Hello,

KDT 웹 개발자 양성 프로젝트

5기!

with

Callback!?

CALLBACK!!

```
function func1(callback) {
  console.log("1번 함수");
  callback();
}

function func2() {
  console.log("2번 함수");
}

function func3() {
  console.log("3번 함수");
}

func1(func2);
```

함수의 매개 변수로
함수를 전달!

즉, 그때 그때 상황에 맞게
다른 함수를 전달 할 수 있습니다!

전달 받은 함수를 함수 내부에서
호출!

즉, 순차적 실행을 안정적으로
수행 가능!!

Func1 함수에서
Func2 함수를 콜백으로 전달!

```
lhs@DESKTOP-86MUCGC
$ node cb.js
1번 함수
2번 함수
```

이런 것도!? 가능!!

```
function func1(callback) {
  console.log("1번 함수");
  callback();
}

function func2() {
  console.log("2번 함수");
}

function func3() {
  console.log("3번 함수");
}

func1(func3);
```

```
lhs@DESKTOP-86MUCGC
$ node cb.js
1번 함수
3번 함수
```

꼭 선언된 함수만 불러야 하나?

```
function func1(callback) {
  console.log("1번 함수");

  function fakeFunc3() {
    console.log("3번 인척 하는 함수");
  }

  callback(fakeFunc3);
}

function func2(callback) {
  console.log("2번 함수");
  callback();
}
function func3() {
  console.log("3번 함수");
}

func1(func2);
```

```
lhs@DESKTOP-86MUCGC MIN
$ node cb.js
1번 함수
2번 함수
3번 인척 하는 함수
```

그렇다면!? (2)

```
function func1(callback) {
  console.log("1번 함수");
  callback();
}
function func2(callback) {
  console.log("2번 함수");
  callback();
}
function func3() {
  console.log("3번 함수");
}
func1(func2);
```

```
function func1(callback) {
  console.log("1번 함수");
  callback();
}
function func2(callback) {
  console.log("2번 함수");
  callback();
}
function func3() {
  console.log("3번 함수");
}

func1(function () {
  console.log("2번 인척하는 새로 만든 익명 함수!");
});
```

```
lhs@DESKTOP-86MUCGC MINGW64 /d/git/kdt-5th (main)
$ node cb.js
1번 함수
2번 인척하는 새로 만든 익명 함수!
```

```
function multiplication(number, callback) {
  let answer = 0;
  setTimeout(function () {
    answer = number * number;
    callback(answer);
  }, 2000);
}

function say(result) {
  console.log(result);
}

multiplication(3, say);
```

계산한 결과 값을 인자로 전달

전달 받은 값을 출력!

```
lhs@DESKTOP-86MUC
$ node cb.js
9
```


여기서 잠깐!!

Callback

+ 익명함수

```
function buySync(item, price, quantity, callback) {
  console.log(`${item} 상품을 ${quantity} 개 골라서 점원에게 주었습니다.`);
  setTimeout(function () {
    console.log("계산이 필요합니다.");
    const total = price * quantity;
    callback(total);
  }, 1000);
}

// function pay(tot) {
//   console.log(`${tot} 원을 지불하였습니다.`);
// }

buySync('포켓몬빵', 1000, 5, function (total) {
  console.log(`${total} 원을 지불하였습니다.`);
});
```


런닝맨
SBS
나 분명히 어딘가에 생긴
사람을 본 적이 있는데...

```
const list = document.querySelector(".list");

list.addEventListener("click", function (e) {
  console.log(e.target);
});
```

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

const PORT = 4000;

const app = express();

app.use(cors());

app.use('/', (req, res) => {
  const str = '안녕하세요. 여기는 백엔드 입니다!';
  const json = JSON.stringify(str);
  res.send(json);
});

app.listen(PORT, () => {
  console.log(`데이터 통신 서버가 ${PORT}에서 작동 중입니다!`);
});
```

Callback

Hell?

콜백 지옥?

```
function funcHell(callback) {
  callback();
}

funcHell(function () {
  console.log("1번 인척하는 새로 만든 익명 함수!");
  funcHell(function () {
    console.log("2번 인척하는 새로 만든 익명 함수!");
    funcHell(function () {
      console.log("3번 인척하는 새로 만든 익명 함수!");
    });
  });
});
```

```
lhs@DESKTOP-86MUCGC MINGW64 /d/git/kdt-5th (main)
$ node cb.js
1번 인척하는 새로 만든 익명 함수!
2번 인척하는 새로 만든 익명 함수!
3번 인척하는 새로 만든 익명 함수!
```

콜백 지옥? 그게 뭐죠?

callbackhell.js

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

floppy.load('disk1', function (data1) {
    floppy.prompt('Please insert disk 2', function() {
        floppy.load('disk2', function (data2) {
            floppy.prompt('Please insert disk 3', f
                floppy.load('disk3', function (data
                    floppy.prompt('Please insert di
                        floppy.load('disk4', functi
                            floppy.prompt('Please i
                                floppy.load('disk5'
                                    floppy.prompt('
                                        floppy.load
                                            //if no
                                        });
                                    });
                                });
                            });
                        });
                    });
                });
            });
        });
    });
});
```

```
foo(() => {
    bar(() => {
        baz(() => {
            qux(() => {
                quux(() => {
                    quuz(() => {
                        corge(() => {
                            grault(() => {
                                run();
                            }).bind(this);
                        }).bind(this);
                    }).bind(this);
                }).bind(this);
            }).bind(this);
        }).bind(this);
    }).bind(this);
}).bind(this);
```

파일 읽기!

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

fs.readFile('readme.txt', 'utf-8', function (err, data) {
  if (err) {
    console.log(err);
  } else {
    console.log(data);
  }
});
```

backend/file.js

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

fs.readFile('readme.txt', 'utf-8', (err, data) => {
  if (err) {
    console.log(err);
  } else {
    console.log(data);
  }
});
```

backend/file.js

파일 쓰기!

