사용자 인증 처리 / 네트워크

HTTP

> W3 상에서 정보를 주고받을 수 있도록 정의된 프로토콜

- **Connectionless**(비연결성) : 요청에 따른 응답을 받고나면 연결을 끊음
- **Stateless**(무상태성) : 클라이언트-서버 관계에서 서버가 클라이언트의 상태를 보존하지 않음

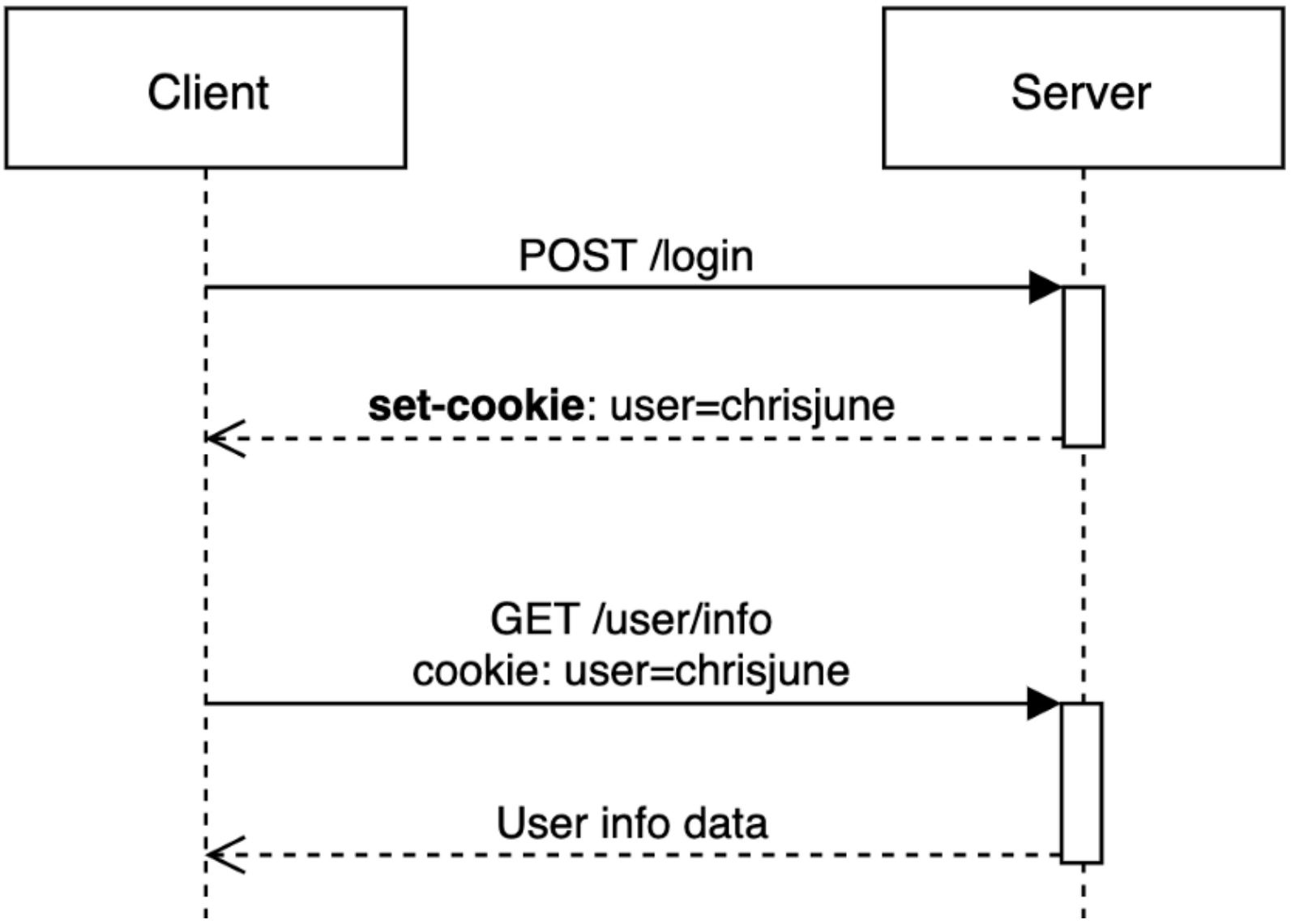
  → 그런데 상태가 없다면 로그인같이 상태가 필요한 경우는?

