Hello,

KDT 웹 개발자 양성 프로젝트

5기!

with

Express Routing

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

const app = express();
const PORT = 4000;

app.get('/', (req, res) => {
  res.send('GET request');
});

app.post('/', (req, res) => {
  res.send('POST request');
});

app.put('/', (req, res) => {
  res.send('PUT request');
});

app.delete('/', (req, res) => {
  res.send('DELETE request');
});
```

Express Router()

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

const app = express();
const userRouter = express.Router();
const PORT = 4000;

app.use('/users', userRouter);
userRouter.get('/', (req, res) => {
  res.end('회원 목록');
});
userRouter.get('/:id', (req, res) => {
  res.end('특정 회원 정보');
});
userRouter.post('/', (req, res) => {
  res.end('회원 등록');
});

app.use('/', (req, res) => {
  res.end('Hello, express world!');
});

app.listen(PORT, () => {
  console.log(`The express server is running at port: ${PORT}`);
});
```

서버에서 받은 데이터로 어떻게 그리죠?

- Res.write 로 하나하나 그려줄 수 있습니다!

```
userRouter.get('/show', (req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/html;charset=UTF-8' });
  res.write('<h1>hello, Dynamic Web page</h1>');
  for (let i = 0; i < USER.length; i++) {
    res.write(`<h2>USER id is ${USER[i].id}`);
    res.write(`<h2>USER name is ${USER[i].name}`);
  }
  res.end('');
});
```

```
rules: {
    'linebreak-style': 0,
    'no-console': 'off',
    'no-plusplus': 'off',
  },
```

View

Engine

EJS

<% Embedded JavaScript %>

Templating: Node, Express, EJS

- 가장 기본이 되는 View Engine
- HTML 문법을 사용
- 단, 레이아웃 기능 X

pug

- HTML 문법을 단순화 하여 사용
- 레이아웃 기능 O

NUNJUCKS

- HTML 문법을 그대로 사용
- 레이아웃 기능 O

EJS

사용하기

View Engine 세팅!

- Ejs 를 설치 합니다!
  - Npm i -D ejs
- Express 에게 어떤 View Engine 으로 웹 페이지를 그릴 것인지도 알려줘야 합니다

```
app.set('view engine', 'ejs');
```

- 동적 웹페이지는 Views 폴더에서 관리 합니다

Index.ejs 파일 만들고 연결하기

- 서버 폴더 가장 외부에 views 폴더 만들기!
- Views 폴더 내부에 index.ejs 파일 만들고
  - <h1> 태그로 Hello, EJS World! 띄우기
- Localhost:4000/users 기본 요청이 들어오면
  - Index.ejs 파일 띄우기

```
userRouter.get('/ejs', (req, res) => {
  res.render('index');
});
```

EJS

동적 웹 시작!

- 서버 코드

```
const USER = [
  {
    id: "tetz",
    name: "이효석",
  },
];


userRouter.get('/ejs', (req, res) => {
  const userLen = USER.length;
  res.render('index', { USER, userCounts: userLen });
});
```

```
<body>
    <h1>회원 목록</h1>
    <h2>
        총 회원 수 <%= userCounts %>
    </h2>
    <ul>
        <% if(userCounts> 0) { %>
            <% for(let i=0; i < userCounts; i++) { %>
                <li>
                    <p>ID: <%= USER[i].id %>
                    </p>
                    <p>이름: <%= USER[i].name %>
                    </p>
                </li>
                <% } %>
                    <% } else { %>
                        <li>
                            회원 정보가 없습니다!
                        </li>
                        <% } %>
    </ul>
</body>
```

Static

사용하기!

Static

- Static 은 브라우저에서 접근이 가능한 폴더의 위치를 지정해 주는 역할을 합니다.
- 사용법은 app.use(express('폴더위치'));

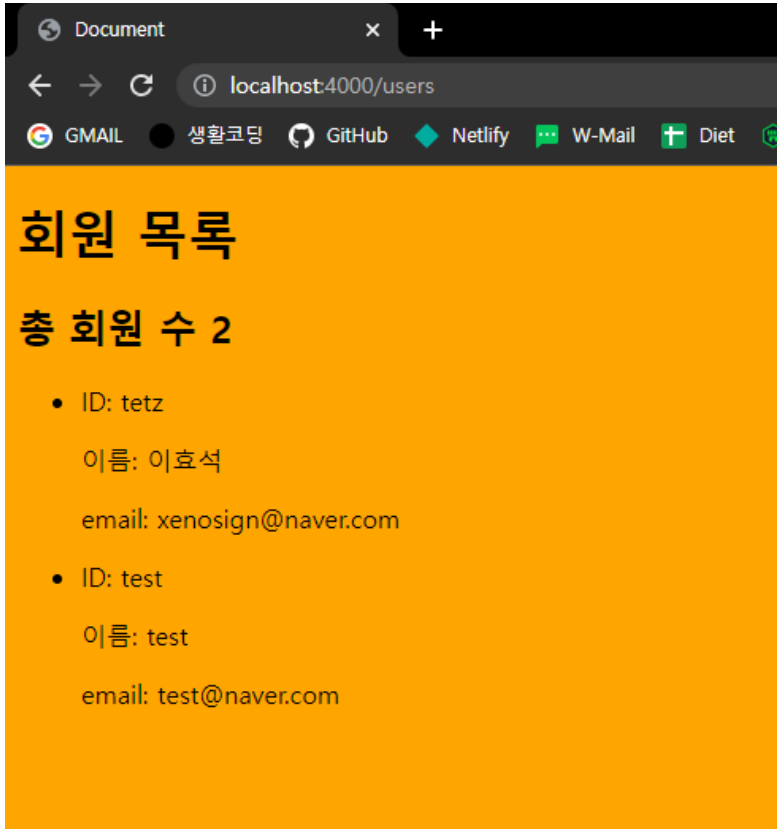
```
app.use(express.static('views'));
```

- 위의 미들웨어를 사용하면 프로젝트에서 사용하는 폴더 경로의 시작점은 localhost:4000/views 가 됩니다!
- ./ 의 상대 경로 → localhost:4000/views/ 와 동일한 의미를 가집니다!

Static

- Index.ejs 파일 css 링크 경로를 아래와 같이 변경하고 페이지를 새로 고침 해 봅시다!

```
<link rel="stylesheet" href="/css/style.css">
```



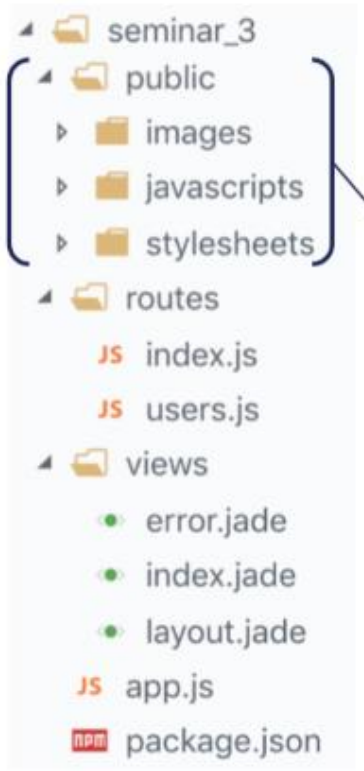
Express

기본 폴더 구조

Express 기본 폴더 구조

- Express 로 만들어진 서비스의 기본 폴더 구조를 배워 봅시다!

