

비교 요약

특징	**데이터베이스(DB)**	**Redis**
속도	느림 (디스크 I/O 의존)	매우 빠름 (메모리 기반)
내구성	강력한 내구성 (데이터 영구 저장)	기본적으로 낮음 (옵션으로 보완 가능)
TTL 관리	직접 구현 필요	기본 제공 (자동 만료 처리 가능)
운영 비용	상대적으로 낮음	메모리 비용 높음
관리 용이성	사용자 데이터와 통합 가능	별도 인프라 및 관리 필요
확장성	수평 확장 가능	클러스터링으로 확장 가능

추천 시나리오

1. **데이터베이스(DB) 추천:**
   - 사용자 정보와 리프레시 토큰을 통합 관리해야 하는 경우.
   - 토큰 요청 빈도가 낮고 내구성이 중요한 시스템.
   - 추가 인프라 구축이 부담스러운 경우.
2. **Redis 추천:**
   - 토큰 요청 빈도가 매우 높고 실시간 처리가 중요한 경우.
   - 토큰의 자동 만료와 빠른 성능이 중요한 시스템.
   - Redis를 이미 사용하고 있는 환경.

3. 로그아웃까지 고려해서 저장소 추천해봐.

▼ 1. 데이터베이스(DB)

장점

1. **명시적 무효화 관리 용이:**
   - 리프레시 토큰을 DB에 저장하면 특정 사용자의 리프레시 토큰을 삭제하거나 상태를 업데이트하여 쉽게 무효화할 수 있습니다.
   - 예: `UPDATE tokens SET revoked = TRUE WHERE user_id = ?`.

단점

1. **성능 저하:**
   - 높은 빈도의 토큰 검증 및 무효화 작업이 발생하면 DB 부하가 커질 수 있습니다.
2. **토큰 만료와 삭제 관리 필요:**
   - 주기적으로 만료된 토큰을 삭제하거나 관리해야 하므로 추가 작

2. **세션 추적:**
    - DB에서 토큰을 사용자 ID 또는 세션 ID와 연결하여 다중 기기에서의 세션 관리가 가능합니다.
    - 특정 기기에서만 로그아웃하거나 모든 기기에서 로그아웃하는 로직 구현이 용이.
3. **내구성:**
    - 토큰이 영구적으로 저장되므로, 서버 재시작이나 장애 상황에서도 로그아웃 상태를 유지할 수 있습니다.

업 필요.

▼ 2. Redis

장점

1. **속도:**
    - 메모리 기반으로 읽기/쓰기 속도가 매우 빠르기 때문에 토큰 검증 및 삭제 작업이 효율적.
2. **TTL(Time-To-Live) 지원:**
    - 각 리프레시 토큰에 TTL을 설정하여 자동 만료가 가능하므로 만료된 토큰 관리를 자동화할 수 있습니다.
3. **즉각적인 로그아웃 처리:**
    - 특정 사용자의 토큰 키를 Redis에서 삭제하면 즉시 로그아웃 효과를 낼 수 있습니다.
    - 예: `DEL user:refresh_token`.

단점

1. **내구성 부족:**
    - 기본 설정에서는 시스템 재시작 시 저장된 데이터가 삭제될 수 있습니다. 이를 방지하려면 Redis의 AOF(Append Only File) 설정이 필요합니다.
2. **추가 인프라 필요:**
    - Redis를 운영하고 관리할 별도 환경이 필요하며, 비용이 증가할 수 있습니다.

▼ 3. 브라우저 기반 저장소 (로컬 스토리지, HTTP Only 쿠키)

장점

1. **단순 구현:**
   - 클라이언트 측에만 토큰을 저장하면 서버 측에 추가 저장소를 설정할 필요가 없습니다.
2. **자체 만료 가능:**
   - 클라이언트에서 토큰 만료 시간을 확인하거나 삭제하여 로그아웃 상태를 유지할 수 있습니다.

단점

1. **명시적 무효화 불가능:**
   - 클라이언트에 저장된 토큰은 서버에서 제어할 수 없기 때문에, 클라이언트가 토큰을 삭제하지 않으면 로그아웃 상태를 강제할 수 없습니다.
   - 서버가 토큰을 무효화하려면 별도의 블랙리스트 로직이 필요합니다.
2. **보안 이슈:**
   - 로컬 스토리지에 저장하면 XSS 공격에 취약하며, HTTP Only 쿠키를 사용하면 CSRF 방어가 필요합니다.