쿠키 & 세션

> HTTP의 무상태성 보완을 위해 등장한 기술들

쿠키

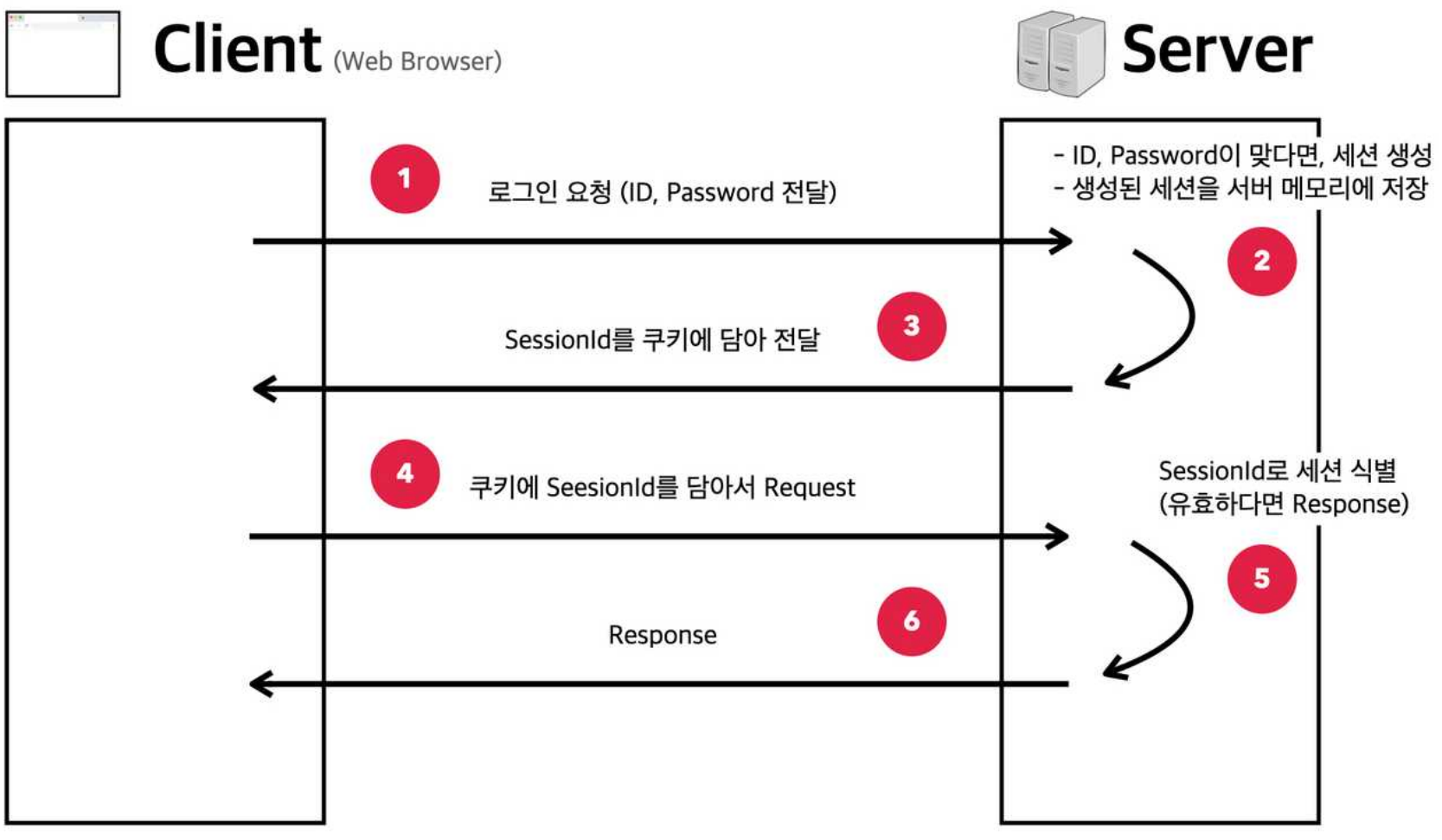
- **클라이언트에 저장**되는 `Key-Value` 쌍의 작은 데이터 파일
- 사용자의 정보를 브라우저에 저장하기 위해서 등장
  - 초창기에는 사용자의 정보를 서버가 보관하기에는 사양이 떨어졌음
  - 그래서 쿠키를 클라이언트가 가지고 있다가 요청할 때 보내도록 함
  - 최대 용량은 4KB, 개수는 20개(한 사이트 당)
- 동작 과정
  1. 클라이언트가 요청을 보내면 서버에서 쿠키를 생성
  2. 서버가 클라이언트로 보내는 응답의 헤더 중 `Set-Cookie` 라는 헤더에 키와 값을 보냄
  3. 응답을 받은 브라우저는 해당 쿠키를 저장
  4. 그 다음 요청부터 자동으로 쿠키를 헤더에 넣어 송신
  5. 브라우저가 종료되어도 쿠키 만료 기간이 남았다면 클라이언트에서 보관



