

삼성청년 SW·AI아카데미

Node.js_Firebase

<알림>

본 강의는 삼성청년SW·AI아카데미의 콘텐츠로
보안서약서에 의거하여
강의 내용을 어떠한 사유로도 임의로 복사, 촬영,
녹음, 복제, 보관, 전송하거나
허가 받지 않은 저장매체를
이용한 보관, 제3자에게 누설, 공개,
또는 사용하는 등의 행위를 금합니다.

Day3-1. Node 에서 MySQL 연동하기

챕터의 포인트

- express 웹 서버
- Node 에서 MySQL 연동하기

express 웹 서버

Node.js의 대표 웹 프레임워크

- 자체 Node.js 만으로도 웹 서버 구축이 가능하나 express를 쓰면 훨씬 간편하다. ex) 자바에서 스프링 쓰기
- http와 Connect 컴포넌트를 기반으로 만들어짐
- Node.js의 API들을 단순화
- 쉬운 확장성



Weekly Downloads

16,727,122

Version

4.17.1

License

MIT

Unpacked Size

208 kB

Total Files

16

Issues

92

Pull Requests

53

Homepage

expressjs.com/

Repository

github.com/expressjs/express

npm 패키지 생성

- 프로젝트 디렉터리 생성

ex) express-class

- VSCode 를 해당 디렉터리 기준으로 실행

- 터미널 열고, 다음 명령어 실행

npm init

- 모두 엔터 입력 후, package.json 파일 생성 확인

```
This utility will walk you through creating a package.json file.  
It only covers the most common items, and tries to guess sensible defaults.
```

```
See `npm help init` for definitive documentation on these fields  
and exactly what they do.
```

```
Use `npm install <pkg>` afterwards to install a package and  
save it as a dependency in the package.json file.
```

```
Press ^C at any time to quit.
```

```
package name: (express-class)
```

```
version: (1.0.0)
```

```
description:
```

```
entry point: (index.js)
```

```
test command:
```

```
git repository:
```

```
keywords:
```

```
author:
```

```
license: (ISC)
```

```
About to write to C:\Users\SSAFY\Desktop\work\express-class\package.json:
```

```
{  
  "name": "express-class",  
  "version": "1.0.0",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "author": "",  
  "license": "ISC",  
  "description": ""  
}
```

```
Is this OK? (yes)
```

```
PS C:\Users\SSAFY\Desktop\work\express-class>
```

package.json

- 패키지 명세서
- 이 파일이 존재한다면, 프로젝트는 "npm 패키지" 라고 할 수 있다.
- package.json 이 있다면, 다른 npm 패키지 설치 가능
- 다음 명령어를 입력해서, 패키지를 설치해보자.

npm install express morgan mysql2

- 각각의 패키지 용도
 - express : 웹서버
 - morgan : 로그 라이브러리
 - mysql2 : MySQL 연결

```
{
  "name": "express-class",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "description": ""
}
```


dependencies

- **package.json** 에 **dependencies** 객체가 추가됨
- **dependencies** (의존성)
 - 현재 패키지가 의존하는 하는 다른 패키지
 - 즉, 이 패키지는 **dependencies** 에 명시된 패키지가 함께 설치되어야만 동작
- **node_modules**
 - **node_modules** 라는 새로운 디렉터리가 생성됨
 - 굉장히 많은 패키지가 설치된 것을 확인할 수 있다.
 - **express**, **morgan**, **mysql2** 확인 가능
 - 왜 세 가지 패키지 이외의 패키지도 많이 깔렸을까?
세 가지 패키지도 다른 패키지들에 의존하기 때문

```
"dependencies": {  
  "express": "^4.19.2",  
  "morgan": "^1.10.0",  
  "mysql2": "^3.10.2"  
}
```

package-lock.json

- **열어보면 1000 줄이 넘어가는 파일**
 - package.json 만으로 충분하지 않다.
 - package.json 요약 명세서
 - package-lock.json 상세 명세서
 - express 역시도 하나의 패키지이므로, "다른 패키지들" 을 dependencies 로 가지고 있다.
 - 다른 개발 PC 또는 서버 컴퓨터로 우리가 만든 패키지(node-practice) 를 옮겼을 때,
 - 원래 개발 PC 와 완벽하게 동일한 환경에서 우리 패키지를 동작시키기 위함
 - 즉, 협업 시엔 package-lock.json 까지 공유

```
1015         "integrity": "sha512-BNGbV  
1016     }  
1017 }  
1018 }  
1019 |
```

global install (전역 설치)

- package.json 이 위치한 프로젝트 디렉터리 이외에도,
- 어떤 경로에서든 패키지 호출 가능
- 커멘드라인에서 실행하는 프로그램은 전역 설치가 기본
- `npm install --location=global nodemon`
 - nodemon 전역 설치
- `npm list --location=global --depth=0`
 - 전역 패키지 확인

```
C:\Users\mincoding\Desktop\node-test>npm list --location=global --depth=0
npm WARN config global `--global`, `--local` are deprecated. Use `--location=global` instead.
C:\Users\mincoding\AppData\Roaming\npm
└─ nodemon@2.0.16
```

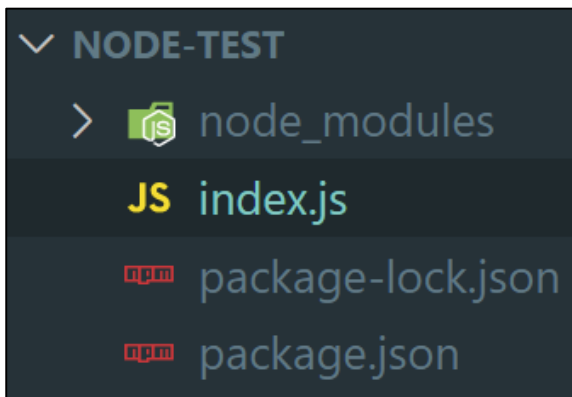
방금 전역 설치한 nodemon 이란?

- 서버 코드 수정 시마다 매번 서버를 재부팅해야하는 불편함
- nodemon 은 코드가 수정되면 자동으로 서버를 재시작



샘플 코드 작성하기

- 간단한 express code를 작성하기.
- index.js 생성



```
const express = require("express");
const morgan = require("morgan");

const app = express();
const PORT = 8080;

app.use(express.json());
app.use(morgan("dev"));

app.get("/api/info", (req, res) => {
  return res.json({
    name: "david",
    job: "tutor",
  });
});

app.listen(PORT, () => console.log(`this application is running in ${PORT}`));
```

```
// express 모듈 가져옴
const express = require("express");
// 로그 라이브러리
const morgan = require("morgan");
// express 함수 호출 후 app 객체 생성
const app = express();
// PORT 상수
const PORT = 8080;
// JSON 사용
app.use(express.json());
// morgan 개발 모드
app.use(morgan("dev"));
```

코드 분석

- 사용자가 `/api/info` 로 http 요청을 보내면,
JSON Format 으로 name 과 job 을 응답
- `get`
 - http 요청의 한 종류
 - REST API 챕터에서 배움
- `"/`
 - API 라우트 경로
다음 장 Routing 에서 배움
- `req, res`
 - 요청객체 (request), 응답객체 (response)
- `return res.json()`
 - JSON Format 으로 응답 객체 res 에 name, job 을 실어서 보낸다.

```
app.get("/api/info", (req, res) => {  
  return res.json({  
    name: "david",  
    job: "tutor",  
  });  
});
```

마지막 줄에서, 서버를 작동시킨다.

- **listen**
 - 서버 요청 대기 상태
- **PORT 8080**
 - 즉, express 서버는 실행과 동시에 8080 포트에서 클라이언트 요청(request) 대기중
- **두번째 파라미터 콜백함수**
 - listen 성공 시 실행. 터미널에 어느 포트에서 서비스되고있는지 안내하는 용도

```
app.listen(PORT, () => console.log(`this application is running in ${PORT}`));
```


express 서버 실행하기

- nodemon ./index.js

```
C:\Users\mincoding\Desktop\node-test>nodemon index.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 index.js`
this server listening on 8080
```

8080 포트에 작동중이라는 안내문 확인

- localhost:8080/api/info
 - 접속 시, 작성해둔 객체 리턴
 - localhost 내 (local) 컴퓨터 (host)
 - /api/host 코드에서 지정한 경로
- 이렇게 활용되는 서버를 API 서버라고 한다.
 - 활용: 사용자 요청을 받아 SQL DB 접근 후
 - 결과를 JSON Format으로 리턴



요청과 응답

요청과 응답은 통신의 기본이다.

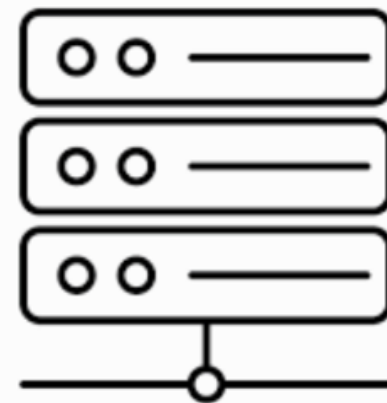


Client (Frontend)

GET /api/info

요청(request, req)

응답(response, res)



Server (backend)

```
localhost:8080/api/info
{
  "name": "david",
  "job": "tutor"
}
```

express의 req, res 알아보기

- req, res

- 각각 요청(req)과 응답(res)에 대한 정보 및, 활용 가능한 메서드를 가진 **거대한 객체**
- 각각 console.log 를 사용해서, 터미널에 출력해보자
- 오른쪽 이미지는 req 객체의 아주 작은 일부분
- 우린 어떻게 활용했는가?
 - res 객체 안에, json 함수를 사용해,
 - 객체를 JSON Format 으로 바꾼 후, 클라이언트로 보냄

```
app.get("/api/info", (req, res) => {  
  console.log(req);  
  // console.log(res);  
  return res.json({  
    name: "david",  
    job: "tutor",  
  });  
});
```

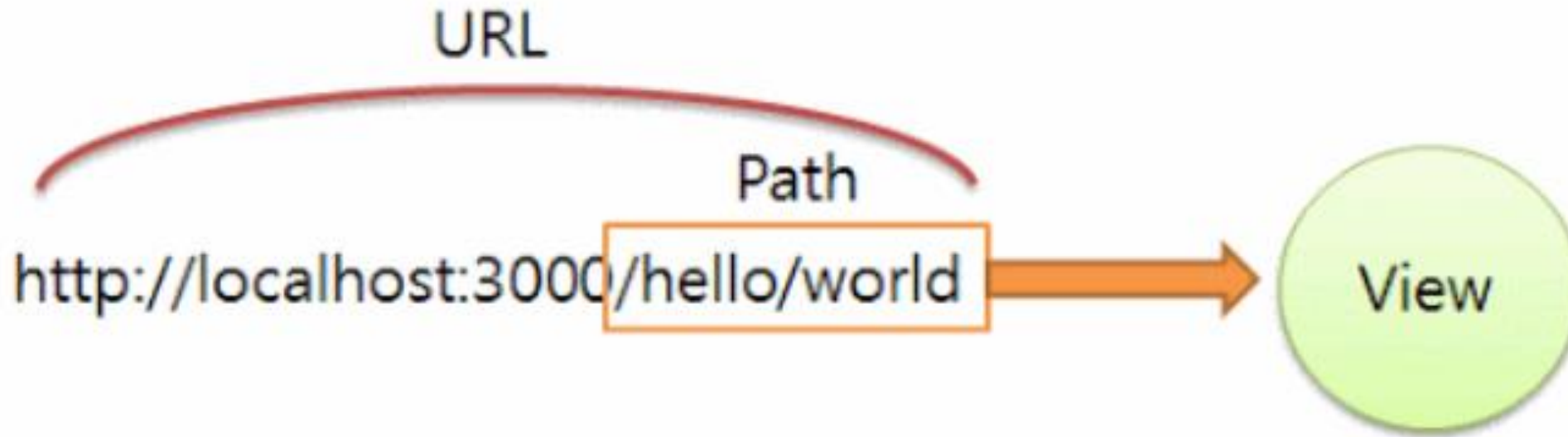
```
<ref *2> IncomingMessage {  
  _readableState: ReadableState {  
    objectMode: false,  
    highWaterMark: 16384,  
    buffer: BufferList { head: null,  
      length: 0,  
      pipes: [],  
      flowing: null,  
      ended: false,  
      endEmitted: false,  
      reading: false,  
      constructed: true,  
      sync: true,  
      needReadable: false,  
      emittedReadable: false,  
      readableListening: false,  
      resumeScheduled: false,  
      errorEmitted: false,  
      emitClose: true,  
      autoDestroy: true,  
      destroyed: false,  
      errored: null,  
      closed: false,
```

req 객체

Routing

URL 라우팅

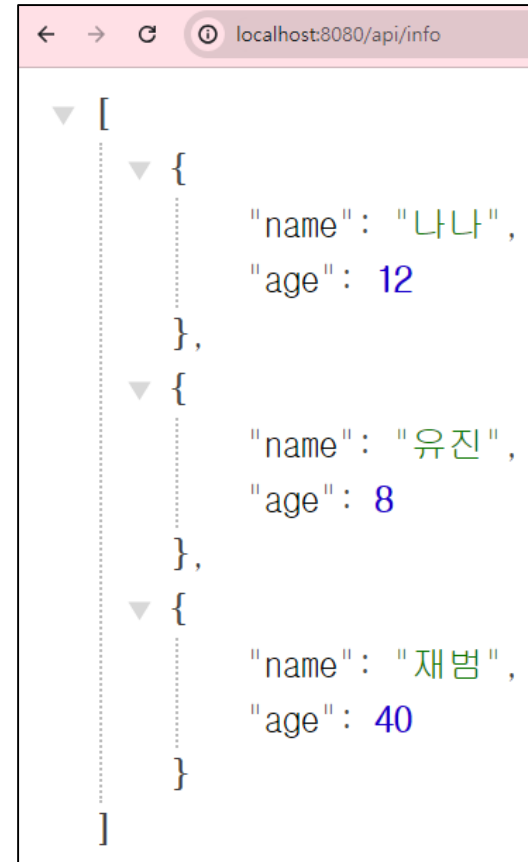
사용자가 접근한 경로에 따라서 그에 맞는 메소드를 호출해주는 기능



JSON을 리턴 해보기

- /info 에 요청을 보낼 시에 return res.json으로 배열을 리턴

```
const infos = [  
  {  
    name: "나나",  
    age: 12,  
  },  
  {  
    name: "유진",  
    age: 8,  
  },  
  {  
    name: "재범",  
    age: 40,  
  },  
];  
  
app.get("/api/info", (req, res) => {  
  return res.json(infos);  
});
```