프로젝트 구조



/public

외부(브라우저 등 클라이언트)에서 접근 가능한 파일 모아둠

리소스들이 올라감

앱 서버에서는 딱히 건들일 없음

- /images : 그림파일들을 저장
- /javascripts : 자바스크립트 파일들을 저장
- /stylesheets : CSS 파일들을 저장

Express

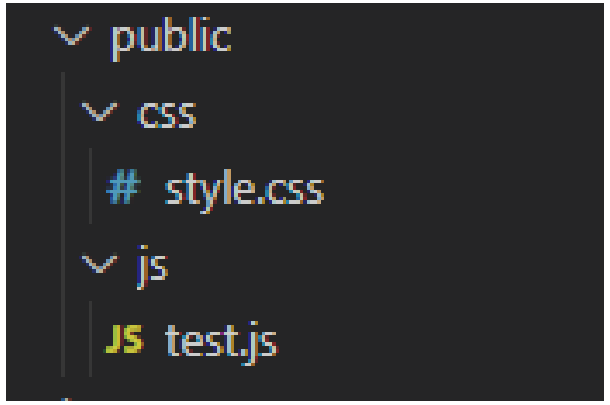
기본 폴더 구조로 변경

Public 폴더 설정

- Public 폴더 만들고 static 설정

```
app.use(express.static('public'));
```

- Public/css 폴더 만들고 css 파일 옮기기
- Public/js 폴더 만들고 test.js 파일 넣기



Public 폴더 설정

- Index.ejs 파일에서 css 파일과 아래 코드로 구성 된 test.js 파일 불러와서 테스트

```
/* eslint-disable */
// @ts-check

alert('test');
```

```
<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>Document</title>
  <link rel="stylesheet" href="./css/style.css">
  <script src="/js/test.js"></script>
</head>
```

public/images 적용

- Public/images 폴더에 원하는 이미지 넣기
- Index.ejs 에서 이미지 태그 추가해서 불러오기!

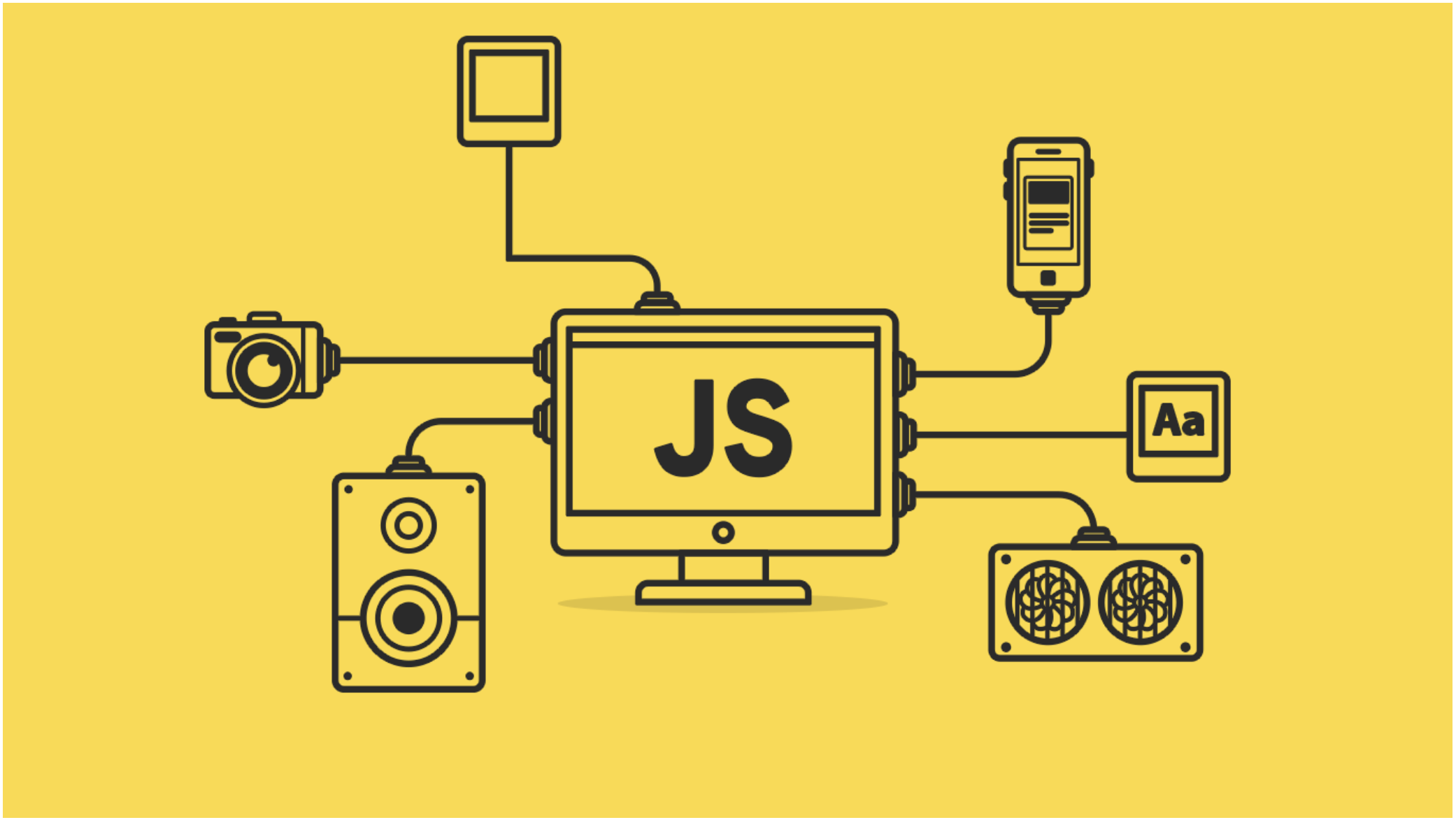
```
<img src="/images/dog.jpg" alt="강아지" />
```


지금 시작합니다

Module

사용하기!

JS Module


JS
Aa

JS 에서 Module 사용하기

- JS 도 Module 을 지원 합니다!
- 다른 사람이 만든 기능을 활용 할 때, 매번 코드를 받아서 붙여 넣기는 귀찮겠죠?
- 이럴 땐, 파일로 받아서 사용하는 편이 더 편할 겁니다!
- 그래서 JS도 파일을 Module 로 불러와서 사용이 가능합니다!
- 그런데 방법이 2가지가 있습니다!
  - CommonJS 방식 / ES6 방식

CommonJS

방식

CommonJS 방식

- Node.js 에서 사용되는 모듈 방식 입니다!
- 키워드로는 require / exports 를 사용합니다.
- Require 로 모듈을 불러 올 때, 코드 어느 곳에서나 불러 올 수 있습니다!

CommonJS 방식

- 전체 모듈로써 내보내고, 전체를 하나의 Obj 로 받아서 사용하는 방법

```
const animals = ['dog', 'cat'];

function showAnimals() {
  animals.map((el) => console.log(el));
}

module.exports = {
  animals,
  showAnimals,
};
```

Animal.js

```
const animals = require('./animals');

console.log(animals);
animals.showAnimals();
```

Module.js

CommonJS 방식

- 내보내고 싶은 것(변수, 함수, 클래스 등등)에 exports 를 붙여서 내보내고, 각각을 따로 선언해서 가져 오는 방식

```
const animals = ['dog', 'cat'];

exports.animals = animals;

exports.showAnimals = function showAnimals() {
  animals.map((el) => console.log(el));
};
```

Animal.js

```
const { animals, showAnimals } = require('./animals');

console.log(animals);
showAnimals();
```

Module.js

CommonJS 방식

- 하나의 객체(or 클래스)에 전부를 넣어놓고 그 객체 자체를 내보내는 방식!

```
const animals = {
  animals: ["dog", "cat"],
  showAnimals() {
    this.animals.map((el) => console.log(el));
  },
};

module.exports = animals;
```

Animal.js

```
const { animals, showAnimals } = require('./animals');

console.log(animals);
showAnimals();
```

Module.js

ES6 방식

ES6 방식

- 2015년에 ES6 가 업데이트 되면서 추가 된 방식입니다.
- 사실 ES6 는 브라우저에서 구동되는 JS에 대한 문법이라서 Node.js 에서는 사용하려면 별도의 처리가 필요합니다!
- 예전에는 파일 확장자를 ~~.mjs 로 처리해서 구분
- 요즘에는 package.json 에 아래의 문구를 추가해 주면 사용이 가능합니다!

```
"type": "module",
```

- ES6 는 CommonJS 처럼 전체를 내보내는 방식이 존재하지 않으며, 필요한 모듈을 설정해서 원하는 것만 내보내고 받는 방식을 씁니다!(사실 CommonJS도 그렇게 변경이 되었습니다!)