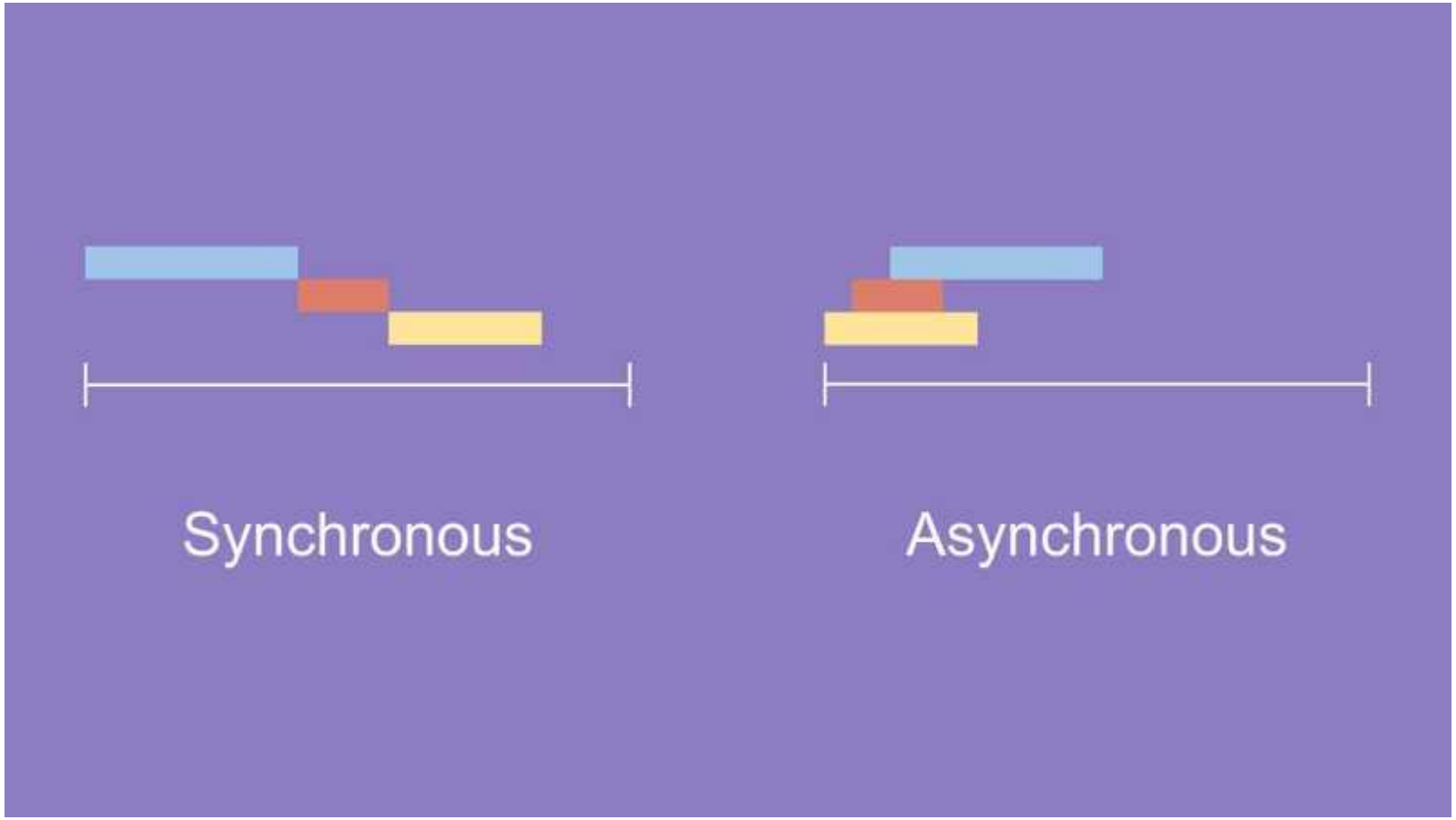
```
const fs = require('fs');

const str = '파일 쓰기가 정상적으로 되는지 테스트 합니다.';

fs.writeFile('readme.txt', str, 'utf-8', (err) => {
  if (err) {
    console.log(err);
  } else {
    console.log('writefile succeed');
  }
});
```

backend/file.js

File-system 과 비동기 프로그래밍

backend/file.js

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

fs.readFile('./readme.txt', (err, data) => {
  if (err) {
    throw err;
  }
  console.log('1번', data.toString());
});
fs.readFile('./readme.txt', (err, data) => {
  if (err) {
    throw err;
  }
  console.log('2번', data.toString());
});
fs.readFile('./readme.txt', (err, data) => {
  if (err) {
    throw err;
  }
  console.log('3번', data.toString());
});
fs.readFile('./readme.txt', (err, data) => {
  if (err) {
    throw err;
  }
  console.log('4번', data.toString());
});
```


Synchronous
Asynchronous

Callback

Hell?!

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

fs.readFile('./readme.txt', (err, data) => {
  if (err) {
    throw err;
  }
  console.log('1번', data.toString());
  fs.readFile('./readme.txt', (err, data) => {
    if (err) {
      throw err;
    }
    console.log('2번', data.toString());
    fs.readFile('./readme.txt', (err, data) => {
      if (err) {
        throw err;
      }
      console.log('3번', data.toString());
      fs.readFile('./readme.txt', (err, data) => {
        if (err) {
          throw err;
        }
        console.log('4번', data.toString());
      });
    });
  });
});
```

backend/cbHell.js

```
lhs@DESKTOP-86MUCGC MINGW64 ~
$ node js/file.js
1번 readme 텍스트 입니다
2번 readme 텍스트 입니다
3번 readme 텍스트 입니다
4번 readme 텍스트 입니다

lhs@DESKTOP-86MUCGC MINGW64 ~
$ node js/file.js
1번 readme 텍스트 입니다
2번 readme 텍스트 입니다
3번 readme 텍스트 입니다
4번 readme 텍스트 입니다

lhs@DESKTOP-86MUCGC MINGW64 ~
$ node js/file.js
1번 readme 텍스트 입니다
2번 readme 텍스트 입니다
3번 readme 텍스트 입니다
4번 readme 텍스트 입니다
```

Promise!

Promise!

- Promise 는 생성자 입니다! 따라서 new 로 사용하죠!

```
const promise = new Promise(function(resolve, reject) {});
```

- resolve, reject 라는 2개의 콜백 함수를 받아서 사용합니다.
- promise 가 할당 되면 이 promise 는 resolve 또는 reject 함수가 callback 될 때 까지 무한 대기 합니다.
- Resolve 는 promise 가 정상적으로 이행 되었을 경우 사용하며, reject 는 반대의 경우에 사용합니다.

```
const promise = new Promise(function (resolve, reject) {
  const tetz = 'old';
  if (tetz === 'old') {
    setTimeout(() => {
      resolve('tetz is old');
    }, 3000);
  } else {
    reject('tetz is getting old');
  }
});

promise
  .then(function (data) {
    console.log(data);
  })
  .catch(function (data) {
    console.log(data);
  });
```

backend/promise.js

```
const fs = require('fs').promises;

fs.readFile('./readme.txt')
  .then((data) => {
    console.log('1번', data.toString());
    return fs.readFile('./readme.txt');
  })
  .then((data) => {
    console.log('2번', data.toString());
    return fs.readFile('./readme.txt');
  })
  .then((data) => {
    console.log('3번', data.toString());
    return fs.readFile('./readme.txt');
  })
  .then((data) => {
    console.log('4번', data.toString());
  })
  .catch((err) => {
    throw err;
  });
```

backend/promiseHell.js

```
lhs@DESKTOP-86MUCGC MINGW64 ~/
$ node js/file.js
1번 readme 텍스트 입니다
2번 readme 텍스트 입니다
3번 readme 텍스트 입니다
4번 readme 텍스트 입니다

lhs@DESKTOP-86MUCGC MINGW64 ~/
$ node js/file.js
1번 readme 텍스트 입니다
2번 readme 텍스트 입니다
3번 readme 텍스트 입니다
4번 readme 텍스트 입니다

lhs@DESKTOP-86MUCGC MINGW64 ~/
$ node js/file.js
1번 readme 텍스트 입니다
2번 readme 텍스트 입니다
3번 readme 텍스트 입니다
4번 readme 텍스트 입니다
```

