CORS 에러 해결하기

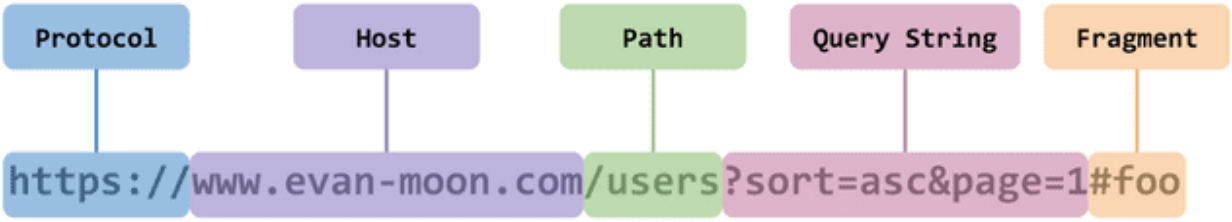
CORS = Cross-Origin Resource Sharing (교차 출처 리소스 공유 정책)

> ❗ Warning
>
> Access to fetch at ‘https://myhompage.com’ from origin ‘http://localhost:3000’ has been blocked by CORS policy: No ‘Access-Control-Allow-Origin’ header is present on the requested resource. If an opaque response serves your needs, set the request’s mode to ‘no-cors’ to fetch the resource with CORS disabled.
>
> 'https://myhomepage.com'에서 'https://localhost:3000' 출처로 가져올 수 있는 액세스가 CORS 정책에 의해 차단되었습니다. 요청된 리소스에 'Access-Control-Allow-Origin' 헤더가 없습니다. 불투명한 응답이 필요에 적합한 경우, 요청 모드를 'no-cors'로 설정하여 CORS가 비활성화된 리소스를 가져오십시오.

위 에러가 무엇이며, 해결하기 위한 방법에 대해 알아보자.

Cross-Origin 이란?

url은 https://domain.com:3000/user?query=name&page=1 과 같이 하나의 문자열 같지만, 사실은 다음과 같이 여러개의 구성 요소로 이루어져 있다.



origin이란 요청 URL의 **scheme(protocol), host, port**를 말하며, 이 중 하나라도 다르다면 cross-origin이다.

다음은 https://www.domain.com:3000 출처에 대한 여러 URL에 따른 동일 출처 비교 표이다.

URL	동일출처 (same-origin)	이유
https://www.domain.com:3000/about	O	프로토콜, 호스트, 포트 번호 동일
https://www.domain.com:3000/about?username=inpa	O	프로토콜, 호스트, 포트 번호 동일
http://www.domain.com:3000	X	프로토콜 다름 (http ≠ https)
https://www.another.co.kr:3000	X	호스트 다름
https://www.domain.com:8888	X	포트 번호 다름
https://www.domain.com	X	포트 번호 다름 (443 ≠ 3000)

동일 출처 정책 (Same-Origin Policy) 이란?

SOP 정책 (Same-Origin Policy) 은 '동일한 출처에서만 리소스를 공유할 수 있다.'라는 법률을 가지고 있다.
즉, 동일 출처(Same-Origin) 서버에 있는 리소스는 자유로이 가져올수 있지만, 다른 출처(Cross-Origin) 서버에 있는 이미지나 유튜브 영상 같은 리소스는 상호작용이 불가능하다는 말이다.

