JWT

JSON Web Token

JSON 포맷을 이용하여 **사용자에 대한 속성을 저장**하는 Claim 기반의 **Web Token**

시작하기에 앞서,

WHY. 토큰기반 인증 ❓

서버 기반 인증 시스템 VS 토큰 기반 인증 시스템

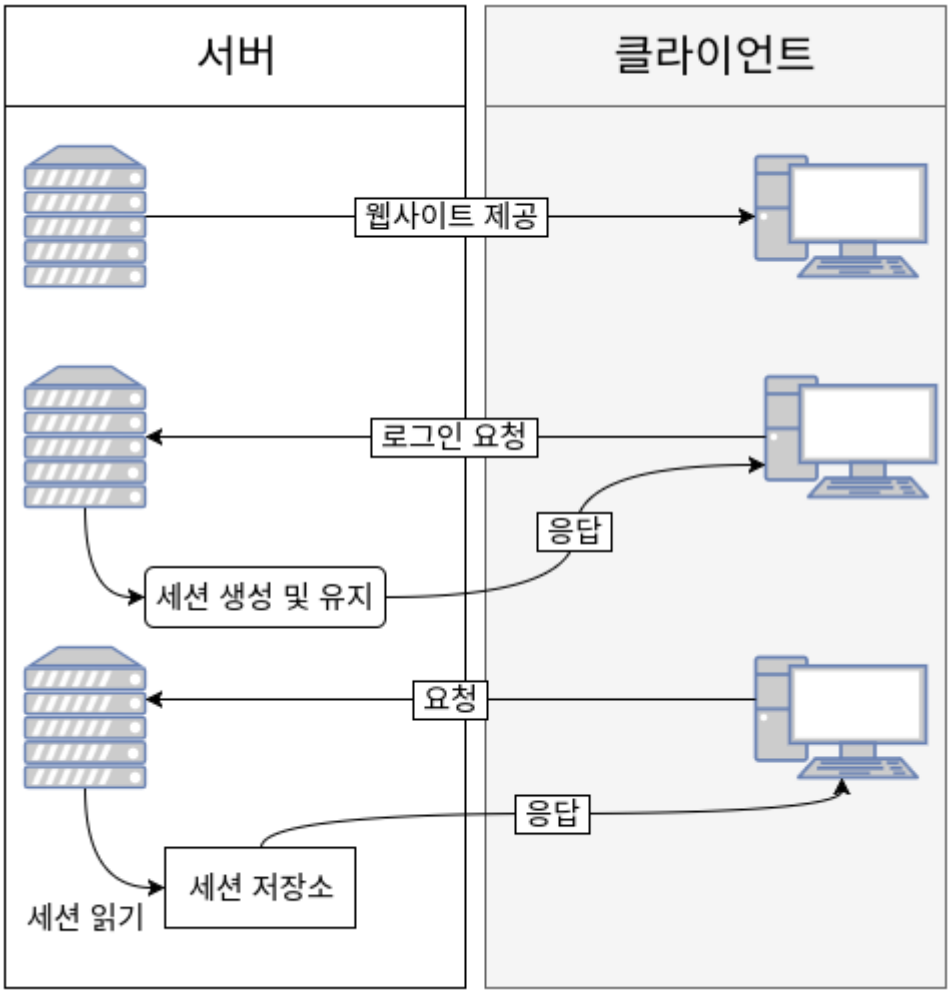
1. 서버 기반의 인증 시스템

▼ **서버** 측에서 사용자들의 정보를 기억

▼ 사용자들의 정보 기억 == 세션 유지

⇒ 메모리, 디스크 또는 데이터베이스 등으로 관리

▼ **Stateful 서버** : 클라이언트와의 통신상태를 계속 추적하고 이를 자신의 서비스 제공에 이용하는 서버



BUT. 소규모 시스템이 아닌 대규모시스템에서 **서버를 확장하기 어려움**

[서버 기반의 인증 시스템 문제점]

◆ **세션**

사용자 증가 시, 서버의 RAM 과부화

◆ **확장성**

사용자 증가 시, 많은 트래픽을 처리하기 위해 여러 프로세스를 돌리거나 컴퓨터를 추가하는 등 서버 확장이 필요
⇒ 세션을 분산시키는 시스템을 설계하는 과정이 매우 어렵고 복잡