배열 안에 동일한 타입의 객체를 정의

/info 접속 시, res.json() 을 사용해
배열을 JSON 형식으로 리턴

해당 경로 접속 시 명시된 리턴을 수행

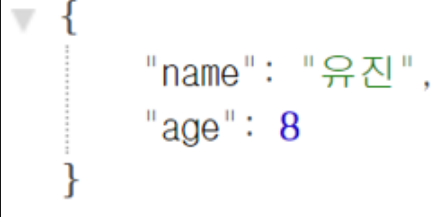
- **/api/info/names**
 - infos 중 이름만 파싱한 배열을 JSON 리턴
 - map 활용해보기
 - `Array.map((element) => { return element })`
- **/api/info/ages**
 - infos 중 나이만 파싱한 배열을 JSON 리턴
- **/api/info/0**
 - infos 배열의 0번째 리턴
- **/api/info/1**
 - infos 배열의 1번째 리턴
- **/api/info/2**
 - infos 배열의 2번째 리턴

```
const infos = [  
  {  
    name: "나나",  
    age: 12,  
  },  
  {  
    name: "유진",  
    age: 8,  
  },  
  {  
    name: "재범",  
    age: 40,  
  },  
];
```



A JSON array containing three string elements: "나나", "유진", and "재범". The array is enclosed in square brackets with a small downward arrow on the left.

/infos/names



A JSON object with two key-value pairs: "name": "유진" and "age": 8. The object is enclosed in curly braces with a small downward arrow on the left.

/infos/1

Node 에서 MySQL 연동하기

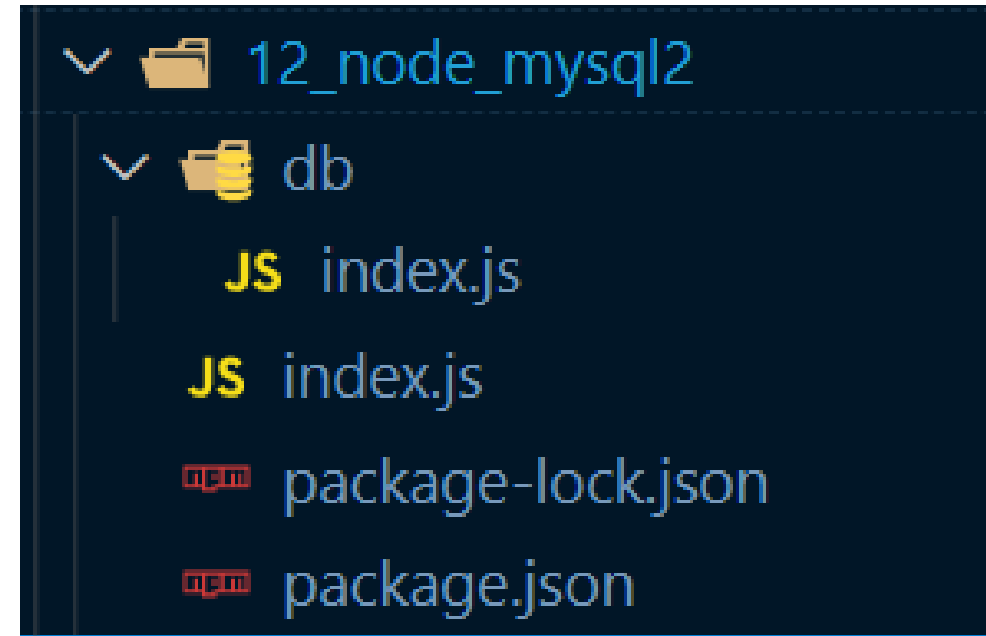
mysql2를 사용하는 이유

- node에서는 기본적으로 mysql에 접근하기 위해서는 라이브러리 사용이 필요
- mysql2라는 라이브러리를 사용한다.
 - 왜 mysql1 이 아니라 2를 사용할까?
 - mysql2부터 promise 방식이 사용 가능 및 개편된 버전
 - npm i mysql2

```
"dependencies": {  
  "cors": "^2.8.5",  
  "express": "^4.18.1",  
  "mysql2": "^2.3.3"  
}
```

폴더 구조 확인하기

- npm init 명령어 실행
- npm i mysql2 express cors
- db 폴더 생성 후 그 안에 index.js 생성



db 폴더 내 index.js를 수정

- **host**: 접속하기 위한 ip 주소 작성
 - localhost의 db에 접근하기 위해서는 localhost 나 127.0.0.1 을 작성
 - 외부 DB를 생성했기때문에 AWS의 주소를 작성
- **user**
 - DB에 접근하기 위해 생성해둔 ID
 - ssafy
- **password**
 - DB에 접근하기 위해 생성해둔 PASSWORD
 - ssafy_1234
- **database**
 - 접근하기 위한 DB(스키마) 의 이름

```
const mysql = require("mysql2/promise");

const pool = mysql.createPool({
  // aws ip
  host: "123.123.123.123",
  // mysql username
  user: "ssafy",
  // mysql user password
  password: "ssafy_1234",
  // db name
  database: "ssafy",
  waitForConnections: true,
  connectionLimit: 10,
  queueLimit: 0,
});

module.exports = { pool };
```

dotenv 활용

`npm i dotenv`

`.env` 파일은 반드시 `.gitignore` 에 등록

```
# dotenv environment variables file
.env

# Node.js build output
node_modules

# Log files
npm-debug.log*
yarn-debug.log*
yarn-error.log*

# IDE - VSCode
.vscode
```

```
DB_HOST=123.123.123.123
DB_USER=ssafy
DB_PASSWORD=ssafy_1234
DB_NAME=ssafy
```

```
require("dotenv").config();
const mysql = require("mysql2/promise");

const pool = mysql.createPool({
  // aws ip
  host: process.env.DB_HOST,
  // mysql username
  user: process.env.DB_USER,
  // mysql user password
  password: process.env.DB_PASSWORD,
  // db name
  database: process.env.DB_NAME,
  waitForConnections: true,
  connectionLimit: 10,
  queueLimit: 0,
});

module.exports = { pool };
```

DB 접속 코드 작성

- db 폴더에는 index.js로 서버 접속을 위한 셋팅이 되어있다.
- 해당 부분에서 pool 부분만 사용하기 위해 가져온 코드
 - `const { pool } = require('./db')`
 - db 폴더의 index.js 의 객체 에서 pool 내용만 따로 가져와서 사용하겠다.
 - `const db = require('./db');`
`const pool = db.pool` 과 같다

- `app.use(express.json())`
 - API 만들 시, get 이외의 http 메서드 사용에 필수

```
const express = require("express");
const { pool } = require("./db");
const app = express();
const PORT = 8080;
const cors = require("cors");
app.use(cors());
app.use(express.json());
app.get("/", (req, res) => {
  return res.json({
    name: "jony",
  });
});
app.listen(PORT, () => `this application is running in ${PORT}`);
```

DB 접속 코드 작성

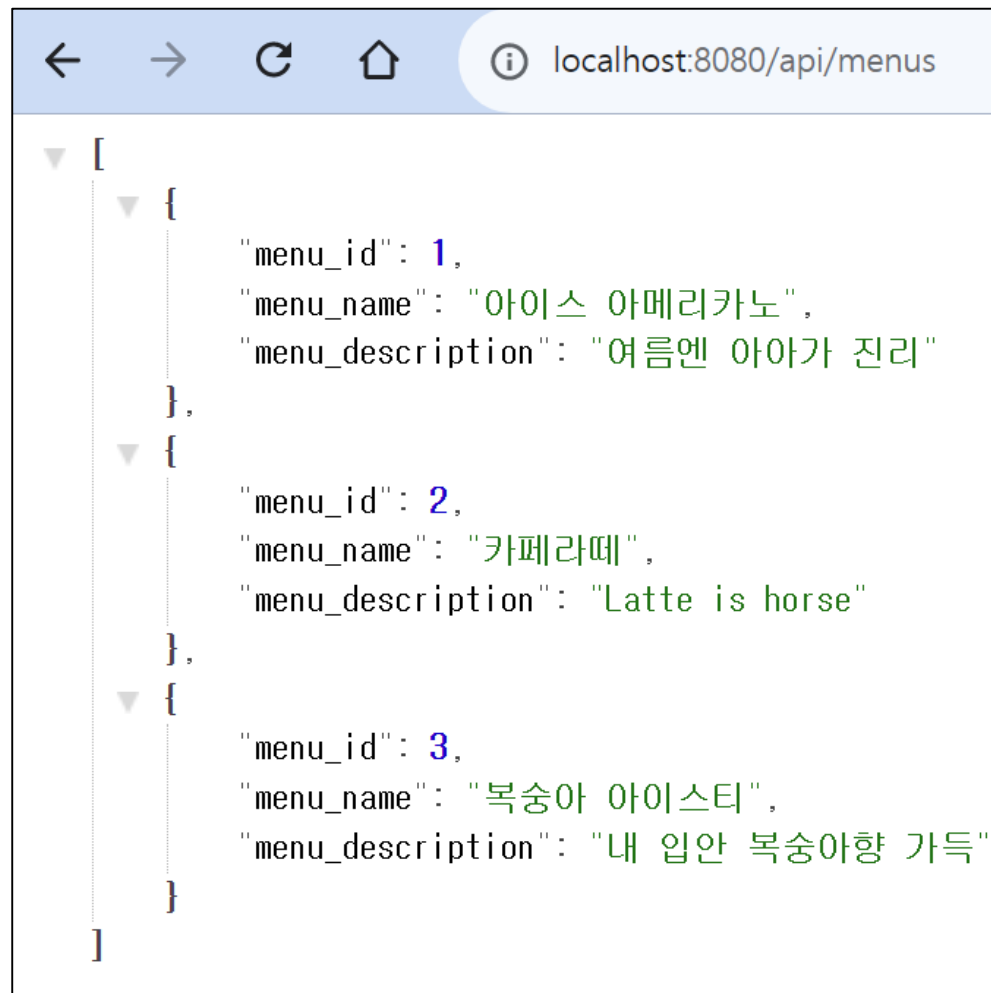
- **async await 활용**

- mysql2는 promise 형식으로 값을 리턴하기 때문에 async/await 활용이 가능
- pool.query 안에 SQL 쿼리문을 작성한다.
- 작성후 첫번째 배열을 리턴

```
app.get("/api/menus", async (req, res) => {  
  try {  
    const data = await pool.query("SELECT * FROM menus");  
    if (data[0]) {  
      return res.json(data[0]);  
    }  
  } catch (error) {  
    return res.json(error);  
  }  
});
```


결과 확인

- localhost:8080/api/menus 접근하기
- 해당 정보를 리턴한다면 성공



```
[
  {
    "menu_id": 1,
    "menu_name": "아이스 아메리카노",
    "menu_description": "여름엔 아아가 진리"
  },
  {
    "menu_id": 2,
    "menu_name": "카페라떼",
    "menu_description": "Latte is horse"
  },
  {
    "menu_id": 3,
    "menu_name": "복숭아 아이스티",
    "menu_description": "내 입안 복숭아향 가득"
  }
]
```

Day3-2. REST API

챕터의 포인트

- REST API 개념
- HTTP Response Status Code
- REST API 설계
- Postman
- params, query, body
- 정적 파일

REST API 개념

- 리모콘 "전원버튼" 누르기
TV 가 켜진다.
- 리모콘 "음량버튼" 을 누르기
 - 소리를 조절한다.
- 리모콘 "채널버튼" 을 누르기
 - 채널이 바뀐다.



인터페이스란?

- 사람은 “리모컨”만 있으면 TV 세부 사용법을 알 필요가 없다.
- 사람이 TV를 제어할 수 있도록 해주는 중간 다리 역할을 “인터페이스” 역할 이라고 한다.
- 사용자가 쉽게 동작 및 사용하는데 도움을 주는 시스템을 뜻한다.
HW Interface 은 물리적 접점을 뜻한다.
 - SW Interface 는 API 소스코드 접근 형태, 추상화 구현 등을 나타낸다.