Async / Await

Await

Syntactic Sugar

{Syntactic Sugar}

- 같은 기능을 하지만 문법 상으로 더 편리하게 바꿔 주는 것을 Syntactic Sugar 라 부릅니다!
- Async, Await 는 promise 의 Syntactic Sugar 입니다!
- Promise 는 기존에 JS 코드 스타일과 다르기 때문에 promise 를 기존 코드 스타일로 사용할 수 있도록 만들어 준 것이 Async, Await 입니다!

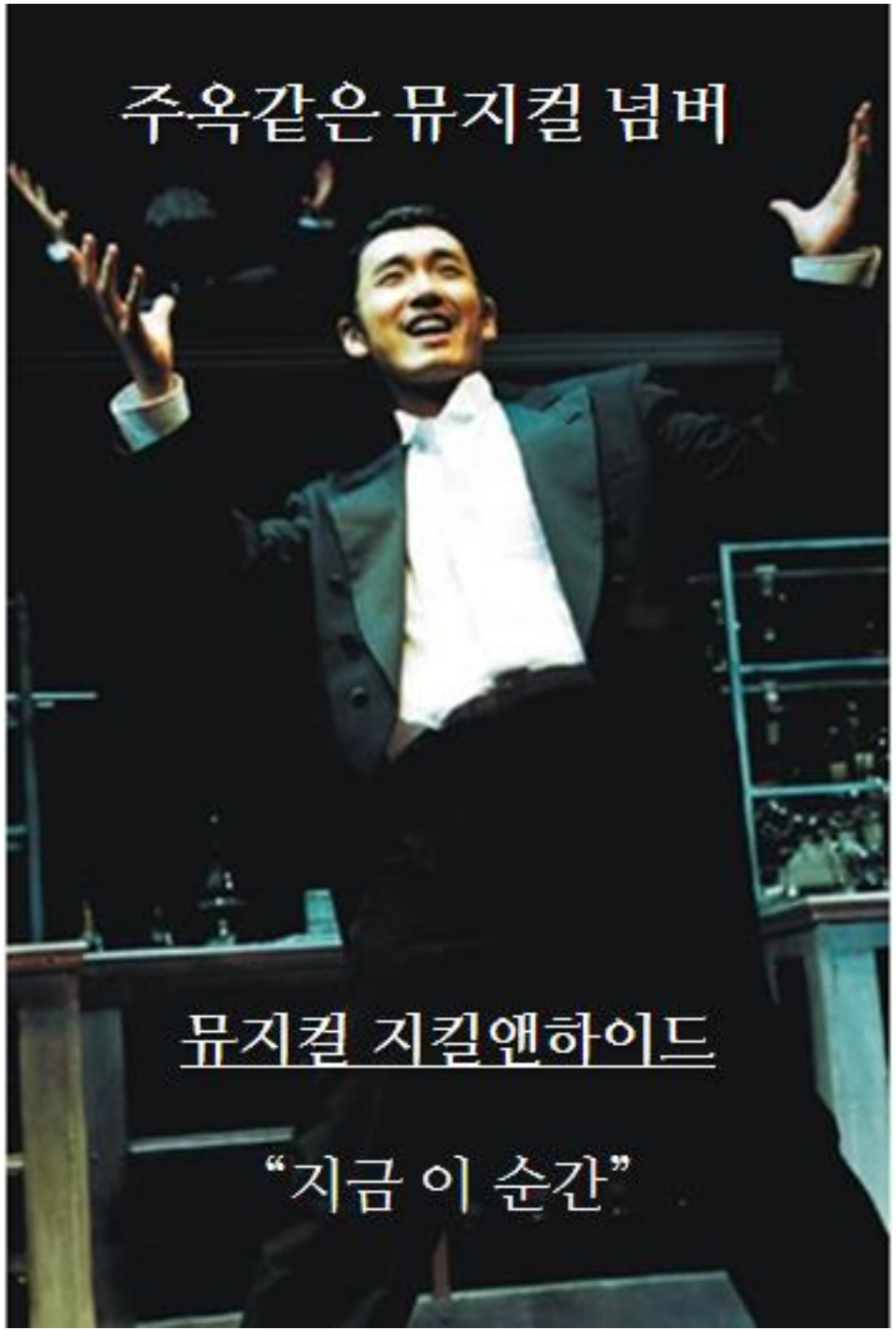
```
promise
  .then(function (profit) {
    console.log(`기다림의 보상으로 ${profit} 원을 벌었습니다!`);
  })
  .catch(function (err) {
    console.log(err);
  });
```

```
async function howMuch() {
  const result = await promise;

  if (result) {
    console.log(`기다림의 보상으로 ${result} 원을 벌었습니다!`);
  }
}

howMuch();
```

```js
const fs = require('fs').promises;

async function main() {
  let data = await fs.readFile('./readme.txt');
  console.log('1번', data.toString());
  data = await fs.readFile('./readme.txt');
  console.log('2번', data.toString());
  data = await fs.readFile('./readme.txt');
  console.log('3번', data.toString());
  data = await fs.readFile('./readme.txt');
  console.log('4번', data.toString());
}

main();
```

```
lhs@DESKTOP-86MUCGC MINGW64
$ node js/file.js
1번 readme 텍스트 입니다
2번 readme 텍스트 입니다
3번 readme 텍스트 입니다
4번 readme 텍스트 입니다

lhs@DESKTOP-86MUCGC MINGW64
$ node js/file.js
1번 readme 텍스트 입니다
2번 readme 텍스트 입니다
3번 readme 텍스트 입니다
4번 readme 텍스트 입니다

lhs@DESKTOP-86MUCGC MINGW64
$ node js/file.js
1번 readme 텍스트 입니다
2번 readme 텍스트 입니다
3번 readme 텍스트 입니다
4번 readme 텍스트 입니다
```

node.js
+
express



Express

http://expressjs.com/

Postman

https://www.postman.com/downloads/

Express Middleware

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

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

app.use('/', (req, res, next) => {
  console.log('Middleware 1');
  next();
});

app.use((req, res, next) => {
  console.log('Middleware 2');
  res.send('res.send!');
  next();
});

app.use((req, res, next) => {
  console.log('Middleware 3');
});

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

```
The express server is running at port: 4000
Middleware 1
Middleware 2
Middleware 3
```

