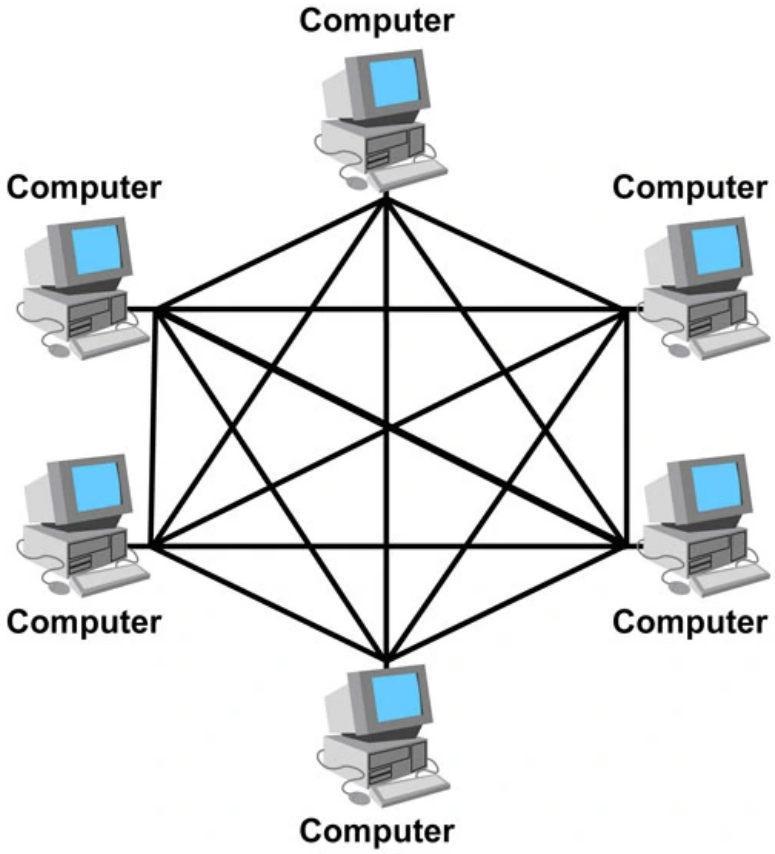
221226_2반_실습

Node.js

Node.js 개요

Node.js 란?

> 💡 **Node**
>
> - 노드(node)는 **컴퓨터 과학에 쓰이는 기초적인 단위**
> - 변과 함께 그래프를 구성하는 요소. 결절(結節), 정점(頂點), 점 등이라 한다. 그래프를 이루는 점과 선 중에서 점을 ***노드*** 또는 절점이라 한다.
> - **노드는 정보를 전송, 수신 및 / 또는 전달할 수있는 다른 장치의 네트워크에있는 모든 물리적 장치입니다.**



- Node.js는 Chrome V8 JavaScript 엔진으로 빌드 된 JavaScript 런타임입니다.

> 💡 ***런타임(Runtime)***
> 은 프로그램이 실행되고 있는 때 존재하는 곳을 말한다. 즉, 컴퓨터 내에서 프로그램이 기동되면, 그것이 바로 그 프로그램의 ***런타임***이다.

- 노드를 통해 **다양한 자바스크립트 애플리케이션을 실행**할 수 있으며, 서버를 실행하는 데 주로 사용
- Node.js는 JavaScript를 서버에서도 사용할 수 있도록 만든 프로그램
- Node.js는 V8이라는 JavaScript 엔진 위에서 동작하는 자바스크립트 런타임(환경)
- Node.js는 서버사이트 스크립트 언어가 아닌 프로그램(환경)
- Node.js는 웹서버와 같이 확장성 있는 네트워크 프로그램 제작이 목적
- 내장 HTTP 서버 라이브러리를 포함하고 있어 웹 서버에서 **아파치 등의 별도 소프트웨어 없이 동작하는 것이 가능,** 이를 통한 웹 서버의 동작에 있어 더 많은 통제에서 벗어나 여러 가지 기능이 가능

Node.js 사용 이유

- JavaScript 를 웹 브라우저에서 독립시킨 것으로 Node.js를 설치하게 되면 터미널프로그램(윈도우의 cmd, 맥의 terminal 등)에서 Node.js를 입력하여 브라우저 없이 바로 실행할 수 있다.
- Node.js를 이용하여 웹 브라우저와 무관한 프로그램을 만들 수 있다.
- Node.js를 이용하여 **서버를 만들 수 있다.**
- 이전까지 Server-Client 웹사이트를 만들 때 웹에서 표시되는 부분은 JavaScript 를 사용하여 만들어야만 했으며, 서버는 Reby, Java 등 다른 언어를 써서 만들었어야 했는데 마침내 **한 가지 언어로 전체 웹 페이지를 만들 수 있게 된 것**이다.
- 이벤트 기반, 논 블로킹 I/O 모델을 사용해 가볍고 효율적

