

SSAFY Node.js 프로젝트

구분	종합 프로젝트 DAY 3
주제	로그 데이터
학습내용	1. 서버 구성 및 동작 2. 통신 API 작성 3. 프론트 페이지 구성 4. 로그 연동 5. 라이브러리 학습
예상구현기간	8H

로그 데이터 프로젝트는 지금까지 배운 Node.js 를 사용하여, 서버를 구성하고 운영하기 위한 로그 프로그램을 만들어 서버의 로그를 실시간으로 확인하는 데 목적이 있다. 수업 시간에 한 내용에 이어, 로그를 시각적으로 Visualization 한다.

#	message	level	timestamp
0	error 메시지	error	2022-05-20 15:26:13
1	warn 메시지	warn	2022-05-20 15:26:13
2	info 메시지	info	2022-05-20 15:26:13
3	error 메시지	error	2022-05-20 15:26:18
4	warn 메시지	warn	2022-05-20 15:26:18
5	info 메시지	info	2022-05-20 15:26:18
6	error 메시지	error	2022-05-20 15:26:19

프로젝트 요구사항 및 과정 설계

로그데이터 서비스를 사용하기 전, 기획 및 설계를 한 후, 계획에 맞추어 한 단계씩 웹 서비스를 제작해본다.

A. 만들어야 하는 기능

- i. 서버 구성 및 동작
- ii. 통신 API 작성
- iii. 프론트 페이지 구성
- iv. 로그 연동(색상 구분)
- v. 라이브러리 학습

B. 추가로 사용하는 Library

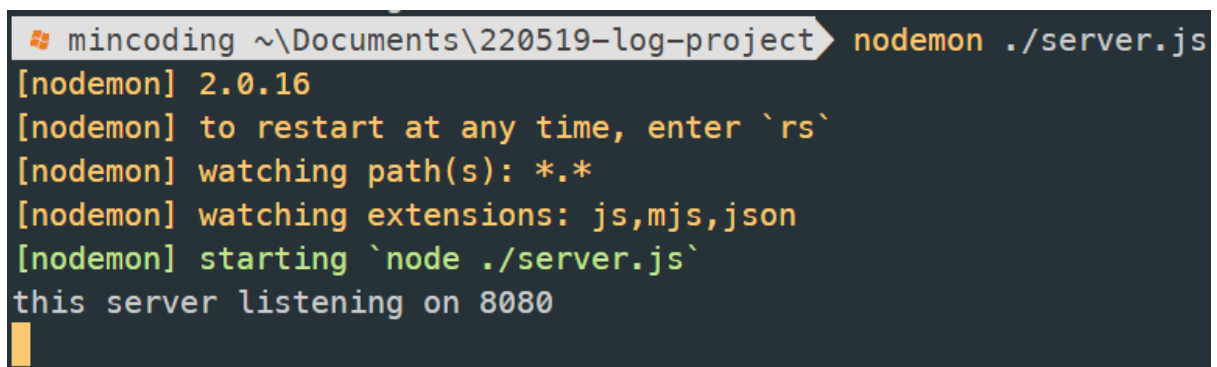
- i. Log Library & Middleware: morgan & winston
- ii. Web Server: express
- iii. Support Library: Winston-daily-rotate-file

서버 구성 및 동작

메터모스트에 제공된 `express_start_template` 을 다운로드받아, 프로젝트 디렉터리에 압축 해제한 후, 다음 명령을 실행한다.

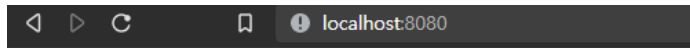
```
$ npm i
```

```
$ nodemon ./server.js
```

A terminal window with a dark background. The title bar shows 'mincoding ~\Documents\220519-log-project'. The command 'nodemon ./server.js' has been entered. The output shows 'nodemon 2.0.16', instructions to restart with 'rs', watched paths and extensions, and a message that the server is listening on port 8080.

```
mincoding ~\Documents\220519-log-project> nodemon ./server.js
[nodemon] 2.0.16
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node ./server.js`
this server listening on 8080
```

다음과 같은 창이 뜨고, 크롬 커서 `localhost:8080` 으로 접속해보면

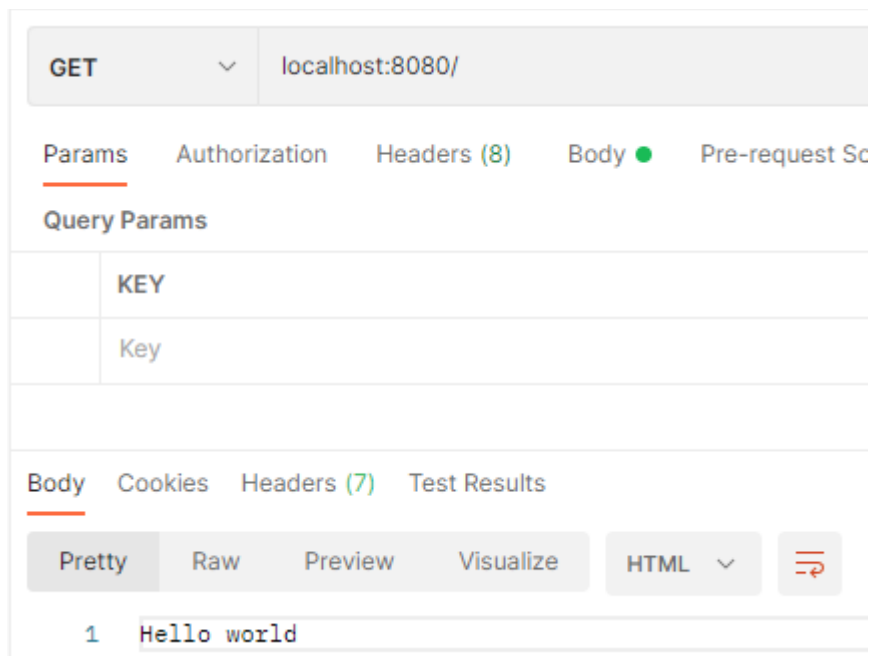


Hello world

express 서버가 정상적으로 동작함을 확인할 수 있다.

postman 으로도 확인해보자.