◆ **CORS** (Cross-Origin Resurce Sharing)

세션을 관리할 때 자주 사용되는 쿠키는 단일 도메인 및 서브 도메인에서만 작동하도록 설계

⇒ 쿠키를 여러 도메인에서 관리하는 것이 번거로움

2. 토큰 기반의 인증 시스템

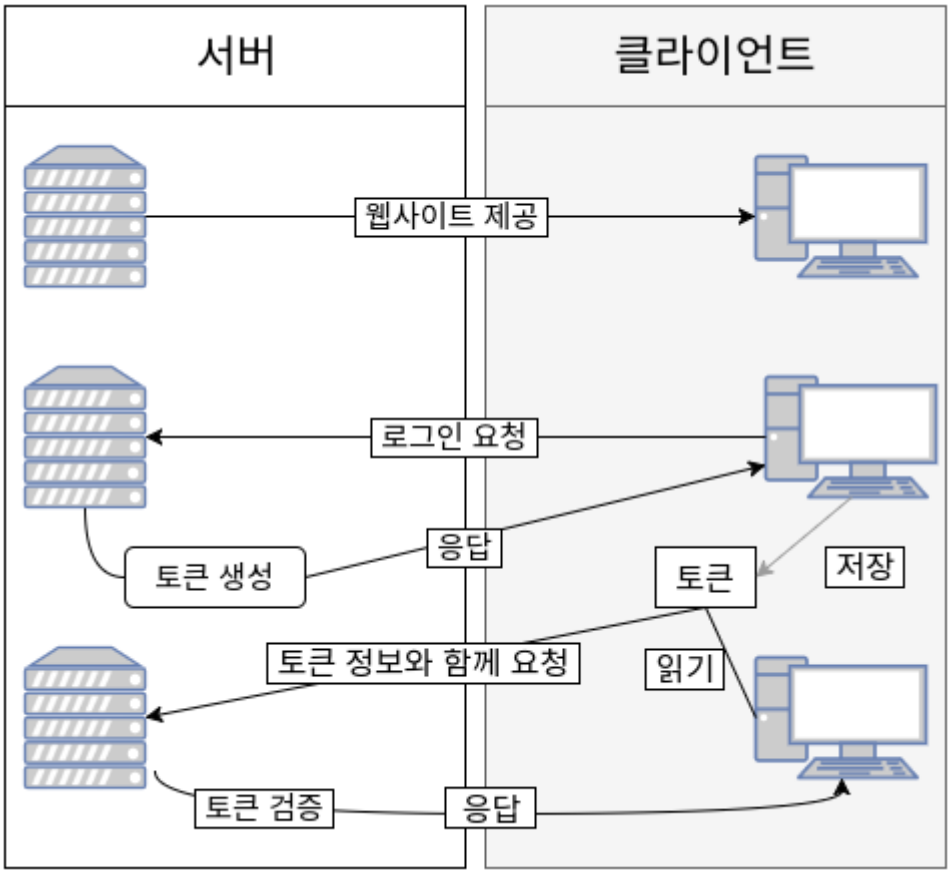
▼ 인증받은 **사용자**들에게 토큰을 발급

▼ 서버에 요청할 때 헤더에 토큰을 전달하여 유효성 검사

▼ 사용자의 인증 정보를 서버나 세션에 유지하지 않음

▼ **Stateless** 구조 : 사용자가 로그인 여부를 신경쓰지않고 시스템 확장 가능

🔷 토큰 기반의 인증 시스템 작동 과정



1. 로그인 정보 입력 및 요청
2. 서버 측에서 해당 정보 검증

3. 인증 성공 시, 서버 측에서 사용자에게 **Signed 토큰** 발급
   (서버에서 정상적으로 발급된 토큰임을 증명하는 Signature를 가지고 있다는 것)
4. 클라이언트 측에서 전달받은 토큰을 저장해두고, **서버에 요청 할 때마다 해당 토큰을 서버에 함께 전달** ⇒ Http 요청 헤더에 토큰을 포함시킴
5. 서버는 토큰을 검증하고, 요청에 응답

[토큰 기반 인증 시스템의 장점]