추천 저장소 (로그아웃 관점)

1. **Redis (추천)**
   - 빠른 속도가 요구되고, 사용자 로그아웃 시 즉각적으로 토큰을 무효화해야 하는 경우 적합합니다.
   - 예:
     - 리프레시 토큰을 Redis에 저장하고, 로그아웃 시 `DEL user:refresh_token` 명령으로 삭제.
     - TTL을 활용하여 만료 관리 자동화.
2. **데이터베이스**
   - 로그아웃 시점 기록, 세션 추적 등과 같은 세부적인 관리가 필요하다면 데이터베이스를 사용하는 것이 적합합니다.
   - 예:
     - 리프레시 토큰에 `revoked` 플래그를 추가하고, 로그아웃 시 해당 값을 `TRUE`로 업데이트.

    - 만료된 토큰 정리를 위한 백그라운드 작업 필요.

3. **브라우저 기반 저장소**
    - 간단한 애플리케이션에서 클라이언트가 직접 로그아웃을 관리하는 경우에만 고려.
    - 보안 및 명시적 무효화가 어렵기 때문에 권장되지 않습니다.

4. Redis + DB 혼합 사용

로그아웃 시 보안과 사용자 경험을 보장하기 위해 Redis와 데이터베이스를 **혼합 사용** 하는 방식도 고려할 수 있습니다.

- **Redis**에서 빠른 검증 및 삭제 처리.
- **DB**에 백업 및 장기 저장.

혼합 사용 방식의 구조

1. Redis

- **역할:**
    - 리프레시 토큰의 **단기 저장소** 역할.
    - 빠른 토큰 검증 및 삭제(로그아웃) 처리.
    - TTL(Time-To-Live)을 설정하여 자동 만료 관리.
- **저장 데이터:**
    - 사용자 ID와 리프레시 토큰의 매핑 정보.
    - 키 형식 예: `user:{userId}:refresh_token`.

2. 데이터베이스

- **역할:**
    - 리프레시 토큰의 **장기 저장소** 및 백업.
    - 사용자 로그아웃 히스토리 관리.
    - 시스템 재시작이나 Redis 장애 시 복구를 위한 영구 저장소.
- **저장 데이터:**
    - 사용자 ID, 리프레시 토큰, 발급 시간, 만료 시간, `revoked` 플래그(무효화 여부).

▼ 작동 원리 🔑⭐

1. **로그인 시:**
    - 리프레시 토큰을 생성한 후 Redis와 DB에 저장.
    - Redis에는 사용자 ID와 토큰을 매핑하여 저장하며, TTL을 설정.
    - DB에는 영구 저장소로 데이터를 기록.

```
Redis 저장: key -> user:{userId}:refresh_token, value
-> {refreshToken}, TTL = 7 days
DB 저장: userId, refreshToken, issuedAt, expiresAt
```

2. **리프레시 토큰 검증 시:**
    - 먼저 Redis에서 토큰을 조회하여 유효성을 검증.
    - Redis에 존재하지 않는 경우 DB를 조회해 복구하거나 무효화 여부 확인.
    - DB에서 복구된 토큰은 Redis로 다시 캐싱.

```
1. Redis 조회 -> 유효하면 OK
2. Redis에 없으면 DB 조회 -> 유효하면 Redis에 재등록
3. DB에서 revoked = TRUE이면 인증 거부
```

3. **로그아웃 시:**
    - Redis에서 해당 사용자의 리프레시 토큰을 삭제.
    - DB에 해당 리프레시 토큰의 `revoked` 플래그를 TRUE로 업데이트.
    - TTL을 이용하지 않고 DB에서 영구적으로 무효화 상태 기록.

```
Redis 삭제: DEL user:{userId}:refresh_token
DB 업데이트: UPDATE tokens SET revoked = TRUE WHERE use
rId = ?
```

4. **시스템 재시작 또는 장애 시:**
    - Redis에 저장된 데이터는 사라질 수 있으므로, 재시작 시 DB에서 유효한 리프레시 토큰을 다시 로드해 Redis에 캐싱.

```
DB 조회: SELECT * FROM tokens WHERE revoked = FALSE AN
D expiresAt > NOW()
```

```
Redis 복구: 각 토큰을 Redis에 다시 등록
```

