

Hello,

KDT 웹 개발자 양성 프로젝트

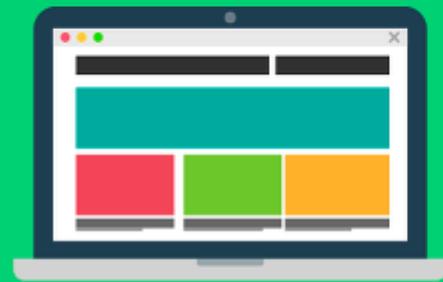
5기!

with





# Back-End



**FRONTEND**



**BACKEND**

**클라이언트**

**서버**

요청

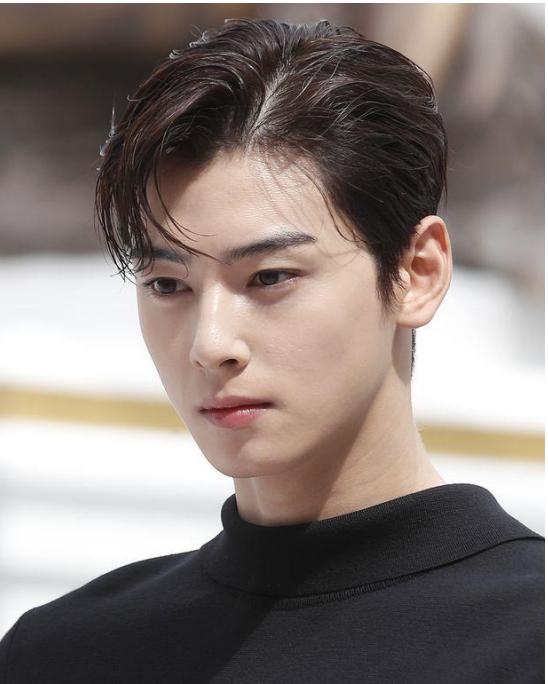
응답



# 남성 연예인

# 남성 연예인

---

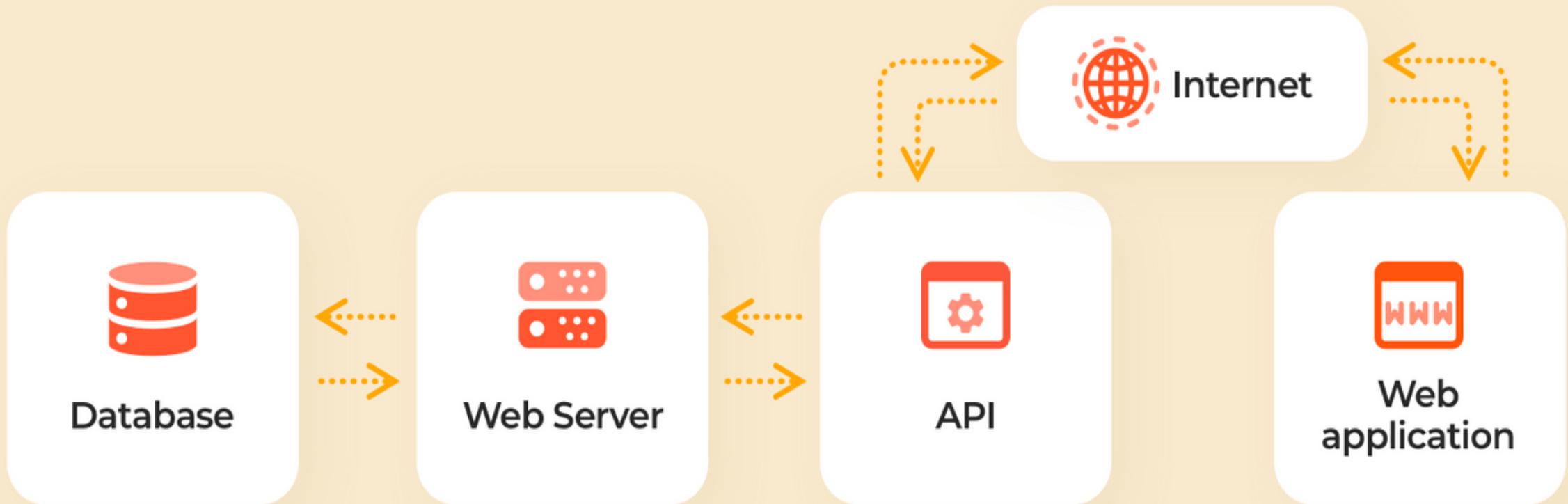




# 남성 연예인 목록

## API

# What is an API?





# API 주소

<http://api.tetzkdt.com/male-stars>

[https://apis.data.go.kr/B551011/KorService/areaBasedList?serviceKey=rfaoGpiapHFq0cUT6bqfERRxy1WVxzOdOpEC3ChyAFPEfONDdRVNETTJKRhqTbPuZ2krpG2mQJMXDbyG74RA%3D%3D&numOfRows=687&pageNo=1&MobileOS=ETC&MobileApp=TripLog&\\_type=json&listYN=Y&arrange=B&contentTypeId=12&areaCode=1](https://apis.data.go.kr/B551011/KorService/areaBasedList?serviceKey=rfaoGpiapHFq0cUT6bqfERRxy1WVxzOdOpEC3ChyAFPEfONDdRVNETTJKRhqTbPuZ2krpG2mQJMXDbyG74RA%3D%3D&numOfRows=687&pageNo=1&MobileOS=ETC&MobileApp=TripLog&_type=json&listYN=Y&arrange=B&contentTypeId=12&areaCode=1)



# http://api.tetzkdt.com/male-stars

```
[  
  {  
    name: "차은우",  
    image: "http://file.tetzkdt.com/malestars/차은우.png"  
  },  
  {  
    name: "고수",  
    image: "http://file.tetzkdt.com/malestars/고수.png"  
  },  
  {  
    name: "이동욱",  
    image: "http://file.tetzkdt.com/malestars/이동욱.png"  
  }  
]
```



# { REST : API }

Representational State Transfer API



<https://apis.data.go.kr/B551011/KorService/areaBasedList?serviceKey=rfaoGpiapHFqOcUT6bqfERRxy1WVxzOdOpEC3ChyAFPEfONDMDRVNETTJKRhqTbPuZ2krpG2mQJMXDbyG74RA%3D%3D&numOfRows=687&pageNo=1&MobileOS=ETC&MobileApp=TripLog& type=json&listYN=Y&arrange=B&contentTypeId=12&areaCode=1>

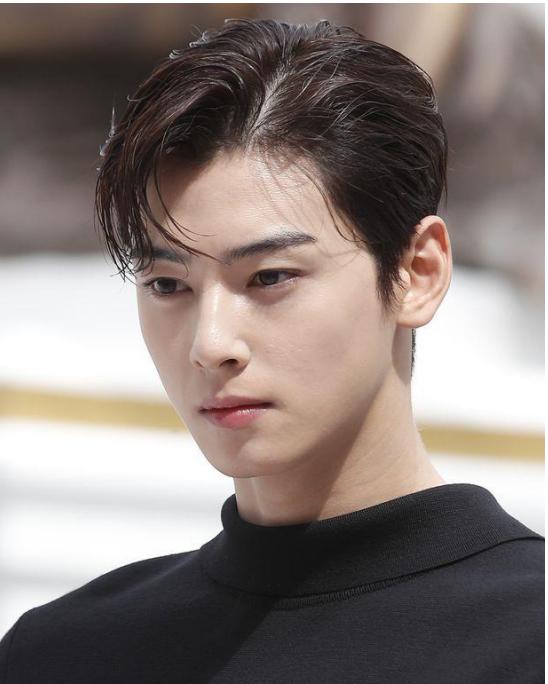
**{JSON}**  
JavaScript Object Notation



# Front-End

# 남성 연예인

---



# 여성 연예인

---





자 이제 시작이야 Back-End



# Node-JS



# 2008









# Node.js 설치

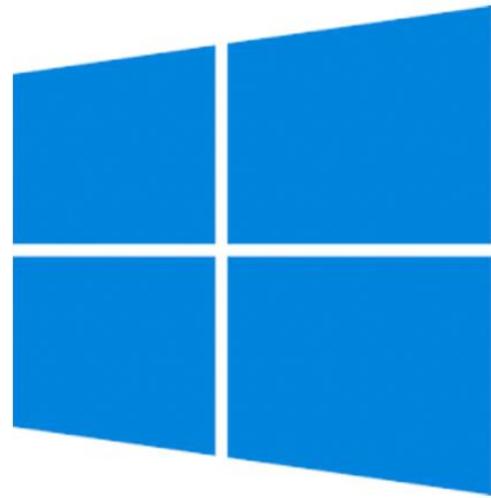


설치

<https://nodejs.org/ko/>



# 버전 관리 Tool 설치



<https://github.com/coreybutler/nvm-windows/releases>



# 버전 관리 Tool 설치



tj/n

#641 "no version found"  
error on Mac with M1  
chip

18 comments



shadowspawn opened on November 27, 2020



<https://github.com/tj/n>



# NPM?

(Node Package  
Manager)



# 1,614,651 개의 패키지!

## └ By the numbers

---

Packages

**1,614,651**

---

Downloads · Last Week

**30,687,522,522**

---

Downloads · Last Month

**126,688,929,038**





# 패키지 관리 시작하기!



- npm init -y

```
tetz@DESKTOP-P7Q40LL MINGW64 ~/Desktop/node-setting
$ npm init -y
npm WARN config global `--global`, `--local` are deprecated. Use `--location=global` instead.
Wrote to C:\Users\tetz\Desktop\node-setting\package.json:

{
  "name": "setting",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

# Package.json 파일 알아보기!