1. **무상태성(Stateless) & 확장성(Scalability)**

   클라이언트와 서버의 연결고리가 없어 확장하기에 매우 적합 ⇒ 여러 서버에서 요청 받을 수 있음

2. **보안성**

   쿠키 사용에 의한 취약점이 사라짐

3. **확장성(Extensibility)**

   로그인 정보가 사용되는 분야의 확장을 의미 ⇒ 토큰에 선택적인 권한만 부여하여 발급 가능
   (EX) OAuth의 경우, Facebook, Google 등과 같은 소셜 계정을 이용하여 다른 웹서비스에서도 로그인 가능

4. **여러 플랫폼 및 도메인**

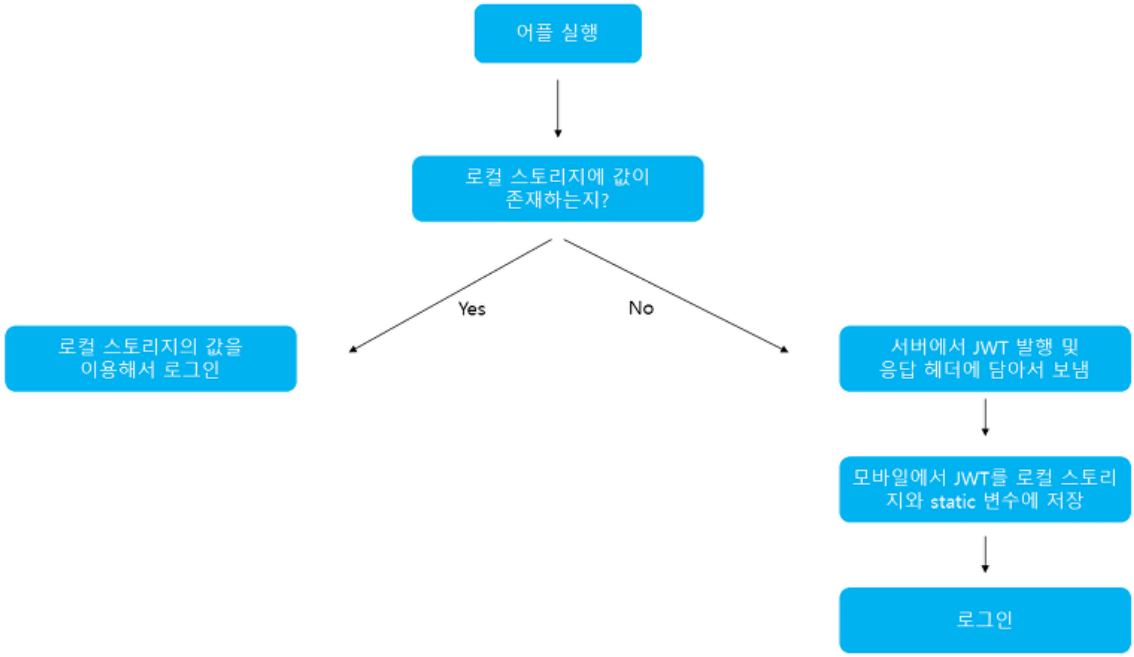
   CORS 문제 해결 ⇒ 토큰 사용 시, 어떤 디바이스 및 도메인에서도 토큰의 유효성 검사를 진행한 후에 요청을 처리할 수 있음
   (EX) assests 파일(Image, html, css, js 등)은 모두 CDN에서 제공하고, 서버 측에서는 API만 다루도록 설계할 수 있음

다시 돌아와서,

JWT (JSON Web Token)

▼ 토큰 자체를 정보로 사용하는 **Self-Contained** 방식

🔷 JWT 로직



1. 로그인 정보 입력 후, 서버가 Access 토큰 부여
2. API 요청 시, Access 토큰을 포함해서 요청
3. 서버는 Access 토큰을 해독 및 검증하여 해당 API 기능 수행
4. 토큰의 기간 만료 시, Access 토큰을 지우고 재로그인하게 함

