






[project3] Node.js 로그

🕒 Created	@2022년 8월 2일 오후 5:00
🕒 Last Edited Time	@2023년 1월 11일 오후 2:09
📌 Type	Project Kickoff 🚀
📌 Status	
👤 Created By	 자룡 이
👤 Last Edited By	 온유 이
👥 Stakeholders	 온유 이  범성 허  최인호
📅 날짜	

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

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

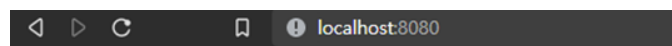
서버 구성 및 동작

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

```
$ npm i
$ 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

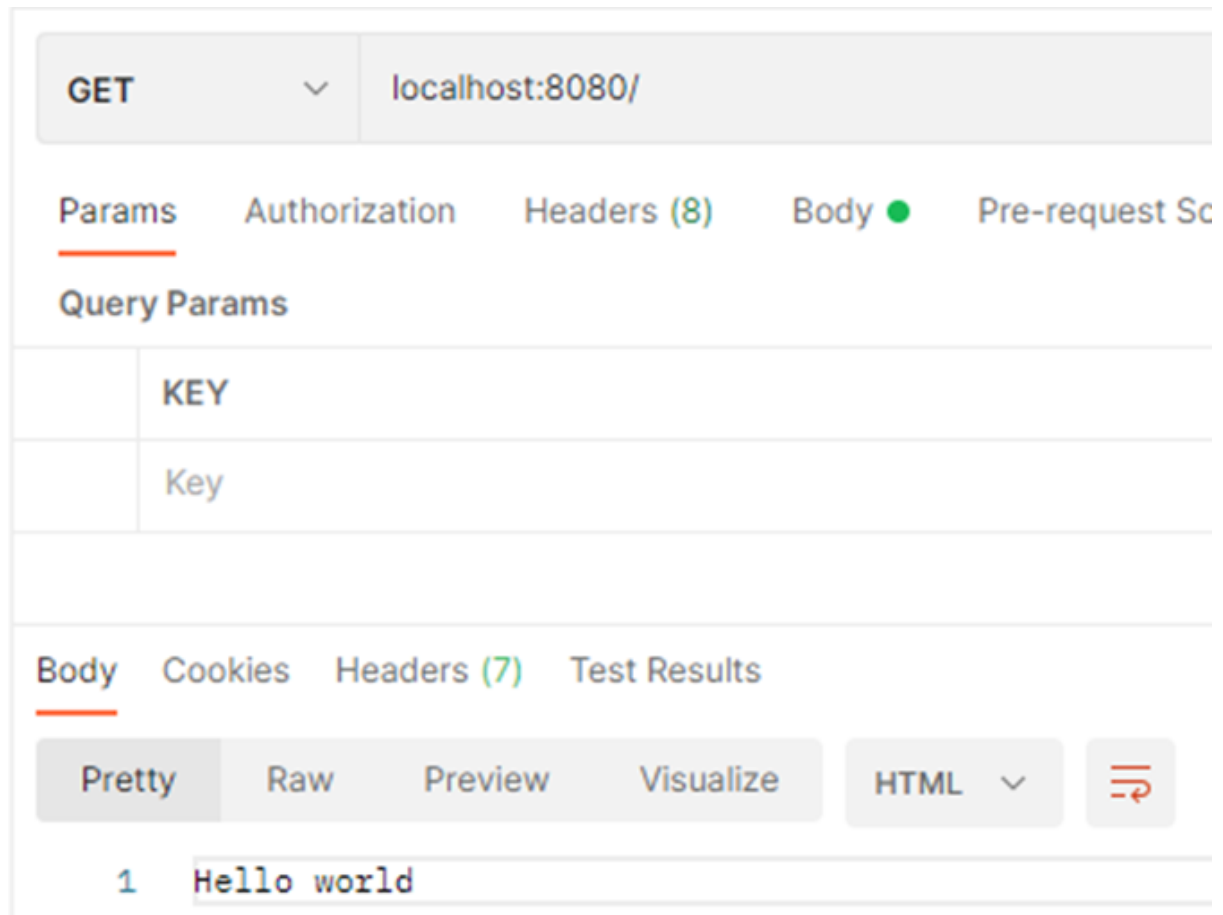


Hello world

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

y

통신 API 작성

`utils/` 디렉토리를 생성하고, `winston.js` 파일을 생성

```
const winston = require("winston");
require("winston-daily-rotate-file");