API

- Application Programming Interface
- 즉, 인터페이스를 소스코드 형태로 구현한 것을 의미

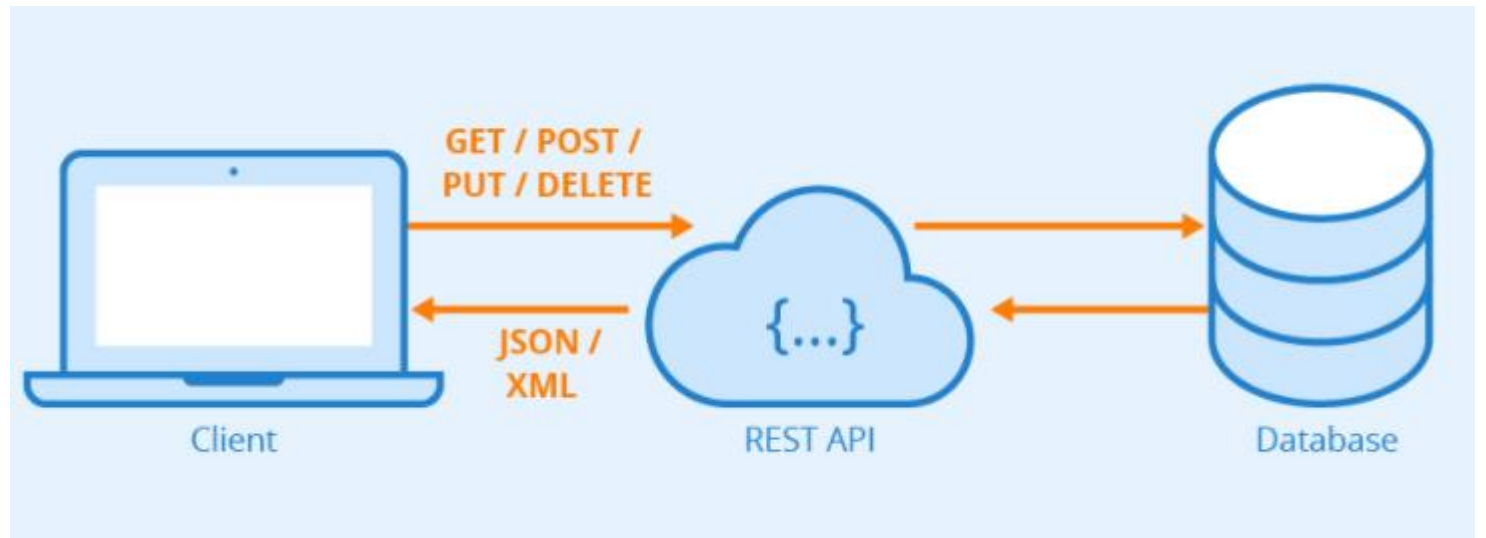
GET /api/menus

모든 메뉴 가져오기

```
localhost:8080/api/menus
[
  {
    "menu_id": 1,
    "menu_name": "아이스 아메리카노",
    "menu_description": "여름엔 아아가 진리",
    "menu_img_link": "/menus/ice-americano.jpg"
  },
  {
    "menu_id": 2,
    "menu_name": "카페라떼",
    "menu_description": "Latte is horse",
    "menu_img_link": "/menus/cafe-latte.jpg"
  },
  {
    "menu_id": 3,
    "menu_name": "복숭아 아이스티",
    "menu_description": "내 입안 복숭아향 가득",
    "menu_img_link": "/menus/peach-icetea.jpg"
  }
]
```

REST API(Representational State Transfer)

- 소프트웨어 개발 아키텍처의 한 형식
- 자원을 이름으로 구분하고 자원의 상태를 주고 받는 모든 것
- 일반적으로 REST라 하면 **HTTP를 통해 CRUD를 실행하는 API**



GET

- 데이터를 읽거나 검색을 할 때 주로 사용된다.
- URL에 데이터를 붙여서 보낸다.
- 캐싱이 가능하다.
- ex) 영화 목록 받아 오기, 검색 결과 확인하기

DELETE

/status/{codes} Return status code or random status code if more than one are given

GET

/status/{codes} Return status code or random status code if more than one are given

PATCH

/status/{codes} Return status code or random status code if more than one are given

POST

/status/{codes} Return status code or random status code if more than one are given

PUT

/status/{codes} Return status code or random status code if more than one are given

POST

- 새로운 리소스를 생성할 때 사용한다.
- URL이 아닌 BODY 부분에 데이터를 넣어서 보낸다.
- ex) 게시물 작성하기

DELETE

/status/{codes} Return status code or random status code if more than one are given

GET

/status/{codes} Return status code or random status code if more than one are given

PATCH

/status/{codes} Return status code or random status code if more than one are given

POST

/status/{codes} Return status code or random status code if more than one are given

PUT

/status/{codes} Return status code or random status code if more than one are given

• PUT

- 전체 데이터를 변경 및 갱신할 때 사용한다
- ex) 회원 정보 수정하기, 게시글 수정하기

• PATCH

- 일부 데이터를 변경 할 때 사용한다
- 전체를 갱신하는 PUT과 다르게 일부를 수정하기때문에 UPDATE 부분에 더 적합하다

DELETE	/status/{codes}	Return status code or random status code if more than one are given
GET	/status/{codes}	Return status code or random status code if more than one are given
PATCH	/status/{codes}	Return status code or random status code if more than one are given
POST	/status/{codes}	Return status code or random status code if more than one are given
PUT	/status/{codes}	Return status code or random status code if more than one are given

DELETE

- 리소스를 삭제 할 때 사용한다.
- ex) 게시물 삭제하기

DELETE

/status/{codes} Return status code or random status code if more than one are given

GET

/status/{codes} Return status code or random status code if more than one are given

PATCH

/status/{codes} Return status code or random status code if more than one are given

POST

/status/{codes} Return status code or random status code if more than one are given

PUT

/status/{codes} Return status code or random status code if more than one are given

HTTP Response Status Code

- 1xx(Information)

- 요청을 받았으며 프로세스를 진행하는 상태
- 100(요청 후 대기중)

- 2xx(Success)

- 요청을 성공적으로 받았으며 인식했고 진행한 상태
- 200(요청 성공)
- 201(생성 완료)

- 3xx(Redirection)

- 클라이언트는 요청을 마치기 위해 추가적으로 동작을 취해줘야 한다.
- 301(새위치로 영구적으로 이동)
- 302(임시이동)

- 4xx(Client Error)

- **클라이언트** 단에서 에러가 발생한 경우 해당 오류가 발생한다.
- 400(잘못된 요청)
- 401(권한 없음)
- 404(찾을 수 없음)

5xx(Server Error)

- 서버 측에 에러가 발생 했을 경우 해당 상태코드가 나타난다.
- 500(내부 서버 오류)
- 503(서버를 사용 할 수 없음)
- 504(시간 초과)

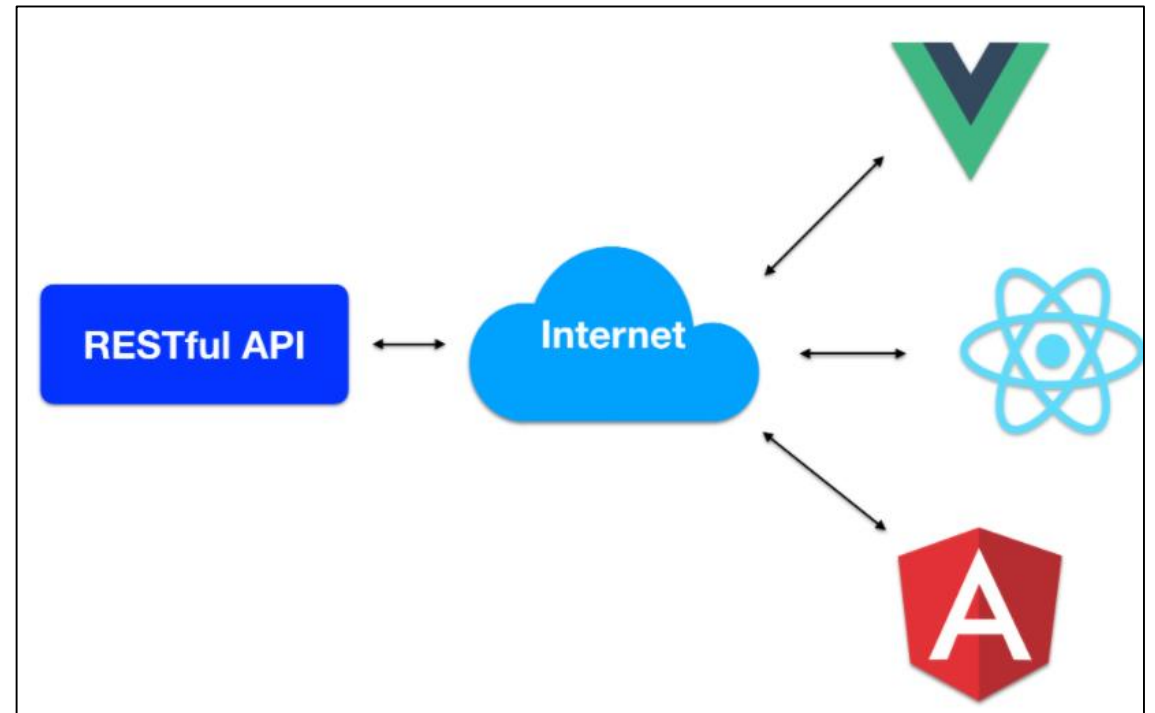
정리

- 가장 많이 보게 되는 상태 코드
 - 200번대
 - 응답이 성공
 - 400번대
 - 클라이언트 이슈(보통 대부분의 에러가 본인에 의해서 나온다.)
 - 500번대
 - 서버 이슈

REST API 설계

RESTful API

- REST 원리를 따르는 시스템
- 이해하기 쉽고 사용하기 쉬운 REST API를 만드는것
- REST API 설계법에 근거한 REST API 설계하기



잘못된 API 설계법

- 유저 정보 관련 설계하기
 - 유저 전체 정보 조회
 - GET /users/all-get
 - 유저 등록
 - POST /users/adduser
 - 특정 유저 조회
 - GET /users/:id/get-information
 - 특정 유저 수정
 - PATCH /users/:id/update
 - 특정 유저 삭제
 - DELETE /users/:id/delete

좋은 API 설계하기

- **유저 정보 조회**

- 유저 전체 정보 조회
 - GET /users
- 유저 등록
 - POST /users
- 특정 유저 조회
 - GET /users/:id
- 특정 유저 수정
 - PATCH /users/:id
- 특정 유저 삭제
 - DELETE /users/:id
- 대상에 대한 행동은 모두 HTTP REQUEST METHOD로 표시한다.
 - 행동과 리소스를 구별해서 설계하는것이 핵심이다.
 - 리소스 : 유저
 - 행동: 조회(GET), 등록(POST), 수정(PATCH), 삭제(DELETE)

API 설계하기

- RESTful API 원칙에 따른 설계 해보기
- 게시물 /boards
 - 게시물 작성
 - 게시물 목록(전체)
 - 게시물 상세 조회
 - 게시물 수정
 - 게시물 삭제
- 댓글 /comments
 - 1번 boards 에 대한 댓글 작성/조회
 - 댓글 수정
 - 댓글 삭제

POST 주요 코드

- `app.use(express.json())`
 - 클라이언트에서 JSON Format 의 형태로 서버에 요청을 보낼수 있도록 해주는 설정
- `req.body`
 - POST 요청시 http body로부터 받아오는 내용

```
app.post("/api/menus", async (req, res) => {  
  try {  
    const data = await pool.query(  
      "INSERT INTO menus (menu_name, menu_description) VALUES (?, ?)",  
      [req.body.menu_name, req.body.menu_description]  
    );  
    return res.json(data);  
  } catch (error) {  
    return res.json(error);  
  }  
});
```


일반 문자열로 넣기

- INSERT INTO menus에 백틱을 활용해서 변수 형태로 한꺼번에 넣는 방법

```
const data = await pool.query(`
  INSERT INTO menus (menu_name, menu_description)
  VALUES
  ("${req.body.menu_name}", "${req.body.menu_description}")
`);
```

? 활용하기

- ?가 들어간 부분들은 pool.query("쿼리문", [?에 들어갈 내용]) 식으로 변경이 가능하다.
- 문자열이 섞여서 복잡할 경우 아래와 같이 해결한다.

```
const data = await pool.query(`INSERT INTO menus (menu_name, menu_description) VALUES (?, ?)`,  
[req.body.menu_name, req.body.menu_description]);
```

API 서버에 접속하는 클라이언트를 만들기

- **강사만 진행**
 - 바탕화면에 클라이언트 생성

```
<h1>POST 테스트</h1>
<div>
  <label for="menu-name">메뉴이름</label>
  <input type="text" id="menu-name" />
</div>
<div>
  <label for="menu-description">메뉴설명
</label>
  <input type="text" id="menu-description" />
</div>
<button>메뉴 등록</button>
```

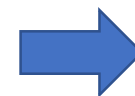
POST 테스트

메뉴이름	<input type="text" value="딸기라떼"/>
메뉴설명	<input type="text" value="딸기는 딸기딸기해"/>
<input type="button" value="메뉴 등록"/>	

API 서버에 접속하는 클라이언트 만들기

- axios.post로 post 요청을 보내면 서버도 post로 요청을 받는다.
- axios.post("주소", { 넘겨줄 객체값 })

```
<script src="https://cdn.jsdelivr.net/npm/axios@1.6.7/dist/axios.min.js"></script>
<script>
  const btn = document.querySelector("button");
  btn.addEventListener("click", async () => {
    const menuName = document.querySelector("#menu-name").value;
    const menuDescription =
      document.querySelector("#menu-description").value;
    const response = await axios.post("http://localhost:8080/api/menus", {
      menu_name: menuName,
      menu_description: menuDescription,
    });
    console.log(response.data);
  });
</script>
```