- 문제점
  - 인증 정보를 브라우저에 저장 → 탈취 위험
  - 용량 및 개수 제한이 있음

세션

- 쿠키와는 반대로, 정보를 서버에 보관하는 방식
  - 쿠키의 트래픽 문제와 보안 문제를 해결
- 동작 방식



1. 클라이언트가 서버에 요청을 보냄
2. 서버에서는 HTTP Request를 통해 쿠키에서 Session ID를 확인

    a. 있다면 사용
    b. 없으면 Set-Cookie를 통해 발행한 Session ID를 보냄
3. 클라이언트는 HTTP Request 헤더에 Session ID를 포함해서 원하는 리소스를 요청
4. 서버는 세션 ID를 통해서 해당 세션을 찾아서 적절하게 응답

- 문제점
  - 인증 정보를 서버에 저장 → 사용자 수에 따른 서버 리소스 소모

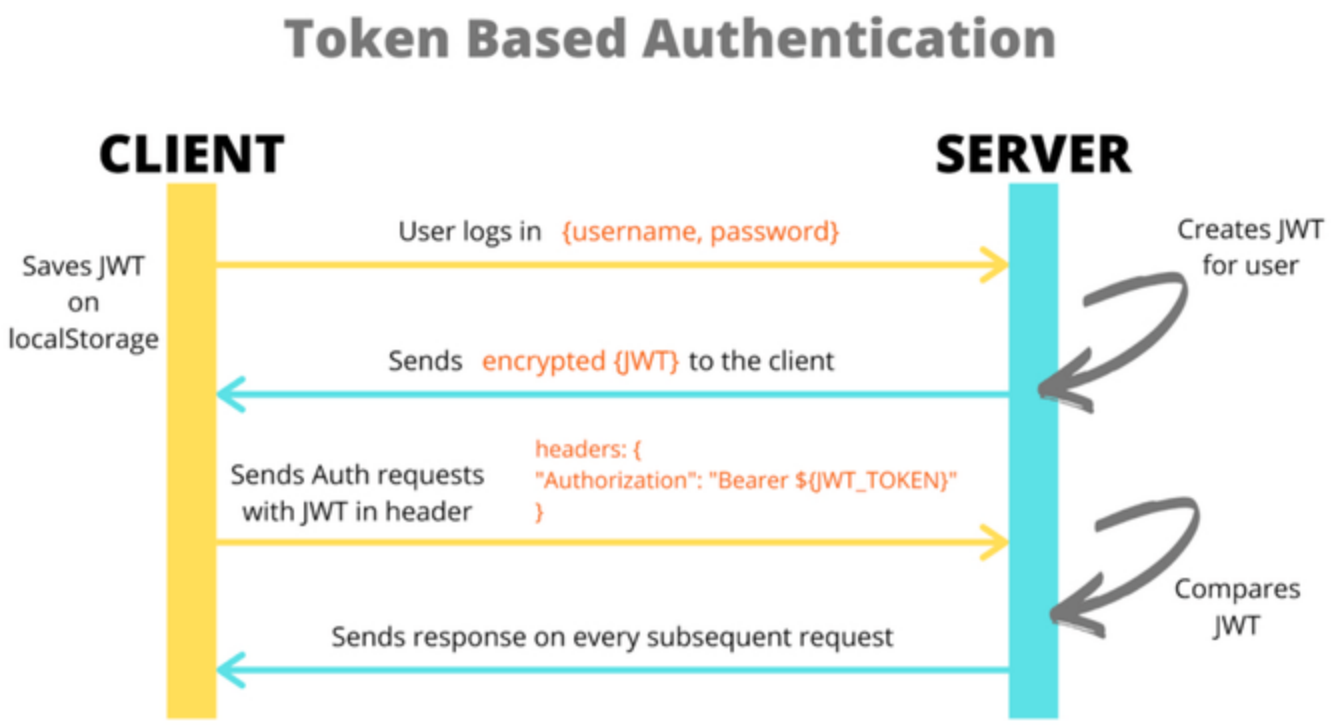
어떤 선택지가 있는지

> 💡 세션 기반 인증과 토큰 기반 인증의 장단점을 파악하고 어떤 로직을 사용할지 결정

세션 기반 인증

> 인증 정보를 서버의 세션 저장소에 저장

세션



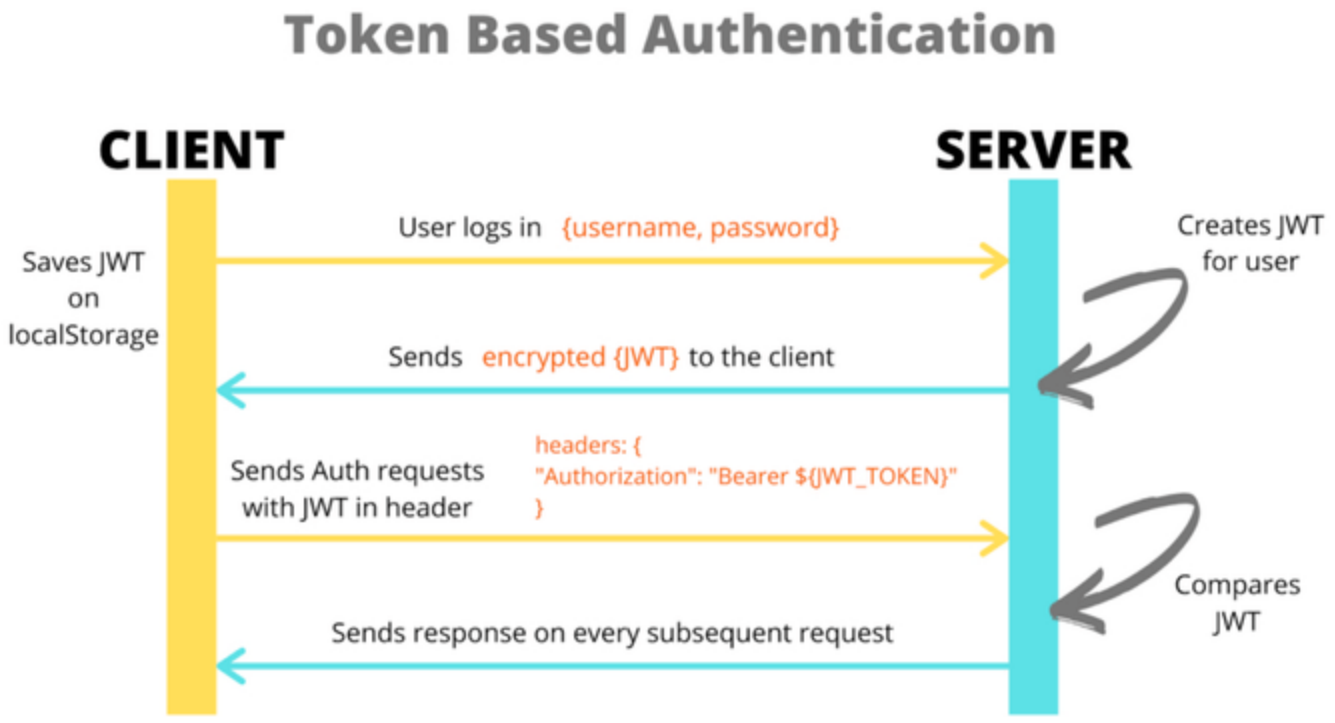
- 장점
  - 사용자의 인증 정보를 서버에서 안전하게 관리
  - 사용 트래픽이 훨씬 적음
- 단점
  - 서버에 상태를 유지해야 함
    - 서버 자원을 사용
    - 세션 불일치 이슈로 인해 수평 확장 시 추가 작업 필요