ES6 방식 - export

- 선언부에 Export 사용하기

```
export const animals = ['dog', 'cat'];

export function showAnimals() {
  animals.map((el) => console.log(el));
}
```

Animal.js

```
import { animals, showAnimals } from './animals.js';

console.log(animals);
showAnimals();
```

Module.js

ES6 방식 - export

- 마지막에 Export 사용하기

```
const animals = ['dog', 'cat'];

function showAnimals() {
  animals.map((el) => console.log(el));
}

export { animals, showAnimals };
```

Animal.js

```
import { animals, showAnimals } from './animals.js';

console.log(animals);
showAnimals();
```

Module.js

ES6 방식 - import

- 가져올 것들이 많으면 * as 를 사용

```
import * as animals from './animals.js';

console.log(animals);
animals.showAnimals();
```

Module.js

- 단, 보통의 경우는 가져올 것을 확실히 명시하는 편이 좋습니다!
  - 메모리 효율 및 처리 속도가 올라갑니다!

ES6 방식 – 모듈 이름 바꾸기(import)

- 모듈 이름을 바꾸고 싶으면 모듈이름 as 새로운모듈이름 으로 변경이 가능 (import, export 동시 적용 가능)

```
import { animals as ani, showAnimals as show } from './animals.js';

console.log(ani);
show();
```

```
export const animals = ['dog', 'cat'];

export function showAnimals() {
  animals.map((el) => console.log(el));
}
```

ES6 방식 – 모듈 이름 바꾸기(export)

- 모듈 이름을 바꾸고 싶으면 모듈이름 as 새로운모듈이름 으로 변경이 가능 (import, export 동시 적용 가능)

```
import { ani, show } from './animals.js';

console.log(ani);
show();
```

```
const animals = ['dog', 'cat'];

function showAnimals() {
  animals.map((el) => console.log(el));
}

export { animals as ani, showAnimals as show };
```

ES6 방식 – export default

- CommonJS 에서 본 것 처럼, 모듈은 크게 2가지 방법으로 작성 됩니다.
  - 1. 복수의 export 가 있는 라이브러리 형태의 모듈
  - 2. 파일 하나의 하나의 개체(Obj, Class)만 있는 모듈
- 보통 파일 하나에 하나의 큰 기능을 커버하는 클래스 또는 Obj 를 만드는 2 번의 방법을 선호합니다!
- 따라서, 이런 파일에는 export default 라는 문법으로 모듈을 내보냅니다!
- 즉, 해당 파일에는 모듈 개체가 하나만 있다! 라고 알려주는 역할을 합니다.

ES6 방식 – export default

- Export default 로 내보내진 모듈을 Import 할 때에는 {} 를 사용하지 않고 불러옵니다!

```
export default class Animal {
  constructor() {
    this.animals = ['dog', 'cat'];
  }

  showAnimals() {
    this.animals.map((el) => console.log(el));
  }
}
```

```
import Animal from './animals.js';

const ani = new Animal();
console.log(ani.animals);
ani.showAnimals();
```

CommonJS 와 ES6 의 차이점

CommonJS 와 ES6 의 차이점

- CommonJS 는 Node.js 에서 사용되고 require / exports 사용
- ES6 는 브라우저에서 사용되고 import / export 사용
- ES6를 Node.js 에서 사용하고 싶으면 package.json 에 `"type": "module"` 추가 필요
- Require 는 코드 어느 지점에서나 사용 가능 / Import 는 코드 최상단에서만 사용 가능
- 하나의 파일에서 둘 다 사용은 불가능!
- 일반적으로 ES6 문법이 필요한 모듈만 불러오는 구조를 가지기에 성능이 우수, 메모리 절약 → 그런데 CommonJS 도 해당 문법이 추가 되었음!

실습, 모듈 사용!

- 각각 human.js 와 student.js 를 만들기
- Human.js 는 [‘철수’, ‘영희’] 데이터와 데이터를 전부 출력하는 showName() 메소드가 있음
- Student.js 는 [‘세호‘, ‘재석’] 데이터와 데이터를 전부 출력하는 showStudent() 메소드가 있음
- Human.js 는 CommonJS 방식으로 모듈을 불러서 데이터를 출력
- Student.js 는 ES6 방식으로 모듈을 불러서 데이터를 출력


미래의 그날이 온다!
BACK TO THE FUTURE
OCT 21 2015
A Universal Picture © 1985 & 2015 Universal Studios.

Routing

처리

```js
const express = require('express');
const router = express.Router();

router.get('/', (req, res) => {
  res.render('users', { USER_ARR, userCounts: USER_ARR.length });
});

router.get('/id/:id', (req, res) => {
  const userData = USER[req.params.id];
  if (userData) {
    res.send(userData);
  } else {
    res.send('ID를 못찾겠어요');
  }
});

module.exports = router;
```

routes/users.js

express.Router()를 사용하여 하나의 Router 모듈를 생성

라우터에 각각의 미들웨어 기능들을 작성하여 모듈에 기능을 추가!

라우터에 작성 된 모든 기능들을 하나의 모듈로써 외부로 내보내기!

```js
const express = require('express');

const app = express();
const PORT = 4000;

const userRouter = require('./routes/users');
app.use('/users', userRouter);
```

router.js

routes/users.js 에서 기능을 정의해서 내보낸 모듈을 userRouter 라는 이름의 변수를 선언하여 담아주기!