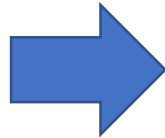
요청 페이로드 소스 보기

```
▼ {menu_name: "딸기", menu_description: "딸기는딸기딸기"}
  menu_description: "딸기는딸기딸기"
  menu_name: "딸기"
```

성공 후 console.log(response.data) 결과값 보기

- **affectedRows: 1**
 - DB에 영향을 받은 row의 수를 의미
 - 하나의 row만 추가했기때문에 1이 나오게 된다.
 - 알맞게 요청이 성공했다는 의미

```
▼ (2) [{...}, null] ⓘ  
  ▼ 0:  
    affectedRows: 1  
    fieldCount: 0  
    info: ""  
    insertId: 4  
    serverStatus: 2  
    warningStatus: 0  
    ▶ [[Prototype]]: Object  
  1: null  
  length: 2  
  ▶ [[Prototype]]: Array(0)
```



```
mysql> select * from menus;  
+-----+-----+-----+  
| menu_id | menu_name      | menu_description |  
+-----+-----+-----+  
| 1 | 아이스 아메리카노 | 여름엔 아아가 진리 |  
| 2 | 카페라떼       | Latte is horse |  
| 3 | 보숫아 아이스티 | 내 입안 보숫아향 가득 |  
| 4 | 딸기라떼       | 딸기는 딸기딸기해 |  
+-----+-----+-----+  
4 rows in set (0.00 sec)
```

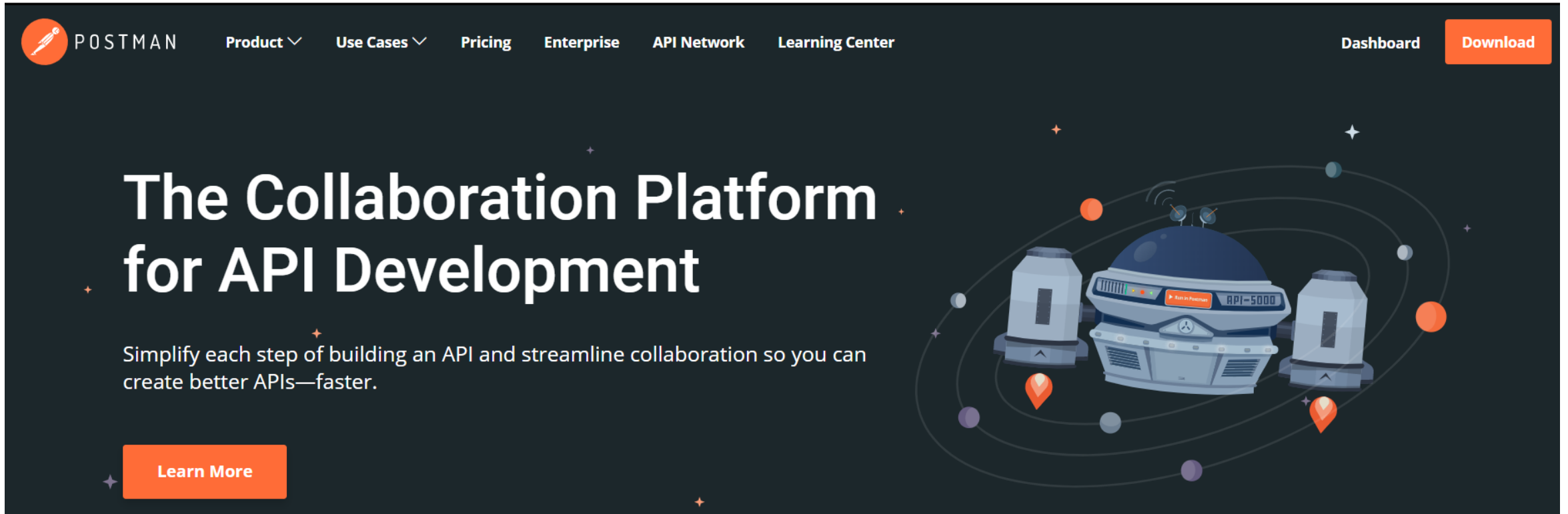
Postman

테스트를 위해 매번 클라이언트를 만드는 건 불편하다

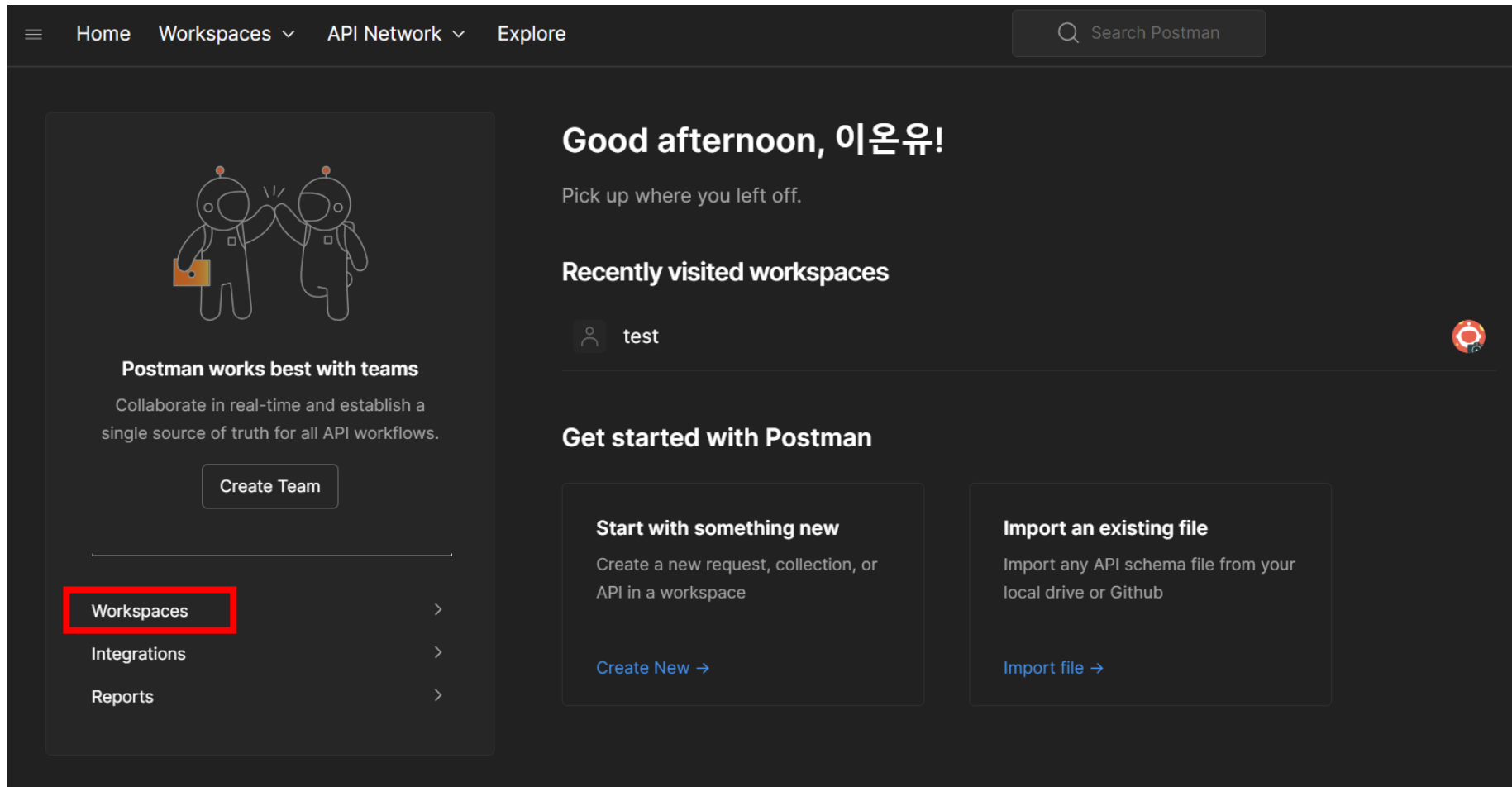
- GET 의 경우, 브라우저 접속만 하면 되지만
POST, PUT, PATCH, DELETE 테스트를 위해선 클라이언트 코드를 따로 작성해야 한다.
- 그러나, 서버 코드는 클라이언트 코드 없이도 테스트 가능해야 한다!

개발한 API를 테스트할 수 있는 플랫폼

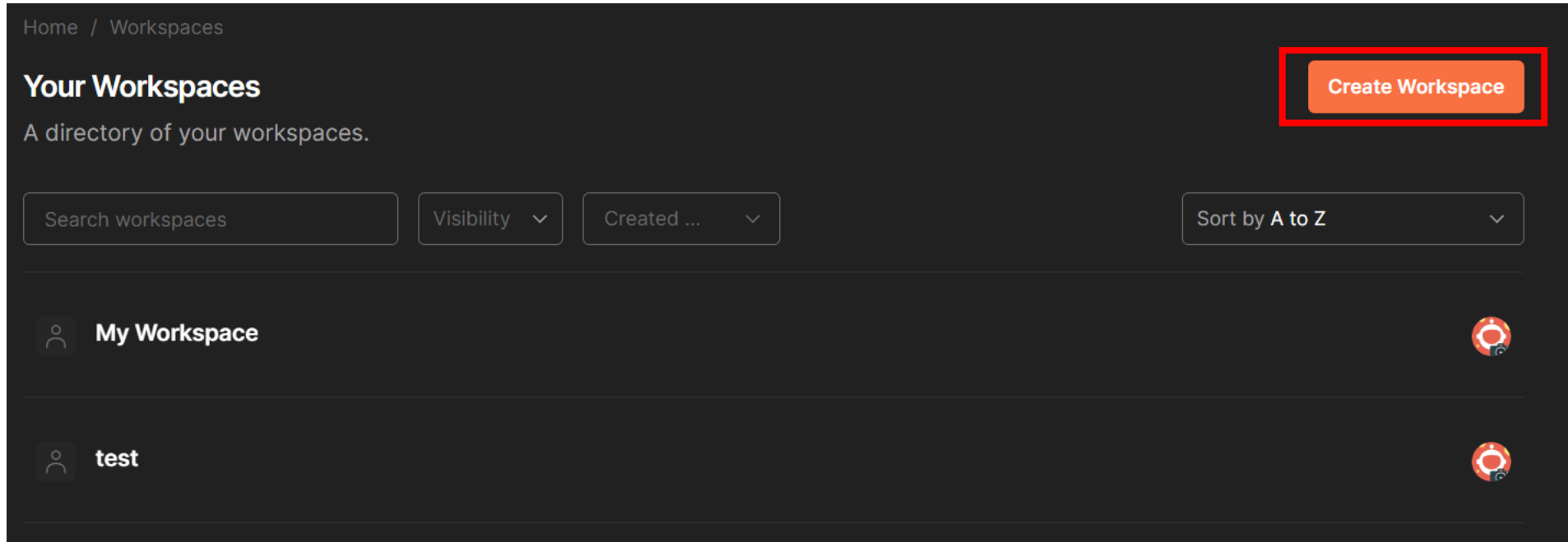
- <https://www.postman.com/>



Workspaces 클릭



Workspace 생성



Workspace 생성하기

Create workspace

Name

Summary

Add a brief summary about this workspace.

Visibility

Determines who can access this workspace.

☒ **Personal**
Only you can access

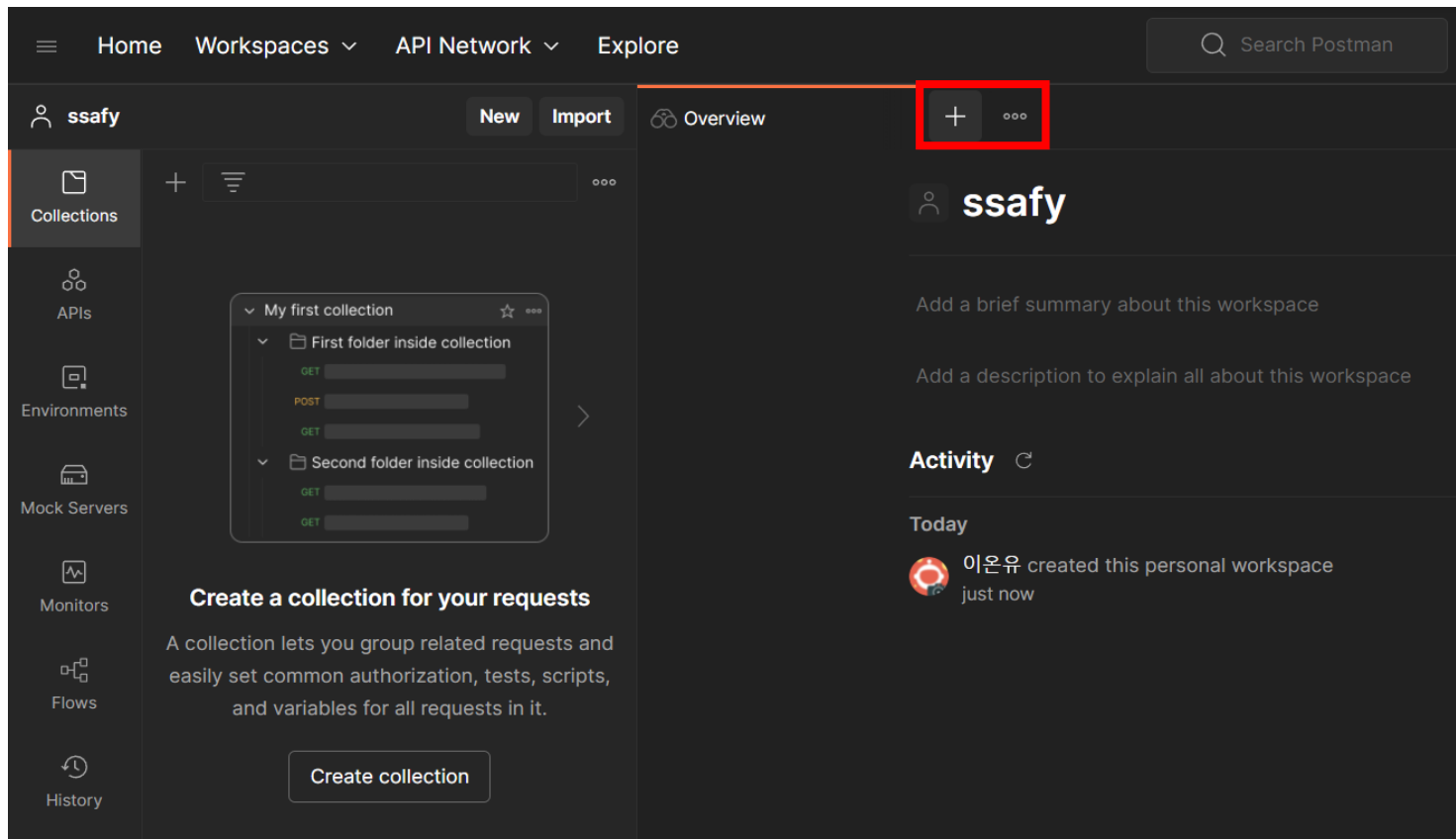
☐ **Private**
Only invited team members can access