토근 기반 인증

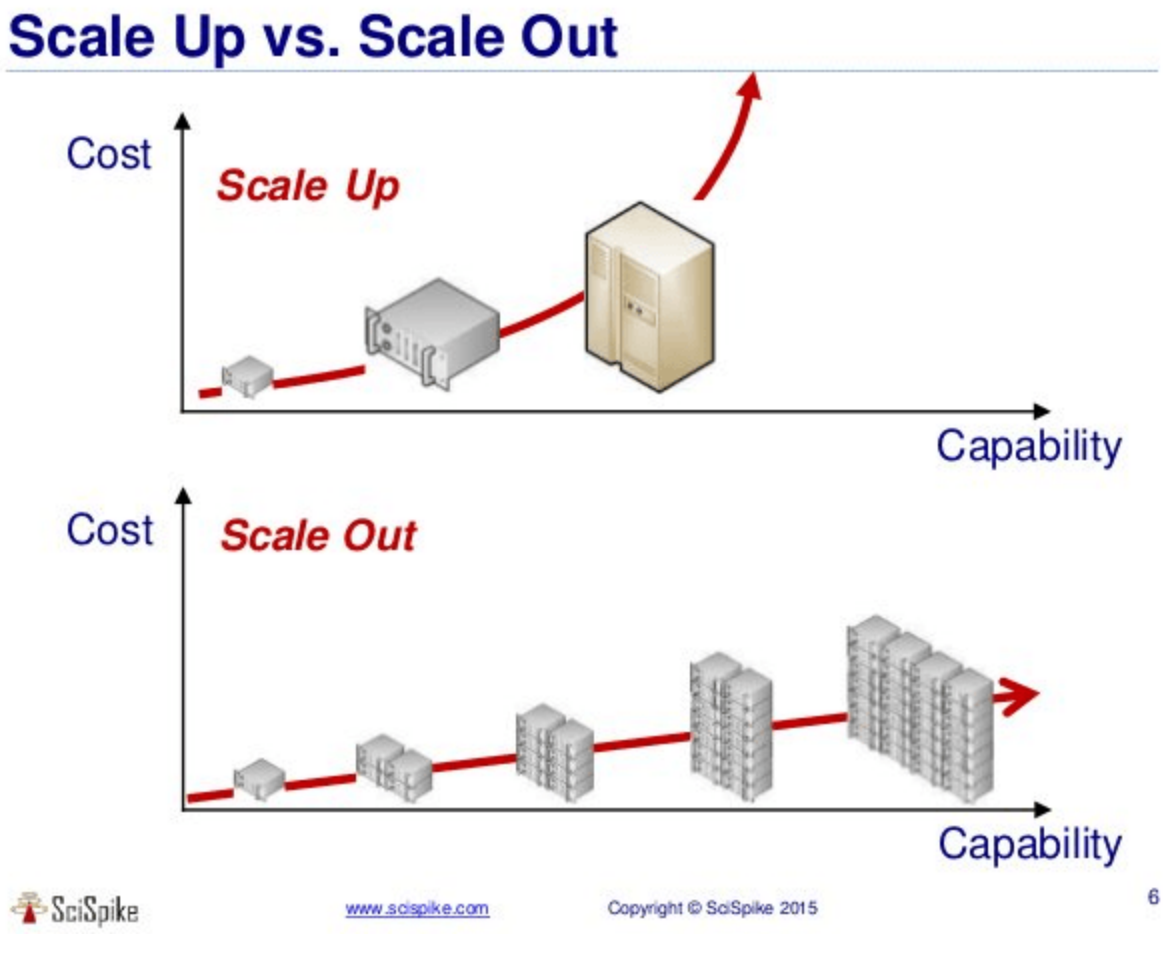
> 인증 정보를 토큰에 담아 클라이언트에 저장

JWT(Json Web Token)