```
{  
  "name": "setting",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC"  
}
```

# Package 설치하기!



- npm install left-pad

```
lhs@DESKTOP-86MUCGC MINGW64 /d/git/kdt-5th-proj1 (main)
$ npm install left-pad
npm WARN deprecated left-pad@1.3.0: use String.prototype.padStart()

added 1 package, and audited 174 packages in 647ms

12 packages are looking for funding
  run `npm fund` for details

4 high severity vulnerabilities

To address all issues (including breaking changes), run:
  npm audit fix --force

Run `npm audit` for details.
```

```
{
  "dependencies": {
    "left-pad": "^1.3.0"
  }
}
```

# Package 삭제하기



- npm uninstall left-pad

```
lhs@DESKTOP-86MUOGC MINGW64 /d/git/kdt-5th-proj1 (main)
$ npm uninstall left-pad

removed 1 package, and audited 173 packages in 678ms

12 packages are looking for funding
  run `npm fund` for details

4 high severity vulnerabilities

To address all issues (including breaking changes), run:
  npm audit fix --force

Run `npm audit` for details.
```

```
{
  "dependencies": {}
}
```

# Package.json 만 활용하기!



- 빈 폴더에 package.json 파일만 카피를 해봅시다
- [Npm install](#)
- 실행을 하면 package.json 를 참고하여 해당 package 가 설치 됩니다!
- 따라서, 추후에 협업 시 node\_modules 폴더를 보낼 필요 없이 package.json 만 공유가 되면 편리하게 관리가 가능합니다~!



# Formatting,

# Linting,

# TypeScript 설정!



# Formatting?

The screenshot shows a dark-themed code editor window with a tab bar at the top labeled "README.md — prettier-config-example". The main area contains the following Markdown content:

```
1 # Prettier support for other languages
2
3 This folder has JavaScript, CSS, HTML, JSON and even this Markdown file all formatted using
4 Prettier
5
6 Note that while formatting [index.js](index.js) Prettier uses 2 spaces per tab, but in the
7 Markdown code blocks it uses 4 spaces.
8
9 ``js
10 const code = true; if (code) {console.log('code is on')}
11 ...
12
13 This is because we override options inside the [.prettierrc.json](.prettierrc.json) file.
```

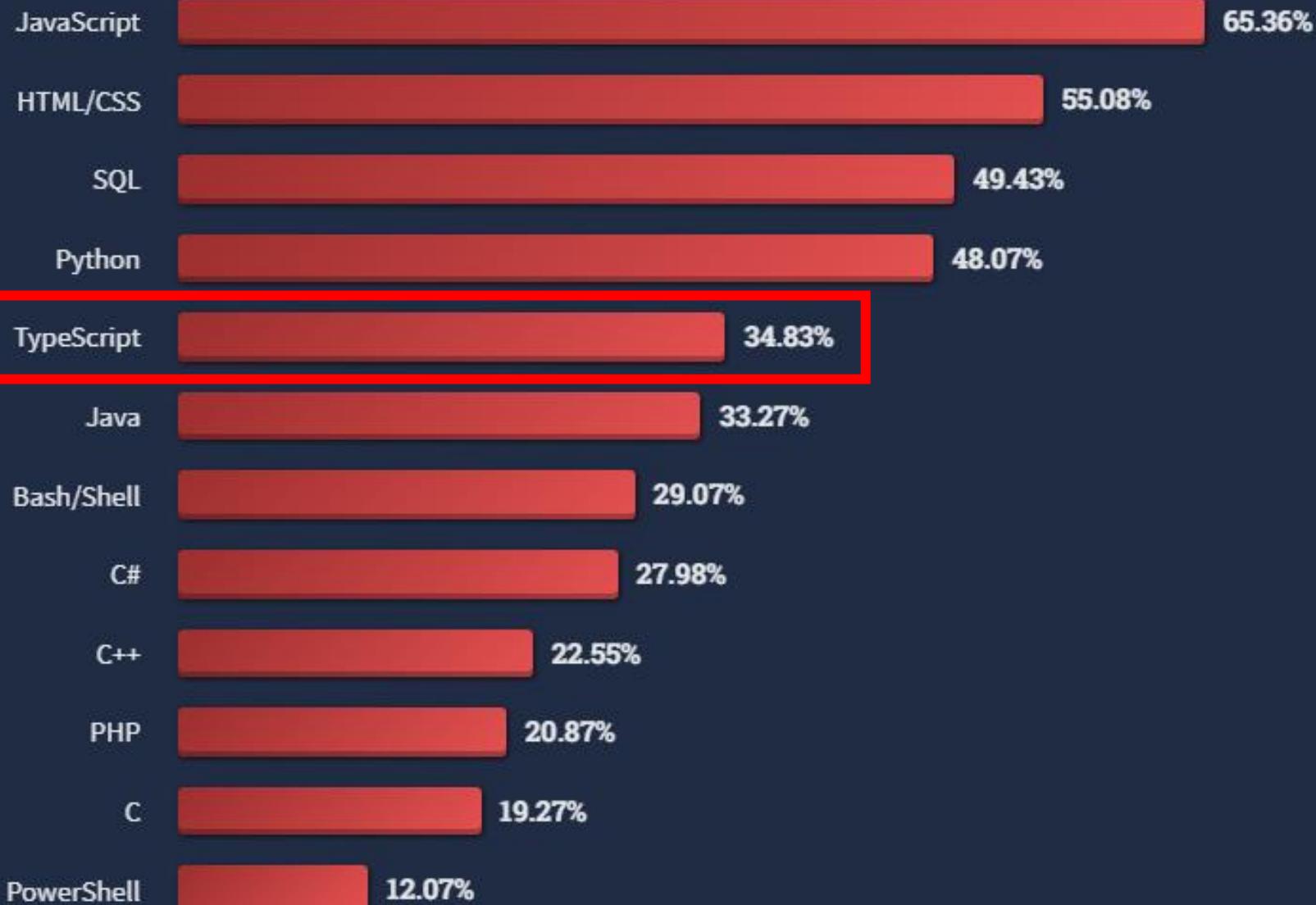
The code editor interface includes standard window controls (red, yellow, green buttons), a title bar, and a toolbar with icons for copy, paste, and other functions.

# Linting?





## Programming, scripting, and markup languages



# TypeScript 적용



- Main.js 파일에 // @ts-check 주석 추가

```
// @ts-check
```

A screenshot of a code editor showing a TypeScript file named main.js. The code contains a comment // @ts-check. A tooltip is displayed over the line const str = 'Hello';, indicating a type mismatch error: "'string' 형식의 인수는 'number' 형식의 매개 변수에 할당될 수 없습니다. ts(2345)". The tooltip also includes a link to "문제 보기 빠른 수정... (Ctrl+.)". The code editor interface shows the file path JS main.js > ... and line numbers 1 through 6.

```
JS main.js > ...
1 // @ts-check
2
3 const str = 'Hello';
4 const num = Math.log(str);
5 console.log(num);
6
```

const str: "Hello"  
'string' 형식의 인수는 'number' 형식의 매개 변수에 할당될 수 없습니다. ts(2345)  
문제 보기 빠른 수정... (Ctrl+.)

- 이제 Type 관련 문제는 TypeScript 가 알려 줍니다!



# TypeScript & Node



# Node & Typescript!

```
in.js > [o] server > http.createServer().callback
// @ts any
const http = require('http');
const res = http.d.ts(487, 9) // 여기서는 'statusCode'이 (가) 선언됩니다.
const PORT = 4000;
const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.end('Hello');
});

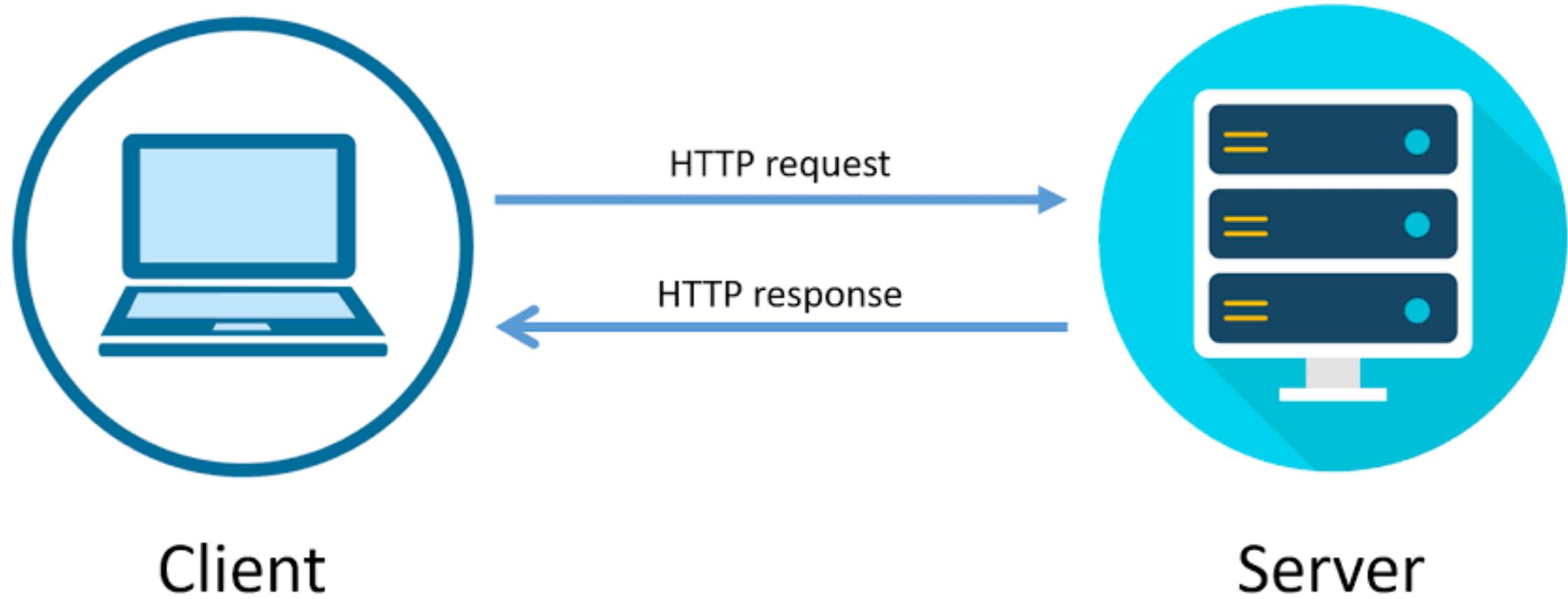
server.listen(PORT, () => {
  console.log(`The server is listening at port: ${PORT}`);
});
```



자 이제 시작이야 Back-End



간단한  
통신 경험하기!



# Express 서버 구축



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

backend/server.js

## frontend/index.html