▼ JWT를 **static 변수**와 **로컬 스토리지**에 저장

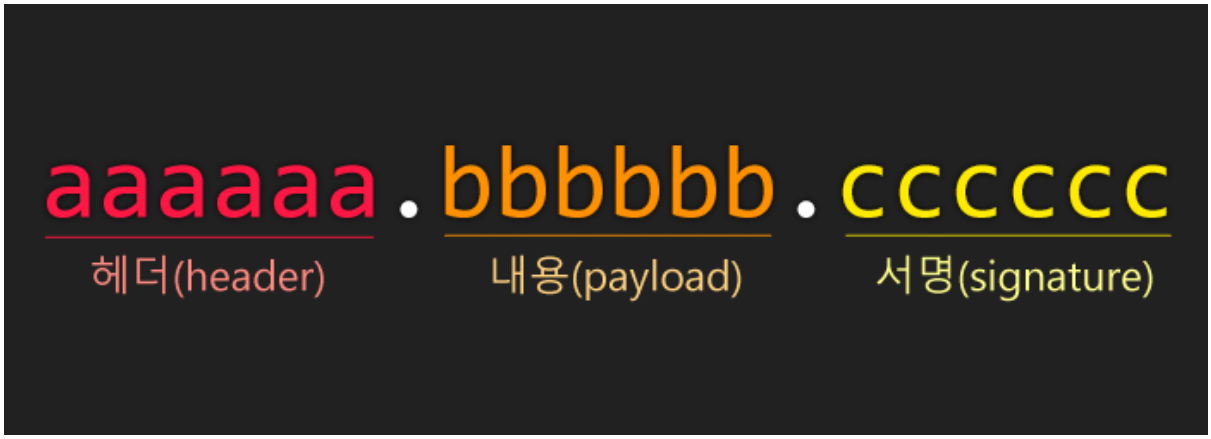
▼ WHY. **static 변수에 저장하는 이유** ❓

HTTP 통신을 할 때마다 **JWT를 HTTP 헤더에 담아서 보내야 함**
⇒ 로컬 스토리지에서 계속 불러오면 **오버헤드** 발생

▼ 로그아웃 시, 로컬 스토리지에 저장된 JWT 데이터 제거

▼ 실제 서비스의 경우, 사용했던 토큰을 blacklist라는 DB 테이블에 넣어 해당 토큰의 접근을 막는 작업을 해줘야 함

🔷 JWT 구조



1. **Header**
   - **typ**과 **alg** 두가지 정도로 구성
     `typ` : 토큰의 타입을 지정
     (EX) JWT
     `alg` : 알고리즘 방식을 지정. 서명(Signature) 및 토큰 검증에 사용
     (EX) HS256(SHA256) 또는 RSA
2. **Payload**
   - 토큰에서 사용할 정보의 조각들인 클레임(Claim) 저장
   - 서버에 보낼 데이터 저장 ⇒ 사용자의 ID 등의 정보, 유효기간 포함
   - 3가지의 클레임으로 나누어짐

     (1) **등록된 클레임**(Registerd Claim)
     - 토큰 정보를 표현하기 위해 이미 정해진 종류의 데이터
     - 선택적 작성 가능, 사용 권장

     ▼ key 종류

     ▼ `iss` : 토큰 발급자(issuer)

     ▼ `sub` : 토큰 제목(subject)

     ▼ `aud` : 토큰 대상자(audience)

     ▼ `exp` : 토큰 만료 시간(expiration), NumericDate 형식으로 되어 있어야 함 (EX) 1480849147370

     ▼ `nbf` : 토큰 활성 날짜(not before), 이 날이 지나기 전의 토큰은 활성화되지 않음

     ▼ `iat` : 토큰 발급 시간(issued at), 토큰 발급 이후의 경과 시간을 알 수 있음

▼ `jti` : JWT 토큰 식별자(JWT ID), 중복 방지를 위해 사용하며, 일회용 토큰(Access Token) 등에 사용

(2) **공개 클레임**(Public Claim)

- 사용자 정의 클레임
- 공개용 정보를 위해 사용
- 충돌 방지를 위해 URI 포맷 이용

(EX) { "https://mangkyu.tistory.com": true }

(3) **비공개 클레임**(Private Claim)