Node.js 특징

- **비동기 I/O 처리 / 이벤트 위주:** Node.js 라이브러리의 모든 API는 비동기식입니다, 멈추지 않는다는거죠 (Non-blocking). Node.js 기반 서버는 API가 실행되었을때, 데이터를 반환할때까지 기다리지 않고 다음 API 를 실행합니다. 그리고 이전에 실행했던 API가 결과값을 반환할 시, NodeJS의 이벤트 알림 메커니즘을 통해 결과값을 받아옵니다.

- **빠른 속도:** 구글 크롬의 V8 자바스크립트 엔진을 사용하여 빠른 코드 실행을 제공합니다.
- **단일 쓰레드 / 뛰어난 확장성:** Node.js는 이벤트 루프와 함께 단일 쓰레드 모델을 사용합니다. 이벤트 메커니즘은 서버가 멈추지않고 반응하도록 해주어 서버의 확장성을 키워줍니다. 반면, 일반적인 웹서버는 (Apache) 요청을 처리하기 위하여 제한된 쓰레드를 생성합니다. Node.js 는 쓰레드를 한개만 사용하고 Apache 같은 웹서버보다 훨씬 많은 요청을 처리할 수 있습니다.
- **노 버퍼링:** Node.js 어플리케이션엔 데이터 버퍼링이 없고, 데이터를 chunk로 출력합니다.
- **라이센스:** Node.js 는 MIT License가 적용되어있습니다.

Node.js 설치

1. 직접 설치

> 💡 node.js 공식 홈페이지
> https://nodejs.org/en/

- LTS(Long Term Supported)
  - 장기적으로 안정되고 신뢰도가 높은 지원이 보장되는 버전
  - 유지/보수와 보안(서버 운영 등)에 초점을 맞춰 대부분 사용자에게 추천되는 버전
  - **짝수 버전(ex. 8.x.x)이 LTS 버전**
- Current(현재 버전)
  - 최신 기능을 제공하고 기존 API의 기능 개선에 초점이 맞춰진 버전
  - 업데이트가 잦고 기능이 변경될 가능성이 높기 때문에 간단한 개발 및 테스트에 적당한 버전
  - **홀수 버전(ex. 9.x.x)이 Current 버전**
- 설치 확인

```
node -v
# v8.9.4
npm -v
# 5.6.0
```

    - 설치 후 버전 확인 명령을 통해 node명령이 정상적으로 동작하는지 확인

2. 패키지 매니저로 설치
    - macOS - Homebrew

```
brew install node@8
```

        - 버전의 Major Number만 입력
    - Windows - Chocolatey

```
choco install nodejs-lts
```

NPM(Node Package Manager)

NPM 이란?

- Node.js 개발자들이 패키지(모듈)의 설치 및 관리를 쉽게 하기 위해 도와주는 매니저(관리 도구)

💡 패키지(모듈) : 프로그램의 구성요소 중 특정 기능을 수행할 수 있는 코드의 집합(라이브러리).

유명한 플랫폼(프로그래밍 언어, OS 등)은 저마다의 패키지 매니저를 가지고 있는데,

다른 유명 패키지 매니저로는

- **Python**의 pip
- **Java**의 Maven, Gradle(Android, React Native에서 많이 봤는데??? 그거 맞습니다.^^)
- **PHP**의 Composer
- **Ruby**의 RubyGems

등이 있고,

Linux환경에 익숙하신 분들은

- **레드햇 계열**의 rpm, yum
- **데비안 계열**의 dpkg, apt
- **맥 OS의** Homebrew

등이 있습니다.

- npm은 JavaScript 및 세계 최대의 소프트웨어 레지스트리 패키지 관리자
- node.js 설치시 같이 설치됨
- npm에는 Node.js에서 사용되는 각종 코드 패키지들이 모여있고, 우리는 그 패키지를 다운로드 받아 사용할 수 있습니다.
- 쉽게 npm은 Node.js 생태계의 앱스토어나 플레이스토어 같은 역할
- npm 레지스트리에는 640,000개가 넘는 패키지가 포함
- 패키지는 의존성(dependencies) 및 버전을 추적할 수 있도록 구성

💡 npm
https://www.npmjs.com/

NPM 장점

- 프로그램을 제작 시 어떤 기능을 구현할 때 자신이 직접 프로그래밍을 하지 않아도 동일한 기능의 남이 만들어놓은 코드를 쉽게 사용이 가능하다.
- 코드의 재사용성이 높아지고 유지 보수가 쉬워질뿐더러 형상관리가 용이해진다.