동일 출처 정책이 필요한 이유

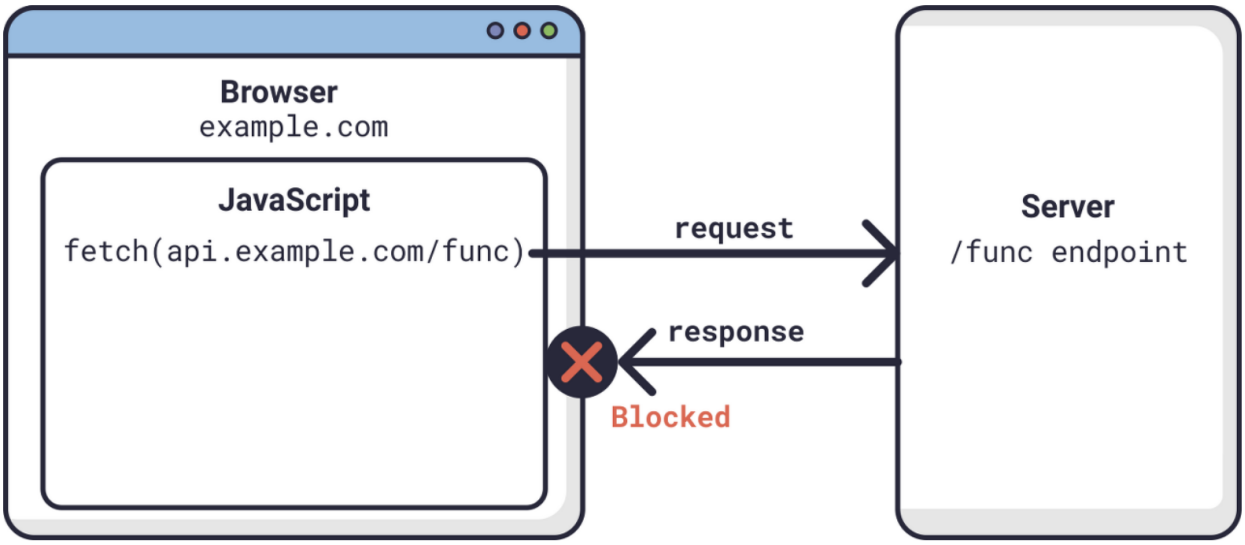
사실 출처가 다른 두 어플리케이션이 자유로이 소통할 수 있는 환경은 꽤 위험한 환경이다. 만일 제약이 없다면, 해커가 CSRF(Cross-Site Request Forgery)나 XSS(Cross-Site Scripting) 등의 방법을 이용해서 우리가 만든 어플리케이션에서 해커가 심어놓은 코드가 실행하여 개인 정보를 가로챌 수 있다.

*CSRF : 인터넷 사용자(희생자)가 자신의 의지와는 무관하게 공격자가 의도한 행위(수정, 삭제, 등록 등)를 특정 웹사이트에 요청하게 만드는 공격

*XSS : 공격자가 상대방의 브라우저에 스크립트가 실행되도록 해 사용자의 세션을 가로채거나, 웹사이트를 변조하거나, 악의적 콘텐츠를 삽입하거나, 피싱 공격을 진행하는 것

출처의 비교와 차단은 어디서하는걸까? ⇒ 브라우저

서버는 CORS를 위반하더라도 정상적으로 응답을 해주고, 응답의 파기 여부는 브라우저가 결정한다.



즉, CORS는 브라우저의 구현 스펙에 포함되는 정책이기 때문에, 브라우저를 통하지 않고 서버 간 통신을 할 때는 이 정책이 적용되지 않는다.
또한 CORS 정책을 위반하는 리소스 요청 때문에 에러가 발생했다고 해도 서버 쪽 로그에는 정상적으로 응답을 했다는 로그만 남기 때문에, CORS가 돌아가는 방식을 정확히 모르면 에러 해결에 난항을 겪을 수도 있다.

그래서 몇 가지예외 조항을 두고 다른 출처의 리소스 요청이라도 이 조항에 해당할 경우에는 허용하기로 했는데, 그 중 하나가 바로 CORS 정책을 지킨 리소스 요청이다.