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

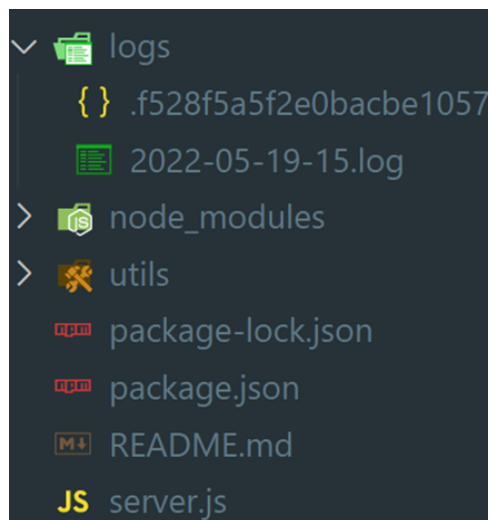
const logger = winston.createLogger({
  transports: [transport],
});

module.exports = { logger };
```

이제 `server.js` 로 옮겨가서,

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

코드를 추가하면,



지정한 형식대로 `logs/` 폴더가 자동 생성되며, `.log` 파일이 생성된다.
`server.js` 에서 로그를 만들어보자.

```
app.get("/", async (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"}
```

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, `error` 는 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("/", async (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, 예러는 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.

```
const winston = require("winston");
// 필요한 라이브러리 가져오기
// format 과 combine 사용
const { format } = require("winston");
const { combine } = format;
require("winston-daily-rotate-file");

const transport = new winston.transports.DailyRotateFile
```

그리고 포스트맨으로 한번 send 해주고 다시 로그 확인하면,

```
{ "level": "error", "message": "error 메시지", "timestamp": "2022-08-03T00:54:24.601Z" }
{ "level": "warn", "message": "warn 메시지", "timestamp": "2022-08-03T00:54:24.601Z" }
{ "level": "info", "message": "info 메시지", "timestamp": "2022-08-03T00:54:24.602Z" }
```

다음과 같이, 언제 로그가 발생했는지까지 찍힌다.

하지만, 이렇게 하면 읽기 어려우므로 로그를 직관적으로 바꿔보겠다.

```
const transport = new winston.transports.DailyRotateFile({
  ...
  maxFiles: "1d",
  format: combine(
    format.timestamp({ format: "YYYY-MM-DD HH:mm:ss" }),
    format.json()
  ),
});
```

`format.timestamp()` 의 파라미터로, 로그로 출력될 시간의 기준을 정해두겠다.

포스트맨으로 send 해보자.

```
{ "level": "error", "message": "error 메시지", "timestamp": "2022-08-03 09:58:50" }
{ "level": "warn", "message": "warn 메시지", "timestamp": "2022-08-03 09:58:50" }
{ "level": "info", "message": "info 메시지", "timestamp": "2022-08-03 09:58:50" }
```

위 세 줄은 포맷 적용 전,

아래 세 줄은 적용한 이후이다.

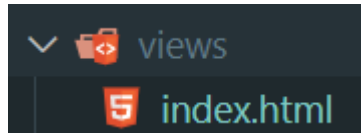
날짜와 시간까지 깔끔하게, 우리가 원하는 형태로 찍힌 것을 확인할 수 있다.

프론트 페이지 구성

`server.js` 에서 프론트 페이지를 구성하기 위해 다음을 추가한다.

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

`views/` 디렉토리를 만든 후, `index.html` 파일을 생성한다.



```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>log project</title>
  </head>
  <body>
    <h1>log project</h1>
  </body>
</html>
```

`<title>` 과 `<h1>` 하나만 추가한 간단한 형태다.

이제, log 를 받는 `route` 와 화면을 보여주는 `route` 를 구분하겠다.

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

app.get("/", async (req, res) => {
  ...
});

app.get("/api/logs", async (req, res) => {
  logger.error("error 메시지");
  logger.warn("warn 메시지");
  logger.info("info 메시지");
  logger.http("http 메시지");
  logger.debug("debug 메시지");
  res.json({
    success: true,
  });
});
```


따로 `get /` 일 경우의 `route` 를 지정하진 않았지만,

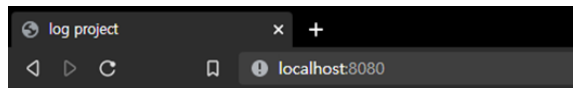
```
app.use(express.static(__dirname + "views"));
```

에 의해, 방금 제작한 `index.html` 이

`localhost:8080` 으로 접속 시에 보일 것이다.