NVM(Node Version Manager)

NVM 이란?

- Node.js 의 버전을 관리하는 도구
- NVM을 사용하는 이유
  - **협업을 할 때, 또는 다양한 프로젝트를 동시에 진행해야 할 때**
  - 다양한 라이브러리 / 프레임워크 / 개발툴의 버전 호환 문제발생
- NVM의 장점
  - 다양한 버전의 Node.js를 설치 가능
  - 명령을 통해 다양한 Node 버전으로 스위칭 가능
  - 디폴트 버전을 설정하거나 / 설치한 버전들의 전체 리스트를 확인하거나 / 필요 없는 버전을 삭제하는 등 버전 관리가 용이
  - `Ruby`의 `rvm`, `rbenv`나 `Python`의 `pyenv`도 같은 역할

node.js 명령어

npm 명령어

1. 버전 확인

```
npm -version
#또는
npm -v
```

2. node.js 프로젝트 시작

```
npm init
```

- Node.js 프로젝트를 시작할때 package.json을 생성해 주는 명령

`package.json`
프로젝트의 정보와 특히 프로젝트가 의존하고 있는(설치한) 패키지(모듈)에 대한 정보가 저장되어 있는 파일.

- package.json 예
  - 기본 package.json

```
{
  "name": "example",
  "version": "1.0.0",
  "description": "example",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "example"
  },
  "keywords": [
    "example"
  ],
  "author": "minchan",
  "license": "MIT"
}
```

  - react native

```
{
  "name": "example",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "android": "react-native run-android",
    "ios": "react-native run-ios",
    "start": "react-native start",
    "test": "jest",
    "lint": "eslint ."
  },
  "dependencies": {
    "@react-native-community/masked-view": "0.1.6",
    "@react-navigation/bottom-tabs": "5.0.0",
    "@react-navigation/drawer": "5.0.0",
    "@react-navigation/native": "5.0.0",
    "@react-navigation/stack": "5.0.0",
    "react": "16.9.0",
    "react-native": "0.61.5",
    "react-native-gesture-handler": "1.5.6",
```

```
    "react-native-reanimated": "1.7.0",
    "react-native-safe-area-context": "0.7.0",
    "react-native-screens": "2.0.0-beta.2"
  },
  "devDependencies": {
    "@babel/core": "7.8.4",
    "@babel/runtime": "7.8.4",
    "@react-native-community/eslint-config": "0.0.7",
    "@types/react": "16.9.19",
    "@types/react-native": "0.61.10",
    "babel-jest": "25.1.0",
    "eslint": "6.8.0",
    "jest": "25.1.0",
    "metro-react-native-babel-preset": "0.58.0",
    "react-test-renderer": "16.9.0",
    "typescript": "3.7.5"
  },
  "jest": {
    "preset": "react-native"
  }
}
```

- 이렇게 package.json에 **프로젝트에 대한 정보**, npm과 연결하여 **사용할 명령**("`scripts`"객체), 프로젝트가 의존하고 있는(설치한) **패키지(모듈)에 대한 정보**("`dependencies`"객체), 프로젝트의 **개발과 관련된 테스트, 컴파일, 코드 작성 형태와 같은 패키지(모듈)에 대한 정보**("`devDependencies`"객체) 등이 저장되어 있는 것을 확인하실 수 있습니다.

3. 패키지 설치
   - 문법

```
npm install (option) [package]
```

   - 옵션
     - `g` : 패키지가 해당 프로젝트(local)가 아닌 시스템 레벨에 전역(global) 설치되어 다른 Node.js 프로젝트에서도 사용할 수 있게 됩니다.
     - `-save`(`-S`) : package.json의 "dependencies"객체에 추가됩니다. (**npm5부터 default**로 설정되어 더 이상 사용하지 않습니다.)
     - `-save-dev`(`-D`) : package.json의 "devDependencies"객체에 추가됩니다.
     - **@패키지 버전** : 패키지명 뒤에 @패키지 버전을 쓰시면 해당 버전의 패키지가 설치되며 입력하지 않을 시 최신 버전으로 설치가 됩니다.