요청 메소드

Create ➡ POST

Read ➡ GET

Update ➡ PUT

Delete ➡ DELETE

HTTP Status Codes

1XX
INFORMATIONAL

2XX
SUCCESS

3XX
REDIRECTION

4XX
CLIENT ERROR

5XX
SERVER ERROR

백엔드에서

데이터를 받는 방법

Req.Params

URL 로 Data 를 받는 방법, Params

- 여러 개를 받을 때, 어떤 값을 전달하는지 보내는 쪽에서 인지 시키기 위해서 이렇게 사용도 가능합니다.

```
app.get('/id/:id/name/:name/gender/:gender', (req, res) => {
  console.log(req.params);
  res.send(req.params);
});
```


```
{
   "id": "1",
   "name": "tetz",
   "gender": "male"
}
```

Req.Query

```
app.get('/', (req, res) => {
  console.log(req.query);
  res.send(req.query);
});
```

GET | localhost:4000?test=test&id=3

Params ● | Authorization | Headers (8) | Body | Pre-request Script | Tests | Settings

KEY	VALUE
☑ test	test
☑ id	3


지금 시작합니다

Express Routing

API Routing

- 프론트에서 백엔드로 요청을 보낼 때는 주소 값을 다르게 보내서 요청을 합니다!
- 따라서 백엔드에서는 각각 주소에 따라서 각기 다른 역할을 해주면 됩니다!
- 이렇게 주소에 따라서 각기 다른 역할을 하도록 나누는 방법을 Routing 이라고 합니다!
- Express 에서는 app.메소드명(); 의 형태로 요청 방식을 나누어 처리 할 수 있습니다!

메소드 별

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()

Express Router

- 당연히 Express 에는 Routing 을 위한 미들웨어도 존재 합니다!
- Router() 를 사용하면 특정 url 요청에 대한 것들을 묶어서 처리가 가능합니다!
- 예를 들어 회원 정보에 관한 요청은 전부 /users 이하의 요청으로 받는 방식이죠!

```
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}`);
});
```

간단한

데이터 처리

Express 를 이용해서 간단한 API 만들기

- 아래와 같은 API를 만들 예정입니다!
- GET Localhost:4000/users
  - 회원 목록을 보여주기
- GET Localhost:4000/users/id/:id
  - 특정 회원 정보 보여주기
- POST Localhost:4000/users/add?id=test&name=test
  - 회원 추가하기

회원 목록 보여주기 API

- GET Localhost:4000/users
- 위의 요청이 들어오면 회원 정보 Obj 를 그대로 전달하여 목록을 띄우겠습니다!

```
const USER = {
  1: {
    id: 'tetz',
    name: '이효석',
  },
};

userRouter.get('/', (req, res) => {
  res.send(USER);
});
```

특정 회원 정보 보여주기 API

- GET Localhost:4000/users/id/:id
- Id 정보를 params 로 받아와서 처리를 해줍니다.
- 단, 해당 id를 가지는 회원이 없으면 예외 처리를 해줍니다!
- 그리고 id 값 입력을 안하면 그 예외 처리는 Express가 해줍니다!

Express

```
userRouter.get('/id/:id', (req, res) => {
  const userData = USER[req.params.id];
  if (userData) {
    res.send(userData);
  } else {
    res.send('ID not found');
  }
});
```

회원 추가하기 API

- POST Localhost:4000/users?id=test&name=test
- Req.query 로 회원 가입할 정보를 받아와서 새로운 회원을 만들어 줍니다!
- 물론, id 또는 name 이 정상적으로 안들어온 경우는 예외 처리를 해야합니다!

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

    USER[Object.keys(USER).length + 1] = newUser;

    res.send('회원 등록 완료');
  } else {
    res.end('Unexpected query');
  }
});
```

실습, 회원 수정 – 삭제 API 만들기!

- 회원 수정, 삭제 API를 만들어 주세요 ☺
- 회원 수정 테스트 URL
  - PUT Localhost:4000/users/modify/1?id=test&name=test
- 회원 삭제 테스트 URL
  - DELETE Localhost:4000/users/delete/1
- 슬랙 과제 제출에 댓글로 제출해 주세요!

HTML

HTML

- 정적 언어(서버 통신 X)
- JS 없이 기능 추가 불가능

- 정적 언어(서버 통신 X)

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',
  },
```

그렇다다면...

- CSS 설정은요?
- CSS 파일 불러오기는 되나요?
- 내부에서 사용하는 JS 코드는 어찌 불러오죠?
- 저거 저래 가지고 쓰겠어요?


우리가 국력(?)이요!!!

View

Engine

EJS

<% Embedded JavaScript %>

Templating: Node, Express, EJS

pug

NUNJUCKS

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!

동적 웹 시작하기!

- 이제 서버에서 데이터를 받아서, 해당 데이터를 그리는 방법을 배워 봅시다!
- 먼저 ejs 로 데이터를 넘기는 방식은
- Res.render('ejs 파일명', { 전달하고 싶은 데이터 });
- 전달하고 싶은 데이터는 오브젝트 형태로 전달이 됩니다!
- 오브젝트로 전달 된 데이터는 ejs 파일 내부에서 <%= %> 문법을 사용하여 오브젝트의 필드명으로 바로 호출이 가능합니다!

- 서버 코드

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


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

- 뷰 코드

```
<body>
    <h1>회원 목록</h1>
    <h2>
        총 회원 수 <%= userCounts %>
    </h2>
    <ul>
        <li>
            <p>ID: <%= USER[0].id%>
            </p>
            <p>이름: <%= USER[0].name%>
            </p>
        </li>
    </ul>
</body>
```

EJS

문법 배우기!

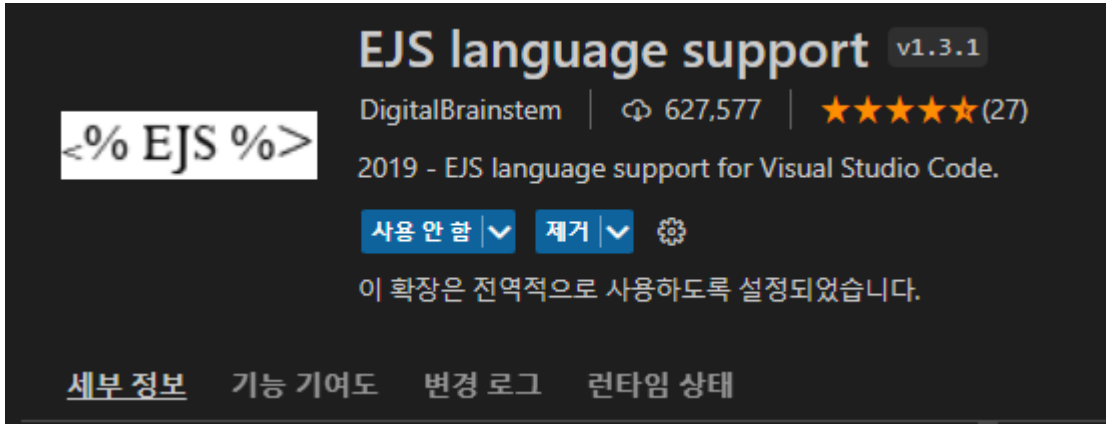
If / for 문 사용하기

- 데이터를 받을 때에는 <%= %> 문법을 사용 했습니다.
- JS의 문법을 HTML 코드에 적용하고 싶다면?
- <% %> 내부에 코드를 사용하면 됩니다!
- 그럼 if 와 for 문을 조합해서 회원 목록을 전부 띄우는 페이지를 만들어 봅시다!

```
<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>
```

아... 그런데 좀 보기가 좀 그렇죠?