http://localhost:4000/users 로 들어오는 모든 요청은 이제 userRouter 에 들어있는 코드가 처리하도록 설정

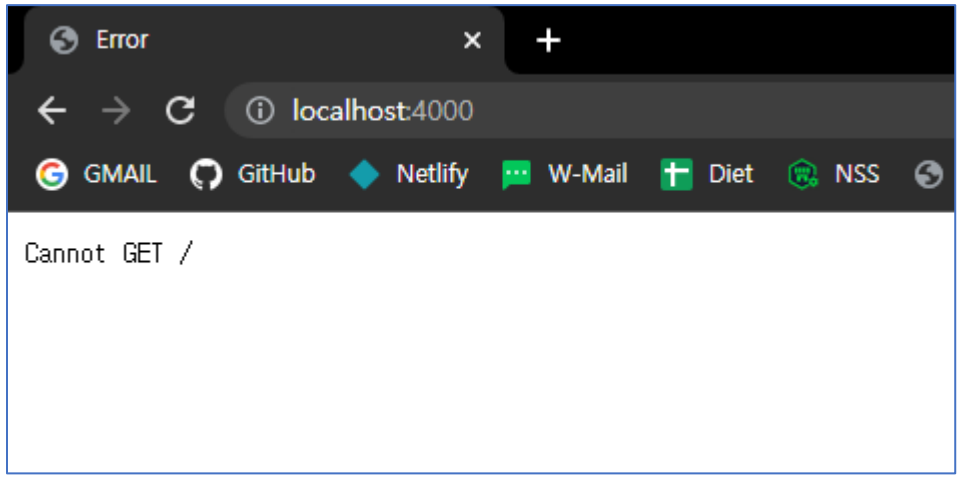

깔끔하게

```
const express = require('express');
const app = express();
const PORT = 4000;

app.set('view engine', 'ejs');
app.use(express.static('public'));
app.use('/users', userRouter);

userRouter.get('/', (req, res) => {
  res.render('users', { USER_ARR, userCounts: USER_ARR.length });
});

userRouter.get('/id/:id', (req, res) => {
  const userData = USER[req.params.id];
  if (userData) {
    res.send(userData);
  } else {
    res.send('ID를 못찾겠어요');
  }
});

userRouter.post('/add', (req, res) => {
  if (!req.query.id || !req.query.name) return res.end('쿼리 입력
이 잘못 되었습니다');

  const newUser = {
    id: req.query.id,
    name: req.query.name,
  };
  USER[Object.keys(USER).length + 1] = newUser;
  res.send('회원 등록 완료');
});

app.listen(PORT, () => {
  console.log(`${PORT} 번에서 서버 실행`);
});
```

```
// @ts-check
const express = require('express');
const userRouter = require('./routes/users');

const app = express();
const PORT = 4000;

app.set('view engine', 'ejs');

app.use(express.static('public'));
app.use('/users', userRouter);

app.get('/', (req, res) => {
  res.send('Hello, Express world!');
});

app.listen(PORT, () => {
  console.log(`${PORT} 번에서 서버 실행`);
});
```

메인 index 페이지 만들기

- 아무런 주소 없이 localhost:4000 으로 접속 하면



- 아무래도 인덱스 페이지가 필요 하겠죠?

Hello, EJS

localhost:4000

GMAIL GitHub Netlify W-Mail Diet NSS

Welcome to tetz's web

유저 서비스

Hello, EJS

localhost:4000/users

GMAIL GitHub Netlify W-Mail Diet

홈으로

회원 목록

총 회원 수 2

- ID: tetz

  이름: 이효석

  이메일: tetz@spreatics.com

- ID: kimi

  이름: 이기원

  이메일: tetz@spreatics.com

메인 index 페이지 만들기

- localhost:4000 으로 접속 하면 index.ejs 페이지가 뜨도록 설정하기
- 해당 페이지에서 유저 서비스를 클릭하면 회원 목록 페이지가 보이도록 설정 해봅시다!
- 회원 목록 페이지에서는 홈으로 a 태그를 만들어서 index.ejs 로 이동이 가능 하도록 설정
- 해당 기능은 mainRouter 로 만들어서 처리할 예정입니다!

Routes/index.js 파일 작업

- 메인 역할을 하는 라우터 파일은 index.js 로 작업해야 합니다!

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

const router = express.Router();

router.get('/', (req, res) => {
  res.render('index');
});

module.exports = router;
```

- 기본 주소(http://localhost:4000/) 로 들어오면 index.ejs 파일을 띄웁시다!

Router.js 에서 모듈 불러오기!

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

const mainRouter = require('./routes');
const userRouter = require('./routes/users');

const app = express();
const PORT = 4000;

app.set('view engine', 'ejs');

app.use(express.static('public'));
app.use('/', mainRouter);
app.use('/users', userRouter);
```

Index.js 라는 파일명은
생략 가능!

파일명이 생략되면
Node.js 가 알아서
Index.js 를 찾습니다!

- 기본 주소로 들어오는 요청은 mainRouter 가 처리하도록 설정

Index.ejs 파일

```
<!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>Document</title>
</head>

<body>
  <h1>Welcome to Tetz's WEB</h1>
  <h2><a href="/users">유저 서비스</a></h2>
</body>

</html>
```

Error 핸들링!

Error 핸들링

- users 라우터에서 문제가 발생하면 res.send 로 err 메시지를 보내주고는 있지만 이 방법은 편의를 위한 방법입니다! → 에러 상황에 대한 statusCode 전달이 안되기 때문이죠!
- 서버 Err 가 발생하면 정석적으로는 err 를 발생 시킨 다음 err 메시지와 status 코드를 전달해 줘야 합니다!
- 그럼, users 라우터의 각각의 상황에서 Err 처리를 해줍시다! + 기존 객체로 구현했던 코드도 배열 코드로 변경 해봅시다!


CT
곤장 2호 발령
MBC
대 역 죄 인

객체 → 배열

- 기존 USER 로 선언해 놨던 객체는 삭제 합시다!
- 그리고 USER_ARR 로 선언했던 배열을 USER 로 대체!

특정 아이디 찾기 미들웨어(API)

```
router.get('/id/:id', (req, res) => {
  const userData = USER.find((user) => user.id === req.params.id);
  if (userData) {
    res.send(userData);
  } else {
    res.send('해당 ID를 가진 회원이 없습니다!');
  }
});
```

회원 추가 미들웨어(API)

```
router.post('/add', (req, res) => {
  if (req.query.id && req.query.name && req.query.email) {
    const newUser = {
      id: req.query.id,
      name: req.query.name,
      email: req.query.email,
    };

    USER.push(newUser);

    res.send('회원 추가 완료!');
  } else {
    res.send('쿼리 입력이 잘못 되었습니다!');
  }
});
```

회원 정보 수정 미들웨어(API)

```
router.put('/modify/:id', (req, res) => {
  if (req.query.email && req.query.name) {
    const userIndex = USER.findIndex((user) => user.id === req.params.id);
    if (userIndex !== -1) {
      USER[userIndex] = {
        id: req.params.id,
        name: req.query.name,
        email: req.query.email,
      };
      res.send('회원 정보 수정 완료!');
    } else {
      res.send('해당 ID를 가진 회원이 존재하지 않습니다!');
    }
  } else {
    res.send('쿼리 입력이 잘못 되었습니다!');
  }
});
```

회원 삭제 미들웨어(API)

```
router.delete('/delete/:id', (req, res) => {
  const userIndex = USER.findIndex((user) => user.id === req.params.id);
  if (userIndex !== -1) {
    USER.splice(userIndex, 1);
    res.send('회원 삭제 완료');
  } else {
    res.send('해당 ID를 가진 회원이 존재하지 않습니다!');
  }
});
```

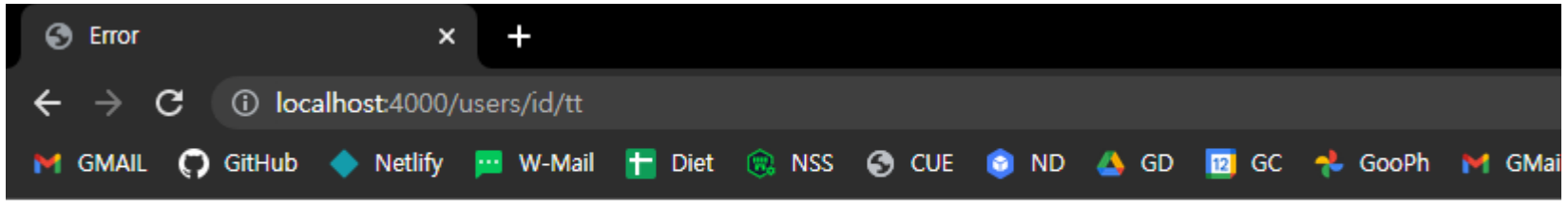
Error 핸들링!

Error 던지기!

```
router.get('/id/:id', (req, res) => {
  const userData = USER.find((user) => user.id === req.params.id);
  if (userData) {
    res.send(userData);
  } else {
    const err = new Error('해당 ID를 가진 회원이 없습니다!');
    err.statusCode = 404;
    throw err;
  }
});
```

- 이런 방식으로 err 가 발생한 지점에서 new Error 를 통해 err 객체를 만들어서 throw 로 전달하면 됩니다!