> 💡 **npm install(npm i) 사용팁**
>
> 만약 패키지명을 입력하지 않고 npm install(npm i)만 입력할 시 package.json의 "dependencies"객체에 명시되어 있는 패키지(모듈)들을 모두 설치하게 됩니다.
>
> 따라서 프로젝트의 형상관리를 위해 `GitHub`와 같은 저장소에 업로드할 때, 무겁고 수많은 패키지(모듈)들을 전부 업로드하는 것이 아니라, `package.json`만 업로드해놓으시면 나중에 프로젝트를 내려받았을 때 npm i 명령어를 통해서 기존 프로젝트에서 사용하던 패키지(모듈)들을 손쉽게 원상복귀시키실 수 있습니다!
>
> (.gitignore파일에 node_modules를 추가하여 패키지(모듈)들이 commit되지 않게 설정해두세요.)

4. 패키지 삭제
   - 문법

```
npm uninstall [package]
```

   - npm uninstall 명령 뒤에 삭제하실 패키지명을 입력하시면 설치된 패키지가 node_modules폴더에서 삭제될 뿐만 아니라 package.json의 "dependencies"객체에서도 삭제됩니다.
   - **단, 설치하실 때 옵션을 사용하셨다면 삭제하실 때도 같은 옵션을 넣어주세요.**
5. 패키지 업데이트
   - 문법

```
npm update [package]
```

   - 설치된 패키지를 최신 버전으로 업데이트합니다.
   - 의존성이 엮여있는 패키지를 함부로 업데이트하면 잘 돌아가던 프로젝트가 갑자기 에러의 굴레에 휘말릴 수 있으니 기존의 버전을 기억해두시거나 신중하게 업데이트 하셔야 합니다.
6. 초기화
   - 명령어

```
npm cache clean
npm rebuild
```

- 두 가지 명령을 차례로 명령해주시면 됩니다.

> 💡 npm cache clean 명령은 npm의 cache를 지우는 명령이고 npm rebuild 명령은 npm을 새롭게 재설치 하는 명령입니다. 주로 npm 명령어가 안 먹히거나 기타 잡다한 버그가 생겼을 시 해결하기 위한 조치 방법으로 쓰이니, 꼭 기억해 두셨다가 npm에 문제가 발생했을 때 사용해보시면 좋을 것 같습니다!

7. `package.json`

- `npm init` 명령어로 초기화를 해주면 `package.json` 파일이 자동으로 생성되는데, 이 때 `package.json` 파일

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

- `scripts`
  - shell script를 지정해 두면 필요할 때 실행 가능
  - `test` 라는 스크립트가 지정되어 있는데, `npm test` 를 입력하여 해당 스크립트를 실행하면 다음과 같은 메세지가 출력되는 것을 확인

```
npm test
#Error: no test specified\" && exit 1
```

  - 예제

```
#만약 새로운 스크립트를 지정하고 싶다면 다음과 같이 새로운 라인에 스크립트를 추가하고 실행
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "msg": "echo \"Hello, World\""
},
```

```
npm run msg
```

nvm 명령어

1. 특정 버전의 node.js 설치
   - 문법

```
nvm install [version]
```

   - 예제

```

# node.js 버전 설치하기
nvm install 0.10
nvm install v0.1.2
nvm install v8

# node 최신 버전 설치 (설치 당시 기준)
nvm install node

# node LTS 최신버전 설치
nvm install --lts
```

2. 설치된 버전 확인 및 삭제

```
# 설치된 node.js 목록 확인하기
nvm ls

# 설치할 수 있는 모든 Node 버전 조회 (재미삼아 해보지마세요 겁나많음... 황급히 control C 두드리기)
$ nvm ls-remote

# 특정 버전의 node 사용하기
$ nvm use <version>

# 현재 사용중인 버전 확인하기
$ nvm current

# node.js 설치 경로 확인하기
```

```
$ which node

# 필요없는 node 버전 삭제하기
$ nvm uninstall <version>
```

3. 기본 버전 설정

```
$ nvm alias default 8.9.4

# 설치되어 있는 가장 최신버전의 node를 디폴트로 사용하기
$ nvm alias default node
```

- 만일 새로운 쉘을 실행할 경우 `node` 의 버전이 `system` 버전으로 리셋되는데요, 이를 고정하기 위한 커맨드는 다음과 같습니다.

Express

Express 개요

Express 란?