```
<!DOCTYPE html>
<html lang="en">

<body>
  <h1 class="header">Hello, Protocol</h1>
  <button onclick="fetchData()">백엔드 통신 경험하기</button>
</body>

<script>
  const headerEl = document.querySelector('.header');

  const fetchData = () => {
    fetch('http://localhost:4000')
      .then((res) => {
        return res.json();
      })
      .then((data) => {
        headerEl.innerHTML = data;
      })
  }
</script>
</html>
```



# Hello, Protocol

[백엔드 통신 경험하기](#)

안녕하세요. 여기는 백엔드입니다!

[백엔드 통신 경험하기](#)



# JSON

# JSON? → JavaScript Object Notation



```
1  {
2      "string": "Hi",
3      "number": 2.5,
4      "boolean": true,
5      "null": null,
6      "object": { "name": "Kyle", "age": 24 },
7      "array": ["Hello", 5, false, null, { "key": "value", "number": 6 }],
8      "arrayOfObjects": [
9          { "name": "Jerry", "age": 28 },
10         { "name": "Sally", "age": 26 }
11     ]
12 }
13 }
```





SBS

그런데  
말입니다





저건 어찌 저렇게  
작동하는지  
잘 모르겠는데?

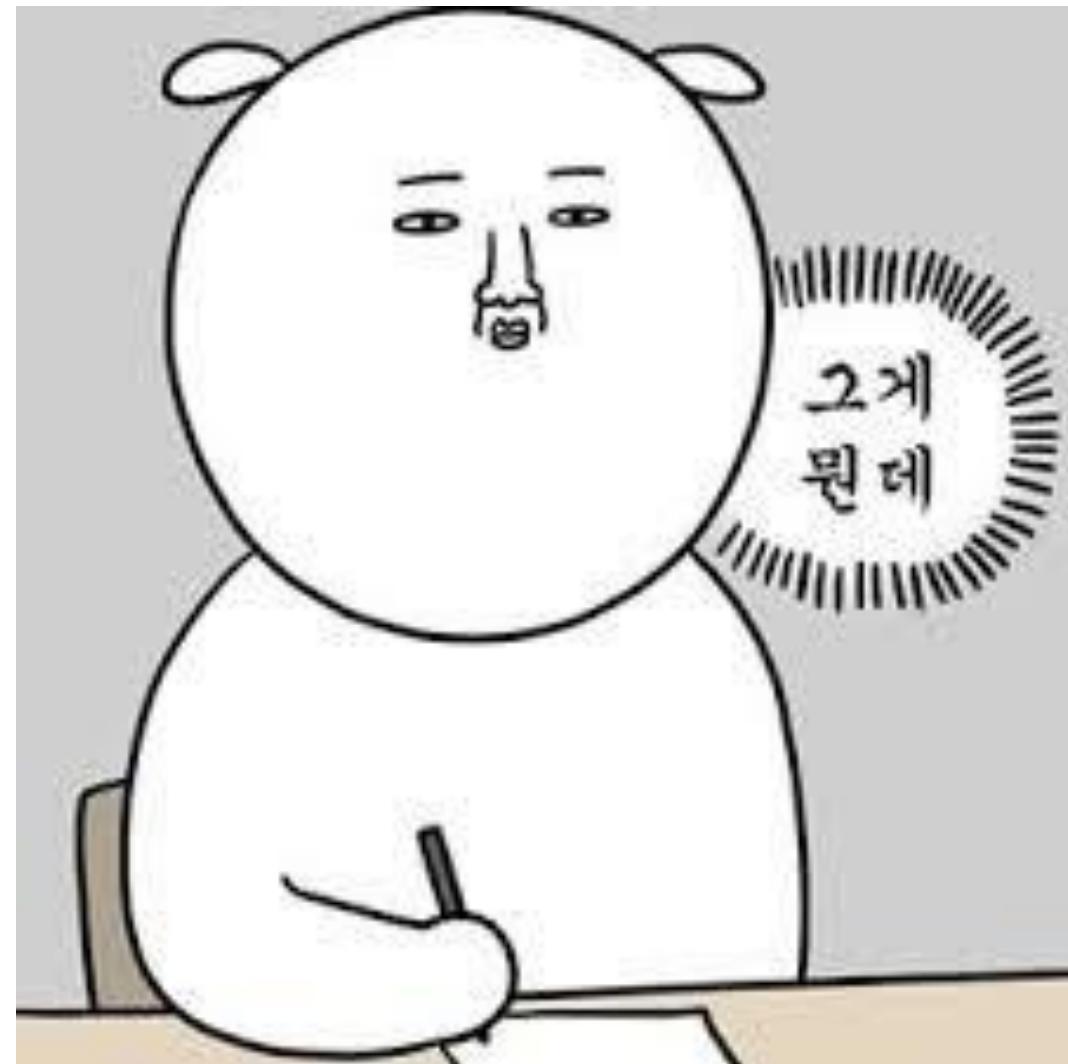




**지금 시작합니다**



# Callback!?



# Callback!!



- 말 그대로 함수를 부르는 것을 뜻합니다!
- 일단 함수를 보시죠~!



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



```
1hs@DESKTOP-86MUCGC MINGW64 /d/git/kdt-5th (main)  
$ node cb.js  
1번 함수  
2번 함수  
3번 함수
```

# Callback!!



- 위와 같은 함수는 실행에 시간이 크게 걸릴 것이 없으므로 안정적으로 1, 2, 3번 함수 순서대로 출력이 될 것입니다~!
- 그런데 좀 더 안정적으로 순차적으로 실행을 시키고 싶다면~!?



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

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

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

func1();
```

```
1hs@DESKTOP-86MUCGC MINGW64 /d/git/kdt-5th (main)
$ node cb.js
1번 함수
2번 함수
3번 함수
```

# Callback!!



- 이렇게 함수 내부에서 함수를 부를 수가 있습니다!
- 그러면 좀 더 안정적으로 순차적 실행을 보장 받을 수 있을 것입니다!



# Callback!!



- 그런데 인간의 욕심은 끝이 없어서 이런 걸 원하게 됩니다!
- 꼭 정의된 함수만 불러와야 함!? 그때 바로 정의해서 쓰면 안되나?
- 그때 그때 다른 함수를 불러오고 싶어지네!?



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

반드시 func2() 만 호출이  
되는 구조를 가짐

그때 그때 다른 함수를 원하는  
대로 불러서 쓸 수가 없습니다!



**CALLBACK!!**

# Callback!!



- 매개 변수로 값만 전달 하는게 아니라, 함수를 전달을 해보자는 아이디어가 나왔습니다!
- 기존 처럼 함수 내부에서 다른 함수를 호출하는 것 → 가능
- 그때 그때 상황에 맞게 함수를 변경해서 호출하는 것 → 가능
- 호출을 하는 수준을 넘어 그 자리에서 바로 정의해서 사용 → 가능
- 즉, 다양한 것을 시도 할 수 있게 됩니다!



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

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

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

```
func1(func2);
```

```
lhs@DESKTOP-86MUCG0:  
$ node cb.js  
1번 함수  
2번 함수
```

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!

# 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
```

# 그렇다면!? (1)



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

```
lhs@DESKTOP-86MUCGI  
$ node cb.js  
1번 함수  
2번 함수  
3번 함수
```

# 꼭 선언된 함수만 불러야 하나?



```
function func1(callback) {  
    console.log("1번 함수");  
  
    function fakeFunc3() {  
        console.log("3번 인적 하는 함수");  
    }  
  
    callback(fakeFunc3);  
}  
  
function func2(callback) {  
    console.log("2번 함수");  
    callback();  
}  
function func3() {  
    console.log("3번 함수");  
}  
  
func1(func2);
```

```
lhs@DESKTOP-86MUOCG: ~ %  
$ node cb.js  
1번 함수  
2번 함수  
3번 인적 하는 함수
```

# 꼭 선언된 함수만 불러야 하나?



- 그런데 이건 좀 그렇겠죠?
- 일단 코드 가독성 → 구림
- 함수 내부에 함수를 정의? → 구림
- 한번만 쓰면 되는데, 굳이 함수 선언을!? → 구림



# 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 인적하는 새로 만든 익명 함수!");  
});
```

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

# 결과 값을 전달하기!



- 이번에는 콜백으로 결과 값을 한번 전달해 봅시다~!
- 콜백은 함수 이므로 함수의 인자로 결과 값을 전달 할 수 있습니다~!



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

계산한 결과 값을 인자로 전달

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

전달 받은 값을 출력!

```
multiplication(3, say);
```

```
lhs@DESKTOP-86MUO0  
$ node cb.js  
9
```



여기서 잠깐!!



```
function buySync(item, price, quantity) {
  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.`);
}

const totalMoney = buySync('포켓몬빵', 1000, 5);

pay(totalMoney);
```



```
lhs@DESKTOP-86MUCGC MINGW64 ~/Desktop/업무/KDT_4th/
$ node callback.js
포켓몬빵 상품을 5 개 골라서 점원에게 주었습니다.
undefined 원을 지불하였습니다.
계산이 필요합니다.
```



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

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

buySync("포켓몬빵", 1000, 5);
```

```
lhs@DESKTOP-86MUCGC MINGW64 /d/git/kdt-5th (main)
$ node cb.js
포켓몬빵 상품을 5 개 골라서 점원에게 주었습니다.
계산이 필요합니다.
5000 원을 지불하였습니다.
```



```
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, pay);
```

```
lhs@DESKTOP-86MUOGC MINGW64 /d/git/kdt-5th (main)  
$ node cb.js  
포켓몬빵 상품을 5 개 골라서 점원에게 주었습니다.  
계산이 필요합니다.  
5000 원을 지불하였습니다.
```





# Callback

+ 익명함수

# Callback 은 즉시 만들어서 전달 가능!



- Callback 함수를 꼭 만들어서 전달할 필요는 없으며, 즉시 만들어서 전달이 가능합니다!
- 이럴 때는 익명 함수를 사용하죠!
- 그리고 상당히 많이 사용이 됩니다!!