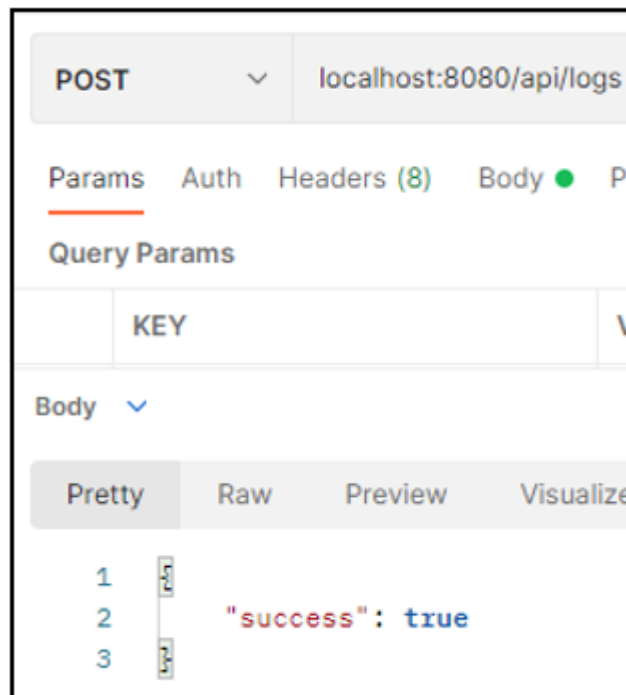
`POST /api/logs` 로 요청 시에 `log` 를 찍도록 했고,

성공 시 `{ success: true }` `JSON` 을 리턴하도록 처리했다.



log project

`localhost:8080` 으로 접속한 경우



`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>
```

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

다음과 같이, 콘솔로그만 찍어보겠다.

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>
```

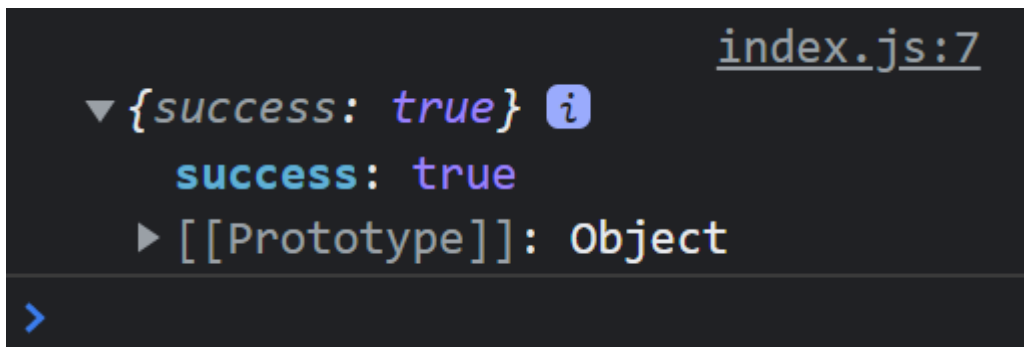
`index.js` 를 다음과 같이 수정한다.

```
const url = "http://localhost:8080/api/logs";

const getData = async () => {
  try {
    const response = await axios.get(url);
    if (response.data) {
      console.log(response.data);
    }
  } catch (error) {
    console.log(error);
  }
};

getData();
```

`GET /api/logs` 로 요청할 것이고,
비동기 방식으로 `JSON` 데이터를 가져온다.



로그에, `{ success: true }` 가 잘 찍힌 것이 확인된다.

즉, 프론트엔드와 백엔드의 연결 성공이다.

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

로그는 현재 어떤 형태인가?

```
{ "level": "error", "message": "error 메시지", "timestamp": "2022-08-03 10:48:06" }
{ "level": "warn", "message": "warn 메시지", "timestamp": "2022-08-03 10:48:06" }
{ "level": "info", "message": "info 메시지", "timestamp": "2022-08-03 10:48:06" }
```

이건 `object` 나 `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 만들자.

```
const insert = (str, index, target) => {
  const front = str.slice(0, index);
  const back = str.slice(index, str.length);
  return front + target + back;
};

let retData = {};
app.post("/api/logs", (req, res) => {
  logger.error("error 메시지");
  logger.warn("warn 메시지");
  logger.info("info 메시지");
  logger.http("http 메시지");
  logger.debug("debug 메시지");
  fs.readFile("./logs/2022-08-03-10.log", "utf8", (err, data) => {
    retData = data;
    let idx = -1;
    while (1) {
      idx = retData.indexOf("}", idx + 1);
      if (idx === -1) {
        break;
      }
      retData = insert(retData, idx + 1, ",");
    }
    retData = "[" + retData.slice(0, retData.length - 3) + "]";
    retData = JSON.parse(retData);
    console.log(retData);
  });
  return res.json({
    success: true,
  });
});
```