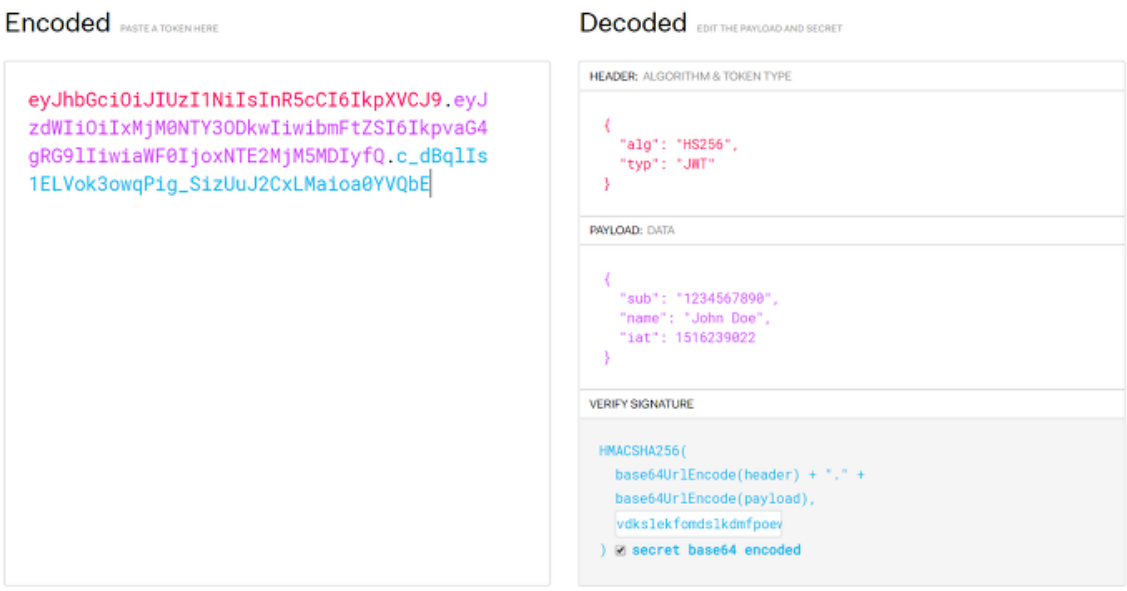
- 사용자 정의 클레임
- 서버와 클라이언트 사이에 임의로 지정한 정보를 저장

(EX) { "token_type": access }

- **Json**(Key/Value) 형태로 다수의 정보 저장 가능

3. **Signature**

- 토큰을 인코딩하거나 유효성 검증 할 때 사용하는 고유한 **암호화 코드**
- Header와 Payload의 값을 각각 **Base64로 인코딩**하여, 비밀키를 이용해 Header에서 정의한 **알고리즘으로 해싱**하고, 그 값을 **다시 Base64로 인코딩**하여 생성