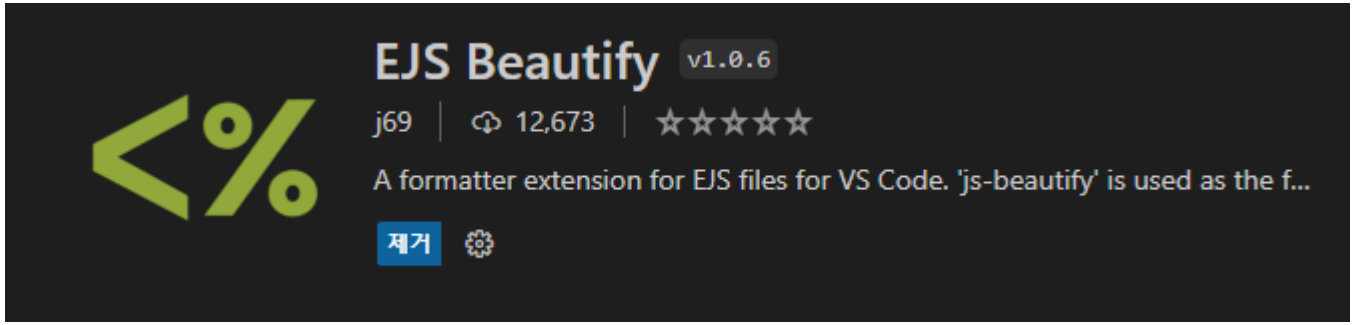
- EJS 형식의 경우 보기가 좀 그렇네요!?
- 그럼 좀 이쁘게 봅시다!

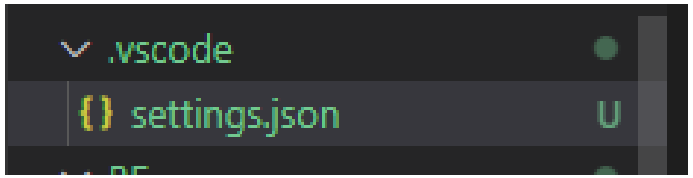



```
<% if(userCounts> 0) { %>
    <% for(let i=0; i < userCounts; i++) { %>
        <li>
            <p>ID: <%= USER[i].id %>
            </p>
            <p>이름: <%= USER[i].name %>
            </p>
        </li>
        <% } %>
            <% } else { %>
                <li>
                    회원 정보가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

아... 그런데 좀 보기가 좀 그렇죠?

- Prettier 같은 기능도 추가를 해보겠습니다!






```
{
  "[html]": {
    "editor.defaultFormatter": "j69.ejs-beautify"
  },
  "emmet.includeLanguages": {
    "ejs": "html"
  }
}
```

```
<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>
```

실습, 데이터에 email 필드를 추가!

- 지금은 데이터에 id, name 만 있습니다! 여기에 email 를 추가해 주세요!
- 해당 email 을 처리하기 위한 Back-end 와 Front-end 코드를 구성해 주세요! ☺

CSS 적용하기

Index.ejs 파일에 CSS 를 적용해 봅시다!

- views 폴더에 css 폴더를 만들고 style.css 파일을 만들어 주세요.
- CSS로 Index.ejs 의 body 태그의 색을 원하는 색으로 변경해 주세요!
- Index.ejs 에서 css 파일을 link 해 주세요!

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

- 그리고 페이지를 새로 고치게 되면!?

CSS 적용이 안되네요!?

에러 메시지를 봅시다!

```
Refused to apply style from 'http://localhost:4000/css/style.css' because its MIME type ('text/html') is not
a supported stylesheet MIME type, and strict MIME checking is enabled.
```

- Css 파일의 위치를 localhost:4000/css/style.css 에서 찾고 있네요!?
- 생각해 보면 우린 views 폴더 아래의 css 폴더에 넣었으니까? 폴더 위치를 변경해 주어야 겠네요?

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

그래도 안되네요!?

- 사실 안되는게 맞습니다!
- 물론 간단하게 생각하면 실제로 구성한 Back-end 의 폴더 구조와 URL 로 접근하는 구조가 동일하는게 맞습니다!
- 그런데, 이렇게 URL과 Back-end 의 폴더 구조가 무조건 동일해야 한다면 어떤 문제가 생길까요?
- 나쁜 사람들이 나쁜 마음을 먹으면 몇몇 페이지 접속 주소, 또는 API 주소만 보고도 Back-end 서버의 구조를 파악 할 수 있겠죠?
- 만약 만든 서비스가 중요한 회원 정보를 담고 있다면!? 그렇다면 생각보다 손 쉽게 해킹이 가능해 집니다!

그래도 안되네요!?

- 따라서, express 같은 프레임 워크에서는 url 주소를 바탕으로 Back-end 폴더의 구조를 알 수 없도록 static 이라는 미들웨어를 제공합니다!

Static

사용하기!

Static

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

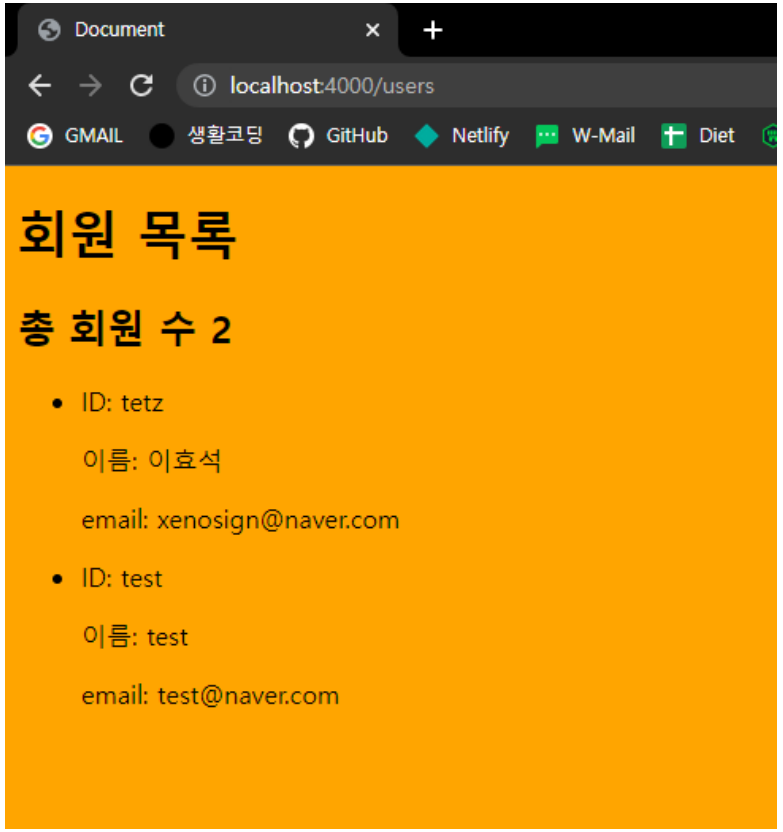
```
app.use(express.static('views'));
```

- 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">
```