브라우저의 CORS 기본 동작 살펴보기

1. 클라이언트에서 HTTP요청의 헤더에 Origin을 담아 전달
2. 서버는 응답헤더에 Access-Control-Allow-Origin을 담아 클라이언트로 전달
3. 클라이언트에서 Origin과 서버가 보내준 Access-Control-Allow-Origin을 비교

결국 CORS 해결책은 서버의 허용이 필요

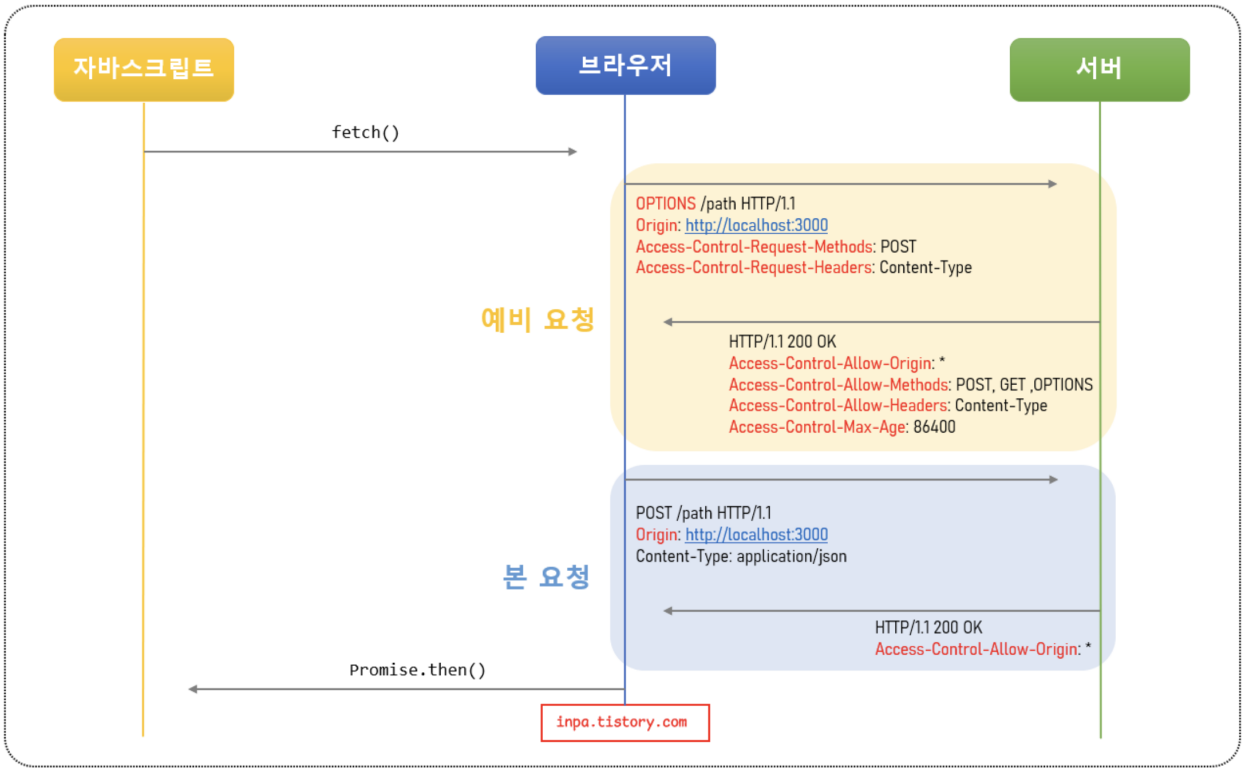
서버에서 Access-Control-Allow-Origin 헤더에 허용할 출처를 기재해서 클라이언트에 응답하면 되는 것이다.

CORS 작동 방식 3가지 시나리오

위에서 살펴본 CORS 동작 흐름은 이해하기 쉽게 하기 위해 기본적인 작동 흐름을 보여준 것이고, 실제로는 CORS가 동작하는 방식은 한 가지가 아니라 세 가지의 시나리오에 따라 변경된다.

1. **예비 요청 (Preflight Request)**

   사실 브라우저는 요청을 보낼때 한번에 바로 보내지않고, 먼저 예비 요청을 보내 서버와 잘 통신되는지 확인한 후본 요청을 보낸다. 예비 요청의 역할은 본 요청을 보내기 전에 브라우저 스스로 안전한 요청인지 미리 확인하는 것이다. 이때 브라우저가 예비요청을 보내는 것을 Preflight라고 부르며, 이 예비요청의 HTTP 메소드를 GET이나 POST가 아닌 OPTIONS라는 요청이 사용된다는 것이 특징이다. 예를들어 자바스크립트로 다음 api 요청을 보낸다고 가정해보자.