GET 방식으로, localhost:8080/ 로 줄 것이다.



Hello World 가 잘 찍힌 것을 확인할 수 있다.

서버 구성에 필요한 라이브러리를 설치한다

```
$ npm i winston winston-daily-rotate-file
```

```
"dependencies": {
  "cors": "^2.8.5",
  "express": "^4.17.2",
  "morgan": "^1.10.0",
  "winston": "^3.7.2",
  "winston-daily-rotate-file": "^4.6.1"
}
```

dependencies 에 추가되었는지 확인

통신 API 작성

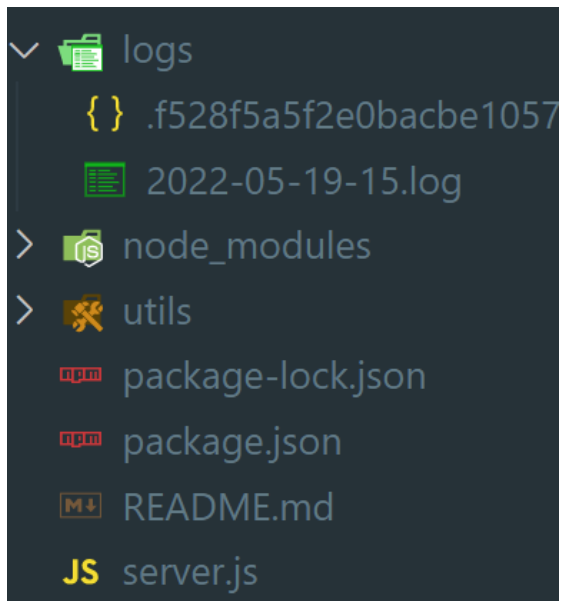
utils 디렉터리를 생성하고, winston.js 파일을 생성

```
1  const winston = require("winston");
2  require("winston-daily-rotate-file");
3
4  const transport = new winston.transports.DailyRotateFile({
5    level: "info",
6    filename: "./logs/%DATE%.log",
7    datePattern: "YYYY-MM-DD-HH",
8    zippedArchive: true,
9    maxSize: "20m",
10   maxFiles: "1d",
11 });
12
13 const logger = winston.createLogger({
14   transports: [transport],
15 });
16
17 module.exports = { logger };
```

이제 server.js 로 옮겨가서,

```
const { logger } = require("../utils/winston");
```

코드를 추가하면,



지정한 형식대로 logs 폴더가 자동 생성되며, 로그 파일이 생성된다.

server.js 에서 로그를 만들어보자.

```
app.get("/", (req, res) => {  
  logger.error("error 메시지");  
  logger.warn("warn 메시지");  
  logger.info("info 메시지");  
  logger.http("http 메시지");  
  logger.debug("debug 메시지");  
  res.send("Hello world");  
});
```

포스트맨에서 해당 경로 접근 시, log 파일에 로그가 출력된다.

```
{"level":"error","message":"error 메시지"}
{"level":"warn","message":"warn 메시지"}
{"level":"info","message":"info 메시지"}
{"level":"error","message":"error 메시지"}
{"level":"warn","message":"warn 메시지"}
{"level":"info","message":"info 메시지"}
```

총 두 번의 신호를 보냈을 때, error, warn, info 가 반복해서 두 번 찍힌 결과다.

그런데, 대체 왜 info 까지 찍힌 것일까? 무엇이 error 고, 무엇이 warn 인가?

이건, error 가 일어나거나 warn 이 일어난 상황이 아니다. 굉장히 중요한 개념인데, error는 컴퓨터에서 자동으로 정하는 게 아니다. 개발자가 error 라고 지정하면 error 고, warn 이라고 지정하면 warn 인 것이다. 즉, 이 메시지는 실제 error 가 일어나서 찍힌 게 아니고, 사용자가 정해둔 level 에 따라서 찍혔을 뿐이다.

그 기준은 어디에 있을까? winston.js 로 가보면,

```
const transport = new winston.transports.DailyRotateFile({
  level: "info",
  filename: "./logs/%DATE%.log",
  datePattern: "YYYY-MM-DD-HH",
  zippedArchive: true,
  maxSize: "20m",
  maxFiles: "1d",
});
```

level 을 info 로 정해놓았다. 이 뜻은, info 가 존재하는 곳까지 내려가서 찍히게 level 을 정해놓았다는 뜻인데, server.js 로 가서 우리가 작성한 로그 메시지를 보면 다음과 같다.

```
app.get("/", (req, res) => {
  logger.error("error 메시지");
  logger.warn("warn 메시지");
  logger.info("info 메시지");
  logger.http("http 메시지");
  logger.debug("debug 메시지");
  res.send("Hello world");
});
```

무슨 뜻인가? / 신호를 받으면 info 메세지까지 찍힐 것이다. 우리가 정해둔 레벨이 info 까지이기 때문이다.