☐ **Team**
All team members can access

☐ **Public**
Everyone can view

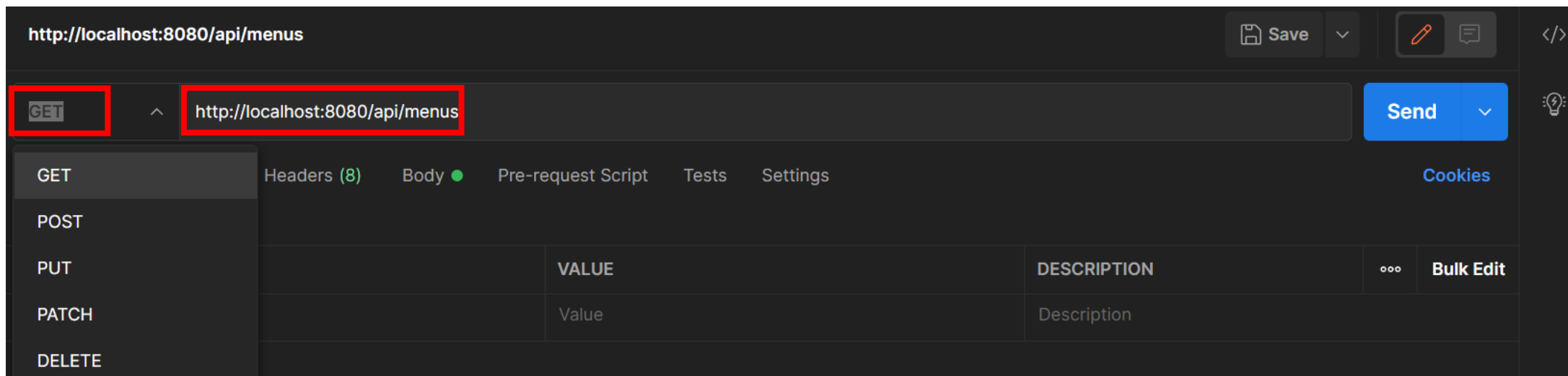
+버튼 누르기

- + 버튼을 눌러 기초 작업 환경을 구성해준다.



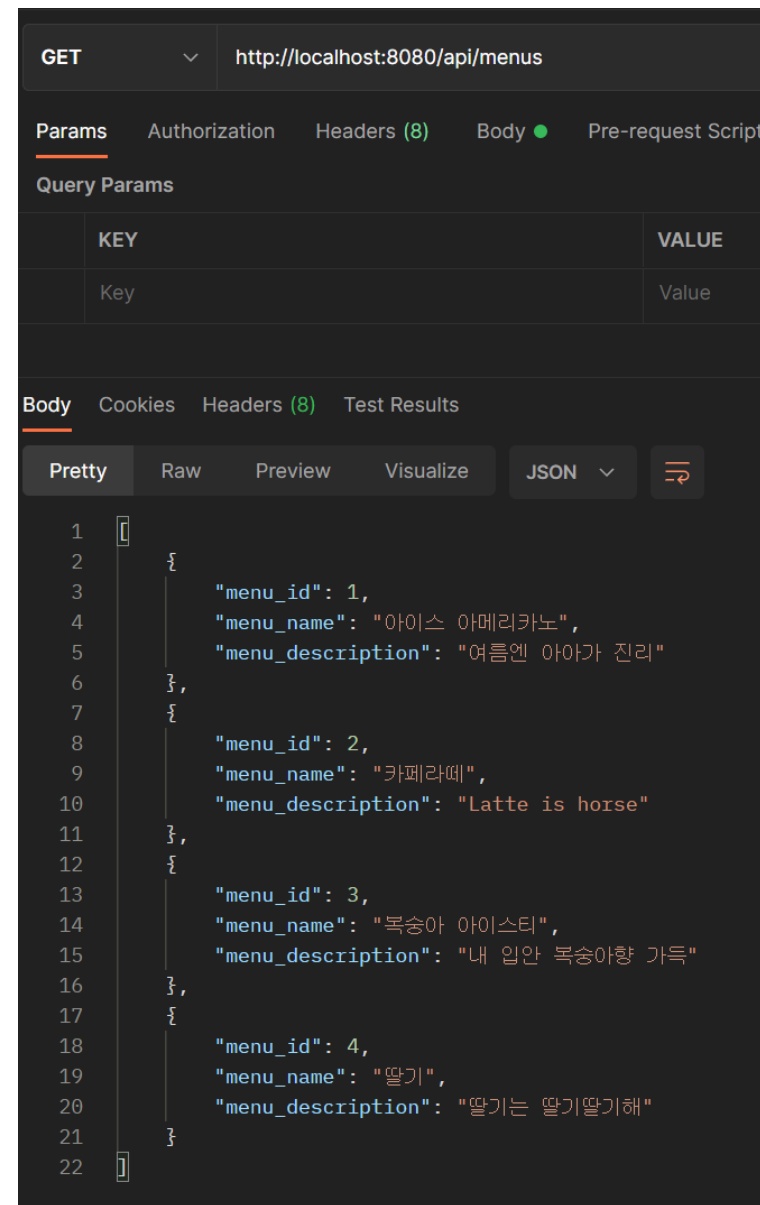
POSTMAN 사용하기

- HTTP REQUEST METHOD 확인하기
 - GET 방식이면 서버에서 GET 방식으로 요청을 받게 되고
POST 방식이면 서버에서 POST 방식으로 요청을 받게 된다.
서로 일치하는 요청인지를 확인!
- 주소 입력
 - 접근할 서버의 API주소를 기입
- Send 버튼을 누르면 해당 METHOD로 주소에 요청을 보낸다.



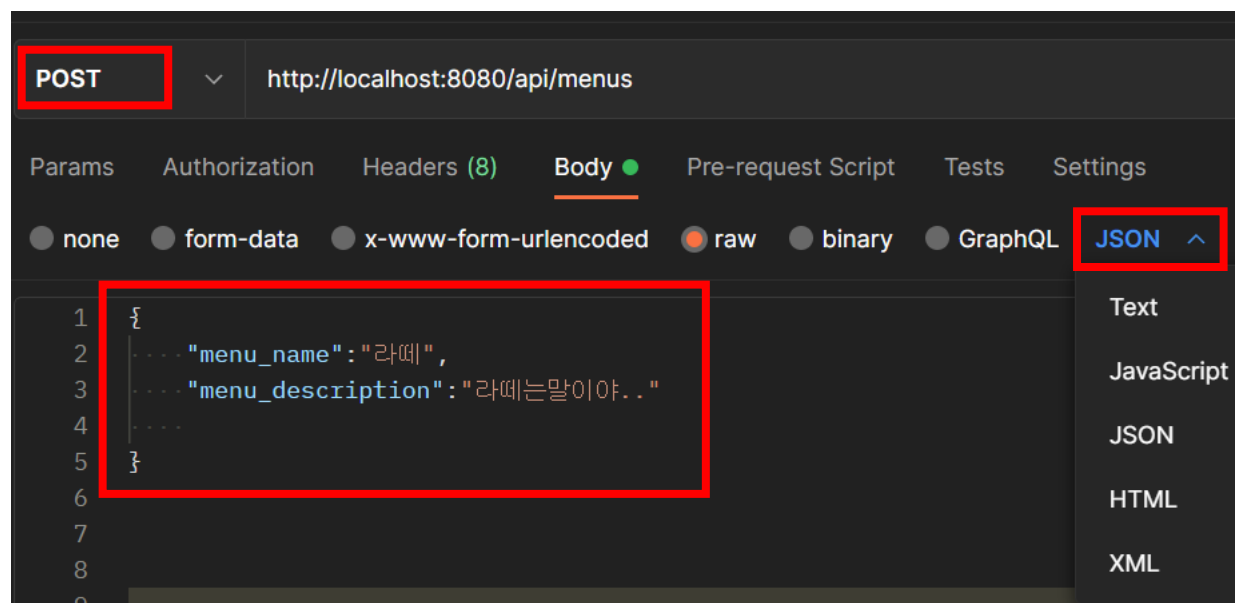
send로 요청

- 아래 body에 요청 결과에 따른 값이 리턴
- GET, POST, PATCH, PUT, DELETE 모든 요청이 가능



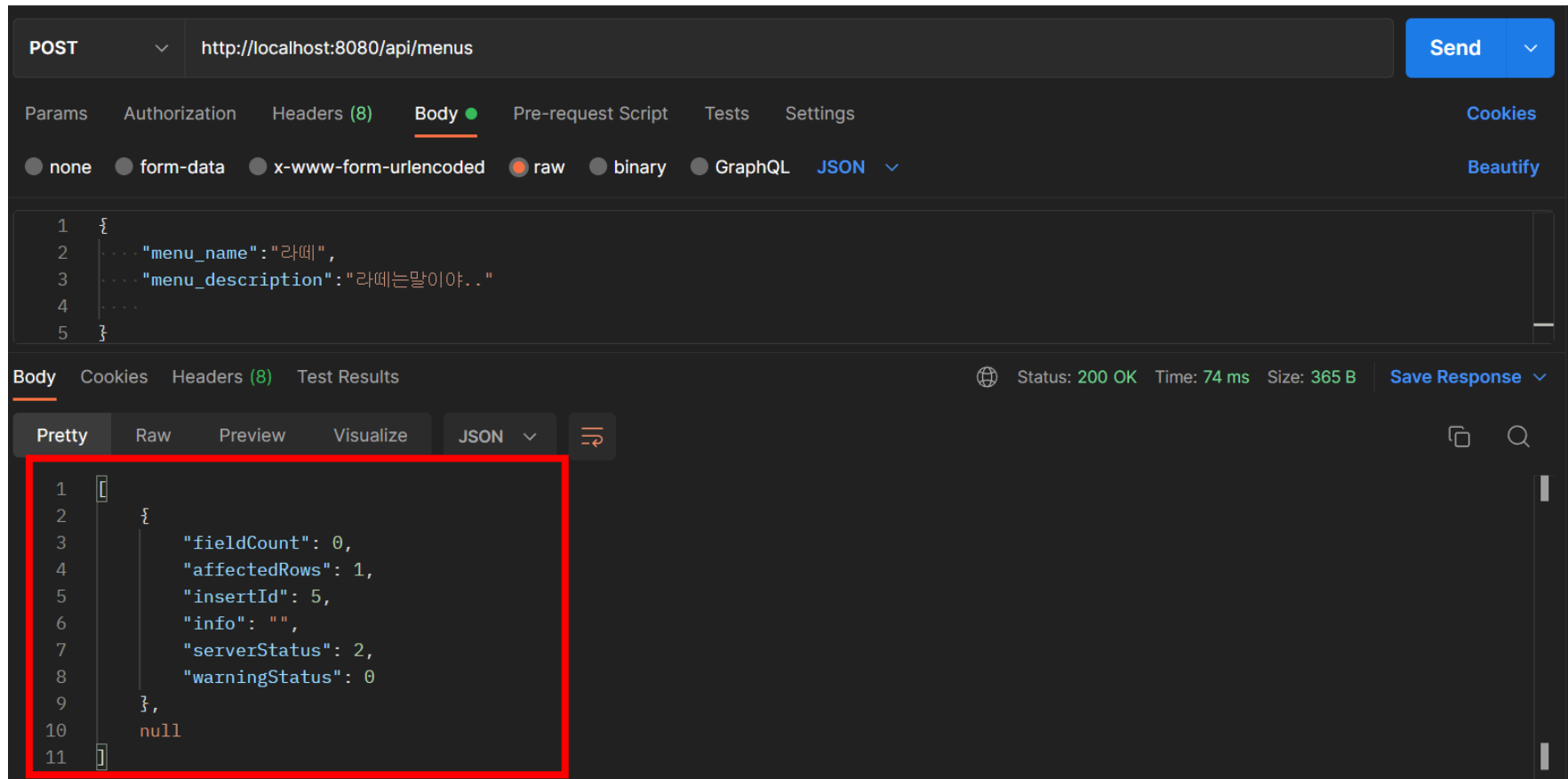
POST 요청해보기

- POST로 HTTP REQUEST METHOD 변경
 - url도 서버와 일치하는지 확인
- Postman에서 요청하는법
 - body -> raw 클릭 후 형식을 JSON 형식으로 바꿔준다.
 - 해당 body의 JSON DATA들을 express 에서는 req.body() 형태로 받아온다.