Error 던지기!




```
Error: 해당 ID를 가진 회원이 없습니다!
    at D:₩git₩KDT-5th-BE₩03_express₩routes₩users.js:29:17
    at Layer.handle [as handle_request] (D:₩git₩KDT-5th-BE₩node_modules₩express₩lib₩router₩layer.js:95:5)
    at next (D:₩git₩KDT-5th-BE₩node_modules₩express₩lib₩router₩route.js:144:13)
    at Route.dispatch (D:₩git₩KDT-5th-BE₩node_modules₩express₩lib₩router₩route.js:114:3)
    at Layer.handle [as handle_request] (D:₩git₩KDT-5th-BE₩node_modules₩express₩lib₩router₩layer.js:95:5)
    at D:₩git₩KDT-5th-BE₩node_modules₩express₩lib₩router₩index.js:284:15
    at param (D:₩git₩KDT-5th-BE₩node_modules₩express₩lib₩router₩index.js:365:14)
    at param (D:₩git₩KDT-5th-BE₩node_modules₩express₩lib₩router₩index.js:376:14)
    at Function.process_params (D:₩git₩KDT-5th-BE₩node_modules₩express₩lib₩router₩index.js:421:3)
    at next (D:₩git₩KDT-5th-BE₩node_modules₩express₩lib₩router₩index.js:280:10)
```

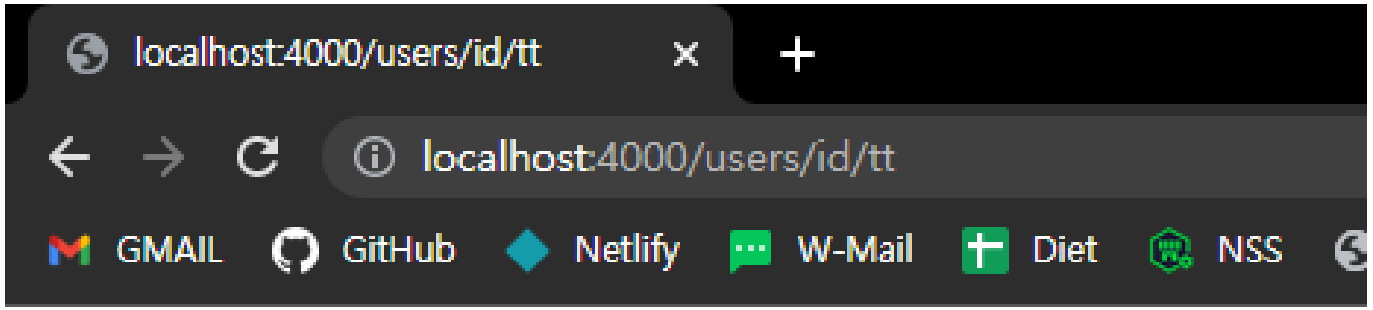
- 그럼 요렇게 Err 가 발생하는데 이런 페이지는 가급적 피하는 편이 좋으므로 app.js 에서 전에 Err 를 핸들링 해봅시다!

App.js 에서 에러 핸들링!

- App.js 의 미들 웨어중 마지막 미들웨어에 Throw 된 err 를 받는 미들웨어를 추가해 줍시다!

```
app.use((err, req, res, next) => {
  console.log(err.stack);
  res.status(err.statusCode);
  res.send(err.message);
});
```

- 해당 미들웨어는 err 인자와 res 를 같이 써야 하므로 app.use() 메소드의 매개변수를 전부 사용해 줘야 합니다. 3개만 쓰면 첫번째는 req, 두번째는 res, 세번째는 next 로 인식합니다 → 그리고 무엇보다 err 를 못받습니다!



해당 ID를 가진 회원이 없습니다!

App.js 에서 에러 핸들링!

- 서버 사이드(백엔드 개발자 입장)에서는 어떤 에러인지 알아야 하므로 console.log(err.stack) 을 내보내서 브라우저에서 보던 Error 내역을 확인 합니다!

```
Error: 해당 ID를 가진 회원이 없습니다!
    at D:\git\KDT-5th-BE\03_express\routes\users.js:28:17
    at Layer.handle [as handle_request] (D:\git\KDT-5th-BE\node_modules\express\lib\router\layer.js:95:5)
    at next (D:\git\KDT-5th-BE\node_modules\express\lib\router\route.js:144:13)
    at Route.dispatch (D:\git\KDT-5th-BE\node_modules\express\lib\router\route.js:114:3)
    at Layer.handle [as handle_request] (D:\git\KDT-5th-BE\node_modules\express\lib\router\layer.js:95:5)
    at D:\git\KDT-5th-BE\node_modules\express\lib\router\index.js:284:15
    at param (D:\git\KDT-5th-BE\node_modules\express\lib\router\index.js:365:14)
    at param (D:\git\KDT-5th-BE\node_modules\express\lib\router\index.js:376:14)
    at Function.process_params (D:\git\KDT-5th-BE\node_modules\express\lib\router\index.js:421:3)
    at next (D:\git\KDT-5th-BE\node_modules\express\lib\router\index.js:280:10)
```

Form 으로

데이터 전송하기!

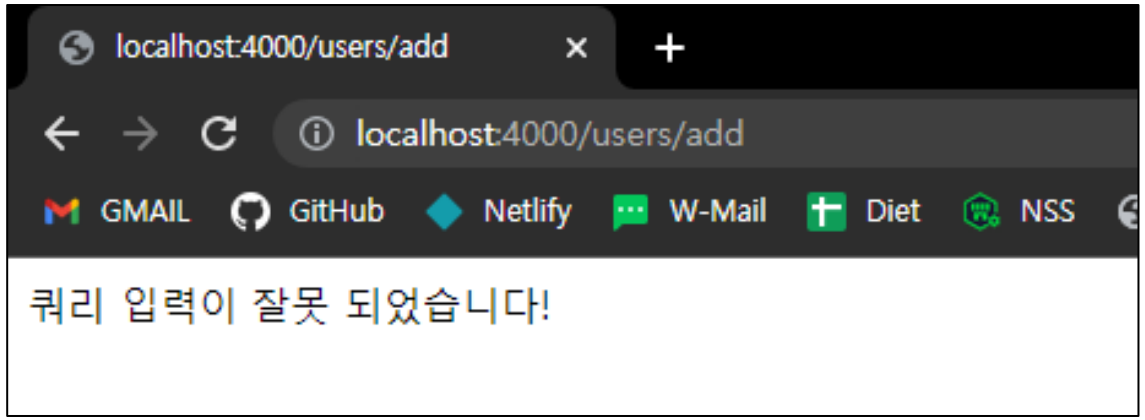
프론트의 form 으로 백에게 데이터 보내기!

- 지금까지는 postman 같은 프로그램을 통해서 서버 통신을 했지만 이번에는 프론트 페이지에서 서버로 데이터를 전달해 봅시다!
- Users.ejs 에 form 추가하기
- Form
  - Action 속성 → 보내고자 하는 주소 값이 됩니다.
  - Method 속성 → 보내는 method 설정
  - Input 의 name 속성은 서버에서 받을 때의 필드 값이 됩니다.
  - 버튼 type 으로 submit 을 하면 해당 폼의 내용을 지정한 방식 + 주소로 전달 합니다!

```
<form action="/users/add" method="POST">
    <div>
      <label>아이디</label>
      <input type="text" name="id" />
    </div>
    <div>
      <label>이름</label>
      <input type="text" name="name" />
    </div>
    <div>
      <label>이메일</label>
      <input type="email" name="email" />
    </div>
    <button type="submit">Submit</button>
  </form>
```

아이디
이름
이메일
Submit

그런데 데이터가 안들어 옵니다!