Static

- 꼭, 하나의 폴더만 등록할 필요는 없습니다!

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

- 여러 폴더를 설정하면 해당 폴더를 각각 찾아 다니면서 파일을 찾습니다
- 물론 파일이 없으면 에러가 발생!

절대 경로 사용하기

- 지금 현재 파일이 위치하는 절대 경로는 __dirname 으로 구할 수 있습니다!

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

```
D:\git\kdt-5th\BE
```

- 따라서 views 폴더 static 설정을 요렇게 할수도 있죠!

요청 경로별 폴더 지정

- 프론트에서 요청되는 경로에 따라 폴더를 따로 지정 할 수도 있습니다!

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

Static

- 다만 브라우저에서 접근 가능한 기본 폴더를 설정하는 만큼 너무 많은 폴더를 정하는 것은 피하는 편이 좋습니다.
- 보통 public 이라는 폴더를 기본으로 설정해 놓고 해당 폴더만 설정하는 경우가 많습니다!
- 그리고 public 폴더의 경우 브라우저 등의 클라이언트가 접근이 가능한 폴더로써 보통 CSS, 브라우저에서 사용하는 JS, 이미지 등을 위치 시킵니다!

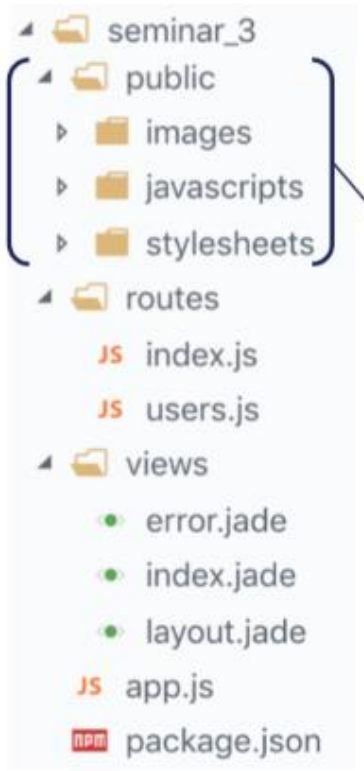
Express

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

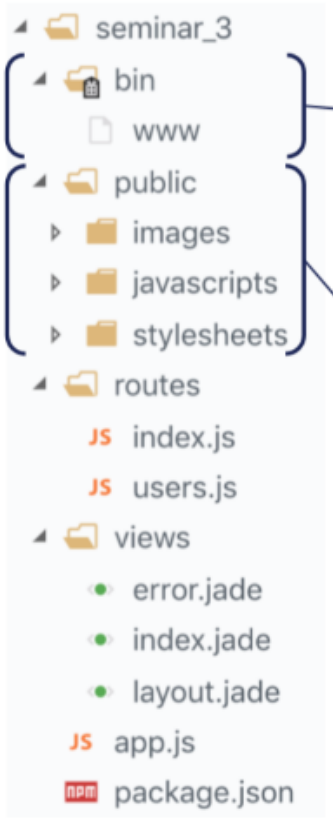
앱 서버에서는 딱히 건들일 없음

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

Express 기본 폴더 구조

- Express 로 만들어진 서비스의 기본 폴더 구조를 배워 봅시다!

프로젝트 구조



/bin/www

서버를 실행하는 스크립트

프로젝트 돌아가는 포트번호 바꿀 수 있음

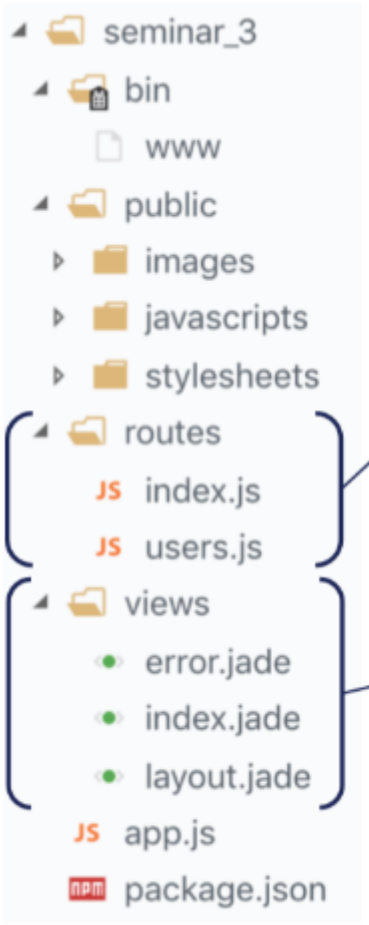
/public

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



/routes

페이지 라우팅과 관련된 파일 저장 (주소별 라우터들을 모아둠)

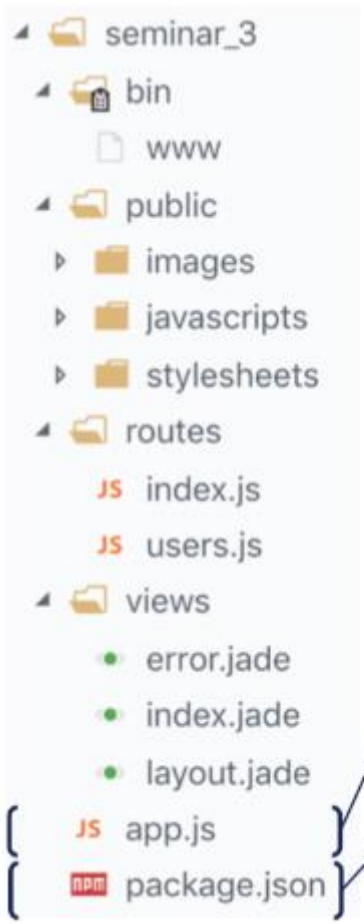
url 별로 실행되는 실제 서버 로직

index.js 파일로 라우팅 관리

/views

jade, ejs 파일 등 템플릿 파일 모아둠

웹 서버 사용시 이 폴더의 파일들을 사용해 렌더링



/app.js

핵심적인 서버 역할 (프로젝트의 중심)