만약, debug 로 레벨을 정해놓았다면, http, debug 로그까지 모두 찍힐 것이다.

```
const transport = new winston.transports.DailyRotateFile({
  level: "debug",
  filename: "./logs/%DATE%.log",
  datePattern: "YYYY-MM-DD-HH",
  zippedArchive: true,
  maxSize: "20m",
  maxFiles: "1d",
});
```

server.js 에서 debug 로 레벨을 바꾸고, 다시 postman 으로, / 로 get 을 보내보겠다.

```
{
  "level": "error",
  "message": "error 메시지"
}
{
  "level": "warn",
  "message": "warn 메시지"
}
{
  "level": "info",
  "message": "info 메시지"
}
{
  "level": "http",
  "message": "http 메시지"
}
{
  "level": "debug",
  "message": "debug 메시지"
}
```

이번엔 debug 까지 모두 찍힌 것을 확인할 수 있다.

즉, 에러는 개발자가 정의하는 것일 뿐이다.

다시 info 로 바꿔두고, 계속 진행하겠다.

우리의 로그를 자세히 보면, 로그가 찍힌 시간이 없어 로그가 언제 발생했는지 파악하기 매우 불편하다. 로그에 시간을 추가해보겠다.

```
1  const winston = require("winston");
2  const { format } = require("winston");
3  const { combine } = format;
4  require("winston-daily-rotate-file");
5
6  const transport = new winston.transports.DailyRotateFile({
7    level: "info",
8    filename: "./logs/%DATE%.log",
9    datePattern: "YYYY-MM-DD-HH",
10   zippedArchive: true,
11   maxSize: "20m",
12   maxFiles: "1d",
13   format: combine(format.timestamp(), format.json()),
14 });
```

line 2, 3 필요한 라이브러리 가져오기.

우리는 format 과 combine 을 사용할 것이다.

line 13 transport 객체 생성자에 format: combine() 을 추가

format.timestamp(), format.json() 두 가지 파라미터를 추가해줬다.

그리고 포스트맨으로 한번 send 해주고 다시 로그 확인하면,

```
{"level":"error","message":"error 메시지","timestamp":"2022-05-20T01:07:21.072Z"}
{"level":"warn","message":"warn 메시지","timestamp":"2022-05-20T01:07:21.073Z"}
{"level":"info","message":"info 메시지","timestamp":"2022-05-20T01:07:21.073Z"}
```

다음과 같이, 언제 로그가 발생했는지까지 찍힌다. 하지만, 이렇게하면 읽기 어려우므로 로그를 직관적으로 바꿔보겠다.


```
format: combine(
  format.timestamp({ format: "YYYY-MM-DD HH:mm:ss" }),
  format.json()
),
```

format.timestamp의 파라미터로, 로그로 출력될 시간의 기준을 정해두겠다.

포스트맨으로 send 해보자.

```
{
  "level": "error",
  "message": "error 메시지",
  "timestamp": "2022-05-20T01:07:21.072Z"
},
{
  "level": "warn",
  "message": "warn 메시지",
  "timestamp": "2022-05-20T01:07:21.073Z"
},
{
  "level": "info",
  "message": "info 메시지",
  "timestamp": "2022-05-20T01:07:21.073Z"
},
{
  "level": "error",
  "message": "error 메시지",
  "timestamp": "2022-05-20 10:13:21"
},
{
  "level": "warn",
  "message": "warn 메시지",
  "timestamp": "2022-05-20 10:13:21"
},
{
  "level": "info",
  "message": "info 메시지",
  "timestamp": "2022-05-20 10:13:21"
}
```

위 세 줄은 포맷 적용 전,

아래 세 줄은 적용한 이후이다.

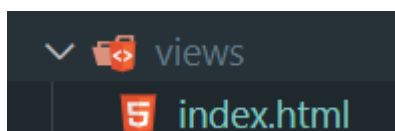
날짜와 시간까지 깔끔하게, 우리가 원하는 형태로 찍힌 것을 확인할 수 있다.

프론트 페이지 구성

server.js에서 프론트 페이지를 구성하기 위해 다음을 추가한다.

```
app.use(express.static(__dirname + "/views"));
```

views 디렉토리를 만든 후, index.html 파일을 생성한다.



```
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8" />
5      <meta http-equiv="X-UA-Compatible" />
6      <meta name="viewport" content="width=device-width, initial-scale=1" />
7      <title>log project</title>
8    </head>
9    <body>
10     <h1>log project</h1>
11   </body>
12 </html>
```

title 과 h1 하나만 추가한 간단한 형태다.

이제, log 를 받는 라우트와 화면을 보여주는 라우트를 구분하겠다.