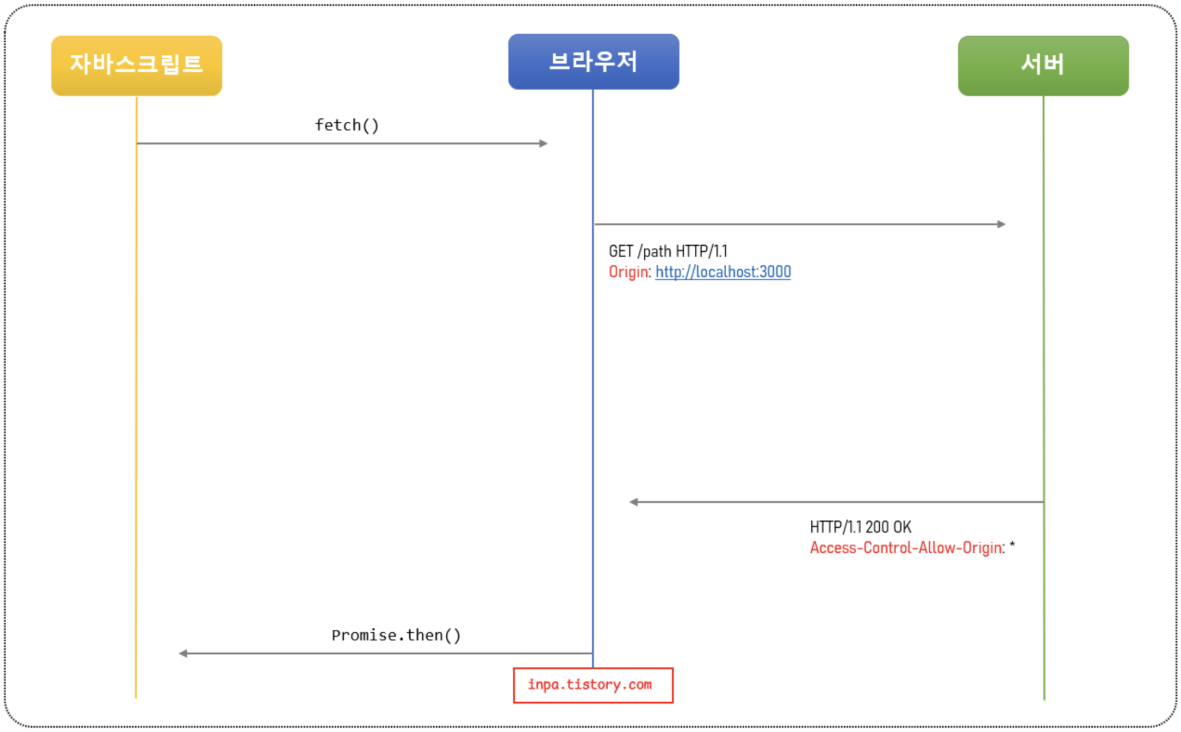



```
1. 자바스크립트의fetch() 메서드를 통해 리소스를 받아오려고 한다.
2. 브라우저는 서버로 HTTP OPTIONS 메소드로 예비 요청(Preflight)을 먼저 보
```

```
   - Origin 헤더에 자신의 출처를 넣는다.
   - Access-Control-Request-Method 헤더에 실제 요청에 사용할 메소드를 실
   - Access-Control-Request-Headers 헤더에 실제 요청에 사용할 헤더들을
3. 서버는 이 예비 요청에 대한 응답으로 어떤 것을 허용하고 어떤것을 금지하고 있
   - Access-Control-Allow-Origin 헤더에 허용되는 Origin들의 목록을 설정
   - Access-Control-Allow-Methods 헤더에 허용되는 메소드들의 목록을 설정
   - Access-Control-Allow-Headers 헤더에 허용되는 헤더들의 목록을 설정한
   - Access-Control-Max-Age 헤더에 해당 예비 요청이 브라우저에 캐시 될 수
4. 이후 브라우저는 보낸 요청과 서버가 응답해준 정책을 비교하여, 해당 요청이 안
5. 서버가 본 요청에 대한 응답을 하면 최종적으로 이 응답 데이터를 자바스립트로
```