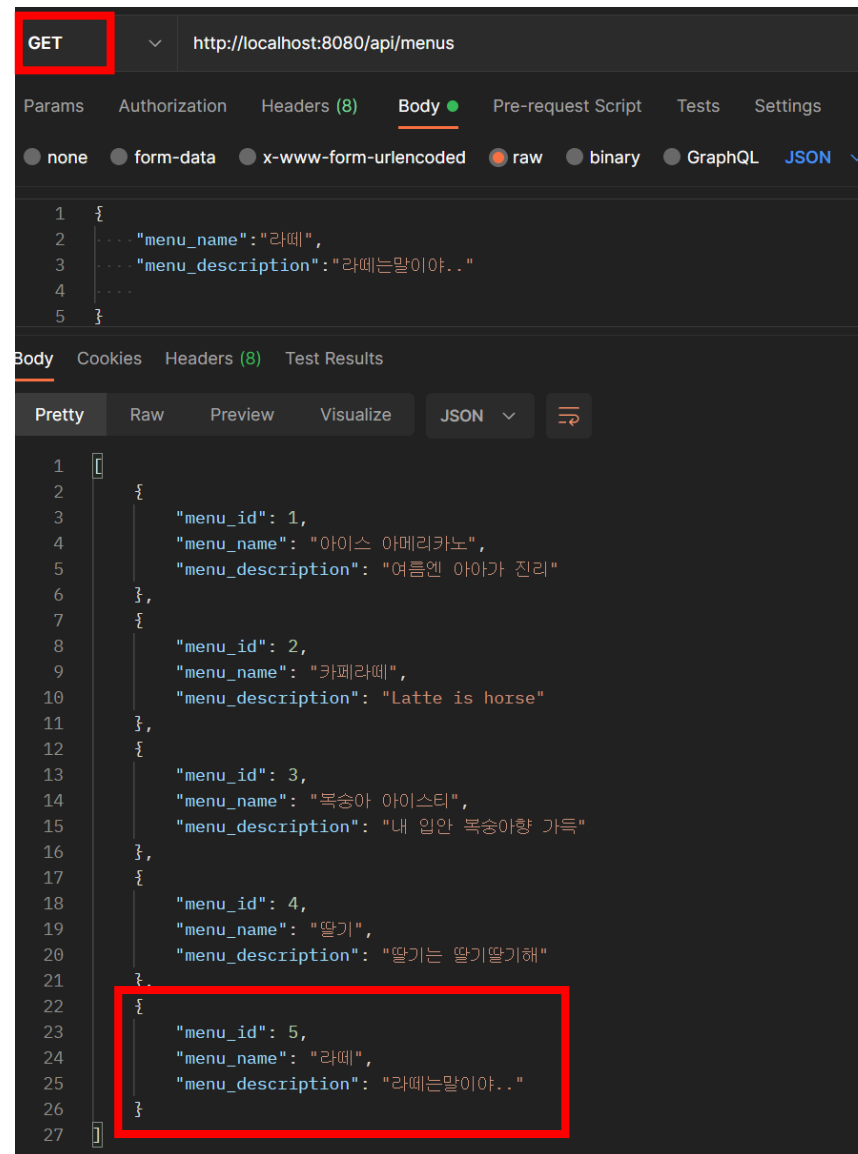
POST 요청해보기

- 성공적으로 요청이 수행되었다.



POST 요청후 값 다시 조회하기

- GET으로 변경한 후 다시 요청을 보낸다.



http 로그를 상세히 보여주는 npm 패키지

morgan

npm v1.10.0 downloads 13.7M/month travis passing coverage 82%

HTTP request logger middleware for node.js

Named after **Dexter**, a show you should not watch until completion.

API

```
var morgan = require('morgan')
```

Install

```
> npm i morgan
```

Repository

github.com/expressjs/morgan

Homepage

github.com/expressjs/morgan#readme

Weekly Downloads

2,688,665



npm i morgan

- `app.use(morgan('dev'))` 형태로 사용

```
{
  "dependencies": {
    "cors": "^2.8.5",
    "express": "^4.18.1",
    "morgan": "^1.10.0",
    "mysql2": "^2.3.3"
  }
}
```

```
const morgan = require("morgan");
app.use(morgan("dev"));
```

Postman으로 요청을 보내고 node의 log 확인

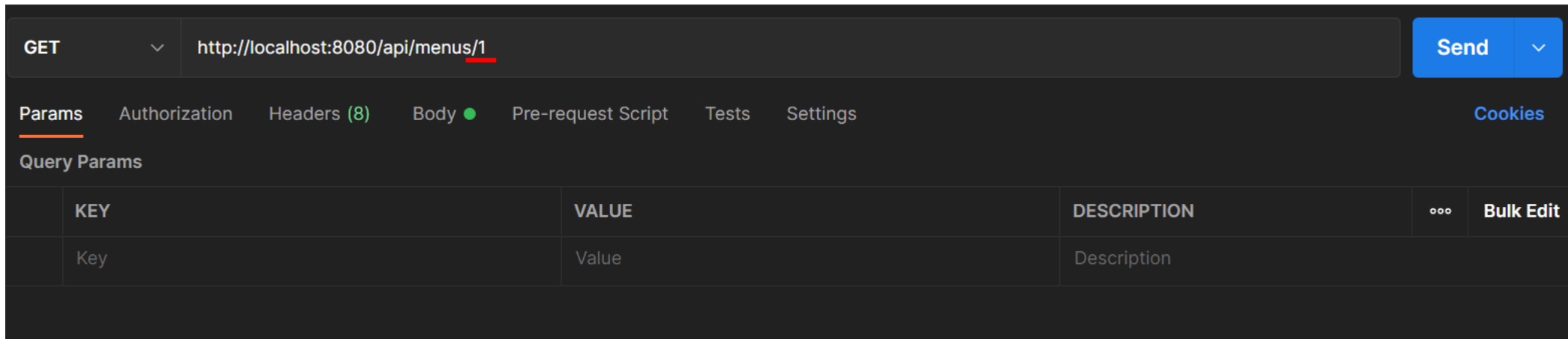
- postman으로 요청 보내기 -> 서버에서 수신 -> morgan이 로그를 기록
- 200 32.830 ms
 - 200
 - Http Response Status Code
 - 32.830 ms
 - 통신 성공까지 걸린 시간

```
[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 index index.js`
this server listening on 8080
GET /api/menus 200 32.830 ms - 672
```

params, query, body

path

- <http://localhost:8080/api/menus/1>
 - menus 데이터의 첫번째를 가져온다.
 - ex) <http://localhost:8080/api/group/1/student>
 - group 1번의 모든 student
 - ex) <http://localhost:8080/api/group/1/student/2>
 - group 1번의 2번 student
 - 계층적 구조를 나타낼 때 사용
 - express에서는 req.params 로 받아온다



app.get('/주소/:변수')

- **app.get('/api/user/:id')**

- :id 부분은 변수로써 어떤 값이 들어올지 모른다
- ex) /api/user/3
 - /api/user의 3번째
- /api/user/chicken
 - /api/user의 chicken user

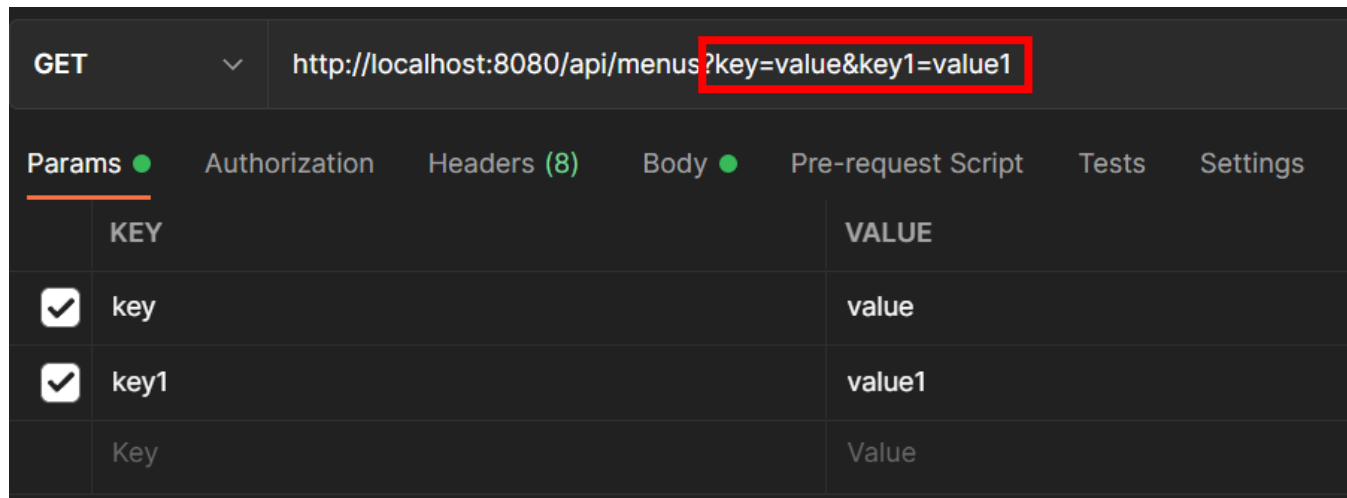
```
app.get("/api/user/:id", (req, res) => {  
    console.log(req.params);  
})
```



```
{ id: '2' }  
GET /api/user/2 200 17.147 ms - 15  
{ id: '1' }  
GET /api/user/1 200 2.484 ms - 15  
{ id: 'chicken' }  
GET /api/user/chicken 200 1.288 ms - 15
```

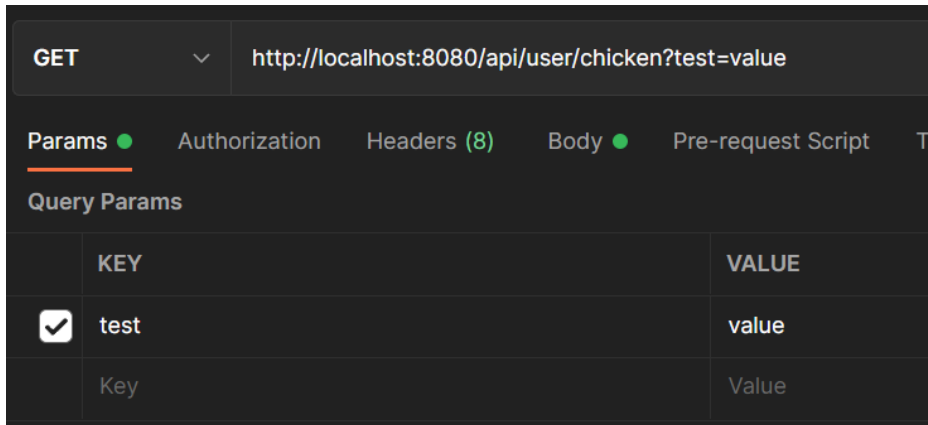
query

- `http://localhost:8080/api/menus?키값=value`
 - ?로 시작해서 key값 = value 형태
- `http://localhost:8080/api/menus?키값=value&키값1=value1`
 - 추가로 전달하고 싶은 경우 &를 붙여서 전달한다.
- http에서는 query params, query string 이라고 한다.
 - express 에서는 req.query 형태로 받아온다.
 - POSTMAN에서 Params라고 표시하는 이유는 Query Params 중 Query 부분을 줄여서 표시한것

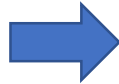


주소?키값=value

- 추가로 전달하고 싶은 경우 &를 붙여서 전달한다.