server.js

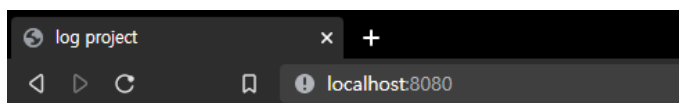
```

13 app.use(express.static(__dirname + "/views"));
14
15 app.post("/api/logs", (req, res) => {
16     logger.error("error 메세지");
17     logger.warn("warn 메세지");
18     logger.info("info 메세지");
19     logger.http("http 메세지");
20     logger.debug("debug 메세지");
21     res.json({ success: true });
22 });

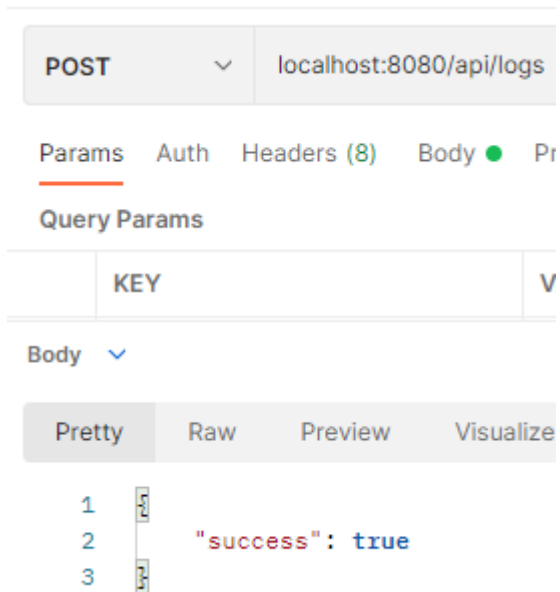
```

- line13 따로 get / 일 경우의 route 를 지정하진 않았지만,
 app.use(express.static(__dirname + "/views"))
 에 의해, 방금 제작한 index.html 이
 localhost:8080 으로 접속 시에 보일 것이다.
- line15 - POST /api/logs 로 요청 시에 log 를 찍도록 했고,
 성공 시 { success: true } json 을 리턴하도록 처리했다.

localhost:8080 으로 접속한 경우



log project



POST /api/logs 로 요청을 보낸 경우

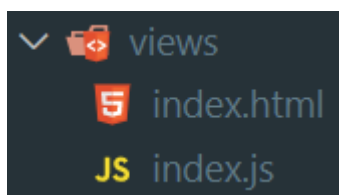
자, 이제 생각을 해보자. 어떤 식으로 진행하면 될까?

비동기통신을 할 것이다.

현재 localhost:8080 으로 접속하면 index.html 이 뜨도록 만들었다.

그러면 index.html 에서 index.js 라는 파일을 따로 만들어서, axios 로 localhost:8080/api/logs 로 POST 요청을 보내서 json 으로 로그 데이터를 받은 후에,

js 에서 해당 로그를 처리해 화면에 붙이면 될 것이다.



index.js 를 만들었고,

```
<body>
  <h1>log project</h1>
  <script src="./index.js"></script>
</body>
```

다음과 같이, 콘솔로그만 찍어보겠다.

```
1 console.log("hello");
```

결과:

```
hello
```

자, 이제 axios 를 설치해야 하는데, NPM 으로 설치하면 쓸 수가 없다.

왜냐? index.html 에 script 로 연결된 index.js 는 node.js 가 아니기 때문이다.
require 로 모듈을 가져오는 기능은 node.js 에서만 가능하고, 브라우저 내에서 사용하는 JS 에선 사용할 수 없다.

그래서, CDN 설치할 것이다. jsdelivr 로 가서, axios cdn 을 가져오자.

```
<body>
  <h1>log project</h1>
  <script src="https://cdn.jsdelivr.net/npm/axios@0.27.2/dist/axios.min.js"></script>
  <script src="./index.js"></script>
</body>
```

다음과 같이 추가하고,

index.js 를 다음과 같이 수정한다.

```

1  const url = "http://localhost:8080/api/logs";
2
3  const getData = async () => {
4    try {
5      const response = await axios.get(url);
6      if (response.data) {
7        console.log(response.data);
8      }
9    } catch (error) {
10     console.log(error);
11   }
12 };
13
14 getData();
15

```

GET /api/logs 로 요청할 것이고,

비동기 방식으로 json 데이터를 가져온다.

그럼 해당 라우트를 만들어야 할 것이므로, server.js 로 가자.

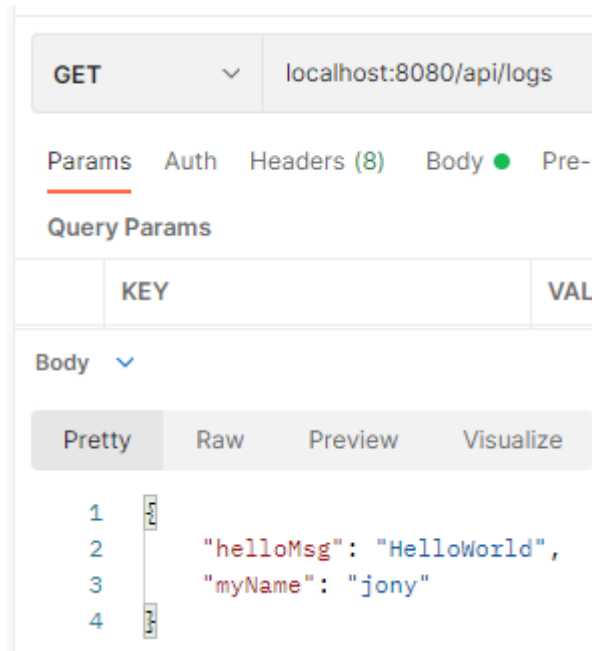
```

24  app.get("/api/logs", (req, res) => {
25    res.json({
26      helloMsg: "HelloWorld",
27      myName: "jony",
28    });
29  });