▼ 이점

1. **빠른 응답 속도 (Redis):**
   - Redis에서 리프레시 토큰을 바로 검증하므로 성능이 매우 빠름.
   - 사용자 경험 향상, 특히 높은 트래픽을 처리하는 시스템에서 유리.
2. **내구성 및 복구 가능성 (DB):**
   - Redis 장애 또는 데이터 손실 시 DB에서 토큰 데이터를 복구 가능.
   - 시스템 안정성을 보장.
3. **보안 강화:**
   - 로그아웃 시 Redis에서 즉시 삭제 처리로 빠르게 무효화.
   - DB에 `revoked` 상태를 기록하여 로그아웃 후에도 동일 토큰 재사용 방지.
4. **TTL 기반 자동 관리:**
   - Redis에서 TTL을 설정하여 만료된 토큰을 자동으로 정리.
   - DB에서는 백그라운드 작업으로 오래된 데이터를 삭제하여 효율적 관리.
5. **다중 기기 로그아웃 지원:**
   - 사용자 ID를 기준으로 Redis와 DB를 조회 및 삭제하여 모든 기기에서 로그아웃 구현 가능.

▼ 구현 예시 (Spring Boot + Redis + MySQL)

Redis 저장 및 검증

```
// Redis에 토큰 저장
redisTemplate.opsForValue().set("user:" + userId + ":ref
resh_token", refreshToken, 7, TimeUnit.DAYS);

// Redis에서 토큰 검증
String storedToken = redisTemplate.opsForValue().get("us
er:" + userId + ":refresh_token");
if (storedToken == null || !storedToken.equals(providedT
oken)) {
```

```
    throw new TokenInvalidException("Invalid or expired
token");
}
```

DB 백업 및 상태 관리

```
// DB에 토큰 저장
TokenEntity token = new TokenEntity(userId, refreshToke
n, issuedAt, expiresAt, false);
tokenRepository.save(token);

// 로그아웃 시 DB 상태 업데이트
tokenRepository.updateRevokedStatus(userId, true);
```

시스템 재시작 시 Redis 복구

```
// DB에서 유효한 토큰 조회 후 Redis에 복구
List<TokenEntity> validTokens = tokenRepository.findAllV
alidTokens();
for (TokenEntity token : validTokens) {
    redisTemplate.opsForValue().set("user:" + token.getU
serId() + ":refresh_token", token.getToken(), 7, TimeUni
t.DAYS);
}
```

Q. 레디스는 메모리 데이터를 disk에 저장하는 기능이 있음.... rdb에 저장하는 이유는?

⇒ disk에 저장하는 기능은 복구시나리오에서 사용 가능함.

⇒ 레디스가 꽉 차면,

메모리를 확보하기 위해 일부 데이터를 제거하는 "eviction" 프로세스가 발생

합니다. 이때, 오래된 데이터부터 지우고 새 데이터가 기록됩니다.

레디스에서 메모리가 꽉 차면 어떻게 처리하는지는 maxmemory-policy 값에 따라 달라집니다. 기본값은 noeviction으로, 메모리가 가득 차면 더 이상 새로운 입력을 받지 않습니다.

What I learned?

Reference

https://velog.io/@ch4570/OAuth-2.0-JWT-Spring-Security%EB%A1%9C-%ED%9A%8C%EC%9B%90-%EA%B8%B0%EB%8A%A5-%EA%B0%9C%EB%B0%9C%ED%95%98%EA%B8%B0-Refresh-Token-%EC%9E%AC%EB%B0%9C%EA%B8%89

https://velog.io/@ch4570/OAuth-2.0-JWT-Spring-Security%EB%A1%9C-%ED%9A%8C%EC%9B%90-%EA%B8%B0%EB%8A%A5-%EA%B0%9C%EB%B0%9C%ED%95%98%EA%B8%B0-JWT-%EA%B0%9C%EB%85%90%EA%B3%BC-Security-%EA%B8%B0%EB%B3%B8-%EC%84%A4%EC%A0%95

---

https://inpa.tistory.com/entry/REDIS-%F0%9F%93%9A-%EB%8D%B0%EC%9D%B4%ED%84%B0-%EC%98%81%EA%B5%AC-%EC%A0%80%EC%9E%A5%ED%95%98%EB%8A%94-%EB%B0%A9%EB%B2%95-%EB%8D%B0%EC%9D%B4%ED%84%B0%EC%9D%98-%EC%98%81%EC%86%8D%EC%84%B1

https://zangzangs.tistory.com/72