JWT를 위해 먼저 알아야 할 것 2

1. 서버기반 인증(세션로그인)

- 서버에서 클라이언트의 정보를 DB 또는 메모리에 저장해두는 것
- 세션: 사용자가 인증시 서버에 저장하는 정보를 일컫음
- MSA 구조로 바뀌면서 적합하지 않은 방식으로 여겨짐

> - **Micro Service Architecture (MSA)** :
> 서비스별 개별 배포가 가능하여 배포 시 전체 서비스 중단이 없다.
> - 독립 배포가 가능하여 개발자의 자율성이 늘어난다.
> - 요구사항을 신속하게 반영 가능하다.
> - 부분적 장애에 대한 수리가 가능하다.
> - 각 서비스 별 개별 프로세스가 구동되는 REST API와 같은 가벼운 방식으로 통신되는 구조
> ↔ Monolithic Architecture

- CORS 에러 발생 가능

1) ✨✨CORS란 무엇인가?

> 출처

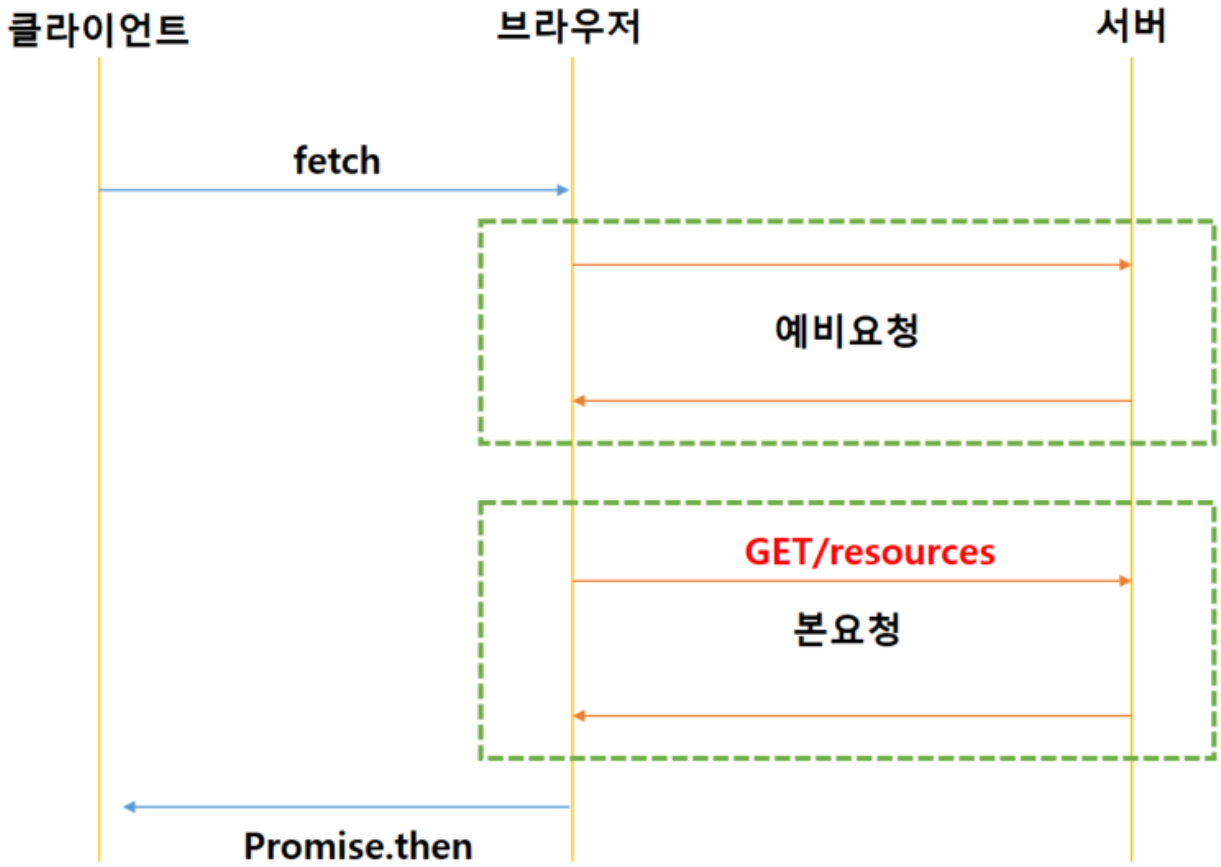
- 다른 *출처* 간의 리소스 공유
    - 출처: Protocol + Host + port #
    - 서버의 위치를 찾아가기 위한 기본적 것들의 조합
- 웹에는 두 가지 정책이 존재하는데 SOP, CORS 이다.
    - SOP(Same-Origin Policy) : 같은 출처에서만 리소스 공유가 가능하도록 한 것
    - CORS: 오픈스페이스 환경에서 다른 출처의 리소스를 사용하 되, 정책을 통해 제한을 두었다.
- 스킴(http://, https://)이나 호스트(도메인)이 다를 경우 같은 출처로 인정하지 않는다.
- 브라우저를 통한 리소스 요청 시, 발생하는 에러
    - 브라우저를 통한 리소스 요청이 일어나지 않으면 발생하지 않는다.