우선, `POST /api/logs` 했을 때 결과부터 보자.

```
[
  {
    level: 'error',
    message: 'error 메시지',
```

```

    timestamp: '2022-08-03 10:56:40'
  },
  {
    level: 'warn',
    message: 'warn 메시지',
    timestamp: '2022-08-03 10:56:40'
  },
  {
    level: 'info',
    message: 'info 메시지',
    timestamp: '2022-08-03 10:56:40'
  }
]

```

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.

```

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

```

`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":"2022-08-03 10:48:06"}
{"level":"warn","message":"warn 메시지","timestamp":"2022-08-03 10:48:06"}
{"level":"info","message":"info 메시지","timestamp":"2022-08-03 10:48:06"}

```

객체가 되려면 무조건 맨 뒤에 콤마 `,` 가 있어야 하는데 없다.

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

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.

```
fs.readFile("./logs/2022-08-03-11.log", "utf8", (err, data) => {  
  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 `5` 로 나오지 `05` 로 나오진 않는다. 그래서 앞에 `0` 을 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.

```
let idx = -1;  
while (1) {  
  idx = retData.indexOf("}", idx + 1);  
  if (idx === -1) {  
    break;  
  }  
  retData = insert(retData, idx + 1, ",");  
}
```

인덱스는 `-1` 로 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 것이기 때문에 `break` 다.

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

```
retData = insert(retData, idx + 1, ",");
```

그리고 파싱한 객체는 배열이 되어야한다. 왜냐면,

```
{ "level": "error", "message": "error 메시지", "timestamp": "2022-08-03 11:35:09" },  
{ "level": "warn", "message": "warn 메시지", "timestamp": "2022-08-03 11:35:09" },  
{ "level": "info", "message": "info 메시지", "timestamp": "2022-08-03 11:35:09" },  
{ "level": "error", "message": "error 메시지", "timestamp": "2022-08-03 11:36:09" },  
{ "level": "warn", "message": "warn 메시지", "timestamp": "2022-08-03 11:36:09" },
```

현재 형태는 객체 객체 객체의 연속인데, 배열로 감싸져야 한다. 그래서 앞뒤에 대괄호 `[]` 를 붙인 것이다.

그리고 맨 마지막 콤마 `,` 를 제거하는 작업이 필요하다. `JSON` 형태로 바꾼 다음 객체로 바뀌 버릴 예정이기 때문에, 맨 마지막 콤마 `,` 가 들어가면 에러 발생한다. 이를 위해 `slice()` 를 사용할 것이다.

슬라이스 무엇인가? 어디부터 어디까지.

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

그랬을 때 결과는 다음과 같다.

```
[{"level": "error", "message": "error 메시지", "timestamp": "2022-08-03 11:35:09"},  
{"level": "warn", "message": "warn 메시지", "timestamp": "2022-08-03 11:35:09"},  
{"level": "info", "message": "info 메시지", "timestamp": "2022-08-03 11:35:09"},  
{"level": "error", "message": "error 메시지", "timestamp": "2022-08-03 11:36:09"},  
{"level": "warn", "message": "warn 메시지", "timestamp": "2022-08-03 11:36:09"},  
{"level": "info", "message": "info 메시지", "timestamp": "2022-08-03 11:36:09"}]
```

정확히, 맨 마지막 엘리먼트의 콤마 `,` 가 제거되었고, 배열을 의미하는 대괄호 `[]` 로 감싸 졌다.