- 웹에서 사용되는 JSON 형식의 토큰

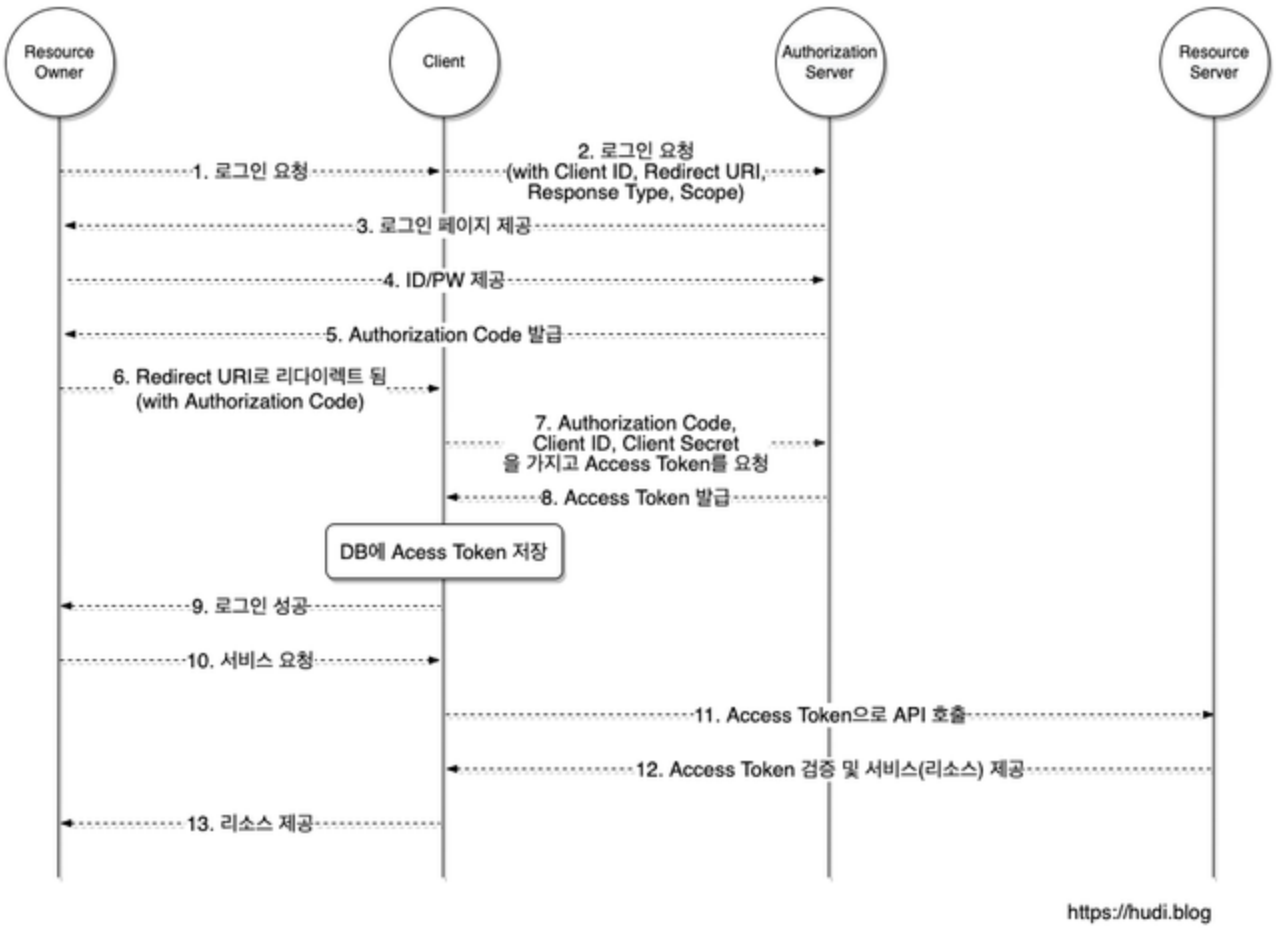


- 장점
  - HTTP의 Stateless(무상태성)을 유지할 수 있음
    - 서버의 확장 시, 수직 확장이 아닌 수평 확장으로도 유지 가능



    - 서버 부담 없음
- 단점
  - 토큰 탈취가 가능하기에 보안 문제 발생 가능
  - 이미 발급된 토큰의 관리(만료, 취소 등...)가 어려움

OAuth



- 장점
  - 복잡한 인증 절차를 외부 서비스에 위임 → 사용자 친화적
- 단점
  - 외부 서비스에 대한 신뢰가 필요
  - Store에 배포할 경우, 자체적인 회원관리 기능 필요

모익

Moi'c의 로그인 방식

- 추후 확장성과 서버의 리소스를 고려해 토큰 기반 인증
- RESTful API
- JWT와 OAuth를 함께 사용
  - OAuth
    - UX 향상
  - JWT
    - Store에 올리기 위해서는 자체적인 로그인이 필수
    - JWT의 문제점 해결

Moi'c에서 구현한 방법

> 사용자 인증을 위해 Access Token, 자동로그인과 보안을 위해 Refresh Token 사용

1. 사용자는 로그인을 하면 AccessToken은 Response로, RefreshToken은 Cookie로 받는다.
2. AccessToken은 항상 header에 추가 되어야 사용자 인증이 가능하다
3. AccessToken이 만료되었다면 Refresh요청을 보내며 만료된 AccessToken과 RefreshToken이 담긴 Cookie를 서버로 보낸다.
4. 서버에서는 검증 로직을 거친 후 AccessToken을 발급한다.