```

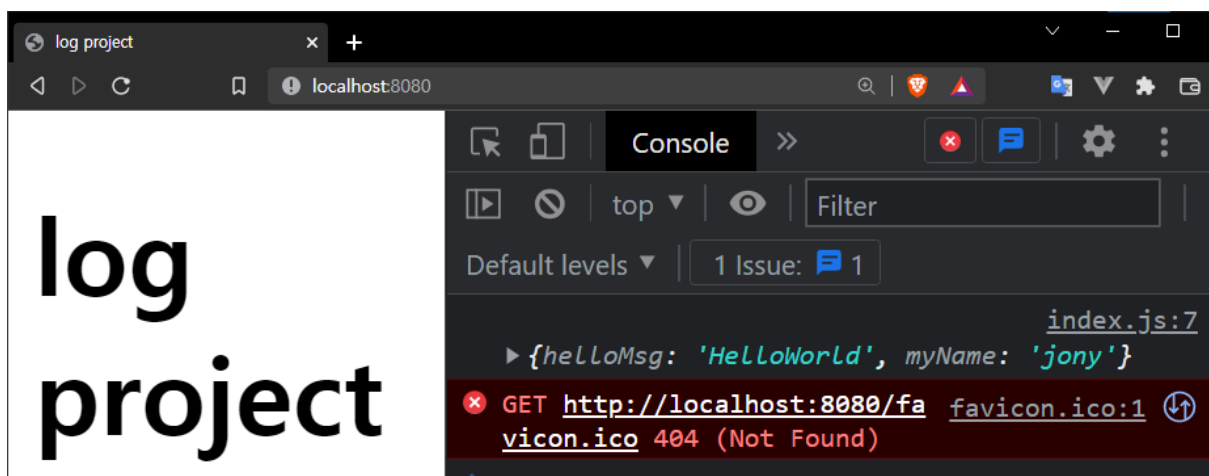
이런 식으로, GET /api/logs 라우트로 접속하면 json 을 보여주는 코드를 달았고,

postman 으로 먼저 테스트해보자.



postman 에선 정상적으로 찍혔다.

그렇다면, index.html 에서도 찍혀야 할 것이다.



로그에, { helloMsg: "HelloWorld", myName: "jony" } 가 잘 찍힌 것이 확인된다.

즉, 프론트엔드와 백엔드의 연결이 성공적으로 확인되었다.

이제 무엇을 해야하나? 오늘 전체 과정중에서 가장 어려운 일이다. 파싱을 할 것이다.

로그는 현재 어떤 형태인가?

```
{ "level": "error", "message": "error 메시지", "timestamp": "2022-05-20 10:13:21" }  
{ "level": "warn", "message": "warn 메시지", "timestamp": "2022-05-20 10:13:21" }  
{ "level": "info", "message": "info 메시지", "timestamp": "2022-05-20 10:13:21" }
```

이건 객체나 json 이 아니다. 그냥 string 일 뿐이다.

목표는, 이걸 파싱해서 객체로 바꾸는 것이다.

일단, 파일시스템을 사용해야한다. 로그 파일에 접근해서 string 을 가져오기 위해서다.

server.js 에 다음을 추가한다.

```
const fs = require("fs");
```

그리고, app.post("/api/logs") 을 다음과 같이 수정하자.


```

16   const insert = (str, index, target) => {
17       const front = str.slice(0, index);
18       const back = str.slice(index, str.length);
19       return front + target + back;
20   };
21
22   let retData = {};
23   app.post("/api/logs", (req, res) => {
24       logger.error("error 메시지");
25       logger.warn("warn 메시지");
26       logger.info("info 메시지");
27       logger.http("http 메시지");
28       logger.debug("debug 메시지");
29       fs.readFile("./logs/2022-05-20-17.log", "utf8", (err, data) => {
30           retData = data;
31           let idx = -1;
32           while (1) {
33               idx = retData.indexOf("}", idx + 1);
34               if (idx === -1) {
35                   break;
36               }
37               retData = insert(retData, idx + 1, ",");
38           }
39           retData = "[" + retData.slice(0, retData.length - 3) + "]";
40           retData = JSON.parse(retData);
41           console.log(retData);
42       });
43       return res.json({
44           success: true,
45       });
46   });

```

우선, POST /api/logs 했을 때 결과부터 보자.

```

    timestamp: '2022-05-20 17:13:21'
  },
  {
    level: 'error',
    message: 'error 메시지 ',
    timestamp: '2022-05-20 17:16:04'
  },
  {
    level: 'warn',
    message: 'warn 메시지 ',
    timestamp: '2022-05-20 17:16:04'
  },
  {
    level: 'info',
    message: 'info 메시지 ',
    timestamp: '2022-05-20 17:16:04'
  }
]

```

콘솔에, "색깔" 로 예쁘게 표시되었다.

이것은 매우 중요한데, json 양식이 아니면 string 이므로, 이런 예쁜 형태로 컬러링되지 않는다.

색깔과 인덴팅이 깔끔하게 콘솔로 찍힌다는 건, 단순히 string 을 받았다는 뜻이 아니라, 제대로 된 json 포맷으로 파싱 성공했다는 뜻이다.

꽤 긴 코드인데, 하나하나 분석해보자.

```

16   const insert = (str, index, target) => {
17     const front = str.slice(0, index);
18     const back = str.slice(index, str.length);
19     return front + target + back;
20   };

```

line16 - insert 함수는 인덱스를 기준으로 문장을 앞뒤로 나눠서,
원하는 문자열을 넣으려고 우리가 만든 함수다.
왜 이걸 쓸까? 콤마 ',' 를 넣기 위해서다.