> **브라우저가 판단한다는 것에 유념**

Case 1) CORS 정책위반 1: Preflight Request

- 요청을 한 번에 보내지 않고 예비 요청과 본 요청으로 나누어 서버에 전송하는 경우

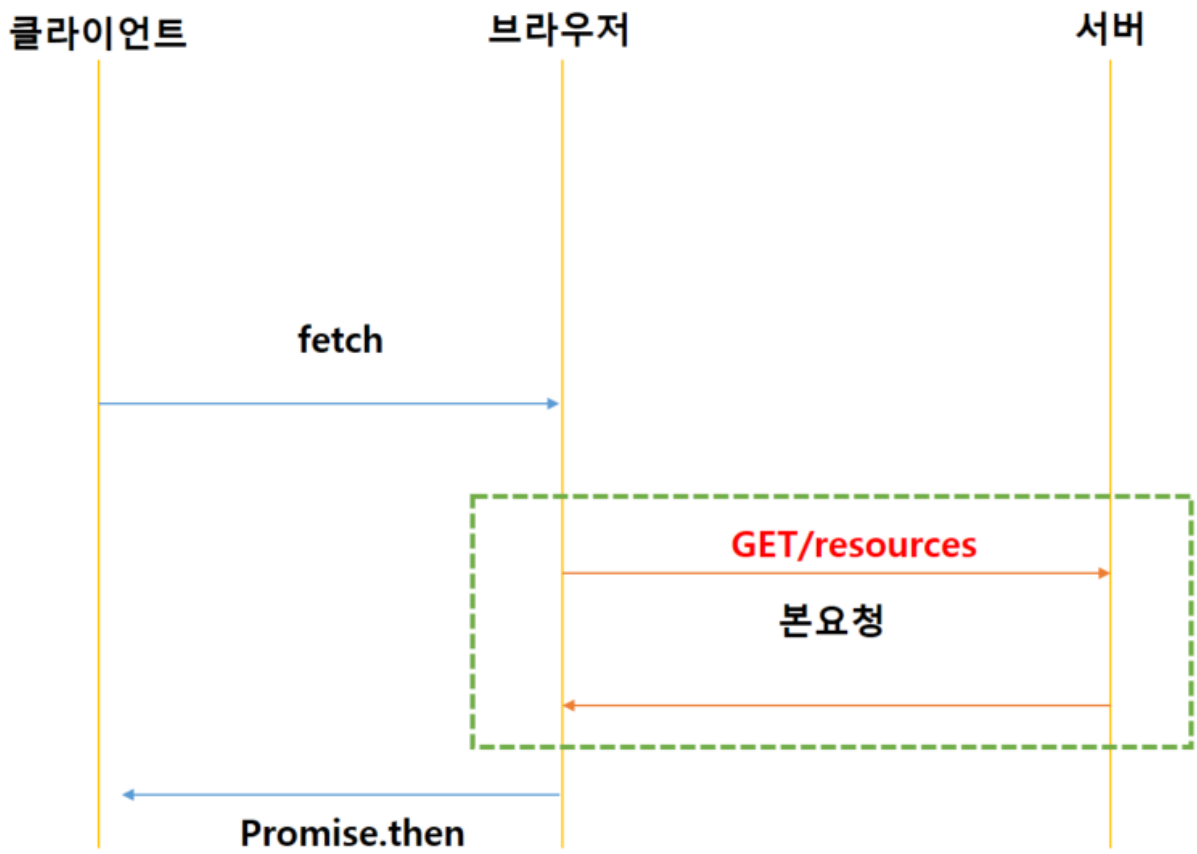
- 본 요청을 보내기 전의 **예비 요청**을 **Preflight Request**라고 한다.



- 예비 요청에 다음 본 요청에 들어갈 `Content-Type`과 `Http Method`가 무엇인지에 관한 정보가 들어있다.
- 클라이언트가 브라우저를 통해 서버에 예비요청을 보내면 서버에서 브라우저에 접근 가능한 출처를 알려준다.
- 클라이언트의 Origin과 서버에서 응답한 접근가능한 주소가 다를 경우 브라우저가 판단하여 CORS 에러를 내뱉는다.

Case 2) Simple Request

- 예비 요청 없이 본 요청만으로 클라이언트에서 요청한 Origin 주소와 서버에서 응답한 접근가능한 주소가 같은지를 브라우저가 판단하는 것



- GET, HEAD, POST 중 하나만 가능하다.
- 헤더에 관한 조건과 Content-type에 관한 조건이 까다롭다.
- 대부분의 HTTP API는 `text/xml` 혹은 `application/json` 형태이기 대문에 본 요청만 보낼 수 있는 요건에 해당하지 않는다.

Case 3) Credentialed Request

- 다른 출처 간 통신에서 보안을 강화하고 싶을 때 사용하는 방식
- 서버에서 `Access-Control-Allow-Origin:*`를 설정해놓았다면 CORS 정책 위반으로 인한 제한을 받지 않는다.
- 그러나 브라우저 인증모드가 `include`일 경우, 모든 출처를 허하는 위 설정이 가능하지 않고 명시적 URL로만 접근이 가능하다.