Moi'c의 AccessToken

- 암호화 키 : 팀원들의 이름을 Base64로 인코딩한 문자열

- 발급자 : moic
- 유효기간 : 30분
- Header
  - Key : Authorization
  - value : AccessToken

Moi'c의 RefreshToken

- 암호화 키 : 팀원들의 이름을 Base64로 인코딩한 문자열
- 발급자 : moic
- 유효기간 : 30일
- Cookie
  - Key : refreshToken
  - value : RefreshToken

검증 로직

AccessToken이 유효할 때

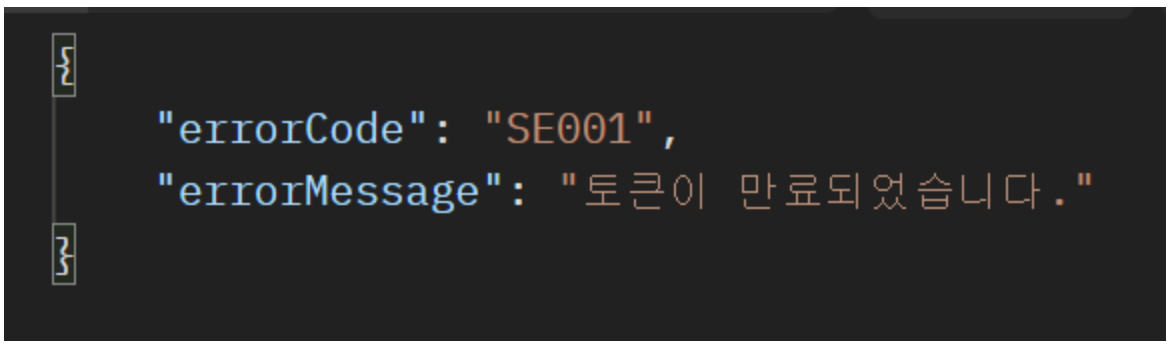
1. JwtAuthenticationFilter에서 Token 검증
   - 토큰의 유효성 검증
   - 토큰의 만료 여부 검증
2. Token에서 사용자 ID 추출
3. 권한 확인 및 서비스 사용

AccessToken이 만료되었을 때

1. Front-end에

   ```
   EXPIRED_TOKEN_ERROR(HttpStatus.INTERNAL_SERVER_ERROR, "SE001", "토큰이 만료되었습니다.")
   ```

   해당 에러 전송



2. Front-end에서 해당 에러를 받으면

   `/auth/refresh`
   로 refresh 요청
   - 이 때 검증을 위해 Body에 만료된 AccessToken과 RefreshToken이 담긴 Cookie를 전송
3. 서버는 먼저 받은 AccessToken을 검증
4. RefreshToken도 만료된 토큰인지 확인
5. 이후 RefreshToken이 서버에서 발급한 Token이 맞는지 Redis에 확인
6. Redis에 저장된 회원 ID와 만료된 AccessToken에서 추출한 회원 ID가 같은지 확인
7. 위 검증 과정이 모두 완료되면 AccessToken 재발급

```
{
    "message": "Refresh",
    "data": {
        "token": "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJwb29paWl1NDEwMCIsImlzcyI6Im1vaWMiLCJpYXQiOjE2OTU4OTY0MjIsImV4cCI6MTY5NTg5NjQ1Mn0.
            p6OPnlnO60cTtrwiRoD0iRMdtM6AjTVZdecK9ZLOAjeN7UI9abv_AcMpFVxYwtWDS8axTvcBoqxma8yjXTSJWQ"
    }
}
```


8. 만약 Refresh도 만료되어 아래와 같은 response를 받는다면 재로그인 요청


```
{
    "errorCode": "SE003",
    "errorMessage": "다시 로그인 해주세요."
}
```


이렇게 해서 좋은 점!!

- AccessToken이 탈취된 경우
  - 짧은 유효기간을 갖고 있어 공격자가 재빨리 사용하지 않으면 의미가 없다
- RefreshToken이 탈취된 경우
  - refresh 요청은 AccessToken이 있어야 가능하므로 소용이 없다