GET	http://localhost:8080/api/user/chicken?test=value
Params	Authorization Headers (8) Body Pre-request Script
Query Params	
KEY	VALUE
<input checked="" type="checkbox"/> test	value
Key	Value



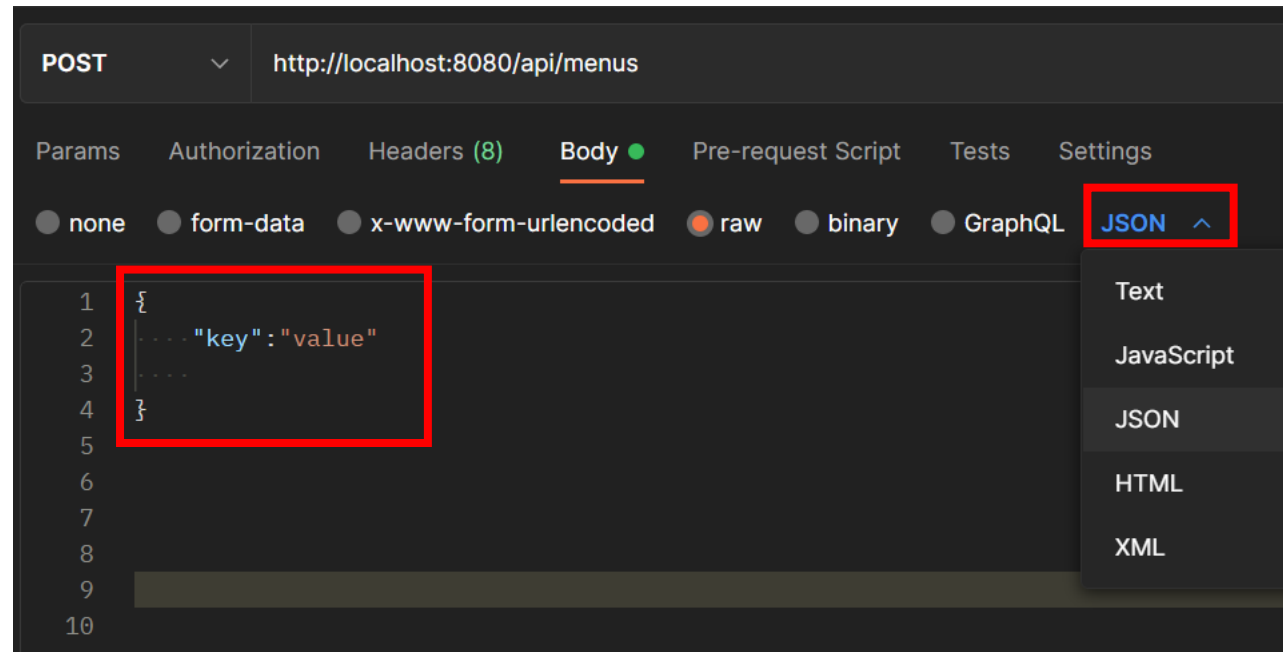
```
app.get("/api/user/:id", (req, res) => {  
  console.log(req.query);  
})
```



```
{ id: 'chicken' }  
GET /api/user/chicken?test=value 200 1.142 ms - 15
```

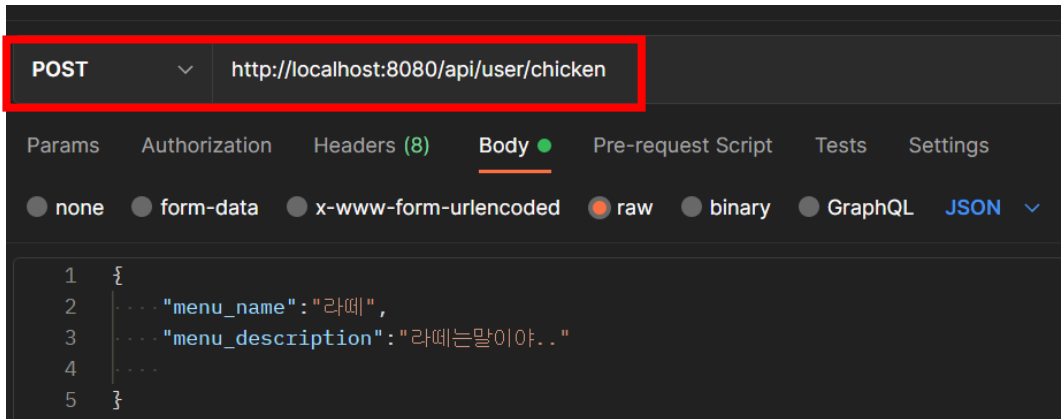
body

- POST, PATCH 요청에 사용
- Postman에서 요청하는법
 - body -> raw 클릭 후 형식을 JSON 형식으로 바꿔준다.
- express에서는 req.body 형태로 받아온다.



body 안에 json 형식으로 데이터를 보낸다

- body 데이터는 POST, PATCH에서 사용한다.
- req.body 안에 데이터가 담긴다



```
app.post("/api/user/:id", (req, res) => {  
  console.log(req.body);  
})
```



```
{ menu_name: '라떼', menu_description: '라떼는말이야..' }  
POST /api/user 200 2.140 ms - 16
```

body 데이터를 가질수 있는 경우

- POST, PUT, PATCH
- GET 과 DELETE는 일반적으로 body 데이터를 가질수 없다.

Request method ⇅	Request has payload body ⇅
GET	Optional
POST	Yes
PUT	Yes
DELETE	Optional
PATCH	Yes

정적 파일

정적 파일

- 직접 변화시키지 않는이상 변하지 않는 파일
 - ex) image, file
- 해당 정적 파일들을 보여줄 서버 구축이 필요

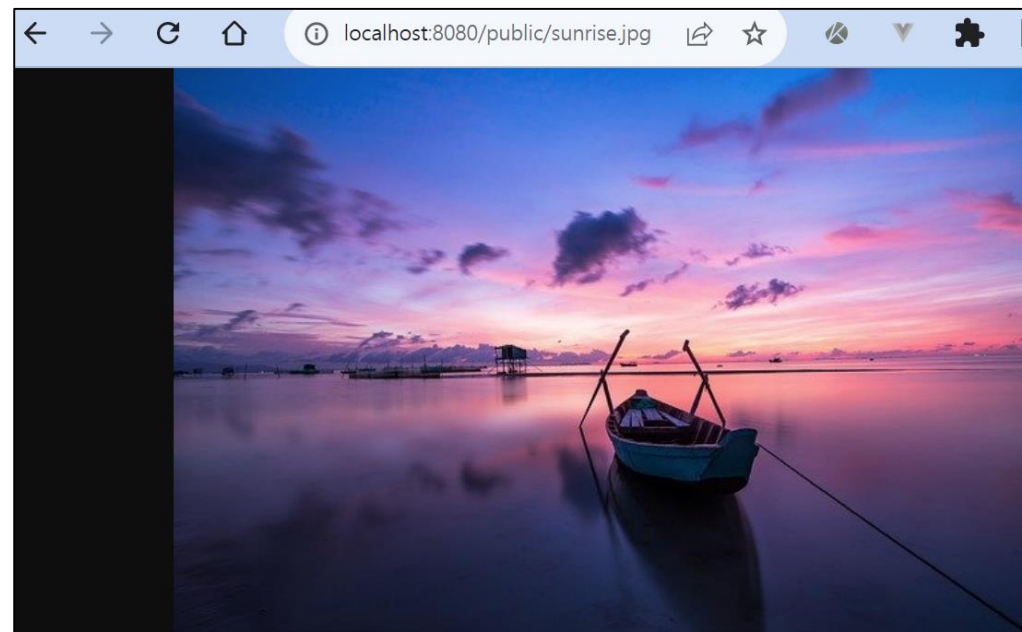
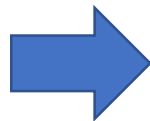
정적 파일 서비스하기

- public 폴더 생성
- `app.use('/경로', express.static('public'))`
 - public 폴더를 정적 폴더로 생성
 - `app.use('/public', express.static('public'))`
 - public 라우터에 접근시 public 안에 있는 폴더 내용들을 보여줄수 있다.
 - `http://localhost:8080/public/sunrise.jpg`

```
> public
JS index.js
package-lock.json
package.json
const express = require("express");

const app = express();
const PORT = 8080;

app.use("/public", express.static("public"));
```



이미지 업로드하기

- **multer 라이브러리 활용**
- **서버**
 - multer 라이브러리를 통해 요청한 파일 업로드
 - 정적 폴더에 업로드 해서 정적 파일 서비스가 진행되도록 하기
 - POSTMAN 을 활용해서 테스트
- **클라이언트**
 - axios를 활용해 파일 전송


multer 라이브러리 활용

- 이미지 업로드를 손쉽게 진행하게 해주는 라이브러리
- `npm i multer`
- 클라이언트로부터 요청 받은 파일을 서버 폴더에 업로드


Install

```
> npm i multer
```

Repository


 github.com/expressjs/multer

Homepage

 github.com/expressjs/multer#readme

Weekly Downloads

3,299,295



multer 정의하기

- **storage**
 - 저장 관련 정보
 - destination
 - 파일이 저장될 경로 지정
 - filename
 - 파일이 저장될 이름 지정
- **limits**
 - 파일 업로드 사이즈 제한

```
const multer = require("multer");
const upload = multer({
  storage: multer.diskStorage({
    destination: (req, file, done) => {
      done(null, "public/");
    },
    filename: (req, file, done) => {
      done(null, file.originalname);
    },
  }),
  limits: { fileSize: 5 * 1024 * 1024 },
});
```

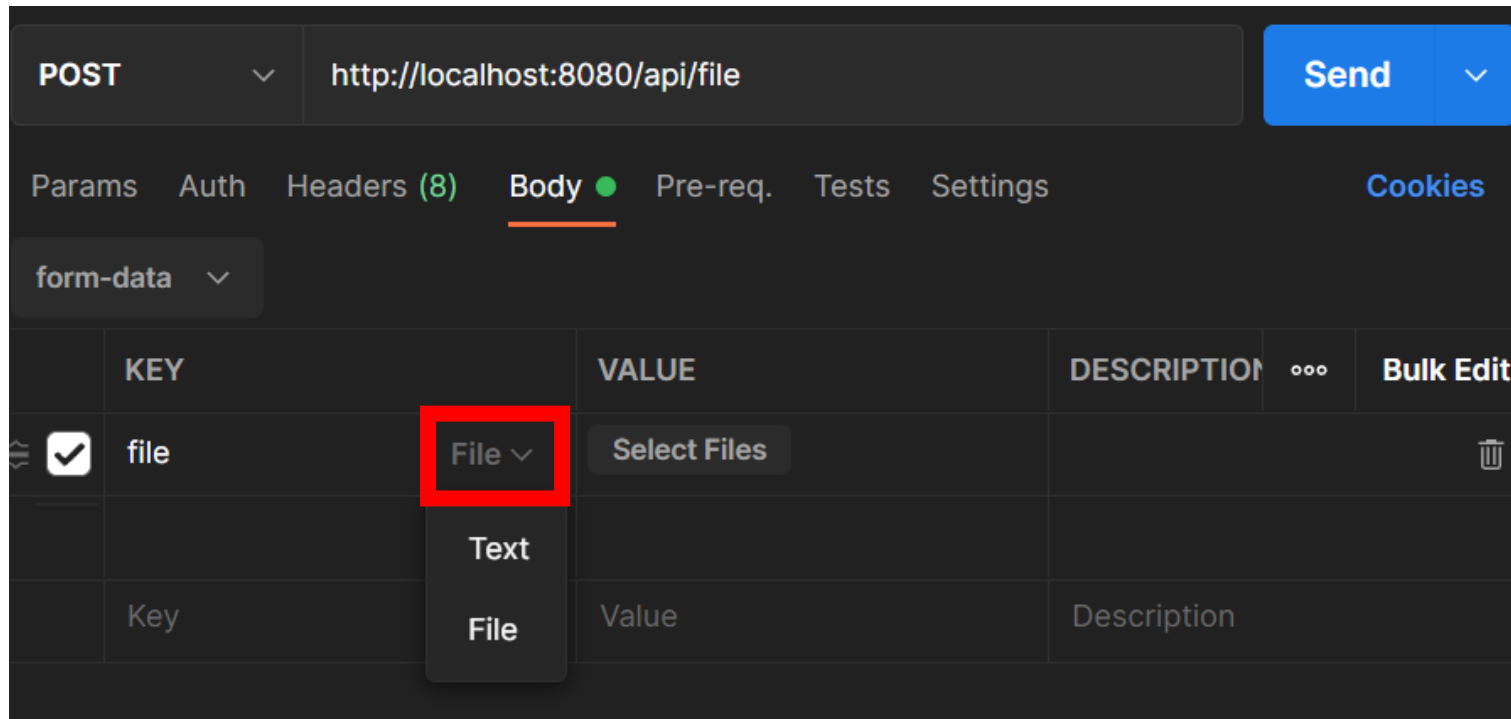
정의된 upload 객체를 요청 앞에 집어넣는다.

- (req,res) 전에 요청을 새로 집어넣게 되면 해당 부분을 먼저 진행하고 나서 그다음으로 넘겨준다.
- `upload.single('file')`
 - file이라는 이름으로 요청이 들어오게 되면 해당 file을 multer 정의에 따라 업로드한다.
- file upload 는 반드시 **POST** 요청만 가능하다.

```
app.post('/api/file', upload.single('file'), (req, res) => {  
    return res.json({test:"OK"})  
})
```