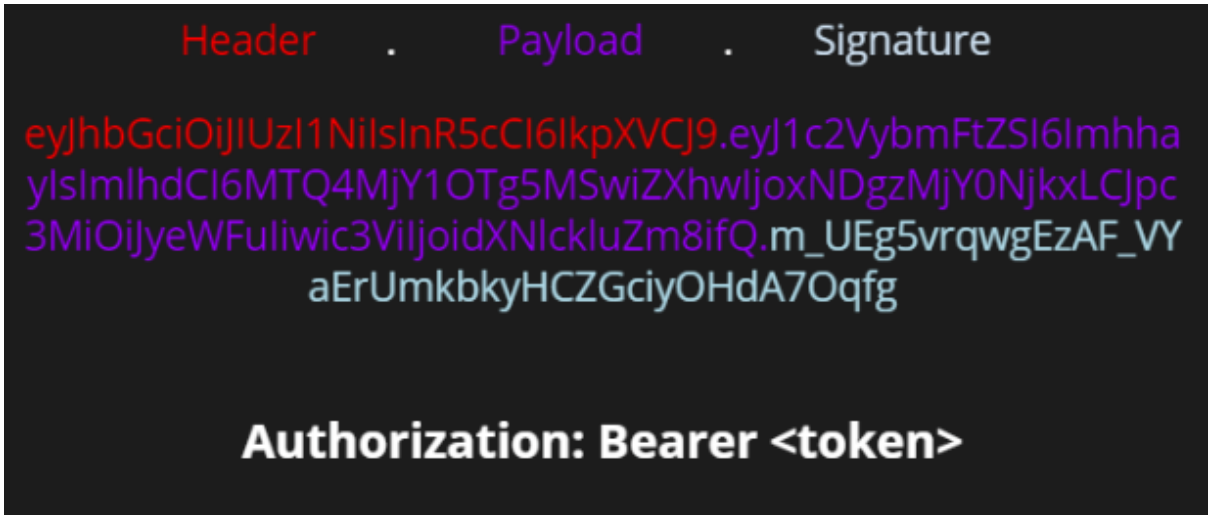


▼ **Base64**로 인코딩 되어 표현됨

▼ Base64 : 암호화된 문자열이 아님. 같은 문자열에 대해 항상 같은 인코딩 문자열 반환

▼ 각각의 부분을 이어주기 위해 **. 구분자**를 사용하여 구분

[JWT 토큰 EX]



▼ 생성된 토큰은 HTTP 통신 할 때 Authorization이라는 Key의 Value로 사용

▼ 일반적으로, Value에는 Bearer이 앞에 붙음

> ⭐ { "Authorization": "Bearer {생성된 토큰 값}", }

❗ 고려사항

- **Self-contained** : 토큰 자체에 정보를 담고 있어 위험성 존재
- **토큰 길이** : 3종류의 클레임을 저장하여 토큰의 길이가 길어 요청이 많아질수록 서버의 자원낭비 및 네크워크 부하 발생
- **Payload 인코딩** : 중간에 Payload를 탈취 후 디코딩하여 데이터 열람 가능 ⇒ JWE로 암호화 하거나 중요 데이터 추가하지 말기

- **Stateless** : 상태를 저장하지 않기 때문에 한번 만들어지면 제어 불가능. 토큰을 임의로 삭제하는 것이 불가능하여 토큰 만료 시간을 꼭 넣어야함

  ⭐ 기존의 Access Token의 유효기간을 짧게 하고, Refresh Token이라는 새로운 토큰을 발급하자

- **Store Token** : 토큰은 클라이언트 측에서 관리해야 하기 때문에 토큰을 저장해야 함

> 유효기간을 짧게 하면서 보안을 챙길 수 있는 방식❓

Refresh Token ❗

- Access Token : 수명이 존재. 기본적으로 30분 - 1시간

  ⇒ 수명이 끝나면 Access Token을 재발급 받아야 함 (재로그인)