- **NodesJS를 사용하여 쉽게 서버를 구성할 수 있게 만든 프레임워크(클래스와 라이브러리의 집합체)**

> 💡 **웹 프레임워크**
> 웹서비스를 개발, 제공하는 과정에서 반복적으로 처리하는 작업의 효율적인 자동화를 위한 클래스와 라이브러리의 모음이다. (Spring, Django, Express.js, Angular JS, Vue.js 등)

- 쉽게 얘기하자면, 서버를 구성할 수 있게 만들어주는 클래스와 라이브러리의 집합체
- node.js 개발 시 개발을 빠르고 손쉽게 할수록 도와주는 역할을 한다. 이것은, 미들웨어 구조 때문에 가능한 것이다. 자바스크립트 코드로 작성된 다양한 기능의 미들웨어는 개발자가 필요한 것만 선택하여 express와 결합해 사용할 수 있다.
- 프레임워크이므로 웹 애플리케이션을 만들기 위한 각종 라이브러리와 미들웨어 등이 내장돼 있어 개발하기 편하고, 수많은 개발자들에게 개발 규칙을 강제하여 코드 및 구조의 통일성을 향상시킬 수 있다.

- nodeJS를 사용한 **REST 서버**를 편리하게 구현하게 해주는 **프레임워크**로는 Koa, Hapi, **express** 등이 있다.

> 💡 **REST**
>
> REST는 Representational State Transfer를 의미한다.
>
> 한글로 풀어서 말을 하자면
>
> **자원을 이름(자원의 표현)으로 구분하여 해당 자원의 상태(정보)를 주고 받는 모든 것을 의미한다고 한다.**
>
> **자원**
>
> (source)은 해당 소프트웨어가 관리하는 모든 것을 의미하며, 뭐 예를 들면
>
> **DB 안에 들어가 있는 데이터 하나하나, 이미지 하나하나 등을 의미할 수 있겠다**
>
> . 상태의 전달은 데이터가 요청되어지는 시점에서 요청받은 자원의 상태(정보)를 전달하는것을 의미하는데, 보통 JSON 형태나 XML 형태를 이용하여 자원의 상태를 전달하게 된다.
>
> REST의 구체적인 개념은 HTTP URL을 통해 자원을 명시하고,
>
> **HTTP Method(POST, GET, DELETE, PUT)**
>
> 를 통해 해당 자원에 대한
>
> **CRUD(CREATE, READ, UPDATE, DELETE)**
>
> 오퍼레이션을 적용하는 것이라고 할 수 있다.

Express의 장점

- Express는 프레임워크이므로 웹 애플리케이션을 만들기 위한 각종 라이브러리와 미들웨어 등이 내장돼 있어 개발하기 편하고, 수많은 개발자들에게 개발 규칙을 강제하여 코드 및 구조의 통일성을 향상시킬 수 있다.

Express 사용하기

HTTP 내장 모듈 vs Express

- HTTP 내장 모듈

```
//http 내장모듈을 사용한 웹서버 띄우기
const http = require('http');

http.createServer(function(request, response){
    response.writeHead(200, {'Content-Type':'text/html'});
    response.write('Hello http webserver!')
    response.end();
}).listen(52773, function(){
    console.log("server running http://127.0.0.1:52773/");
});
```

- Express

```
//express 웹프레임워크를 이용한 서버 띄우기 실습
const express = require('express');
const app = express;
const port = 3000;

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

app.listen(port, () => {
    console.log('Express server listen..')
});
```

nodemon

- 코드 수정을 할 때마다, 서버를 내렸다가 올리려면 귀찮다. 이럴 때 nodemon을 설치하면 소스가 수정될때마다 자동으로 서버를 내렸다 올려주기 때문에 개발하기 편하다.
- nodemon 설치

```
//nodemon을 전역으로 설치
$ npm install -g nodemon

//설치후에는 아래 명령어를 실행하면 된다.
$ nodemon 파일명
```

- nodemon 실행

```
nodemon [js file]
```

Express 명령어

1. package.json 생성하기

```
npm init
```

2. express.js 설치

```
npm install express
```

- 설치 후 package.json 파일 확인

```
"dependencies": {
    "express": "^4.17.1"
  },
```

Express 예제

1. Hello World 출력하기

① package.json에 명령어 추가

```
"scripts": {
  "start": "node index.js"
  ...
}
```

② index.js 파일을 생성 후 코드 추가

```
const express = require('express'); //express를 설치했기 때문에 가져올 수 있다.
const app = express();

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

app.listen(5000, () => {
  console.log(`Example app listening on port ${port}`)
});
```

- `app` : 서버에 접근하여 할 수 있는 행동들이 담긴 "함수". 이 함수를 통해 서버에 port(3000)을 연결할 수도 있고, 어떤 이벤트가 실행된다면 서버에서 특정 값이나 행동을 받아올 수도 있으며, End Point(밑에서 설명)을 만들어낼 수도 있다.