- 바로 Query 로 데이터가 들어오면 행복하겠지만, 폼으로 보낸 데이터는 query 로 값이 들어오지 않습니다!
- 다른 방식으로 받아야 합니다!

그런데 데이터가 안들어 옵니다!

- 이럴 때 편하게 사용하기 위해서 body-parser 라는 모듈을 사용합니다
- Form 에서 전송 된, 정보를 req.body 에 담아서 obj 로 전달해 주는 역할을 합니다
- Body-parser 를 사용하지 않으면 아래와 같은 코드로 데이터를 body 에 넣은 다음, 인코딩 처리 까지 해줘야 한다 → 자동으로 처리해줘서 편리함

```
req.on('data', function(chunk) { body += chunk; });
```

Body-parser 설치

- 설치를 해봅시다!
- Npm install body-parser --save
- Body-parser 는 실제로 프로젝트를 배포한 상태에서도 서버 통신에 사용되므로 --save-dev 가 아닌 --save 옵션으로 설치해야 합니다!

```
const bodyParser = require('body-parser');

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
```

Body-parser 사용

```
const bodyParser = require('body-parser');

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
```

- Json() 은 json 형태로 데이터를 전달한다는 의미 입니다!
- Urlencoded(url-encoded) 옵션은 url 처럼 데이터를 변환하면 localhost:4000/posts?title=title&content=content 해당 데이터를 json 형태 { “title”: “title, “content”: “content” } 라고 전달 합니다.

Extended: false

- query 로 들어온 문자열을
  - true → 외부 모듈인 qs 모듈로 처리 (qs 모듈 설치 필요, npm i qs)
  - false → express 내장 모듈인 queryString 모듈로 처리
- False 옵션은 중첩된 객체 허용 X (https://sjh836.tistory.com/154)

- **qs** library allows you to create a **nested** object from your query string.

```
var qs = require("qs")
var result = qs.parse("person[name]=bobby&person[age]=3")
console.log(result) // { person: { name: 'bobby', age: '3' } }
```

- **query-string** library **does not** support creating a nested object from your query string.

```
var queryString = require("query-string")
var result = queryString.parse("person[name]=bobby&person[age]=3")
console.log(result) // { 'person[age]': '3', 'person[name]': 'bobby' }
```

Body-parser 은 이제 기본으로 제공 됩니다!

- 하도 많이 써서 이제는 express 에 기본 기능으로 추가가 되었습니다.

```
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
```

- 단, express 4.16 이상에서만 먹힙니다!

```
"dependencies": {
  "body-parser": "^1.20.0",
  "express": "^4.18.1",
  "yarn": "^1.22.19"
},
```

Body-parser 의 동작에 대해서 알아 봅시다

- 먼저 body-parser 를 쓰지 않은 상태에서 req.body 값을 찍어서 확인해 봅시다!

```
// app.use(express.json());
// app.use(express.urlencoded({ extended:
false }));
```

```
router.post('/add', (req, res) => {
  console.log(req.body);
  if (req.query.id && req.query.name && req
    const newUser = {
```

```
[nodemon] starting `node rout
4000 번에서 서버 실행
undefined
```

Body-parser 의 동작에 대해서 알아 봅시다

- 앞서 말한 내용 처럼 데이터 처리 및 인코딩 처리가 없어서 undefined 가 찍힙니다!

```
The express server is running at port: 4000
undefined
```

- 주석을 풀고 다시 요청을 보내면?!

```
The express server is running at port: 4000
[Object: null prototype] {
  id: '121',
  name: '2121',
  email: '212@gg.com'
}
```

Req.body 를 처리

- 이제 form 에서 전달 된 정보는 req.body 를 통해 들어 옵니다! → 해당 데이터 처리를 위한 코드 추가!
- Req.query 값은 전달하지 않아도 빈 객체를 전달 하는데 빈 객체는 if 문에서 true 값을 리턴 하므로 다른 방식으로 예외 처리하기!
- 회원 가입이 완료 되면, 다시 회원 목록 페이지를 보여줘서 회원 추가가 된 것을 바로 확인 할 수 있도록 처리!

```
res.redirect('/users');
```

```
router.post('/add', (req, res) => {
  if (Object.keys(req.query).length >= 1) {
    if (req.query.id && req.query.name && req.query.email) {
      const newUser = {
        id: req.query.id,
        name: req.query.name,
        email: req.query.email,
      };
      USER.push(newUser);
      res.send('회원 등록 완료');
    } else {
      const err = new Error('쿼리 입력이 잘못 되었습니다!');
      err.statusCode = 404;
      throw err;
    }
  } else if (req.body) {
    if (req.body.id && req.body.name && req.body.email) {
      const newUser = {
        id: req.body.id,
        name: req.body.name,
        email: req.body.email,
      };
      USER.push(newUser);
      res.redirect('/users');
    } else {
      const err = new Error('쿼리 입력이 잘못 되었습니다!');
      err.statusCode = 404;
      throw err;
    }
  } else {
    const err = new Error('No data');
    err.statusCode = 404;
    throw err;
  }
});
```

- ID : 11

  NAME : 11

  EAMIL : 11@11

실습, posts 서비스 만들기!

- Users 서비스를 만든 것 처럼, Posts 서비스를 만들면 됩니다!
- Posts 서비스의 데이터는 글의 title, content 를 가지고 있습니다.
- 서비스
  - 글 목록을 보여주는 페이지 : GET localhost:4000/posts
  - 글 추가 : POST localhost:4000/posts/add
- 글 추가는 form 으로 title 과 content 를 받아서 추가 합니다
- 모든 기능은 routes/posts.js 에 정의 되어 있어야 합니다!

Front 에서

Back 으로 요청 보내기

Form 은 데이터를 서버로 보내기 위한 태그

- 그렇기 때문에 기본 속성 값에 url 주소를 받기위한 action 과, 타입을 받기 위한 method 를 가지고 있습니다.
- 그렇다면, form 없이 Back-end 로 데이터를 보내는 방법은 없을까요!?
- 당연히 있습니다!
  - XMLHttpRequest
  - JQuery
  - Fetch()

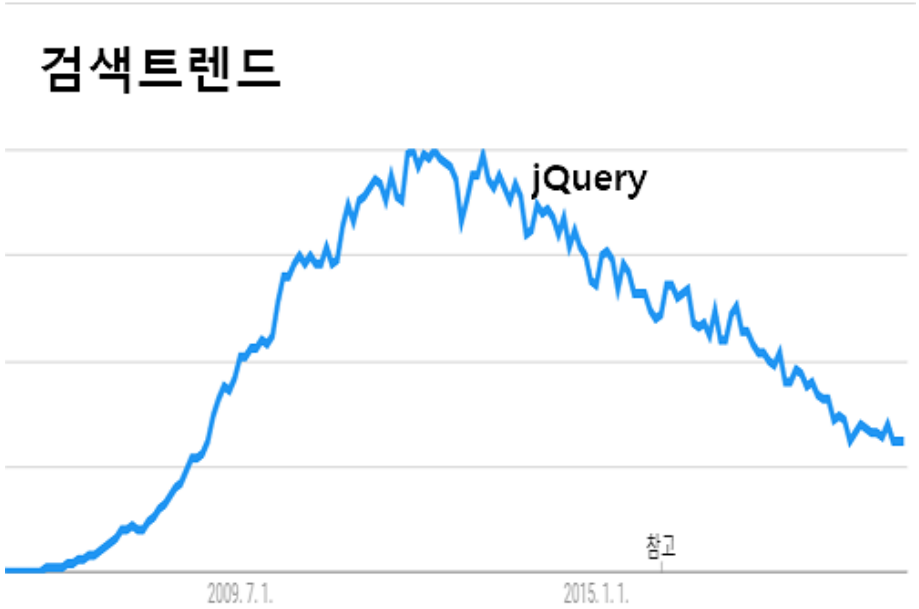
XMLHttpRequest

- 1999년에 클라이언트에서 서버 측 데이터를 받기위해 탄생 하였습니다.
- 아무래도 옛날 방식인 만큼, 최신 기능인 Promise 등을 기본으로 내장하지 못하고 있기 때문에 서버 통신을 동기적으로 처리 하려면 콜백 함수를 제외하면, Promise 를 따로 사용해야 하는 불편함이 있습니다
- ES6 문법이 등장하기 전 까지는 주로 JQuery 를 통해 통신을 했었습니다.

```
<script>
  function deleteUser(id) {
    const xhr = new XMLHttpRequest();
    xhr.open('DELETE', `http://localhost:4000/users/${id}`);
    xhr.responseType = 'json';
    xhr.onload = () => {
      console.log(xhr.response);
      location.reload();
    };
    xhr.send();
  }
</script>
```