```
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(total) {  
//   console.log(`#${total} 원을 지불하였습니다.`);  
// }
```

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





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

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



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



```
// @ts-check

const http = require('http');

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.end('Hello');
});

const PORT = 4000;
server.listen(PORT, () => {
  console.log(`The server is listening at port: ${PORT}`);
});
```



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



```
<!DOCTYPE html>
<html lang="en">

<body>
  <h1 class="header">Hello, Protocol</h1>
  <button onclick="fetchData()">백엔드 통신 경험하기</button>
</body>

<script>
  const headerEl = document.querySelector('.header');

  const fetchData = () => {
    fetch('http://localhost:4000')
      .then((res) => {
        return res.json();
      })
      .then((data) => {
        headerEl.innerHTML = data;
      })
  }
</script>
</html>
```



# Callback

# Hell?

# 콜백 지옥?



```
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 인철하는 새로 만든 익명 함수!");  
});
```

여기서 모든 함수를  
새로 정의하는 익명함수로  
대체해 봅시다!

# 콜백 지옥?



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

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

# 콜백 지옥? 그게 뭐죠?



```
callbackhell.js
```

```
1
2     var floppy = require('floppy');
3
4     floppy.load('disk1', function (data1) {
5         floppy.prompt('Please insert disk 2', function() {
6             floppy.load('disk2', function (data2) {
7                 floppy.prompt('Please insert disk 3', f
8                     floppy.load('disk3', function (data3) {
9                         floppy.prompt('Please insert di
10                            floppy.load('disk4', functi
11                                floppy.prompt('Please i
12                                    floppy.load('disk5'
13                                    floppy.prompt('
14                                        floppy.load(
15                                            //if no
16                                            });
17                                            });
18                                            });
19                                            });
20                                            });
21                                            });
22                                            });
23                                            });
24                                            });
25                                            });
26                                            });
27                                            });
```



```
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);
```





이런 사태가  
발생합니다!

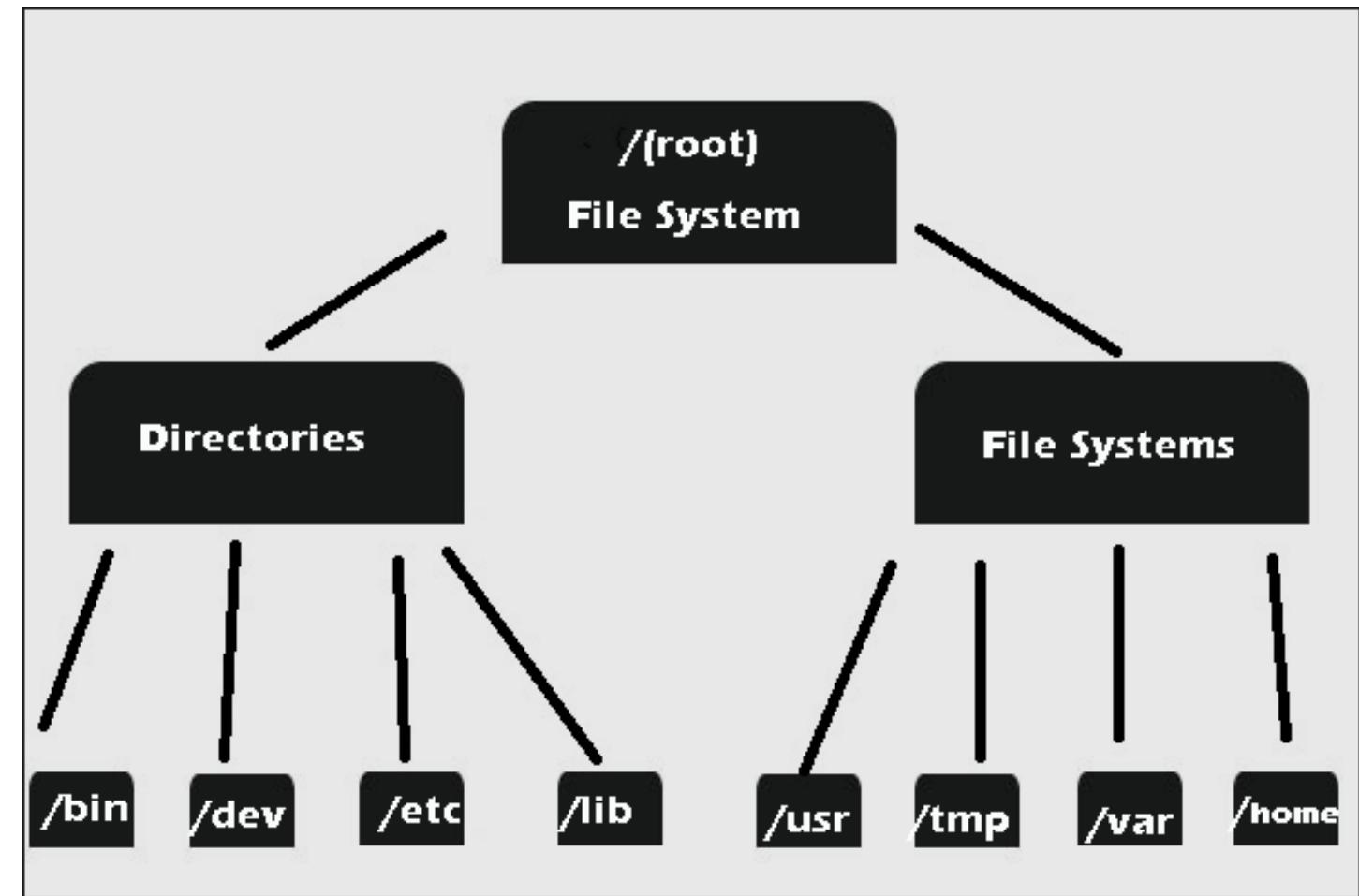


# Callback Hell

체험 해보기!



# File-System on JS





# File-system 은 Node 의 기본 모듈입니다!

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

- 파일을 읽을 때는 **readFile** 메소드를 사용합니다.
- 파일을 읽는 것도 **시간이 필요한 작업**이므로 일종의 **서버 통신과 비슷합니다**  
→ 따라서, 기본적으로 **callback** 을 사용하도록 되어있습니다.
- **fs.readFile('파일위치', '유니코드포맷', callback(err, data) {})**
- **err** 은 파일 읽기가 잘 안되었을 때, Error 코드를 반환 합니다
- **data** 는 파일 읽기가 잘 되었을 때, 읽은 **data** 를 반환합니다.



# 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



# File-system 과 비동기 프로그래밍

- 파일 시스템을 이용해서 비동기 프로그래밍 코드를 짜봅시다!
- JS의 특성으로 인해 각각 readFile 메소드를 동시에 비동기적으로 실행 시켜 봅시다!
- 동시에 readme.txt 파일을 읽어서 console.log 로 출력하는 코드를 작성해 봅시다!



## 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());
});
```

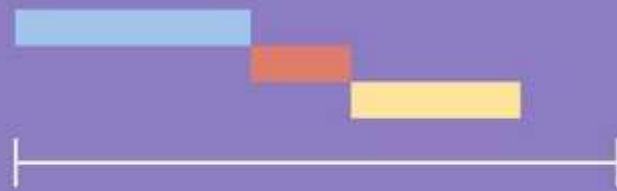


# File-system 과 비동기 프로그래밍

- 과연 결과는 어떻게 될까요!?

```
lhs@DESKTOP-86MUCGC MINGW64 ~/Desktop/업무/KDT/정규 수업/25/node_set
$ node js/file.js
1번 readme 텍스트 입니다
2번 readme 텍스트 입니다
3번 readme 텍스트 입니다
4번 readme 텍스트 입니다
```

```
lhs@DESKTOP-86MUCGC MINGW64 ~/Desktop/업무/KDT/정규 수업/25/node_set
$ node js/file.js
1번 readme 텍스트 입니다
3번 readme 텍스트 입니다
4번 readme 텍스트 입니다
2번 readme 텍스트 입니다
```



Synchronous



Asynchronous



# File-system 과 비동기 프로그래밍

- File 을 읽는 것은 JS가 각각의 Thread 에 실어서 처리하기 때문에 각각의 Thread 상황에 따라 file 읽는 속도가 다르게 됩니다.
- 따라서, 꼭 1, 2, 3, 4 로 실행 된다는 보장이 없죠!
- 그럼 1, 2, 3, 4 순서로 실행 시키려면 어찌하면 될까요?