- app이라는 변수에 express 함수의 반환 값을 저장하였다.
  - 이 app이라는 변수를 사용해서 REST End Point들을 생성할 것이다. End Point란 쉽게 얘기해서, "http://localhost:3001/api/login" 이라는 주소가 있다면, 3001 은 클라이언트에서 서버로 접근하기 위한 port를 나타내는 것이며, /api/login 이 바로 End Point trigger 라고 할 수 있다.
  - 클라이언트에서 서버로 가는 길은 port이다. port의 끝자락에 위치하면서, 서버가 어떤 trigger을 받았을 때 어떤 행동을 해야할 지(어떤 리소스에 접근해야 하는지) 알려주는 URL이 End Point이다.

> 💡 ***REST End Point 란?***
>
> **Endpoint**
>
> 메소드는 같은 URL들에 대해서도 다른 요청을 하게끔 구별하게 해주는 항목이 바로 'Endpoint'입니다.
>
> | HTTP메소드 | URI(자원) | Endpoint의 행위 |
> |---|---|---|
> | POST | http://api.domain.com/books | 새로운 도서정보 생성 |
> | GET | http://api.domain.com/books | 도서정보 목록 조회 |
> | GET | http://api.domain.com/books/1 | 1번 도서정보 조회 |
> | PUT | http://api.domain.com/books/1 | 1번 도서정보 수정 |
> | DELETE | http://api.domain.com/books/1 | 1번 도서정보 삭제 |
>
> 각각 GET, PUT, DELETE 메소드에 따라 다른 요청을 하는 것을 알 수 있습니다.
>
> 결국 Endpoint란 API가 서버에서 자원(resource)에 접근할 수 있도록 하는 URL입니다.

- `app.get()` :
  - REST API의 한가지 종류인 **GET 리퀘스트**를 정의하는 부분이다. app.get이라고 작성했기 때문에 get 요청으로 정의가 되고 app.post로 작성할 경우 post 요청으로 정의가 된다. REST API의 종류 (get, post, update, delete 등등)을 사용하여 **End Point**를 작성할 수 있다. (여기서 End Point는 "/"이거 단 하나이다.)
  - 위와 같이 End Point 생성 시 파라미터는 두 가지를 받는다. 첫 번째 파라미터는 **URL 정의 ('/')** 두 번째 파라미터는 해당 url에서 수행할 **작업 및 응답**을 정의할 수 있다. URL 정의를 통해서, "http://localhost:3000" 일 때, 두 번째 파라미터 함수를 실행한다. (그래서 trigger라고 했다!)

    - 이 함수에는 두 개의 파라미터를 받는데 **요청에 해당하는 req (request)** 와 **응답에 해당하는 res (response)**이다. 요청에 대한 정보는 req에 저장되어있고 응답할 때 res 파라미터를 사용하여 응답 정보를 송신한다. 위 코드는 res 의 **send** 메서드를 통해, h1 이라는 HTML 태그를, client 컴퓨터가 "http://localhost:3000" 라는 주소로 페이지를 띄웠을 때 보이게끔 보내주는(send) 것이다.
- `send()` : 다양한 유형의 응답을 전송하는 메소드입니다. 여기서는 'Hello World!'라는 문자열을 사용하였으므로 response Header 내에 Content-Type을 자동으로 'text/html'로 설정합니다.
- `listen()` : 서버에 접속하기 위해 필요한 메소드입니다. (5000) 이라고 되어있는 것은 포트를 5000번으로 지정해주었다는 뜻입니다. listen 메소드는 지정된 호스트 및 포트에서 연결을 바인딩하고 수신합니다.
    - **클라이언트와 서버를 연결할 때 필요한 포트 정의 및 실행 시 callback 함수**
      를 받는다. 첫 번째 파라미터에는 port 번호가 3000번임을 명시하였고, 두 번째 파라미터인 콜백 함수에서 **서버 구축 성공 시** "listening on *:3000"이라는 로그를 실행하도록 실행했다.