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

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

```
[
  {
    level: 'error',
    message: 'error 메시지',
    timestamp: '2022-08-03 10:56:40'
  },
  {
    level: 'warn',
    message: 'warn 메시지',
    timestamp: '2022-08-03 10:56:40'
  },
  {
    level: 'info',
    message: 'info 메시지',
    timestamp: '2022-08-03 10:56:40'
  }
]
```

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.

```
const url = "http://localhost:8080/api/logs";

const getData = async () => {
  try {
    await axios.post(url);
    const response = await axios.get(url);
    if (response.data) {
      console.log(response.data);
    }
  } catch (error) {
    console.log(error);
  }
};
```



```
getData();
```

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,

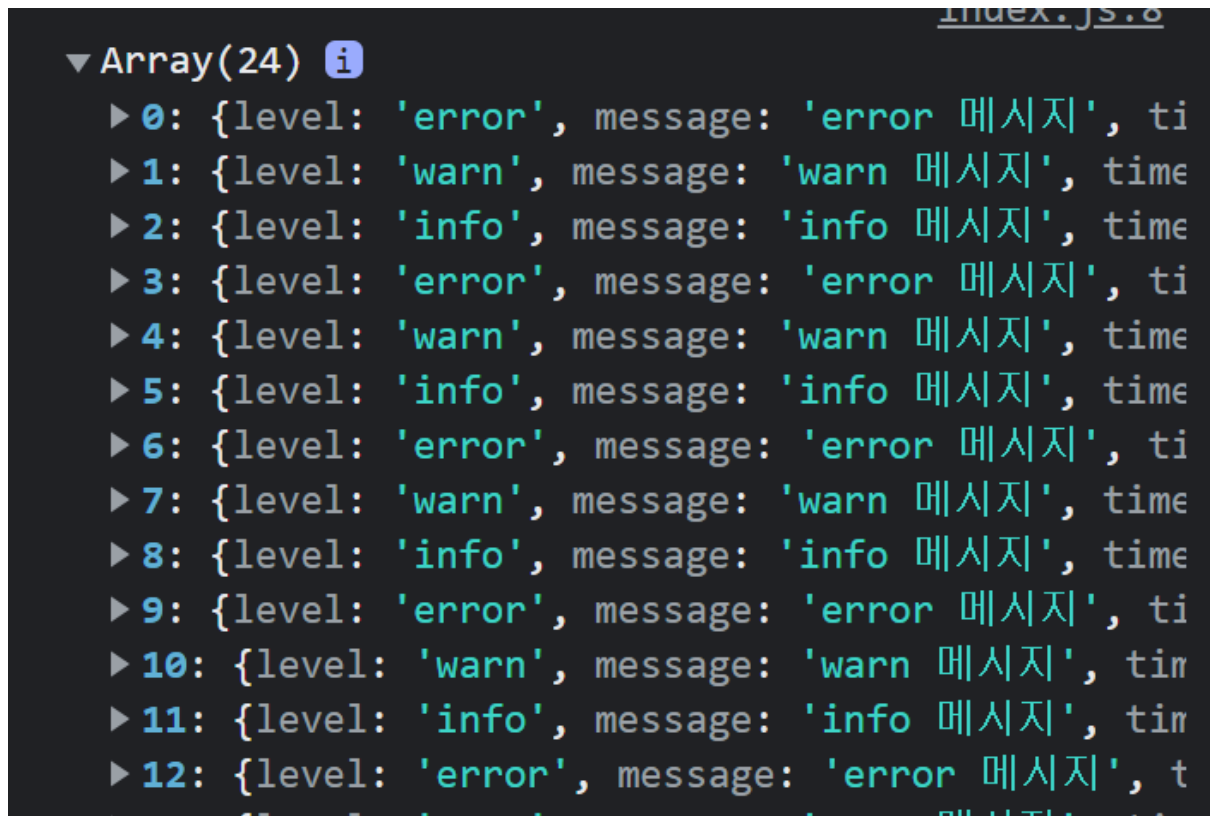
`POST` 먼저 실행해서 `retData` 를 파싱된 로그로 채움

`GET` 실행하여 파싱 성공한 `retData` 가져옴

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

```
app.get("/api/logs", async (req, res) => {  
  res.json(retData);  
});
```

그리고, 크롬 키고 `localhost:8080` 에 접속하여 `console.log` 확인해보면,



```
▼ Array(24) ⓘ  
  ▶ 0: {level: 'error', message: 'error 메시지', ti  
  ▶ 1: {level: 'warn', message: 'warn 메시지', time  
  ▶ 2: {level: 'info', message: 'info 메시지', time  
  ▶ 3: {level: 'error', message: 'error 메시지', ti  
  ▶ 4: {level: 'warn', message: 'warn 메시지', time  
  ▶ 5: {level: 'info', message: 'info 메시지', time  
  ▶ 6: {level: 'error', message: 'error 메시지', ti  
  ▶ 7: {level: 'warn', message: 'warn 메시지', time  
  ▶ 8: {level: 'info', message: 'info 메시지', time  
  ▶ 9: {level: 'error', message: 'error 메시지', ti  
  ▶ 10: {level: 'warn', message: 'warn 메시지', tir  
  ▶ 11: {level: 'info', message: 'info 메시지', tir  
  ▶ 12: {level: 'error', message: 'error 메시지', t  
  ▶ 13: {level: 'warn', message: 'warn 메시지', tim
```