# Callback

# Hello!?



# Callback 지옥으로 구현하기!

- 앞서 배운 것처럼 fs 는 callback 을 기본적으로 지원하므로 callback 이 호출 되면 바로 다음 fileRead 메소드를 실행 시키고, 다시 callback 에 fileRead 를 실행 시키는 콜백 지옥 형태로 코드를 구성하여 봅시다!



```
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 텍스트입니다
2번 readme 텍스트입니다
3번 readme 텍스트입니다
4번 readme 텍스트입니다

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

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



```
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);
```





# Callback 지옥으로 구현하기!

- 비록 코드는 좀 구리지만!? 의도했던 것 처럼 1, 2, 3, 4 가 순서대로 실행이 됩니다!
- 그럼, 이 callback 지옥을 어찌 탈출 할 수 있을까요!?



이상  
콜백지옥 탈출 방법에  
대해 알아 봤습니다!



# Promise!





# Promise!

- Promise 는 생성자 입니다! 따라서 new 로 사용하죠!

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

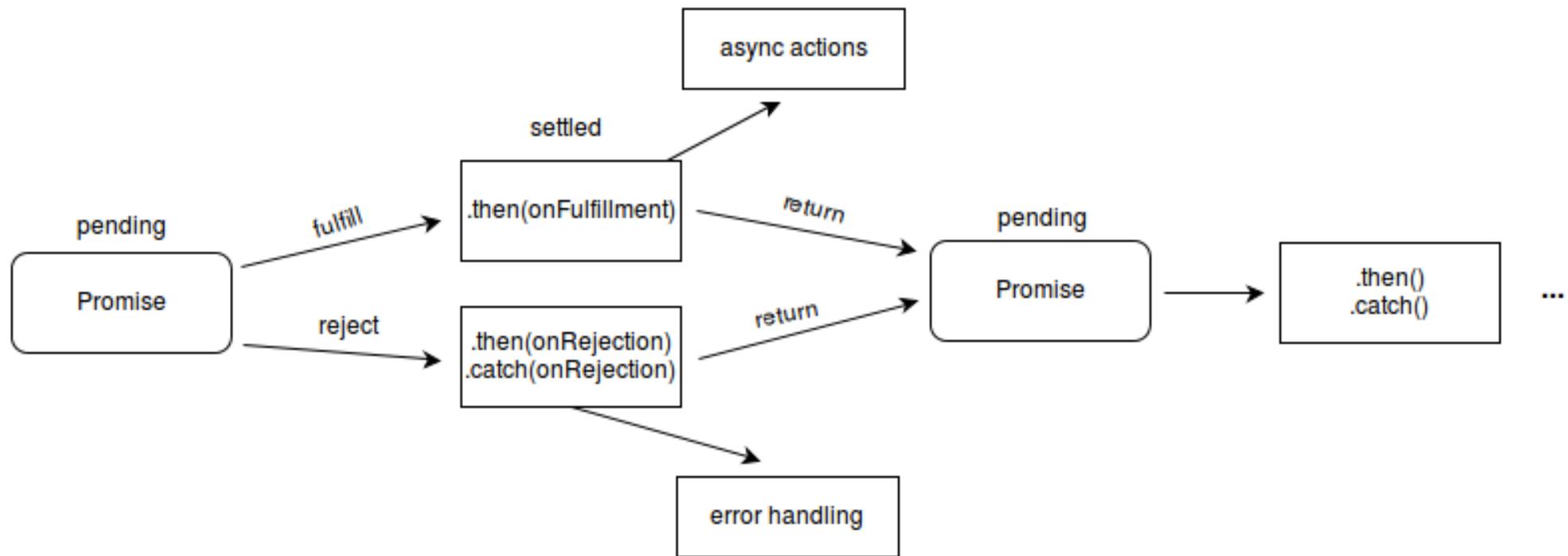
- resolve, reject 라는 2개의 콜백 함수를 받아서 사용합니다.
- promise 가 할당 되면 이 promise 는 resolve 또는 reject 함수가 callback 될 때 까지 무한 대기 합니다.
- Resolve 는 promise 가 정상적으로 이행 되었을 경우 사용하며, reject 는 반대의 경우에 사용합니다.



# Promise!

- resolve 는 추후에 then 으로 받으며, reject 는 catch 로 받습니다!
- resolve, reject 콜백 함수의 경우는 데이터를 매개변수로 보낼 수 있습니다.
- Resolve, reject 가 사용되지 않으면 promise 는 해당 콜백이 나올 때 까지 pending 상태가 되어 기다립니다!

```
1hs@DESKTOP-86MUCGC MINGW64 ~/Desktop/업무/KDT/정규 수업/25/node_set
$ node js/promise.js
Promise { <pending> }
```



출처 : MDN 공식문서

- 대기(pending): 이행하거나 거부되지 않은 초기 상태.
- 이행(fulfilled): 연산이 성공적으로 완료됨.
- 거부(rejected): 연산이 실패함.



Promise

A photograph of a man from the chest up, wearing a traditional blue robe over a white shirt. He is standing outdoors, looking towards the camera with a neutral expression. In the background, there is a large, calm body of water, a stone wall, and a building with a dark, curved tiled roof. The word "Promise" is overlaid in the upper left portion of the image in a large, bold, orange sans-serif font.



```
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 lucky = false;

const promise = new Promise((resolve, reject) => {
    console.log("주식이 오르기를 기다리기 시작합니다!");

    setTimeout(() => {
        console.log("3년의 시간이 흐르고....");
        if (lucky) {
            const profit = 3000;
            resolve(profit);
        } else {
            reject("아.... 망했어요");
        }
    }, 1000);
});

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

```

backend/promise2.js



# 콜백 지옥을 Promise 로 변경하기

- `fs.promises` 를 사용해서 콜백 지옥을 promise 코드로 변경해 봅시다
- `fs.promises` 해당 메소드의 판단 여부를 스스로 판단 후 →
- 파일 읽기가 성공 하면 `resolve`
- 실패하면 `reject` 를 알아서 알아서 반환 합니다! → 고로 편리합니다!

```
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-86MUOGC MINGW64 ~/
```

```
$ node js/file.js
```

```
1번 readme 텍스트 입니다  
2번 readme 텍스트 입니다  
3번 readme 텍스트 입니다  
4번 readme 텍스트 입니다
```

```
lhs@DESKTOP-86MUOGC MINGW64 ~/
```

```
$ node js/file.js
```

```
1번 readme 텍스트 입니다  
2번 readme 텍스트 입니다  
3번 readme 텍스트 입니다  
4번 readme 텍스트 입니다
```

```
lhs@DESKTOP-86MUOGC MINGW64 ~/
```

```
$ node js/file.js
```

```
1번 readme 텍스트 입니다  
2번 readme 텍스트 입니다  
3번 readme 텍스트 입니다  
4번 readme 텍스트 입니다
```





# Async / Await



# 최신 기술인 Async, Await 도 적용!

- Function 앞에 `async` 를 붙이면 해당 함수는 항상 `Promise` 를 반환
- 즉, `async` 가 붙은 함수에서 `return` 을 쓰면 아래와 동일한 역할을 합니다.

```
async function f1() {  
    return 1;  
}  
  
async function f2() {  
    return Promise.resolve(1);  
}
```

- `Async` 가 붙은 함수 내부에는 `Await` 키워드 사용이 가능!
- `Await` 은 `promise` 가 결과(`resolve`, `reject`)를 가져다 줄 때 까지 기다립니다.



# Async, Await 도 적용!

- 단, `async` 는 함수를 정의하는 상황에서 쓰이므로 함수 정의 후, 해당 함수를 외부에서 한번 사용해 줘야합니다!



# Await



```
const lucky = false;

const promise = new Promise((resolve, reject) => {
  console.log("주식이 오르기를 기다리기 시작합니다!");
  setTimeout(() => {
    console.log("3년의 시간이 흐르고....");
    if (lucky) {
      const profit = 3000;
      resolve(profit);
    } else {
      reject("아.... 망했어요");
    }
  }, 1000);
});
```





```
async function howMuch() {  
  try {  
    const result = await promise;  
  
    if (result) {  
      console.log(`기다림의 보상으로 ${result} 원을 벌었습니다!`);  
    }  
  } catch (err) {  
    console.log(err);  
  }  
}  
  
howMuch();
```

```
lhs@DESKTOP-86MUCGC MINGW64 /d/git/kdt-5th (main)  
$ node promise.js  
주식이 오르기를 기다리기 시작합니다!  
3년의 시간이 흐르고....  
기다림의 보상으로 3000 원을 벌었습니다!
```

```
lhs@DESKTOP-86MUCGC MINGW64 /d/git/kdt-5th (main)  
$ node promise.js  
주식이 오르기를 기다리기 시작합니다!  
3년의 시간이 흐르고....  
망했어요
```

# 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();
```