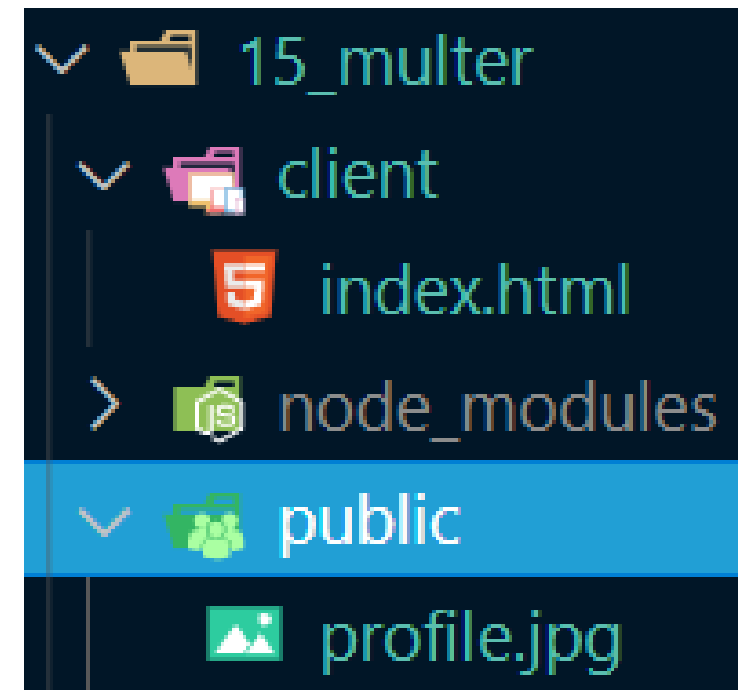
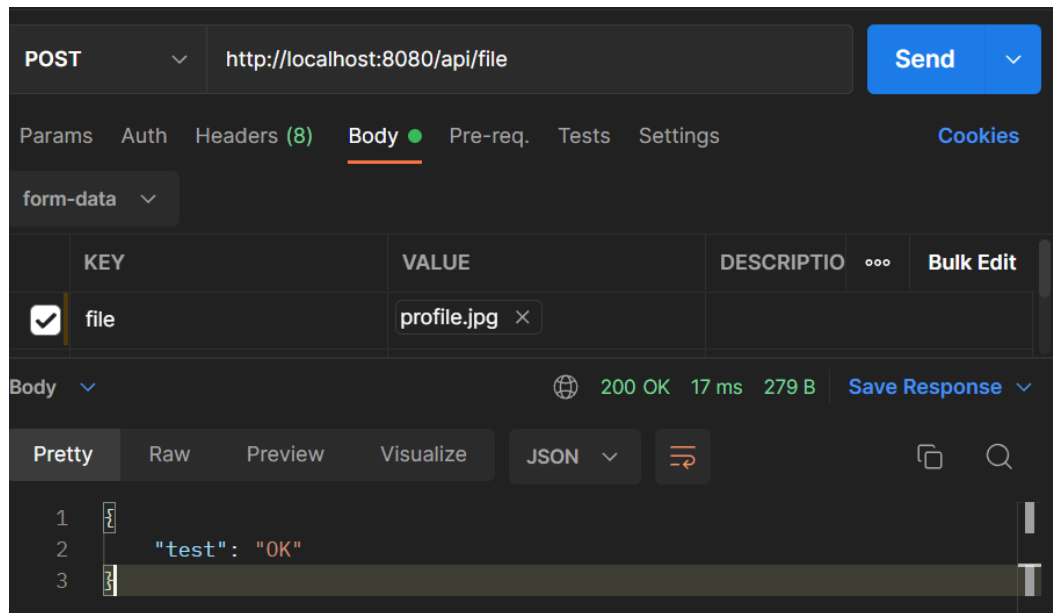
POSTMAN에서 업로드하기

- body
 - form-data 활용
 - 기존 Text -> File로 클릭하면 File 또한 전송이 가능
 - file 이라는 key값으로 selectFiles를 담아서 보낸다.



form-data를 활용해 전송

- 전송후 public 폴더에 정상적으로 profile.jpg가 업로드 된것을 확인



클라이언트 생성

- 바탕화면에 별도의 디렉터리 생성 후, index.html 만들기
- Axios 를 활용해 업로드 진행하기

axios를 활용한 파일 업로드시 유의사항

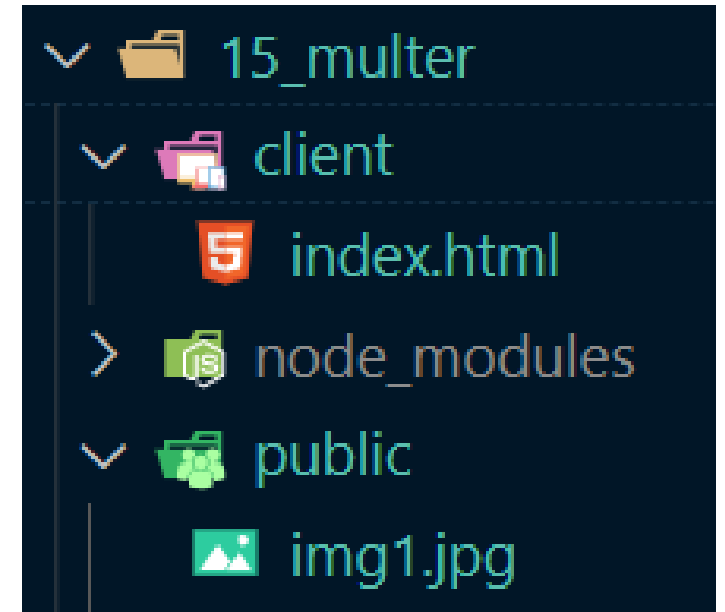
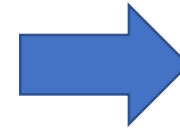
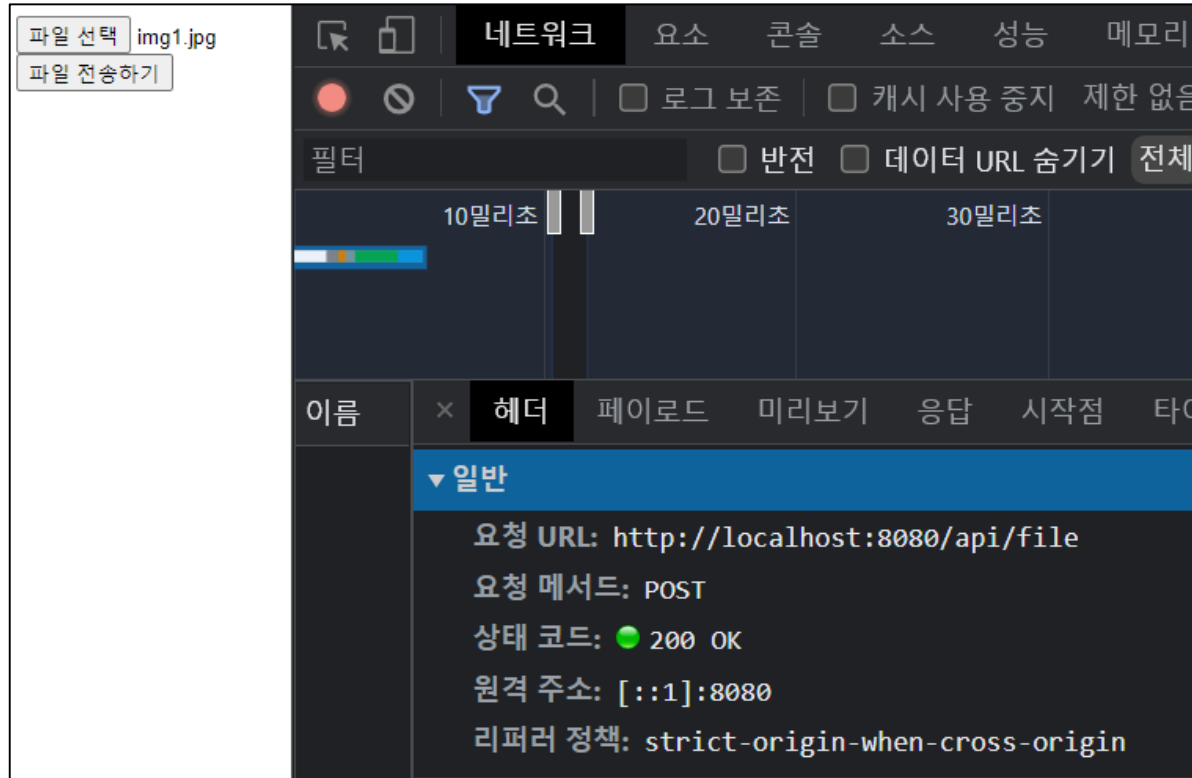
- 파일업로드는 반드시 POST 요청만 가능
- formData
 - form 객체를 만들어서 그안에 file의 정보를 넣어서 file:전달할 파일 식으로 작성
- axios에 header를 추가한다
 - multipart/form-data
 - 파일 업로드를 위한 속성

```
<input type="file" id="input-file" />
<button id="submit-button">파일 전송하기</button>
<script src="https://cdn.jsdelivr.net/npm/axios@1.6.7/dist/axios.min.js"></script>
<script>
  const inputFile = document.querySelector("#input-file");
  const submitButton = document.querySelector("#submit-button");
  submitButton.addEventListener("click", () => {
    console.log(inputFile.files);
    const formData = new FormData();
    formData.append("file", inputFile.files[0]);

    axios.post("http://localhost:8080/api/file", formData, {
      headers: {
        "Content-Type": "multipart/form-data",
      },
    });
  });
</script>
```

결과 테스트

- 파일 전송하기 클릭시 서버에 잘 업로드 된것을 확인



실제로 운영하는 방식

- 클라이언트에서 서버에 파일 업로드
- 서버에서는 해당 파일을 업로드하고 해당 파일의 경로를 DB에 저장

표를 보고 API 만들어 보기

- 넘겨받은 params, query, body는 모두 console.log로 출력한다.
- json 방식으로 리턴 한다.
- 기술된 params, query, body가 정상적으로 넘어오지 않았을 경우 실패 시(json)를 반환한다

	GET	POST	PATCH	DELETE
라우터	/user	/user	/user	/user
params	:id		:id	:id
query	name=chicken			
body		id : num, password: num1	name : hello	
리턴 값(json)	{ getid: true, id : id, name : chicken }	{ signup: true, id : num, password: num1 }	{ update: true, id: id, name: hello }	{delete: true, id: id }
실패 시(json)	{ getid: false }	{ signup: false }	{ update: false }	{ delete : false }

GET /user/123?name=chicken

- GET - HTTP 요청 메서드
- /user - 라우터 주소
- 123 - params
- ?name=chicken - query 파라미터

	GET
라우터	/user
params	:id
query	name=chicken
body	
리턴 값(json)	{ getid: true, id : id, name : chicken }
실패 시(json)	{ getid: false }

GET /user/123?name=chicken

코드 작성하기

- 명세서를 보고 실제로 코드로 변경

	GET
라우터	/user
params	:id
query	name=chicken
body	
리턴 값(json)	{ getid: true, id : id, name : chicken }
실패 시(json)	{ getid: false }

완성된 전체 코드부터
index.js 에 작성해보자.



```
15 // GET /user/123?name=chicken
16 app.get("/user/:id", (req, res) => {
17     console.log(req.params); // { id: '123' }
18     console.log(req.query); // { name: 'chicken' }
19     console.log(req.body); // {}
20     try {
21         if (req.params.id && req.query.name) {
22             return res.json({
23                 getid: true,
24                 id: req.params.id,
25                 name: req.query.name,
26             });
27         }
28         return res.json({ getid: false });
29     } catch (error) {
30         return res.json({ getid: false });
31     }
32 });
```


유의 사항

- params
 - 콜론 (:) 쓰고 parameter 이름
 - 실제 사용할때는 url 에 직접 입력
 - ex) /user/:id 라고 가정
 - /user/123
 - id 는 123
 - /user/jonylee
 - id 는 jonylee
- params, query, body는 전부 req 객체로 부터 존재하는 프로퍼티

```
15 // GET /user/123?name=chicken
16 app.get("/user/:id", (req, res) => {
17   console.log(req.params); // { id: '123' }
18   console.log(req.query); // { name: 'chicken' }
19   console.log(req.body); // {}
```

결과 확인

- params
 - /:id, 즉, 123
- query
 - ?name=chicken
- body
 - 담겨있지 않다.

GET  localhost:8080/user/123?name=chicken



```
this server listening on 8080
{ id: '123' }
{ name: 'chicken' }
{}
GET /user/123?name=chicken 200 4.548 ms - 42
```

추가 코드 작성하기

- params의 id와 query의 name이 존재하는 경우에만 정확한 json을 리턴
- 해당 조건을 만족하지 않거나 에러가 발생하는 경우(통신 실패) getid:false를 리턴

```
20     try {
21         if (req.params.id && req.query.name) {
22             return res.json({
23                 getid: true,
24                 id: req.params.id,
25                 name: req.query.name,
26             });
27         }
28         return res.json({ getid: false });
29     } catch (error) {
30         return res.json({ getid: false });
31     }
```

결과 확인

- 요청에 따른 결과가 잘 나오는 것을 확인 할 수 있다.
- 이렇게 구축한 서버는 멋진 REST API 서버가 된다.

```
1 {  
2     "getid": true,  
3     "id": "123",  
4     "name": "chicken"  
5 }
```


나머지 명세서를 보고 API를 만들어보자

- GET 요청을 했던 내용을 분석을 기반으로
- POST, PATCH, DELETE 명세를 보고 API 서버를 작성

	GET	POST	PATCH	DELETE
라우터	/user	/user	/user	/user
params	:id		:id	:id
query	name=chicken			
body		id : num, password: num1	name : hello	
리턴 값(json)	{ getid: true, id : id, name : chicken }	{ signup: true, id : num, password: num1 }	{ update: true, id: id, name: hello }	{delete: true, id: id }
실패 시(json)	{ getid: false }	{ signup: false }	{ update: false }	{ delete : false }

파일 업로드

파일업로드

- 서버 만들기
- 클라이언트 만들기
- 서버 < - > 클라이언트간 통신하게 만들기
- <https://pixabay.com/ko/> 에 접속해서 마음에 드는 이미지를 업로드하기
- 서버에서 업로드된 이미지를 정적인 웹 주소(URL) 로 접근할 수 있도록 하기

내일 방송에서 만나요!

삼성청년SW·AI 아카데미

과제1(25P)

API Routing 주요코드 + 결과 캡처

과제2 (33p)

DB 테스트 주요 코드 + 결과 캡처

과제3 (40p)

REST API, RESTful API에 대해 조사 후 정리하기

과제4 (54p)

REST API 설계하기

과제5 (61p)

메뉴 추가(POST) 백엔드 API 주요 코드 + POSTMAN 캡처

과제6 (96p)

POSTMAN을 활용한 파일 업로드 테스트 해보기, 주요 코드 + POSTMAN 캡처

과제7 (107p)

주요 코드 + 파일 업로드 + 업로드한 이미지가 정적 서비스(URL)로 보이는지 캡처