미들웨어 관리가 이루어짐

라우팅의 시작점

/package.json

npm의 의존성 파일

현재 프로젝트에 사용된 모듈을 설치하는데 필요한 내용을 담음

--save 옵션을 통해 사용한 모듈 정보 저장

npm install 시 해당 파일에 있는 모듈 전부 설치

Express

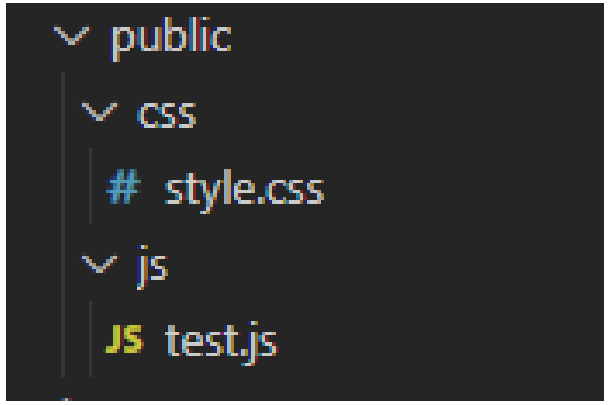
기본 폴더 구조로 변경

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>
```

실습, public/images 적용

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

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

Routing

처리

기능을 파일로 분리하기

- 지금은 모든 기능이 app.js 파일에 몰려 있습니다
- 각각의 기능을 파일로 나눈 뒤, 모듈 타입으로 라우팅 처리해 봅시다!
- 우리는 users 기능만 있으므로 routes/users.js 파일에서 user 에 관련된 기능을 전부 처리해 보겠습니다!

Routes/users.js

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

- Users 의 기능은 router 라는 객체에 담아 모듈로 exports 하고 해당 모듈을 app.js 에서 require 를 사용하여 가져와서 사용할 예정 → router 라는 이름으로 통일해서 담기!
- App.js 에 있던 users 관련 기능을 하나 하나 옮겨 봅시다!
- 데이터를 담당하는 변수 옮기기 + 기존에 만든 router 미들 웨어들 옮기기!
- 단, 라우터의 이름을 router 로 변경!

모듈 빼기, 가져오기

```
module.exports = router;
```

- 하나의 모듈로서 불러올 예정이므로 module.exports 사용해서 모듈로 빼주기!
- App.js 에서는 users.js 파일을 require 해주고 해당 라우터가 사용할 주소 사용 해주기!

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

Index 는 메인 페이지에서 사용!

- Index 라는 이름은 메인 페이지에서 사용할 것이므로, index.ejs 파일을 users.ejs 파일로 변경!
- Uesrs.js 의 코드도 index 가 아닌 users.ejs 파일을 띄우도록 변경

```
router.get('/', (req, res) => {
  const userLen = USER.length;
  res.render('users', {
    USER,
    userCounts: userLen,
    imgSrc: './images/done.png',
  });
});
```

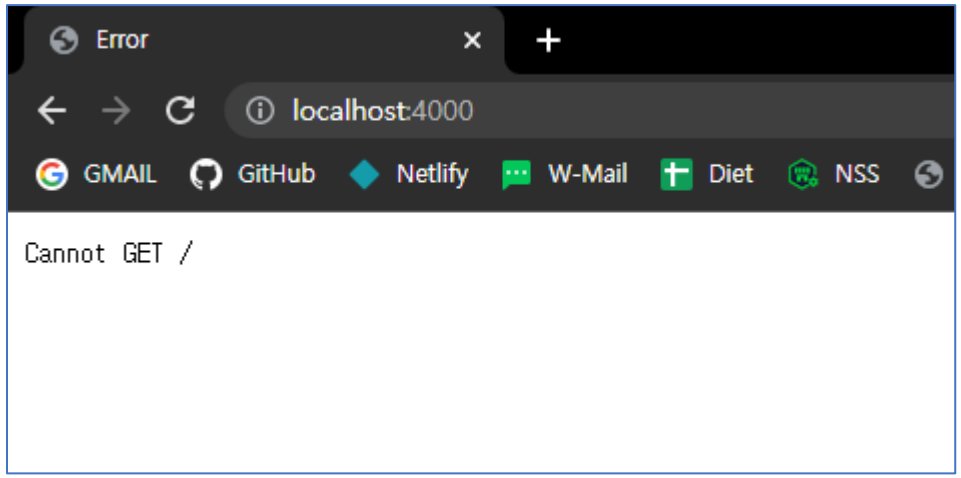
App.js 처리!

- 이제 기능을 분리
- Npm start 를 입력하면 실행이 될 수 있도록, package.json 수정!

```
"scripts": {
    "server": "nodemon app.js",
    "start": "nodemon --watch \"./routes/*.js\" --exec \"npm run server\""
  },
```

실습, index 페이지를 만들기

- 아무런 주소 없이 localhost:4000 으로 접속 하면



- 아무래도 인덱스 페이지가 필요 하겠죠?

실습, index 페이지를 만들기

- localhost:4000 으로 접속 하면 index.ejs 페이지가 뜨도록 설정하기
- Index 페이지는 Welcome to Express Dynamic Web 이란 H1 태그를 가지고 있기
- 기존 localhost:4000/users/ejs 로 가야 보였던 페이지를 이제는 localhost:4000/users 로 가면 보이도록 설정
  - 회원 리스트를 보여주는 페이지의 파일명도 users.ejs 로 변경
  - localhost:4000/users 주소를 입력하면 회원 목록을 보여줬는데 해당 기능은 localhost:4000/users/list 에 할당

실습, index 페이지를 만들기

- Index.ejs 파일에는 users 페이로 이동할 수 있는 <a>태그 넣어서 해당 태그를 클릭하면 users 페이지로 이동하도록 만들기
- Users 페이지 상단에는 index.ejs 로 갈 수 있는 <a> 넣기

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

localhost:4000/users/list

localhost:4000/users/list

GMAIL GitHub Netlify W-Mail Diet NSS

```
// 20221120082713
// http://localhost:4000/users/list

[
  {
    "id": "tetz",
    "name": "이효석",
    "email": "tetz@spreatics.com"
  },
  {
    "id": "kimi",
    "name": "이기원",
    "email": "tetz@spreatics.com"
  }
]
```

Error 핸들링!

Error 핸들링

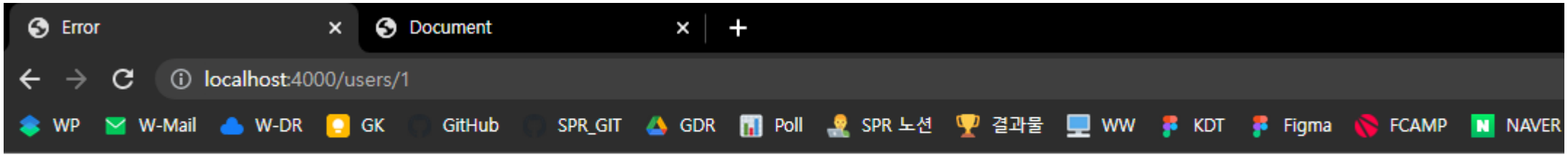
- Users 라우터에서 문제가 발생하면 res.end 로 err 메시지를 보내주고는 있지만 이 방법은 편의를 위한 방법입니다!
- 서버 Err 가 발생하면 정석적으로는 err 를 발생 시킨 다음 err 메시지와 status 코드를 전달해 줘야 합니다!
- 그럼, users 라우터의 각각의 상황에서 Err 처리를 해줍시다!

Error 던지기!