# Async / Await 로

## 콜백지옥 탈출!



```
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());
}

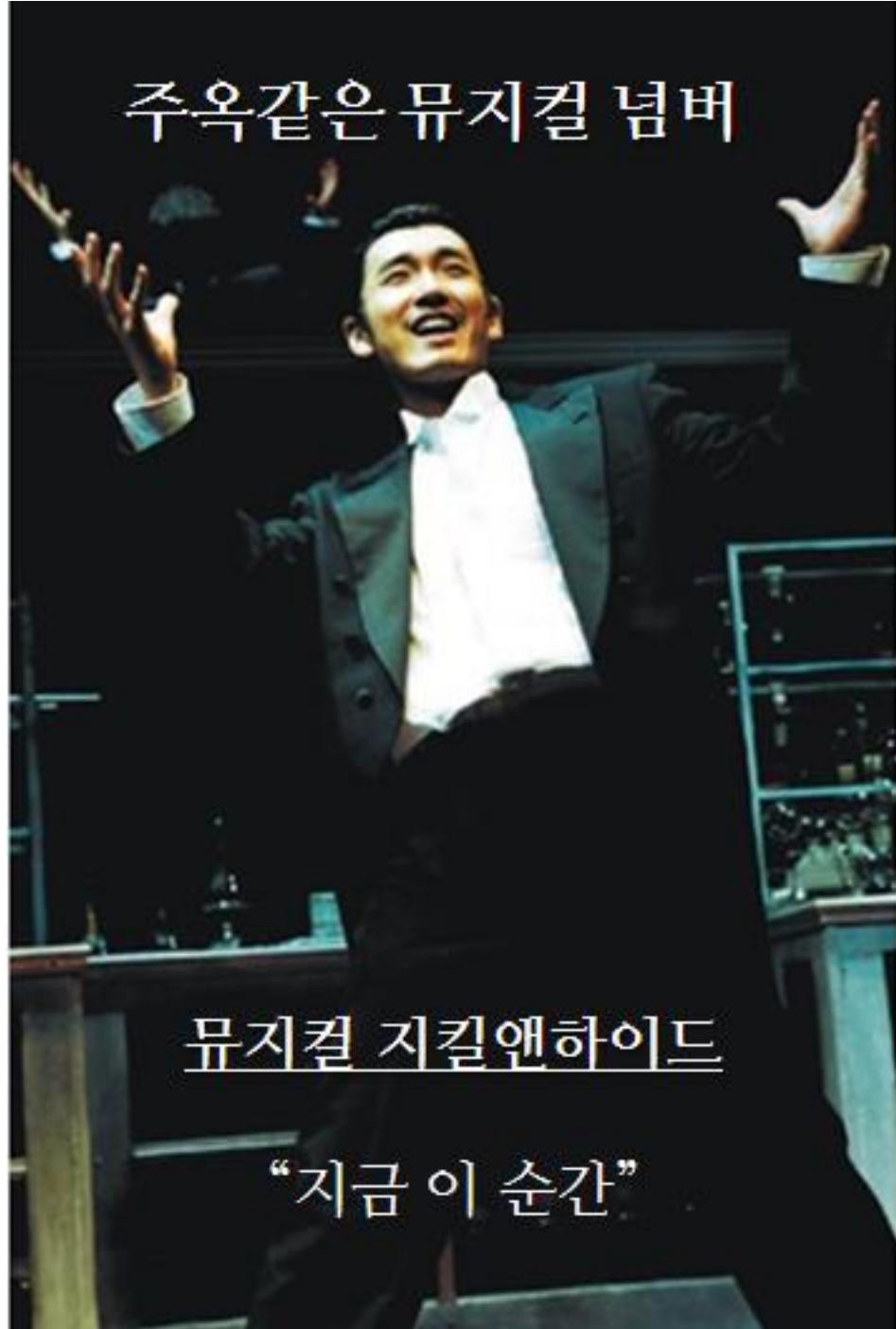
main();
```

```
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
```

  
+  
express





# Express

<http://expressjs.com/>



# Postman

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



# Postman

- 서버 테스트를 하는 프로그램 중, 제일 유명한 프로그램입니다!
- 백엔드 서버 테스트는 요걸로 테스트를 할 겁니다!
- 다운로드 해주시고, 회원 가입 or 구글 로그인 하면 사용이 가능합니다!



≡ Home Workspaces API Network Explore

Search Postman

My Workspaces

Search workspace Create Workspace

Recently visited

My Workspace

More workspaces

No workspaces found

Collections APIs Environments Mock Servers Monitors Flows History

View all workspaces →

localhost:4000

Authorization Headers (8) Body

Raw Preview Visualize Text

Hello, Express world!

A screenshot of the Postman application interface. The top navigation bar includes 'Home', 'Workspaces' (which is highlighted with a blue selection bar and has a red arrow pointing to it), 'API Network', and 'Explore'. A search bar labeled 'Search Postman' is positioned at the top right. The left sidebar contains links for 'Collections', 'APIs', 'Environments', 'Mock Servers', 'Monitors', 'Flows', and 'History', with 'Collections' currently selected (indicated by a red border). The main workspace shows a 'My Workspaces' section with a search bar, a 'Create Workspace' button, and a list of recently visited workspaces ('My Workspace'). Below this is a 'More workspaces' section stating 'No workspaces found'. The central area displays a request configuration for 'localhost:4000' with tabs for 'Authorization', 'Headers (8)', and 'Body'. At the bottom, there are 'Raw', 'Preview', 'Visualize', and 'Text' buttons, and a preview window showing the response 'Hello, Express world!'. A large red arrow points from the top red box down to the 'More workspaces' link.



GET ▼ http://localhost:4000

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

Query Params

KEY	VALUE
Key	Value

Body Cookies Headers (8) Test Results

Pretty Raw Preview Visualize HTML ▼

1 "안녕하세요. 여기는 백엔드입니다!"



# Express Middleware



# Middleware 가 뭐죠?

- 사전적인 의미로는 서로 다른 어플리케이션(프로그램)이 서로 통신을 하는 데 사용되는 소프트웨어를 뜻합니다!
- 즉, 양쪽 어플리케이션 가운데에서 역할을 하는 소프트웨어죠!
- Express 는 백엔드 서비스 구성을 위한 다양한 상황에 맞는 여러가지 서비스를 미들웨어 형태로 제공을 합니다.
- 즉, Express 에서의 미들웨어는 서버에 들어온 요청이 들어와서 응답으로 나갈 때 까지 거치는 모든 함수 또는 기능을 의미한다고 생각하시면 됩니다!



# Middleware 사용하기!

- Express 에서는 app.use 또는 app.method 함수를 이용해서 미들웨어를 사용합니다.
- App.use('요청 주소', (req, res, next) => {}); 의 형태로 사용이 가능합니다!



```
// @ts-check

const express = require('express');

const app = express();

const PORT = 4000;

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

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

```
lhs@DESKTOP-86MUCGC MINGW64 ~/Desktop/업무/K
$ http localhost:4000
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 15
Content-Type: text/html; charset=utf-8
Date: Thu, 01 Sep 2022 23:21:30 GMT
ETag: W/"f-oajqb6Lvrly+9VWwAZjgZh0HtYM"
Keep-Alive: timeout=5
X-Powered-By: Express

Hello, express!
```



# Middleware 사용하기, Next(?)

- Next 는 callback 함수로서 해당 함수가 호출 되면 현재 미들웨어를 종료하고 다음 미들웨어를 실행 시킵니다!

```
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');
});

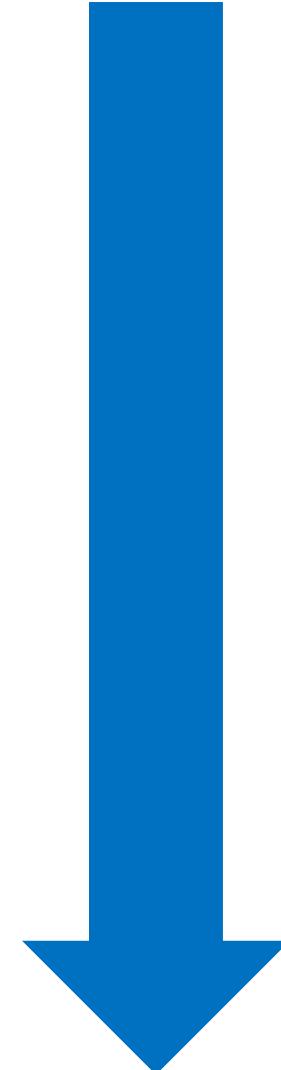
app.listen(PORT, () => {
  console.log(`The express server is running at port: ${PORT}`);
});
```



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



# Middleware 사용하기, Next(?)





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

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

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

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

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

The express server is running at port: 4000  
Middleware 2  
□



# Middleware 사용하기, Next(?)

- Next 를 상용해서 다음 미들웨어를 실행 시킬 때, 이전 미들웨어에서 처리한 값을 전달 하고 싶을 때가 생깁니다!
- 물론, global 영역에서 변수를 하나 만들어서 전달해도 되지만 global 변수는 보통 사용을 안하는 것이 좋습니다. → 변수가 언제 어디에서 변경이 되어 문제를 발생 시킬지 예측이 어렵기 때문이죠.
- 이럴 때에는 req 객체에 필드를 추가해서 전달하는 방법이 많이 사용 됩니다!
- 다만 이 방법도 정석적인 방법은 아니므로 급할 때만 사용하세요!



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

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

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

app.use((req, res, next) => {
  console.log('Middleware 2');
  res.send(`Requested at ${req.reqTime}`);
});

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



# Promise

- 당연히 서버 통신이므로 Promise 사용이 가능합니다!
- Fs 을 이용해서 간단하게 Promise 상황을 만들어서 봅시다!
- 사용이 직관적인 Async / Await 을 이용해서 구현합니다!
- 이 참에 promise {<pending>} 상태도 체험해 봅시다!



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

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

app.use('/', async (req, res, next) => {
  console.log('Middleware 1');
  req.reqTime = new Date();
  req.fileContent = await fs.promises.readFile('package.json', 'utf-8');
  next();
});

app.use((req, res, next) => {
  console.log('Middleware 2');
  console.log(req.fileContent);
  res.send(`Requested at ${req.reqTime}`);
});

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



# Back-End

## 서버의 기본!



# 요청 메소드



Create → POST

Read → GET

Update → PUT

Delete → DELETE

GET localhost:4000/users ● | POST lo



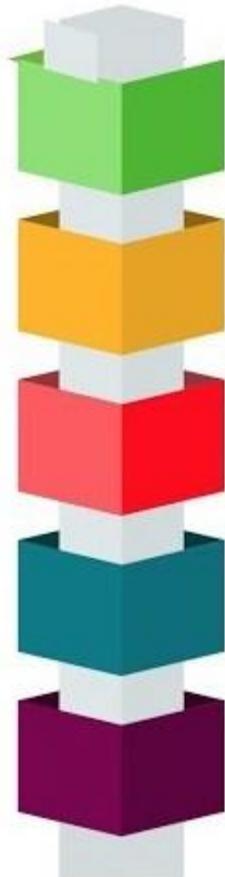
New Collection / localhost:4000/

GET × http://localhost:4000/

- GET
- POST
- PUT
- PATCH
- DELETE
- COPY
- HEAD
- OPTIONS
- LINK
- UNLINK
- PURGE
- LOCK
- UNLOCK
- PROPFIND
- VIEW



# HTTP Status Codes



**1XX**  
**INFORMATIONAL**

**2XX**  
**SUCCESS**

**3XX**  
**REDIRECTION**

**4XX**  
**CLIENT ERROR**

**5XX**  
**SERVER ERROR**



# HTTP Status

- 100 번대 : 정보 / 리퀘스트를 받고 처리 중
- 200 번대 : 성공 / 리퀘스트를 정상 처리
- 300 번대 : 리디렉션 / 처리 완료를 위해서는 추가 동작 필요
- 400 번대 : 클라이언트 에러 / 클라이언트에서 요청을 잘못 보냄
- 500 번대 : 서버 에러 / 리퀘스트는 잘 들어 갔지만 서버에서 처리를 못함



# HTTP Status Codes

## Level 200 (Success)

**200 : OK**

**201 : Created**

**203 : Non-Authoritative  
Information**

**204 : No Content**

## Level 400

**400 : Bad Request**

**401 : Unauthorized**

**403 : Forbidden**

**404 : Not Found**

**409 : Conflict**

## Level 500

**500 : Internal Server Error**

**503 : Service Unavailable**

**501 : Not Implemented**

**504 : Gateway Timeout**

**599 : Network timeout**

**502 : Bad Gateway**



백엔드로

데이터 요청!

# 백엔드 데이터 요청!



- 여러분 백엔드 설계를 할 때 모든 요청을 예상해서 api 를 만들어 놓을 수는 없겠죠?
- 그리고 회원가입을 하는 과정이라면 회원가입을 하는 유저의 ID와 Password는 프론트에서 받을 수 밖에 없겠죠?



요청사항:

가게 : 락고 많이 주세용! 간장은 하나만 주셔도 돼요!

배달 : 조심히 안전하게 와주세요 :)

메뉴명	수량	금액
치즈피자	1	
+ 치즈 크러스트 추가(2,000원)		
+ 꿀(500원)		
+ 피클(400원)		
+ 펩시 (500ml)(1,400원)		



# 백엔드에서 데이터를 받는 방법



# Req.Params



# URL 로 Data 를 받는 방법, Params

- 보통 요청은 주소로 들어오죠? 그렇기 때문에 프론트에서 받아야 하는 정보는 주소 값에 담아서 보냅니다!