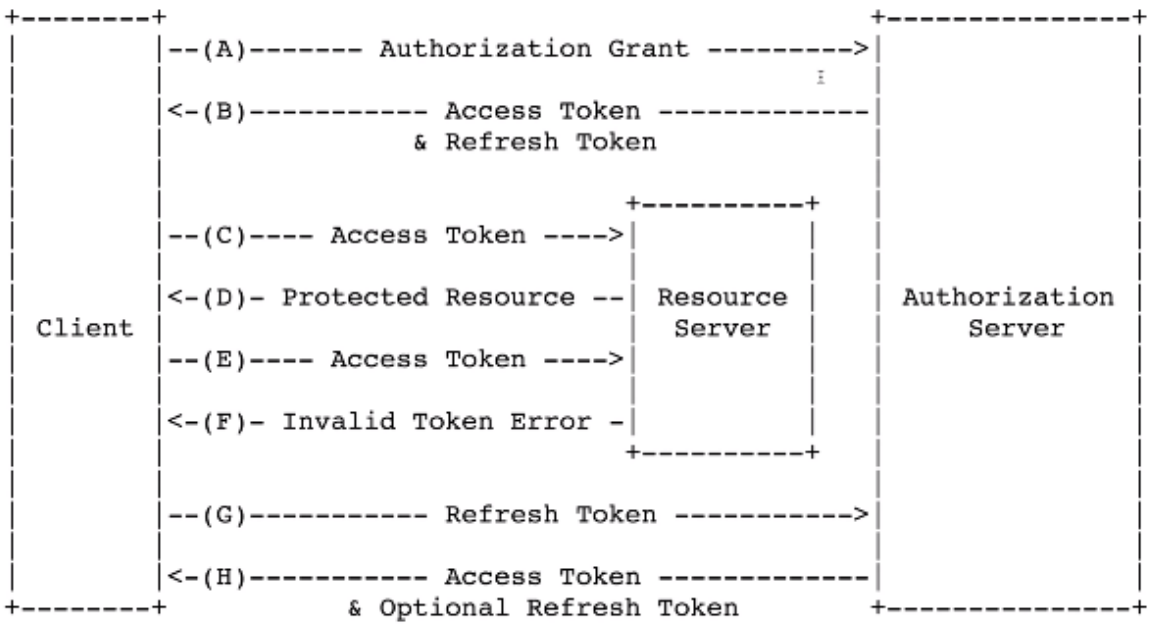
- 쉽게 재발급을 받자 ! **Refresh Token**

  ⇒ 기본 만료 시간 : 1일- 2주

  ⇒ 만약, Access Token이 만료되면 Refresh Token 인증 후 새로운 Access Token 발급

  ⇒ 서버에서 저장. 강제로 토큰 만료 가능

🔷 Refresh Token 처리



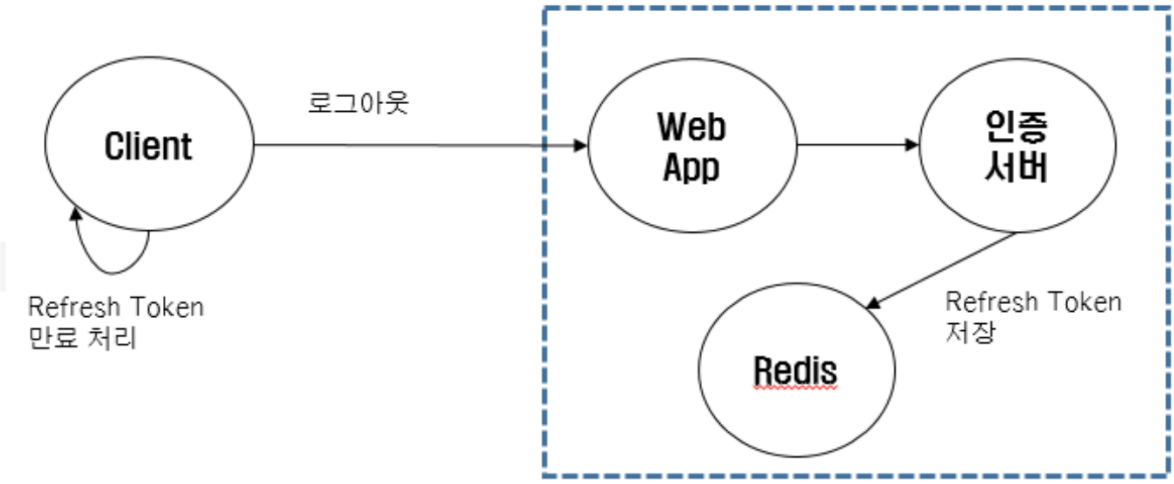
📌 만료 기간이 짧은 **Access Token**

📌 만료 기간이 긴 **Refresh Token**

1. 로그인 요청 및 인증 후, 성공시 **Access Token과 Refresh Token 발급**
2. Refresh Token은 **DB**에 저장. Access Token을 **헤더**에 저장하여 요청
3. Access Token 만료 시, Access Token 발급 요청
4. **Refresh Token 확인** 후, **새로운 Access Token** 발급

BUT. 여전히 토큰 탈취의 가능성 존재

> ⭐ 로그아웃 시, Refresh Token을 **blacklist**에 등록
> **blacklist**에 등록된 Refresh Token의 요청이 들어온다면? 탈취당한 토큰임을 인증!
> **Redis**를 black-list처럼 활용



1. 로그아웃 요청
2. 서버에서 Refresh Token을 받아서 Redis에 저장
3. 클라이언트 측에서 토큰 만료 시키기

[참고사이트]

https://victorydntmd.tistory.com/116

https://mangkyu.tistory.com/57

https://mangkyu.tistory.com/55

https://brownbears.tistory.com/440

https://yonghyunlee.gitlab.io/node/jwt/

https://syundev.tistory.com/28

http://tlog.tammolo.com/tags/jwt/