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, `Bootstrap` `CDN` 부터 가져온다. `css` , `js with popper` 둘 다 가져오자.

```
<table class="table">
  <thead>
    <tr>
      <th class="col">#</th>
      <th class="col">message</th>
      <th class="col">level</th>
      <th class="col">timestamp</th>
    </tr>
  </thead>
  <tbody class="log-table-body">
    <tr>
      <th scope="row">0</th>
      <td>
        <div class="alert alert-primary" role="alert">메세지0</div>
      </td>
      <td>레벨0</td>
      <td>타임스탬프0</td>
    </tr>
    <tr>
      <th scope="row">1</th>
      <td>
        <div class="alert alert-primary" role="alert">메세지1</div>
      </td>
      <td>레벨1</td>
      <td>타임스탬프1</td>
    </tr>
    <tr>
      <th scope="row">2</th>
      <td>
        <div class="alert alert-primary" role="alert">메세지2</div>
      </td>
      <td>레벨2</td>
      <td>타임스탬프2</td>
    </tr>
  </tbody>
</table>
```

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

로그 내용은 `Bootstrap` 에서 제공하는 `alert` 처리했다.

#	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.

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

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

```
const getData = async () => {
  try {
    await axios.post(url);
    const response = await axios.get(url);
    if (response.data) {
      let trTags = "";
      response.data.map((data, idx) => {
        let trTag = inputData(data, idx);
        trTags += trTag;
      });
      logTableBody.innerHTML = trTags;
      changeAlertColor();
    }
  } catch (error) {
    console.log(error);
  }
};
```

`trTags` 는 처음엔 빈 `string` 이다.

여기에, 태그를 쭉 이어서 붙인 다음 최종적으로 붙일 것이다.

`map()` 을 사용해 배열 순회한다. `idx` 도 사용할 것이다.

순회하면서, `inputData()` 라는 함수를 작동시켜 "하나의 `<tr>` 태그" 를 만들 것이다. 바로 다음 코드에 설명이 나온다.

`inputData()` 리턴값으로 `trTag` 를 받고, `trTags` 에 `trTag` 를 이어붙인다.

만들어진 `trTags` 를 `innerHTML` 을 사용해,

기존 `<tbody>` 태그 안의 내용을 대체한다.

`changeAlertColor()` 를 사용해, 에러 레벨 별 색깔을 다르게 만드는데, 조금 있다가 살펴보자.

`inputData()`

```
const inputData = (data, idx) => {
  const sample = `
    <tr>
      <th scope="row">${idx}</th>
      <td>
        <div class="alert alert-primary" role="alert">${data.message}</div>
      </td>
      <td>${data.level}</td>
      <td>${data.timestamp}</td>
    </tr>
  `;
  return sample;
};
```

파라미터로 `data`, `idx` 를 받는다.

`data` - 서버로부터 받아온 배열 중 하나의 엘리먼트

`idx` - 엘리먼트에 해당하는 인덱스

백틱 ``으로, 넘어온 `data` 와 `idx` 에 해당하는 `<tr>` 태그를 만든다.

백엔드에서 받아온 데이터를 잘 파악해서, 원하는 값을 넣자.

이렇게 만든 태그 `sample` 을 리턴하는데, 이 리턴값은 `trTag` 가 되어, 기존 `trTag` 에 이어 붙여질 것이다.

마지막으로, 알람별로 색깔을 다르게 만들어보면 다음과 같다.

```
const changeAlertColor = () => {
  logTableBody.querySelectorAll(".alert").forEach((element) => {
    if (element.innerHTML.includes("warn")) {
      element.classList.remove("alert-primary");
      element.classList.add("alert-warning");
    }
    if (element.innerHTML.includes("info")) {
      element.classList.remove("alert-primary");
      element.classList.add("alert-info");
    }
  });
};
```

```

    if (element.innerHTML.includes("error")) {
      element.classList.remove("alert-primary");
      element.classList.add("alert-danger");
    }
  })
}

```

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

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

`Bootstrap` 에서 제공하는 `alert-primary` 를 `class` 에서 빼버리고,

`Bootstrap` 에서 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.