2. 단순 요청 (Simple Request)

단순 요청은 말그대로 예비 요청(Prefilght)을 생략하고 바로 서버에 직행으로 본 요청을 보낸 후, 서버가 이에 대한 응답의 헤더에 Access-Control-Allow-Origin 헤더를 보내주면 브라우저가 CORS 정책 위반 여부를 검사하는 방식이다.



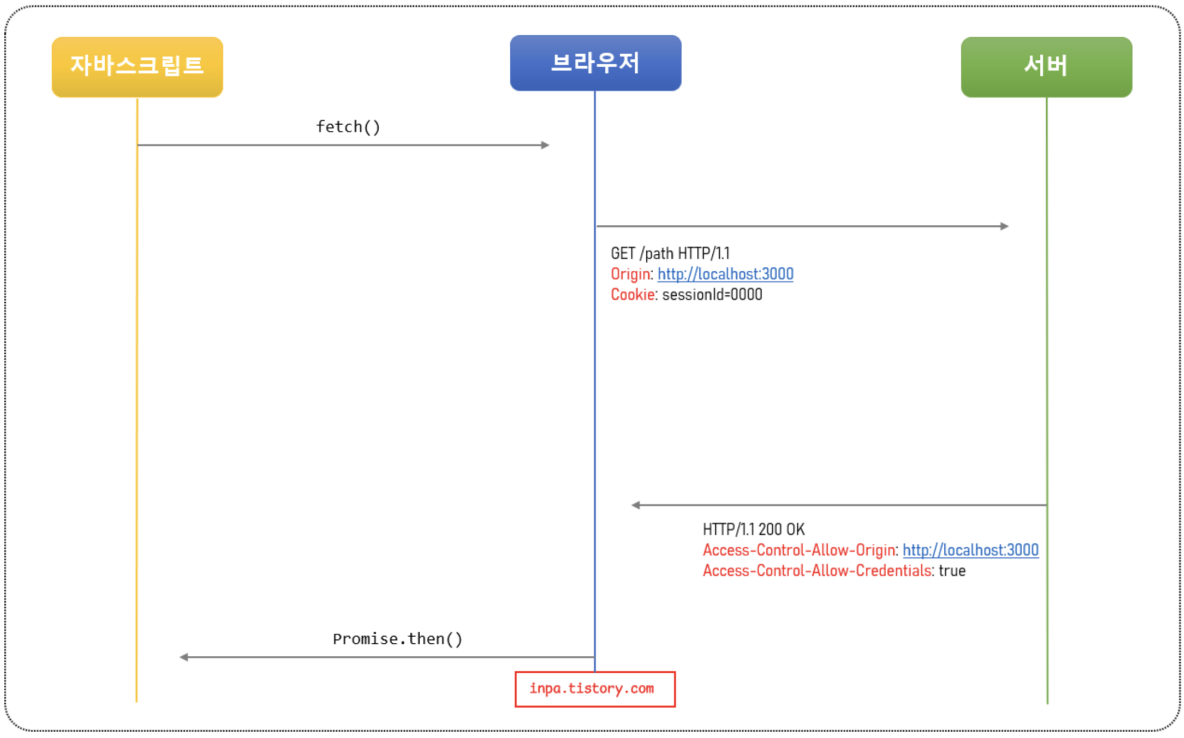
다만, 심플한 만큼 특정 조건을 만족하는 경우에만 예비 요청을 생략할 수 있다.
대표적으로 아래3가지 경우를 만족할때만 가능하다.