```
app.get('/:id', (req, res) => {
  console.log(req.params);
  res.end(`ID 번호가 ${req.params.id} 인 회원 정보`));
});
```

- 받을 url에서 **:파라미터명** 을 미리 정의해 두면 해당 내용은 req.params 에 담겨서 전달이 됩니다!



GET ▼ http://localhost:4000/1

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

Query Params

KEY	VALUE
Key	Value

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize Text ▼ ≡

1 ID 번호가 1 인 회원 정보



# URL로 Data를 받는 방법, Params

- 물론 여러 개를 받을 수도 있습니다.

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

```
The express server is running at port: 4000
{ id: '1', name: 'tetz', gender: 'male' }
```

- Req.params라는 객체에 담겨서 전달이 되어서 편리하게 사용이 가능합니다!



GET ▼ http://localhost:4000/1/tetz/male

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

Query Params

KEY	VALUE
Key	Value

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON ▼

```
1 {  
2   "id": "1",  
3   "name": "tetz",  
4   "gender": "male"  
5 }
```



# URL로 Data를 받는 방법, Params

- 여러 개를 받을 때, 어떤 값을 전달하는지 보내는 쪽에서 인지시키기 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"
}
```



GET ▼ <http://localhost:4000/id/1/name/tetz/gender/male>

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

Query Params

KEY	VALUE
Key	Value

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON ≡

```
1 {  
2   "id": "1",  
3   "name": "tetz",  
4   "gender": "male"  
5 }
```



# Req.Query



# URL 로 Data 를 받는 방법, Query

- Params 의 약점은 정의 된 형태로만 데이터를 받을 수 있습니다.
- 하지만 막 보내고 싶다면!? Query 를 쓰면 됩니다!
- Query 는 url에 ? 를 붙인 뒤, 필드명=값 으로 사용하면 됩니다!
- 여러 개를 보내고 싶으면, & 로 묶어서 여러 개를 보낼 수 있습니다!



```
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
<input checked="" type="checkbox"/> test					<input type="text"/> test	<input type="button" value="VALUE"/>
<input checked="" type="checkbox"/> id					<input type="text"/> 3	<input type="button" value="VALUE"/>

# 실습, req.params 와 req.query



- Parameter 방식과 Query 방식으로 email, password, name, gender 정보를 받아와서 출력하는 백엔드 코드 작성하기!
- 슬랙 과제 제출에 댓글로 제출해 주세요!
- 코드와, 각각 요청을 보내고 받은 postman의 스크린 샷을 보내주세요!



# 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 Routing이라고 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!



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

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

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('/users', userRouter);

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
  - 특정 회원 정보 보여주기
- POST Localhost:4000/users?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 정보를 params로 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', (req, res) => {
  const userData = USER[req.params.id];
  if (userData) {
    res.send(userData);
  } else {
    res.end('ID not found');
  }
});
```



# 회원 추가하기 API

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



```
userRouter.post('/', (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/1?id=test&name=test
- 회원 삭제 테스트 URL
  - DELETE Localhost:4000/users/1
- 슬랙 과제 제출에 댓글로 제출해 주세요!



# HTML



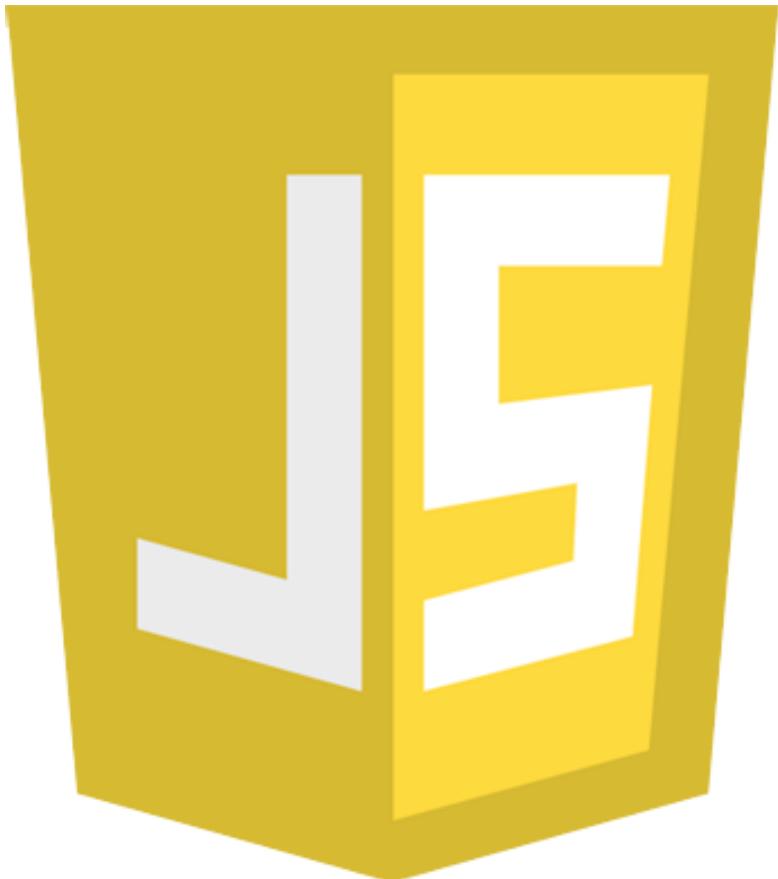


# HTML



- 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',
},
```

# 그렇다면...



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





# View Engine



# EJS

<% Embedded JavaScript %>

Templating: Node, Express, EJS



# pug

# NUNJUCKS



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



- HTML 문법을 단순화 하여 사용
- 레이아웃 가능 O



- HTML 문법을 그대로 사용
- 레이아웃 가능 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

# 동작 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



```
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>
    송 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 보기 가 좀 그렇죠?

- EJS 형식의 경우 보기 가 좀 그렇네요!?
- 그럼 좀 이쁘게 봅시다!

**EJS language support** v1.3.1

DigitalBrainstem | 627,577 | ★★★★★(27)

2019 - EJS language support for Visual Studio Code.

사용 안 함 | 제거 | ⚙

이 확장은 전역적으로 사용하도록 설정되었습니다.

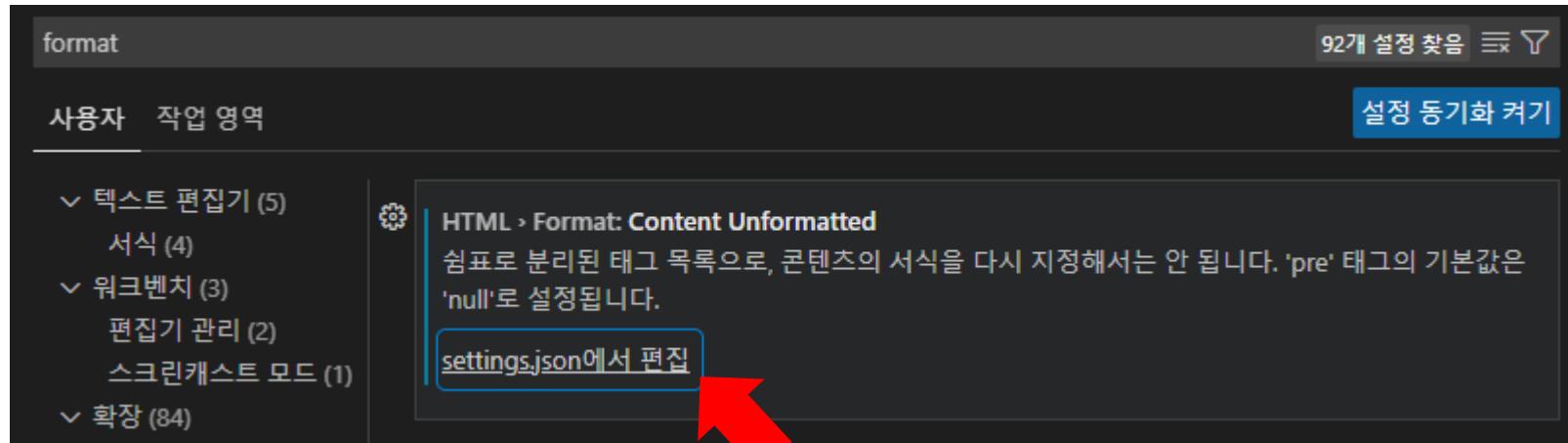
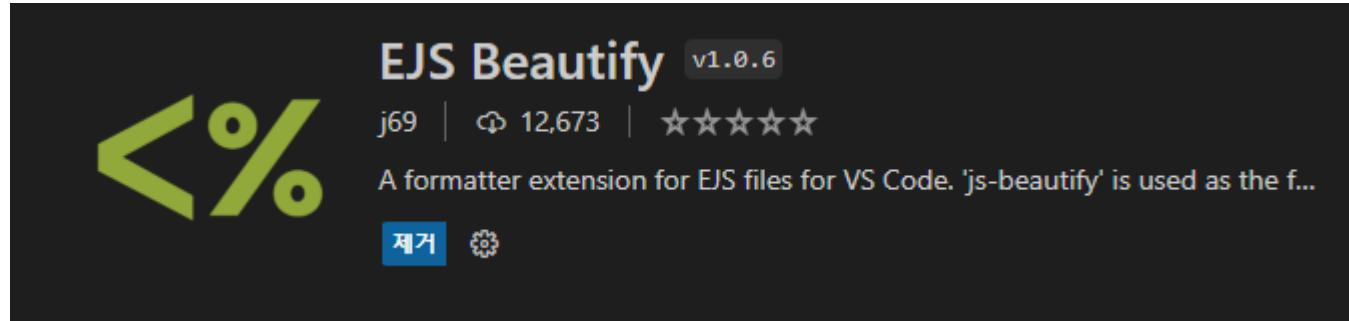
[세부 정보](#) [기능 기여도](#) [변경 로그](#) [런타임 상태](#)