CORS 에러를 해결하는 방법

- 서버에서 `Access-Control-Allow-Origin`을 이용하여 출처를 명시해준다.
  - 와일드 카드(*)는 권유되지 않는다.
  - 이러한 헤더는 Nginx나 Apache와 같은 서버엔진에 추가할 수도 있으나 복잡한 설정 때문에 Spring과 같은 백엔드 서버에서 설정해주는 것이 좋다.
- lcoal에서 프론트앤드 애플리케이션을 개발하는 경우 에러가 자주 발생한다.
  - localhost주소가 아닌, 프론트에서 웹팩을 이용하여 주소를 프록싱해주어 CORS 정책을 지킨 것처럼 브라우저를 속일 수 있다. (우회)

```
module.exports = {
    devServer: {
        proxy: {
```

```
            '/api': {
                target: 'https://api.evan.com',
                changeOrigin: true,
                pathRewrite: {'^/api':''},
            }
        }
    }
}
```

---

2. 토큰기반 인증(토큰로그인)

- 사용자가 자신의 아이덴티티를 확인하고 고유한 액세스 토큰을 받을 수 있는 프로토콜이다.
- 토큰 유효기간동안 동일한 웹페이지 접속이 가능하며, 다른 서비스를 이용하기 위한 자격 증명을 다시할 필요가 없다.

1) JWT란?

- Json Web Token (*RFC7519*)
- JSON 객체를 암호화하여 만든 String 값
- 토큰 자체에 데이터를 포함하고 있다. (인증과 더불어 인가의 정보도 담고 있다.)
- 클라이언트에게 발급된 JWT를 이용하여 권한이 필요한 리소스를 서버에 요청하는데 이용할 수 있다.
- 서버는 토큰이 유효한지 판단하여 제한된 리소스를 제공할 수 있다.
- 즉, JWT는 사용자의 **인증**을 위해 필요한 정보를 전달하는 객체로 사용하는 것이다.

2) JWT을 발급하는 인증서버와 리소스 서버를 구분하지 않는 이유는 무엇일까?

- 리소스 서버와 인증 서버가 분리된 경우, 매번 인증서버에 무효화 여부를 물어보아야 해 성능에 영향을 받을 수 있다.
- 이는 인증서버가 병목현상(bottleneck)을 일으킬 수 있다는 것을 의미한다.

3) 액세스 토큰과 리프레시 토큰 두 개가 존재하는 이유는 무엇일까?

- 액세스 토큰의 유효기간은 짧고 리프레시 토큰의 유효기간은 상대적으로 길다.
- 액세스 토큰은 짧은 유효기간을 가지는 대신 무효화를 지원하지 않는다.
- 반면, 리프레시 토큰은 긴 유효기간을 가지는 대신 무효화를 지원한다.
- 리프레시 토큰은 탈취되면 안되기 때문에 보안성을 인정받은 https와 같은 안전한 전송수단을 이용해야 한다.
- 리프레시 토큰을 통해 엑세스 토큰을 발급할 수 있다. 리프레시 토큰은 상대적으로 액세스 토큰에 비해 사용 빈도가 줄기 때문에 탈취 가능성을 줄일 수 있다.
- 리소스 서버와 인증 서버를 분리하지 않는다고 가정하자. 이 때, 액세스 토큰의 유효기간이 길다면 무효화를 하는데 DB와 같은 서버 스토리지가 필요하기 때문에, 이는 latency에 영향을 준다.
- 따라서 액세스 토큰의 짧은 유효기간을 가지게 하여 서버 스토리지와 같은 자원을 접근하지 않도록 하여 latency를 줄였다.
- refresh 토큰의 사용빈도를 줄임으로써 탈취 가능성을 줄이고 설사 탈취되더라도 refresh 토큰을 이용하여 access token을 발급하는 번거로운 과정을 거치기 때문에 서버에서 비정상적 요청인지 판단할 수 있는 시간을 벌 수 있다.

- access token도 탈취가능성이 있기 때문에 https를 반드시 기본으로 사용하여 클라이언트와 서버 간 전송되는 데이터를 암호화하도록 하자.

4) 토큰은 DB 에 저장해야하는 것일까?

- 클라이언트에서 액세스 토큰으로 서버에 인증을 받을 때, 만약 액세스 토큰의 유효기간이 만료되었다면 리프레시 토큰으로 새로 액세스 토큰을 발급받아야 한다.
- 이 때, 서버에 리프레시 토큰을 저장을 하게된다면 클라이언트로 액세스 토큰을 발급하기 위해 리프레시 토큰을 묻는 과정을 생략할 수 있다.
- 그러나, 보안적 측면에서 서버에 리프레시 토큰을 저장하는 것은 바람직하지 못하다. 서버가 해커에게 노출될 경우 리프레시 토큰이 노출될 위험이 있어 서버의 보안관리에 철저해야하기 때문이다.