> 1. 요청의 메소드는 GET, HEAD, POST 중 하나여야 한다.
> 2. Accept,Accept-Language,Content-Language,Content-Type,DPR,Downlink,Save-Data,Viewport-Width,Width 헤더일 경우에만 적용된다.
> 3. Content-Type 헤더가 application/x-www-form-urlencoded,multipart/form-data,text/plain 중 하나여야한다.

이처럼 다소 까다로운 조건들이 많기 때문에, 위 조건을 모두 만족되어 단순 요청이 일어나는 상황은 드물다고 보면 된다 .왜냐하면 대부분 HTTP API 요청은 text/xml 이나 application/json 으로 통신하기 때문에 3번째 Content-Type이 위반되기 때문이다.따라서 대부분의 API 요청은 그냥 예비 요청(preflight)으로 이루어진다 라고 이해하면 된다.

3. **인증된 요청 (Credentialed Request)**

   인증된 요청은 클라이언트에서 서버에게 자격 인증 정보(Credential)를 실어 요청할때 사용되는 요청이다.
   여기서 말하는 자격 인증 정보란 세션 ID가 저장되어있는 쿠키(Cookie) 혹은
   Authorization 헤더에 설정하는 토큰 값 등을 일컫는다.



1. 클라이언트에서 인증 정보를 보내도록 설정하기

기본적으로 브라우저가 제공하는 요청 API 들은 별도의 옵션 없이 브라우저의 쿠키와 같은 인증과 관련된 데이터를 함부로 요청 데이터에 담지 않도록 되어있다. 이때 요청에 인증과 관련된 정보를 담을 수 있게 해주는 옵션이 바로 credentials 옵션이다. 이 옵션에는 3가지의 값을 사용할 수 있으며, 각 값들이 가지는 의미는 아래와 같다.

옵션 값	설명
same-origin(기본값)	같은 출처 간 요청에만 인증 정보를 담을 수 있다.
include	모든 요청에 인증 정보를 담을 수 있다.
omit	모든 요청에 인증 정보를 담지 않는다

만일 이러한 별도의 설정을 해주지 않으면 쿠키 등의 인증 정보는 절대로 자동으로 서버에게 전송되지 않는다. 서버에 인증된 요청을 보내는 방법으로는 fetch 메서드를 사용하거나

axios, jQuery 라이브리리 등 다양하다. 어떤 메서드를 사용하느냐에 따라 약간 credentials 옵션을 지정하는 문법이 다르다.

```
// fetch 메서드
fetch("https://example.com:1234/users/login", {
    method: "POST",
    credentials: "include",// 클라이언트와 서버가 통신할때 쿠키와 같
  body: JSON.stringify({
        userId: 1,
    }),
})
```

```
// axios 라이브러리
axios.post('https://example.com:1234/users/login', {
     profile: { username: username, password: password }
 }, {
    withCredentials: true// 클라이언트와 서버가 통신할때 쿠키와 같은
})
```

2. 서버에서 인증된 요청에 대한 헤더 설정하기

서버도 마찬가지로 이러한 인증된 요청에 대해 일반적인 CORS 요청과는 다르게 대응해줘야 한다.

- 응답 헤더의Access-Control-Allow-Credentials 항목을 true로 설정해야 한다.
- 응답 헤더의Access-Control-Allow-Origin 의 값에 와일드카드 문자("*")는 사용할 수 없다.
- 응답 헤더의Access-Control-Allow-Methods 의 값에 와일드카드 문자("*")는 사용할 수 없다.
- 응답 헤더의Access-Control-Allow-Headers 의 값에 와일드카드 문자("*")는 사용할 수 없다.

즉, 응답의 Access-Control-Allow-Origin 헤더가 와일드카드(*)가 아닌 분명한 Origin으로 설정되어야 하고, Access-Control-Allow-Credentials 헤더는 true로 설정되어야 한다는

뜻이다. 그렇지 않으면 브라우저의 CORS 정책에 의해 응답이 거부된다. (인증 정보는 민감한 정보이기 때문에 출처를 정확하게 설정해주어야 한다)

CORS를 해결하는 방법

1. Chrome 확장 프로그램 이용

Chrome에서는 CORS 문제를 해결하기 위한 확장 프로그램을 제공해준다.