우리 로그 잘 보면,

```
{"level":"error","message":"error 메시지","timestamp":"2022-05-20 13:51:43"}  
{"level":"warn","message":"warn 메시지","timestamp":"2022-05-20 13:51:43"}  
{"level":"info","message":"info 메시지","timestamp":"2022-05-20 13:51:43"}
```

객체가 되려면 무조건 맨 뒤에 콤마가 있어야 하는데 없다.

```
22 let retData = {};
```

line22 전역변수 retData 를 빈 객체로 만들어둔다.
이 안에, 파싱 성공한 log JSON 을 담아서 프론트엔드로 보낼 것이다.
그런데 빈 객체라는 뜻은, 프론트엔드에서 그저
GET /api/logs 만 시도할 경우, 아무 데이터도 못 가져옴을 뜻한다.
즉, 프론트엔드에선 먼저 POST /api/logs 명령을 한 후,
GET /api/logs 로, 로그를 가져올 것이다.

```
29 fs.readFile("./logs/2022-05-20-17.log", "utf8", (err, data) => {  
30   retData = data;
```

그리고, 파일을 읽는다. 물론 이 파일은 실제 존재하는 로그파일이어야 한다.

이 프로그램의 가장 큰 단점은, 시간에 따라서 파일명이 바뀌다보니깐 프로그램 작동 전에 현재 날짜의 현재 시간에 해당하는 로그 파일로 바꿔줘야 한다는 것이다.

그냥 Date 객체 사용해서 쓰면 되지 않나? 상당히 귀찮은 일인데,

Date 객체의 경우, 예를 들어 5월이면 5로 나오지 05로 나오진 않는다. 그래서 앞에 0을 상황에 따라 붙여주는 파싱을 해야하는데, 만들수는 있겠지만 우리 수업의 범위에서 벗어나기때문에 생략한다.

현재 읽고자 하는 파일은 2022-05-20-17 인데, 이것은 2022년 05월 20일 17시에 발생한 로그라는 뜻이다.

그리고 로그파일 전체를 string 으로, 변수 data 로 받은 다음,
전역변수 retData 에 저장한다.

```
31     let idx = -1;
32     while (1) {
33         idx = retData.indexOf("}", idx + 1);
34         if (idx === -1) {
35             break;
36         }
37         retData = insert(retData, idx + 1, ",");
38     }
```

인덱스는 -1로 시작하고, 무한루프를 사용한다.

왜? 언제 끝날지 모르기때문이다. 상황에 따라 로그의 길이는 다르다.

그리고, } 를 찾을 때마다 그 인덱스를 받고,

찾기 시작하는 시작점은 두번째 파라미터로, 계속 바뀔 것이다.

만약, } 를 찾지 못하면 문장이 끝난 것이기때문에 break 다.

그리고 우리가 만든 insert 를 사용할 것인데, 찾은 인덱스에서 +1 한 지점을 찾아, 문장을 둘로 쪼개고 콤마를 넣는다.

```
39     retData = "[" + retData.slice(0, retData.length - 3) + "];
```

그리고 파싱한 객체는 배열이 되어야한다. 왜냐면,

```
GET /api/make-logs 200 2.367 ms - 16
{"level":"error","message":"error 메시지","timestamp":"2022-05-20 14:07:49"},
{"level":"warn","message":"warn 메시지","timestamp":"2022-05-20 14:07:49"},
{"level":"info","message":"info 메시지","timestamp":"2022-05-20 14:07:49"},
{"level":"error","message":"error 메시지","timestamp":"2022-05-20 14:08:07"},
{"level":"warn","message":"warn 메시지","timestamp":"2022-05-20 14:08:07"},
```

현재 형태는 객체 객체 객체의 연속인데, 배열로 감싸져야 한다. 그래서 앞뒤에 대괄호를 붙인 것이다.

그리고 맨 마지막 콤마를 제거하는 작업이 필요하다. json 형태로 바꾼 다음 객체로 바뀌버릴 예정이기 때문에, 맨 마지막 콤마가 들어가면 에러 발생한다. 이를 위해 slice 를 사용할것이다.

슬라이스 무엇인가? 어디부터 어디까지.

여기선, 0부터 `retData.length - 3` 인데, 왜냐면 로그에 엔터가 포함되어있어서 정확히 -3 을 해야만 콤마 직전 위치까지 가기 때문이다.

그랬을 때 결과는 다음과 같다.

```
{"level":"warn","message":"warn 메시지","timestamp":"2022-05-20 14:08:51"},
{"level":"info","message":"info 메시지","timestamp":"2022-05-20 14:08:51"},
{"level":"error","message":"error 메시지","timestamp":"2022-05-20 14:09:21"},
{"level":"warn","message":"warn 메시지","timestamp":"2022-05-20 14:09:21"},
{"level":"info","message":"info 메시지","timestamp":"2022-05-20 14:09:21"}]
```

정확히, 맨 마지막 엘리먼트의 콤마가 제거되었고, 배열을 의미하는 대괄호로 감싸졌다.

```
40      retData = JSON.parse(retData);
```

마무리로, `json` 을 객체로 바꾸는 `JSON.parse()` 를 사용하면 결과는 다음과 같다.

```
timestamp: '2022-05-20 17:13:21'
},
{
  level: 'error',
  message: 'error 메시지 ',
  timestamp: '2022-05-20 17:16:04'
},
{
  level: 'warn',
  message: 'warn 메시지 ',
  timestamp: '2022-05-20 17:16:04'
},
{
  level: 'info',
  message: 'info 메시지 ',
  timestamp: '2022-05-20 17:16:04'
}
]
```

일단, string 일 때와는 다르게 색깔이 칠해졌고, 인덴팅이 예쁘게 적용되었다. json 은 문법이 엄격하기 때문에, 콤마 하나, 중괄호 하나라도 잘못되면 절대 이런 형태로 나오지 않는다.