> 💡 더 자세하고 다양한 설명은 `express API` 참조

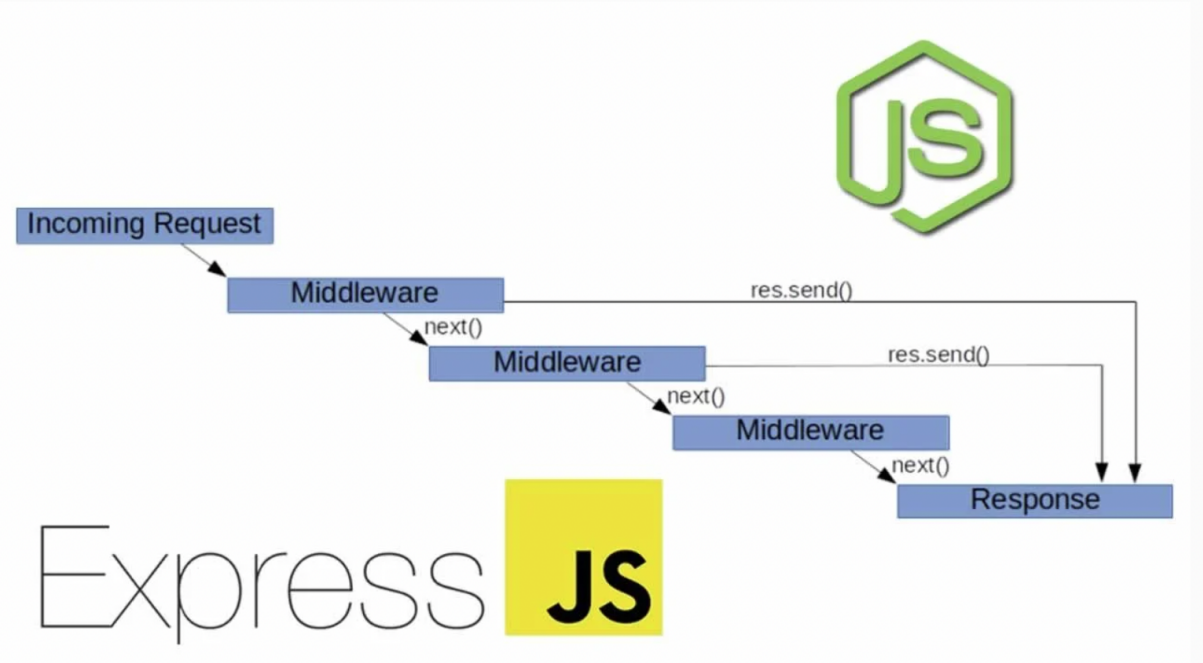
③ 실행

```
npm run start
```

- http://localhost:5000/ 에 접속해 hello world 출력 확인.

Middleware

Middleware 란?



- 요청(`Request`)과 응답(`Response`) 사이 중간(middle)에서 **핸들링(목적에 맞게 처리를 하는)해주는 익스프레스의 핵심 기능(함수)**

> express 공식 문서에 따르면 **"미들웨어 함수는 요청 오브젝트(req), 응답 오브젝트 (res), 그리고 애플리케이션의 요청-응답 주기 중 그 다음의 미들웨어 함수 대한 액세스 권한을 갖는 함수입니다."** 라고 설명한다.

- Express.js는 요청이 들어올 때 그에 따른 응답을 보내주는데, 응답을 보내주기 전에 미들웨어가 지정한 동작을 수행한다.
- Express에서는 함수로 미들웨어를 구현이 가능하다.
- Express에서 미들웨어를 사용하는 과정은 미들웨어 함수를 호출하여 사용한다.
- express는 미들웨어의 집합이라 해도 과언이 아니다.

- express를 사용하는 것이 미들웨어를 활용한다는 뜻과 같다.

Middleware 종류

1. 애플리케이션 레벨 미들웨어(Application Level Middleware)
    - `app.use()`나 `app.get`, `app.post`, `app.put` 과 같이 HTTP 메서드를 사용하여 app의 인스턴스에 결합시킨다.
    - `app.use()` 및 `app.METHOD()` 함수를 이용해 애플리케이션 미들웨어를 앱 오브젝트의 인스턴스에 바인드한다. 이때 `METHOD`는 미들웨어 함수가 처리하는 요청(GET, PUT 또는 POST 등)의 소문자로 된 HTTP 메소드입니다.
    - 예제
        - 예제1

```
var app = express();

app.use(function (req, res, next) {
  console.log('Time:', Date.now());
  next();
});
```

            - 이 함수는 앱이 요청을 수신할 때마다 실행
        - 예제2

```
app.use('/user/:id', function (req, res, next) {
  console.log('Request Type:', req.method);
  next();
});
```

            - `/user/:id` 경로에 마운트되는 미들웨어 함수가 표시
            - 이 함수는 `/user/:id` 경로에 대한 모든 유형의 HTTP 요청에 대해 실행
        - 예제3

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

app.get('/', function(request, response, next) {
  console.log('first middleware');
    next();
})
//first middleware
```