> ✅ https://chromewebstore.google.com/detail/allow-cors-access-control/lhobafahddgcelffkeicbaginigeejlf

그러면 브라우저 오른쪽 상단에서 확장 프로그램을 활성화 시킬 수 있다. 해당 프로그램을 활성화 시키게 되면,로컬(localhost) 환경에서 API를 테스트 시, CORS 문제를 해결할 수 있다.

2. 프록시 서버 이용하기

중간에 요청을 가로채서 HTTP 응답 헤더에 `Access-Control-Allow-Origin : *` 를 설정해서 응답해준다.

3. 서버에서 Access-Control-Allow-Origin 헤더 세팅하기

- Access-Control-Allow-Origin
- Access-Control-Allow-Methods
- Access-Control-Allow-Headers

```
var http = require('http');

const PORT = process.env.PORT || 3000;
```

```
var httpServer = http.createServer(function (request, respons
    // Setting up Headers
    response.setHeader('Access-Control-Allow-origin', '*'); /
    response.setHeader('Access-Control-Allow-Methods', 'GET,
    response.setHeader('Access-Control-Allow-Credentials', 't

    // ...

    response.writeHead(200, { 'Content-Type': 'text/plain' }
    response.end('ok');
});

httpServer.listen(PORT, () => {
    console.log('Server is running at port 3000...');
});
```

이때, 와일드 카드를 사용하기 보다는 **직접 명시**해주는 것이 보안상으로도 안전하고, 에러 발생을 방지할 수 있다.

4. **미들웨어 라이브러리 사용하기**

   클라이언트 : http-proxy-middleware

   CORS를 해결하는 방법 중 하나가 바로 http-proxy-middleware 라이브러리를 사용하는 것이다. 배포하고 나서는 동일한 출처에 요청을 하므로 CORS 에러가 발생하지 않지만,배포하기 전 개발 단계가 문제가 되는데 로컬 환경에서 React는 3000번 포트를 사용하고 Express는 5000번 포트를 사용하기 때문이다. 그러면 3000번 포트에서 5000번 포트로 요청을 보내니까 당연히 CORS 문제가 발생하게 되는데 이를 로컬 환경일 경우에 한정해서 http-proxy-middleware 라이브러리를 사용하여 클라이언트단에서 쉽게 해결할 수 있다. http-proxy-middleware 라이브러리를 설치하고, setupProxy.js 라는 파일을 src 폴더 내에 만들고 아래와 같이 코드를 작성하면된다.

```
const { createProxyMiddleware } = require("http-proxy-m
```

```
module.exports = function (app) {
  app.use(
    "/api",
      createProxyMiddleware({
        target: "http://localhost:5000",
        changeOrigin: true,
    })
  )
}
```

위와 같이 코드를 작성해 두면, 로컬 환경에서 http://localhost:3000/api로 시작되는 요청을 라이브러리가 http://localhost:5000/api 로 프록싱 해주게 된다. 따라서 브라우저는 클라이언트와 서버의 출처가 다르지만 같은 것으로 받아들이게 되어 CORS 문제를 일으키지 않는다.

서버 : CORS 미들웨어 사용하기

서버를 Express로 구축한 경우 Node.js 미들웨어 중 하나인 CORS를 사용하여 쉽게 문제를 해결할 수 있다. origin에 허용하고자 하는 도메인을 넣어주면 response 헤더에 Access-Control-Allow-Origin 내용이 추가가 된다.

```
const express = require('express');
const cors = require('cors');
const app = express();

const corsOptions = {
  origin: '허용하고자 하는 도메인',
};

app.use(cors(corsOptions));
```

app.use(cors()) 이런 식으로 하게 되면 모든 출처에서 오는 요청을 허용하는 것이므로 지양하는 게 좋다.

<참고자료>

https://inpa.tistory.com/entry/WEB-📚-CORS-💯-정리-해결-방법-👏

https://evan-moon.github.io/2020/05/21/about-cors/