```
router.get('/:id', (req, res) => {
  const userData = USER.find((user) => user.id === req.params.id);
  if (userData) {
    res.send(userData);
  } else {
    const err = new Error('ID not found');
    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: ID not found
    at C:₩Users₩tetz₩Desktop₩KDT₩__수업 자료₩정규 수업₩29₩backend₩routes₩users.js:34:17
    at Layer.handle [as handle_request] (C:₩Users₩tetz₩Desktop₩KDT₩__수업 자료₩정규 수업₩29₩backend₩node_modules₩express₩lib₩router₩layer.js:95:5)
    at next (C:₩Users₩tetz₩Desktop₩KDT₩__수업 자료₩정규 수업₩29₩backend₩node_modules₩express₩lib₩router₩route.js:144:13)
    at Route.dispatch (C:₩Users₩tetz₩Desktop₩KDT₩__수업 자료₩정규 수업₩29₩backend₩node_modules₩express₩lib₩router₩route.js:114:3)
    at Layer.handle [as handle_request] (C:₩Users₩tetz₩Desktop₩KDT₩__수업 자료₩정규 수업₩29₩backend₩node_modules₩express₩lib₩router₩layer.js:95:5)
    at C:₩Users₩tetz₩Desktop₩KDT₩__수업 자료₩정규 수업₩29₩backend₩node_modules₩express₩lib₩router₩index.js:284:15
    at param (C:₩Users₩tetz₩Desktop₩KDT₩__수업 자료₩정규 수업₩29₩backend₩node_modules₩express₩lib₩router₩index.js:365:14)
    at param (C:₩Users₩tetz₩Desktop₩KDT₩__수업 자료₩정규 수업₩29₩backend₩node_modules₩express₩lib₩router₩index.js:376:14)
    at Function.process_params (C:₩Users₩tetz₩Desktop₩KDT₩__수업 자료₩정규 수업₩29₩backend₩node_modules₩express₩lib₩router₩index.js:421:3)
    at next (C:₩Users₩tetz₩Desktop₩KDT₩__수업 자료₩정규 수업₩29₩backend₩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!

App.js 에서 에러 핸들링!

- 서버 사이드에서는 어떤 에러인지 알아야 하므로 console.log(err.stack) 을 내보내서 브라우저에서 보던 Error 내역을 확인 합니다!

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" 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>
```

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

- 그래도 체험은 하셔야 하니까 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('/', (req, res) => {
  console.log(req.body);
```

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 보여 주도록 res.send 대신 res.redirect 사용!

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

```
router.post('/', (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('Unexpected query');
      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.query.email,
      };
      USER.push(newUser);
      res.redirect('/users');
    } else {
      const err = new Error('Unexpected query');
      err.statusCode = 404;
      throw err;
    }
  } else {
    const err = new Error('No data');
    err.statusCode = 404;
    throw err;
  }
});
```

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 전체 조회 : GET localhost:4000/posts/list
  - 글 추가 : POST localhost:4000/posts
- 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()

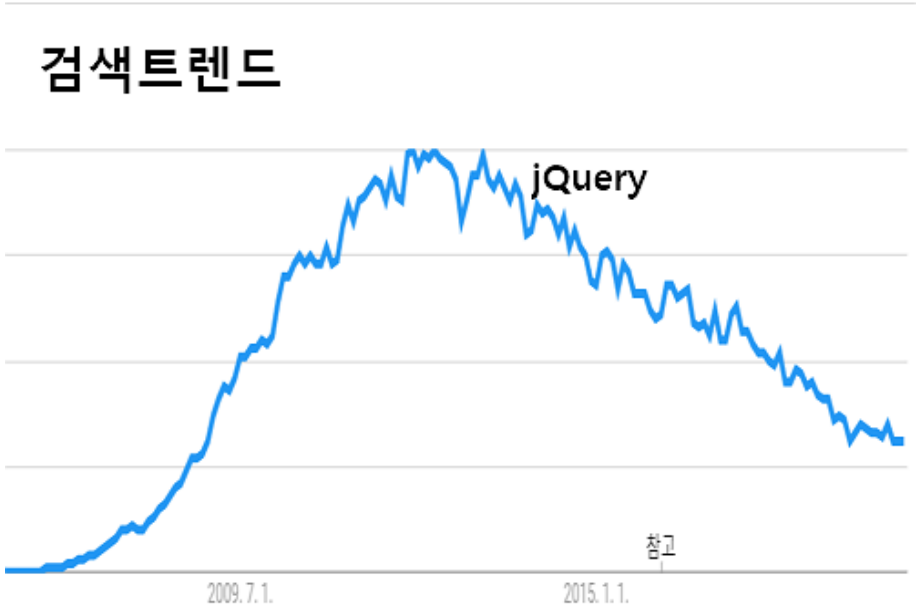
XMLHttpRequest

- 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/${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();
    })
```

```
<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>
```

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 목록 보여주기
});

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

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

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

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

router.delete('/title/: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 articleLen = ARTICLE.length;
  res.render('board', { ARTICLE, articleCounts: articleLen });
});
```

```
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, 메소드는 POST 로 전달

```
<form action="/board" 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 로 들어왔으므로 / 처리
- 메소드는 post 이므로 router.post('/', (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/title/:title 로 전달 / 메소드는 POST

```
<form action="/board/title/<%=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/title/:title 로 들어왔으므로 /title/:title 처리
- 여기서 만약 파라미터 설정이 /:title 이라고 한다면?
  - Localhost:4000/board/write 이라는 요청이 들어오면?
  - 해당 write 라는 값이 글 쓰기 모드 이동을 위한 주소 호출인지, 글 수정을 하려는 글의 제목이 write 인지 구별이 불가능
  - 결과적으로 더 먼저 선언되어 있는 미들웨어가 처리하게 되어 문제 발생!
- 메소드는 post 이므로 router.post('/title/:title', (req, res)= > {}) 로 처리
- 프론트에서 예외 처리를 하였지만, 데이터가 누락되는 경우를 대비

```
router.post('/title/: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!