즉, 완벽하게 우리가 쓰기 딱 좋은 객체가 되었다.

```
return res.json({
  success: true,
});
```

그러나, 여기서 정작 프론트엔드로 잘 만든 JSON 을 넘겨주진 않는다.

그저 success: true 라는 객체 하나 줄 뿐이다.

왜? 생각을 해보면, POST /api/logs 는 "로그를 만드는 일만" 한다. 즉, 로그를 가져오는 일은 구분해서 처리해야한다.

일단, views/index.js 로 가보자.

```

1  const url = "http://localhost:8080/api/logs";
2
3  const getData = async () => {
4      try {
5          await axios.post(url);
6          const response = await axios.get(url);
7          if (response.data) {
8              console.log(response.data);
9          }
10     } catch (error) {
11         console.log(error);
12     }
13 };
14
15 getData();
16

```

다음과 같이 수정했다. GET 이든 POST 든 URL 은 동일하다.

만약 get 먼저 실행시켜버리면 빈 객체만 가져올 것이다.

애초에 retData = {} 으로 설정되어 있기 때문이다.

그래서, 다음과 같이 서버와 통신하는데,

line5 POST 먼저 실행해서 retData 를 파싱된 로그로 채움

line6 GET 실행하여 파싱 성공한 retData 가져옴

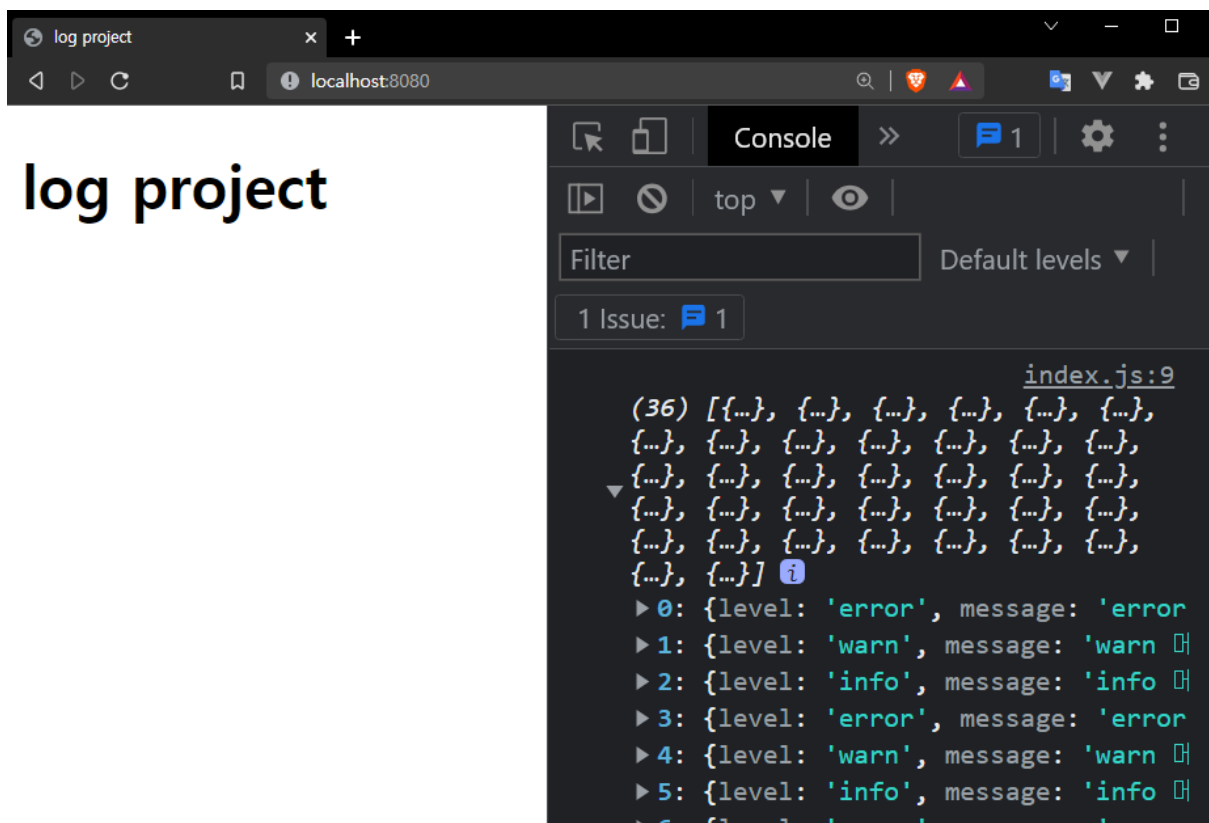
그럼 server.js 에선 GET /api/logs 를 다음과 같이 수정하기만 하면 될 것이다.

```

48   app.get("/api/logs", (req, res) => {
49     res.json(retData);
50   });

```

그리고, 크롬 키고 localhost:8080 에 접속하여 console 확인해보면,



성공적으로, 백엔드에서 로그를 JSON 으로 받아왔다.

이것도 마찬가지로, 콤마 하나, 중괄호 하나라도 JSON 양식과 어긋나면 그저 string 으로 들어올 것이다. JSON 으로 정확히 들어왔을 때에만, 다음과 같이 예쁜 콘솔로 찍히는 것이다.

이제 무엇을 하면 되는가? /views 안에 index.html 과 index.js 를 적절히 수정하여, 서버에서 받아온 JSON 을 잘 붙이기만 하면 된다.