JQuery

- 기존 구린 JS 의 구세주 역할을 하던 라이브러리 입니다.
- 2006년 등장하여 편리한 DOM 기능, 데이터 통신 등 다양한 분야에서 사용
- JQuery 의 기능들이 JS 에 기본 내장 되면서 점차 사용이 줄어드는 추세



```
<script>
  function deleteUser(id) {
    console.log(`http://localhost:4000/users/${id}`);
    $.post(`http://localhost:4000/users/${id}`, function (res) {
      console.log(res);
      location.reload();
    }).done(function (data) {
      console.log(data);
    }).fail(function (err) {
      console.log(err);
    })
  }
</script>
```

Fetch

- 브라우저에서 서버 사이드 통신을 위해 2015년 ES6 에서 추가된 기능입니다.
- 기존의 XMLHttpRequest 의 문제점을 개선하고 Promise 를 기본으로 내장하여 서버 통신을 코드를 백엔드 코드와 비슷하게 짤 수 있습니다!

```
<script>
  function deleteUser(id) {
    fetch(`http://localhost:4000/users/${id}`, {
      method: 'delete',
      headers: {
        'Content-type': 'application/json'
      },
    }).then((res) => {
      console.log(res);
      location.reload();
    })
  }
</script>
```

Fetch 를 이용해서

삭제 기능 구현!

Fetch를 이용한 삭제 기능 구현!

- 삭제 기능 자체는 이미 Back-end API에 잘 구현이 되어 있습니다
- 우리가 할 일은 fetch 를 이용해서 back-end 에 요청만 잘 보내면 됩니다!
- 먼저, li 요소에 삭제 버튼을 추가해 봅시다!
- 삭제 버튼은 간단하게 <a> 태그로 구현
- <a> 태그에 onclick 으로 삭제 함수를 연결 → 해당 함수에서 fetch 로 삭제 api 요청!

```
<ul>
    <% if(userCounts> 0) { %>
      <% for(let i=0; i < userCounts; i++) { %>
        <li>
          <p>ID: <%= USER[i].id %>
          </p>
          <p>이름: <%= USER[i].name %>
          </p>
          <p>email: <%= USER[i].email %>
          </p>
          <a href="" onclick="deleteUser();">삭제</a>
        </li>
        <% } %>
    <% } else { %>
        <li>
          회원 정보가 없습니다!
        </li>
    <% } %>
  </ul>
```

삭제를 하려면!?

- USER 의 id 를 알아야만 삭제가 가능합니다!
- 그럼, deleteUser 에 어떤 방식으로 id 를 전달하면 될까요!?

Ejs 를 활용하면 됩니다!

- USER[i].id 값을 전달하면 되므로

```
<a href="" onclick="deleteUser('<%= USER[i].id %>');">삭제</a>
```

- 위와 같이 deleteUser 함수의 매개 변수로 USER[i].id 값을 전달 합니다!

deleteUser(id) 함수 구현

- 그럼 실제적으로 삭제 기능을 하는 deleteUser 함수를 만들어 봅시다
- fetch 의 주소로는 매개 변수로 받은 id 값을 넣어서 전달!

```
fetch(`http://localhost:4000/users/delete/${id}`,
```

- 전달 값으로는 method, headers 를 전달 (데이터 전달이 필요하면 body 에 담아서 전달)

```
{
      method: 'delete',
      headers: {
        'Content-type': 'application/json'
      },
    }
```

deleteUser(id) 함수 구현

- 통신이 완료 되면 then 으로 받습니다.

```
}).then((res) => {
      console.log(res);
      location.reload();
    })
```

Promise 버전

```
<script>
  function deleteUser(id) {
    fetch(`http://localhost:4000/users/${id}`, {
      method: 'delete',
      headers: {
        'Content-type': 'application/json'
      },
    }).then((res) => {
      console.log(res);
      location.reload();
    })
  }
</script>
```

Async/Await 버전

```
<script>
    async function deleteUser(id) {
      const res = await fetch(`http://localhost:4000/users/delete/${id}`, {
        method: 'delete',
        headers: {
          'Content-type': 'application/json'
        }
      });

      if (res) location.reload();
    }
  </script>
```

실습, 삭제 버튼을 posts 에도 추가!

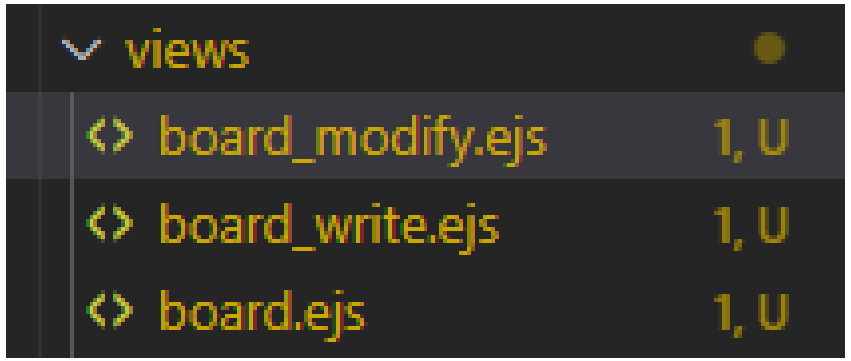
- Posts.ejs 파일에도 post 삭제를 위한 버튼을 추가하고 하고, 해당 버튼으로 글을 삭제 할 수 있도록 만들어 주세요! ☺

게시판

만들기!!

View part!

- 프론트 코드는 제가 작업한 파일을 바탕으로 사용하겠습니다!
  - https://github.com/xenosign/backend2/blob/main/views/board.ejs
  - https://github.com/xenosign/backend2/blob/main/views/board_write.ejs
  - https://github.com/xenosign/backend2/blob/main/views/board_modify.ejs



Board.js 로 라우팅!