```
<% 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 보기 가 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!?





# 어러 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">
```

A screenshot of a web browser window titled "Document". The address bar shows "localhost:4000/users". The page content is displayed on a yellow background. It starts with "회원 목록" (Member List) and "총 회원 수 2". Below this, there are two entries:

- ID: tetz  
이름: 이효석  
email: xenosign@naver.com
- ID: test  
이름: test  
email: test@naver.com



# 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!

# 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

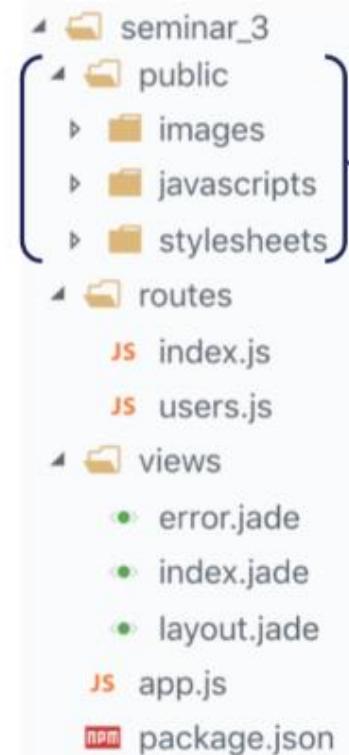
# 기본 폴더 구조



# Express 기본 폴더 구조

- Express로 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

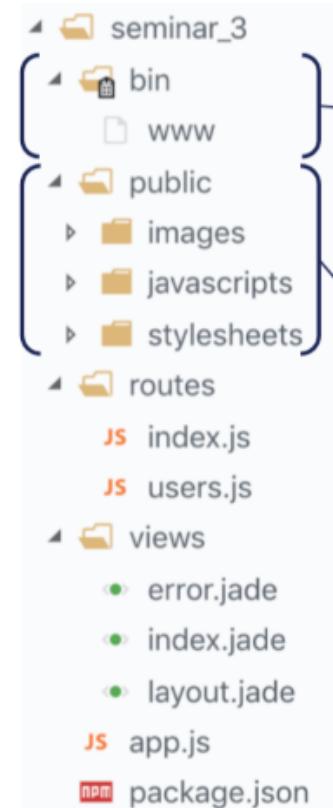
- /images : 그림파일들을 저장
- /javascripts : 자바스크립트 파일들을 저장
- /stylesheets : CSS 파일들을 저장



# Express 기본 폴더 구조

- Express로 만들어진 서비스의 기본 폴더 구조를 배워 봅시다!

## 프로젝트 구조



### /bin/www

서버를 실행하는 스크립트

프로젝트 돌아가는 포트번호 바꿀 수 있음

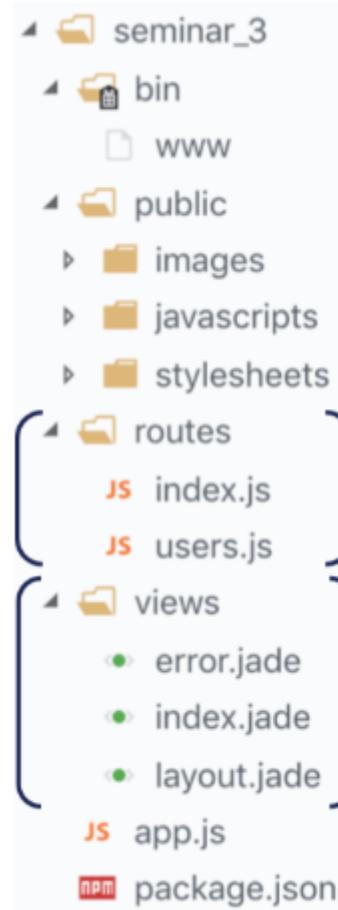
### /public

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)

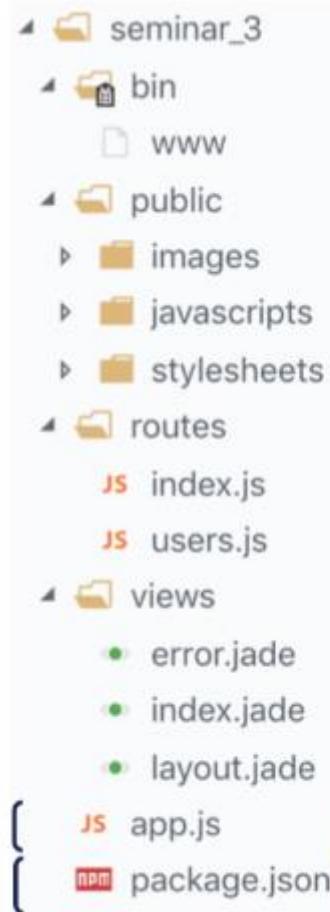
url 별로 실행되는 실제 서버 로직

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
  ⌄ css
    # style.css
  ⌄ js
    JS test.js
```



# 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!



# 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 user에 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!
- Users.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 페이지 라우팅!

- 아무런 주소 없이 localhost:4000 으로 접속 하였을 때, views 폴더에 있는 index.ejs 파일을 띄워주는 백엔드 구성하기
- Index.ejs 은 간단하게 h1 태그로 Welcome, Express service 라고 띄워주기!
- 단, 해당 기능을 index.js 파일로 따로 만들어 라우팅 할 것!



# Error 핸들링!



# Error 핸들링

- Users 라우터에서 문제가 발생하면 res.end로 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 만들 어서 throw 로 전달하면 됩니다!



# Error 던지기!

localhost:4000/users/1

```
Error: ID not found
at C:\Users\tetz\Desktop\KDT\__수업 자료\정규 수업\backend\routes\users.js:34:17
at Layer.handle [as handle_request] (C:\Users\tetz\Desktop\KDT\__수업 자료\정규 수업\backend\node_modules\express\lib\router\layer.js:95:5)
at next (C:\Users\tetz\Desktop\KDT\__수업 자료\정규 수업\backend\node_modules\express\lib\router\route.js:144:13)
at Route.dispatch (C:\Users\tetz\Desktop\KDT\__수업 자료\정규 수업\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\backend\node_modules\express\lib\router\layer.js:95:5)
at C:\Users\tetz\Desktop\KDT\__수업 자료\정규 수업\backend\node_modules\express\lib\router\index.js:284:15
at param (C:\Users\tetz\Desktop\KDT\__수업 자료\정규 수업\backend\node_modules\express\lib\router\index.js:365:14)
at param (C:\Users\tetz\Desktop\KDT\__수업 자료\정규 수업\backend\node_modules\express\lib\router\index.js:376:14)
at Function.process_params (C:\Users\tetz\Desktop\KDT\__수업 자료\정규 수업\backend\node_modules\express\lib\router\index.js:421:3)
at next (C:\Users\tetz\Desktop\KDT\__수업 자료\정규 수업\backend\node_modules\express\lib\router\index.js:280:10)
```

- 그럼 이렇게 Err 가 발생하는데 이런 페이지는 가급적 피하는 편이 좋으므로 app.js에서 전에 Err 를 핸들링 해봅시다!



# App.js 에서 에러 핸들링!

- App.js 의 미들웨어중 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 인자를 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 은 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
- 하도 많이 써서 이제는 express 의 기본 기능으로 들어 갔습니다
- Form 에서 전송 된, 정보를 req.body 에 담아서 obj 로 전달해 주는 역할을 합니다.

```
const bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({ extended: true }));
```

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



# 이제 req.query 와 req.body 를 같이 처리!

- Form 으로 들어온 데이터는 이제 req.body 에 담겨 있으므로 같이 처리 해 줘야 합니다.



```
router.post('/', (req, res) => {
  if (req.query.id) {
    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.body.email,
      };
      USER.push(newUser);
      res.send('회원 등록 완료');
    } 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 전체 조회 : GET localhost:4000/posts
  - 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!



수고하셨습니다!