먼저, 부트스트랩 CDN 부터 가져온다. css, js with popper 둘 다 가져오자.


```

17 <table class="table">
18   <thead>
19     <tr>
20       <th scope="col">#</th>
21       <th scope="col">message</th>
22       <th scope="col">level</th>
23       <th scope="col">timestamp</th>
24     </tr>
25   </thead>
26   <tbody class="log-table-body">
27     <tr>
28       <th scope="row">0</th>
29       <td>
30         <div class="alert alert-primary" role="alert">메세지0</div>
31       </td>
32       <td>레벨0</td>
33       <td>타임스탬프0</td>
34     </tr>
35 > <tr> ...
42   </tr>
43 > <tr> ...
50   </tr>
51 </tbody>

```

테이블을 만들 것이고, 세 줄 정도 하드코딩해서 테이블이 잘 만들어지는지 테스트부터 해볼 것이다.

로그 내용은 부트스트랩에서 제공하는 알람 처리했다.

line26 log-table-body 라는 클래스를 달아두었다.

index.js 에서 querySelector() 로 받아온 다음,

innerHTML 을 사용할 것이다.

line35 - 50 이것은 그저 line27 - 34 를 반복해서 총 세 개의 하드코딩된 데이터를 보기 위한 용도라 접어두었다.

결과:

#	message	level	timestamp
0	메세지0	레벨0	타임스탬프0
1	메세지0	레벨1	타임스탬프1
2	메세지0	레벨2	타임스탬프2

하드코딩된 테이블이 잘 나오는것이 확인된다.

이제, 이 안에 실제 로그 데이터를 넣기만 하면 된다.

```
4 const logTableBody = document.querySelector(".log-table-body");
```

<tbody> 에 해당하는 .log-table-body 를 가져오고,

```

36  const getData = async () => {
37      try {
38          await axios.post(url);
39          const response = await axios.get(url);
40          if (response.data) {
41              let trTags = "";
42              response.data.map((data, idx) => {
43                  let trTag = inputData(data, idx);
44                  trTags += trTag;
45              });
46              logTableBody.innerHTML = trTags;
47              changeAlertColor();
48          }
49      } catch (error) {
50          console.log(error);
51      }
52  };

```

line41 trTags 는 처음엔 빈 string 이다.

여기에, 태그를 쭉 이어서 붙인 다음 최종적으로 붙일 것이다.

line42 - 45 map 을 사용해 배열 순회한다.

idx 까지 파라미터로 받고,

inputData 라는 함수를 작동시켜 "하나의 tr 태그" 를 만들 것이다

line44 trTags 에 trTag 를 이어붙인다.

line46 만들어진 trTags 를 innerHTML 을 사용해,

기존 tbody 태그 안의 내용을 대체한다.

line47 changeAlertColor() 를 사용해, 에러 레벨 별 색깔을 다르게 만든다.

```

5  const inputData = (data, idx) => {
6    const sample = `
7      <tr>
8        <th scope="row">${idx}</th>
9        <td>
10         <div class="alert alert-primary" role="alert">${data.message}</div>
11        </td>
12        <td>${data.level}</td>
13        <td>${data.timestamp}</td>
14      </tr>
15    `;
16    return sample;
17  };

```

inputData 의 내용이다.

line5 파라미터로 data, index 를 받는다.

data - 서버로부터 받아온 배열 중 하나의 엘리먼트

idx - 엘리먼트에 해당하는 인덱스

line6 - 15 백틱으로, 넘어온 data 와 idx 에 해당하는 tr 태그를 만든다.

백엔드에서 받아온 데이터를 잘 파악해서, 원하는 값을 넣으면 된다.

line16 만든 태그를 리턴한다.

리턴된 sample 은 trTag 가 되어, trTags 에 이어붙여질 것이다.

마지막으로 알람별로 색깔을 다르게 만들어보면 다음과 같다.

```

19  const changeAlertColor = () => {
20    logTableBody.querySelectorAll(".alert").forEach((element) => {
21      if (element.innerHTML.includes("warn")) {
22        element.classList.remove("alert-primary");
23        element.classList.add("alert-warning");
24      }
25      if (element.innerHTML.includes("info")) {
26        element.classList.remove("alert-primary");
27        element.classList.add("alert-info");
28      }
29      if (element.innerHTML.includes("error")) {
30        element.classList.remove("alert-primary");
31        element.classList.add("alert-danger");
32      }
33    });
34  };

```

.alert 을 querySelectorAll 로 가져와서, forEach 를 사용해 각각을 검증해 색깔을 바꾼다.

만약, 해당 엘리먼트에 warn 이 포함되어 있다면, 즉 로그 내용이 warn 이라면,

부트스트랩에서 제공하는 alert-primary 를 클래스에서 빼버리고,

부트스트랩에서 제공하는 alert-warning 으로 대체한다.

이렇게, info, error 세 가지 경우에 교체를 진행한다.

결과는 다음과 같다.

#	message	level	timestamp
0	error 메시지	error	2022-05-20 15:26:13
1	warn 메시지	warn	2022-05-20 15:26:13
2	info 메시지	info	2022-05-20 15:26:13
3	error 메시지	error	2022-05-20 15:26:18
4	warn 메시지	warn	2022-05-20 15:26:18
5	info 메시지	info	2022-05-20 15:26:18
6	error 메시지	error	2022-05-20 15:26:19

심화과제

버튼을 총 3개 만든다.

warn, info, error

각각의 버튼을 눌렀을 때, 해당 로그만 보이도록 만들어보자.