- 먼저 board.js 파일을 만들고 express 모듈 추가 등의 기본 코드만을 추가하고 모듈화 작업

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

const router = express.Router();

module.exports = router;
```

- App.js 에서 해당 모듈 불러오고 주소 라우팅 설정

```
const boardRouter = require('./routes/board’);
app.use('/board', boardRouter);
```

Board.js 기본 기능 정의

- 글 전체 목록 보여주기
  - GET /board/ → 글 전체 목록 보여주기
- 글 쓰기
  - GET /board/write → 수정 모드로 이동
  - POST /board/ → 글 추가 기능 수행
- 글 수정
  - GET /board/modify → 글 수정 모드로 이동
  - POST /board/title/:title → 파라미터로 들어온 title 값과 동일한 글을 수정
- 글 삭제
  - DELETE /board/title/:title → 파라미터로 들어온 title 값과 동일한 글을 삭제

Board.js 기본 코드 작업

- 데이터로 사용할 배열 선언

```
const ARTICLE = [
  {
    title: 'title',
    content:
      'Lorem ipsum, dolor sit amet consectetur adipisicing elit. Quia delectus iusto
fugiat autem cupiditate adipisci quas, in consectetur repudiandae, soluta, suscipit
debitis veniam nobis aspernatur blanditiis ex ipsum tempore impedit.',
  },
  {
    title: 'title2',
    content:
      'Lorem ipsum, dolor sit amet consectetur adipisicing elit. Quia delectus iusto
fugiat autem cupiditate adipisci quas, in consectetur repudiandae, soluta, suscipit
debitis veniam nobis aspernatur blanditiis ex ipsum tempore impedit.',
  },
];
```

Board.js 기본 코드 작업

- 요청 주소 미들 웨어 만들기

```
router.get('/', (req, res) => {
  // 글 전체 목록이 보이는 페이지
});

router.get('/write', (req, res) => {
  // 글 쓰기 모드로 이동
});

router.post('/write', (req, res) => {
  // 글 추가
});

router.get('/modify/:title', (req, res) => {
  // 글 수정 모드로 이동
});

router.post('/modify/:title', (req, res) => {
  // 글 수정
});
router.delete('/delete/:title', (req, res) => {
  // 글 삭제
});
```

전체 목록 보여주기

전체 목록 보여주기

- 글 전체 목록을 데이터에 담아서 board.ejs 파일 그려주기
- res.render(뷰파일명, 데이터); 사용
- Ejs 코드에서 사용한 변수명과 일치하는지 체크 필요

```
router.get('/', (req, res) => {
  const articleCounts = ARTICLE.length;
  res.render('board', { ARTICLE, articleCounts });
});
```

```
div class="board_write">
        <span>현재 등록 글 :   <%= articleCounts %></span>
        <a class="btn red" href="/board/write">글쓰기</a>
      </div>
      <div class="board_body">
        <ul class="board">
          <% if (articleCounts> 0) { %>
            <% for(let i=0; i < articleCounts; i++) { %>
              <li>
                <div class="title">
                  <%= ARTICLE[i].title %>
```

글 쓰기 모드로 이동

글 쓰기 모드로 이동

- Board.ejs 뷰에서 글쓰기 버튼을 클릭 → /board/write 라는 주소로 이동
- 해당 요청이 들어오면 글 쓰기 페이지 board_write.ejs 를 그려주기

```
<div class="board_write">
    <span>현재 등록 글 :   <%= articleCounts %></span>
    <a class="btn red" href="/board/write">글쓰기</a>
</div>
```

- 데이터 전달 필요 X

```
router.get('/write', (req, res) => {
  res.render('board_write');
});
```

글 추가 기능

글 추가 기능(프론트)

- 글쓰기 모드에서 입력 받은 title, content 를 새로운 글로 추가하기
- 아무것도 안 쓸 경우의 예외를 처리하기 위해 필수 입력 지정(required)
- 주소는 /board/write, 메소드는 POST 로 전달

```
<form action="/board/write" method="POST" class="board_form">
    <div class="form_title">
      <h3>제목</h3>
      <input type="text" name="title" required />
    </div>
    <div class="form_content">
      <h3>내용</h3>
      <textarea name="content" id="content" cols="30" rows="10" required></textarea>
    </div>
    <button type="submit">글 작성하기</button>
</form>
```

글 추가 기능(백)

- 주소는 /board/write 로 들어왔으므로 /write 처리
- 메소드는 post 이므로 router.post('/write', (req, res)= > {}) 로 처리
- 프론트에서 예외 처리를 하였지만, 데이터가 누락되는 경우를 대비

실습, 추가 기능 백엔드 코드 작성하기!

- Req.bdoy 로 들어온 title / content 를 백엔드의 ARTICLE 배열에 추가하는 코드를 만들어 봅시다!
- 추가가 완료 되면 /board 로 리다이렉트 해줘서 추가 된 글이 바로 보이도록 설정해 주세요!
- Req.body 의 값이 잘 들어오지 않았으면 Err 를 발생 시키면 됩니다!

글 수정 모드로 이동

글 수정 모드로 이동

- 뷰에서 수정 버튼을 클릭 → /board/modify/:title 라는 주소로 이동
- title 파라미터는 ejs 에서 직접 입력하여 전달

```
<a class="btn orange" href="board/modify/<%= ARTICLE[i].title %>">수정</a>
```

- 데이터 전달 필요 O, ARTICLE 배열에서 제목이 같은 글을 찾아 전달

```
router.get('/modify/:title', (req, res) => {
  const arrIndex = ARTICLE.findIndex(
    (article) => article.title === req.params.title
  );
  const selectedArticle = ARTICLE[arrIndex];
  res.render('board_modify', { selectedArticle });
});
```

글 수정 모드 뷰 페이지

- 전달 받은 데이터(selectedArticle)를 <input> <textarea> 값으로 전달
- 데이터 전송은 수정할 글의 title 을 파라미터로 담아서

  board/modify/:title 로 전달 / 메소드는 POST

```
<form action="/board/modify/<%=selectedArticle.title%>" method="POST" class="board_form">
   <div class="form_title">
     <h3>제목</h3>
     <input type="text" name="title" value="<%= selectedArticle.title %>" required />
   </div>
   <div class="form_content">
     <h3>내용</h3>
     <textarea name="content" id="content" cols="30" rows="10" required>
     <%= selectedArticle.content %></textarea>
   </div>
  <button type="submit">글 수정하기</button>
</form>
```

PUT!?

- 순수 HTML은 데이터를 전송할 때 GET, POST 만 지원했었습니다
- 그래서 method 속성에 PUT, DELETE 를 입력해도 정상적으로 전달 X
- 이럴 때 PUT, DELETE 로 전달하고 싶다면?

```
<form action="#" method="post">
        <input type="hidden" name="_method" value="put" />
  </form>
```

- 따라서 보통 수정, 삭제도 POST 로 구현하고 주소 또는 데이터로 구분하는 경우가 많아요 → 그래서 POST를 쓸 겁니다! ☺
- 그리고 form 태그에 한해서만 POST, PUT, DELETE 전송이 가능합니다!

글 수정 기능

글 수정 기능(백)

- 주소는 /board/modify/:title 로 요청이 들어 옵니다!
- 메소드는 post 이므로 router.post('/modify/:title', (req, res)= > {})
  로 처리
- 프론트에서 예외 처리를 하였지만, 데이터가 누락되는 경우를 대비

```
router.post('/modify/:title', (req, res) => {
  if (req.body.title && req.body.content) {
    const arrIndex = ARTICLE.findIndex(
      (_article) => _article.title === req.params.title
    );
    if (arrIndex !== -1) {
      ARTICLE[arrIndex].title = req.body.title;
      ARTICLE[arrIndex].content = req.body.content;
      res.redirect('/board');
    } else {
      const err = new Error('해당 제목의 글이 없습니다.');
      err.statusCode = 404;
      throw err;
    }
  } else {
    const err = new Error('요청 쿼리 이상');
    err.statusCode = 404;
    throw err;
  }
});
```

글 삭제 기능

글 삭제 기능(프론트)

- 삭제는 DELETE 요청을 해야 합니다! → 그런데 지금 우리는 A 태그에서만 요청을 보낼 수 있습니다!
- 따라서, 이전에 배운 JS의 fetch 를 사용합니다!

```
<a class="btn blue" href="#" onclick="deleteArticle('<%= ARTICLE[i].title %>')">삭제</a>
```

실습, 삭제 기능 프론트 & 백엔드 코드 작성

- 프론트에서 Fetch 를 이용해서 글 삭제 요청을 보내는 코드와, 백엔드에서 이를 받아서 실제적으로 글을 삭제하는 코드를 작성해 주세요!


수고하셨습니다!