```
//미들웨어의 args로 Request객체, Response객체, next 값이 오는데,
//next는 다음 미들웨어 함수를 실행시킬 수 있다.
//next()가 실행되면 다음 미들웨어를 실행시킨다.

app.use(function(request, response, next) {
  console.log('secone middleware');
    next();
}, function(request, response, next) {
  console.log('third middleware');
    response.end("Hello World");
})
//second middleware
//third middleware
```

2. 라우터 레벨 미들웨어(Router Level Middleware)
   - 라우터 미들웨어는 `express.Router()` 인스턴스에 바인드되는 방식으로 동작한다.
   - `express.Router()` 인스턴스에 바인드된다는 점을 제외하면 어플리케이션 레벨 미들웨어와 동일하게 작동한다.
   - **Router 객체를 이용**해 `router.use()` 나 `router.get()` `.post()` `.put()` ...등 **HTTP Method 함수를 사용**하여 라우터 레벨 미들웨어를 로드할 수 있다.
   - 또한 Router 객체는 그 자체로 미들웨어이다. app.use()나 router.route()의 arg로 사용될 수 있다.

```
const router = express.Router()
```

   - 위처럼 Router 객체를 반드시 생성한 뒤에 app.use()같은 메서드를 이용해 마운트 시켜야 사용할 수 있다.
   - 라우터를 사용하는 이유는 특정 Root URL을 기점으로 기능이나 로직 별 라우팅을 나누어 관리할 수 있다는 장점이 있기 때문이다.
   - 예제
     - 예제1

```
var express = require('express')
var app = express()
var router = express.Router()

//미들웨어 함수에 경로를 지정하지 않았으므로, 모든 라우터 요청에 대해 동작한다. (현재 시간이 콘솔에 찍힌다)
router.use(function (req, res, next) {
  console.log('Time:', Date.now())
  next()
```

```
})

//'/user/10'이라는 URL로 GET/POST 등 요청이 오면
//Request URL : /user/10
//Request Type : GET(혹은 POST)
//와 같이 콘솔에 찍힌다.
router.use('/user/:id', function (req, res, next) {
  console.log('Request URL:', req.originalUrl)
  next()
}, function (req, res, next) {
  console.log('Request Type:', req.method)
  next()
})

//'/user/:id' URL에 대해 GET요청이 오면, :id의 값에 따라 분기가 나뉜다.
//0일 경우 : 밑의 router.get()으로 넘어간다(skip) ==> 'special'을 렌더링 한다.
//0이 아닐 경우 ==> regular를 렌더링 한다.
router.get('/user/:id', function (req, res, next) {
  // if the user ID is 0, skip to the next router
  if (req.params.id === '0') next('route')
  // otherwise pass control to the next middleware function in this stack
  else next()
}, function (req, res, next) {
  // render a regular page
  res.render('regular')
})

// handler for the /user/:id path, which renders a special page
router.get('/user/:id', function (req, res, next) {
  console.log(req.params.id)
  res.render('special')
})

// mount the router on the app
app.use('/', router)
```

3. 오류 처리 미들웨어(Error Handling Middleware)
   - 에러 핸들링 미들웨어는 **항상 4개의 인자**를 갖는다.
   - 에러 핸들링 미들웨어로서 확인시켜주기 위해 4개의 인자를 반드시 제공해야 한다.
   - next가 필요 없더라도(사용되지 않더라도), 이 특색을 유지하기 위해 명시해주어야 한다. 그렇지 않으면 next 객체가 일반 미들웨어로 해석되어 에러 핸들링에 실패할 것이다.
   - 예제

```
app.use(function (err, req, res, next) {
  console.error(err.stack)
  res.status(500).send('Something broke!')
})
```

(1) err : 상위 미들웨어에서 next() 통해 넘긴 err를 받을 인자

(2) request

(3) response

(4) next

4. 기본 제공 미들웨어(Built-in Middleware)
   - Built-in Middleware는 말 그대로 내장된 미들웨어이다.
   - Java의 lang.util.List 같은 내장된 기능
   - Express 4.x 버전을 시작하며, Express는 더 이상 Connect에 의존하지 않는다.
   - 이전 Express에 포함되었던 미들웨어 함수들은 이제 모듈들로 분리되었다.
   - Express는 다음과 같은 미들웨어 함수들이 내장되어 있다.
     - Express.static : HTML 파일, Images 같이 정적인 도움을 준다.
     - Express.json : JSON을 담은 요청을 파싱할 수 있다.
     - Express.urlencoded : URL인코딩을 해준다.
   - 사용법

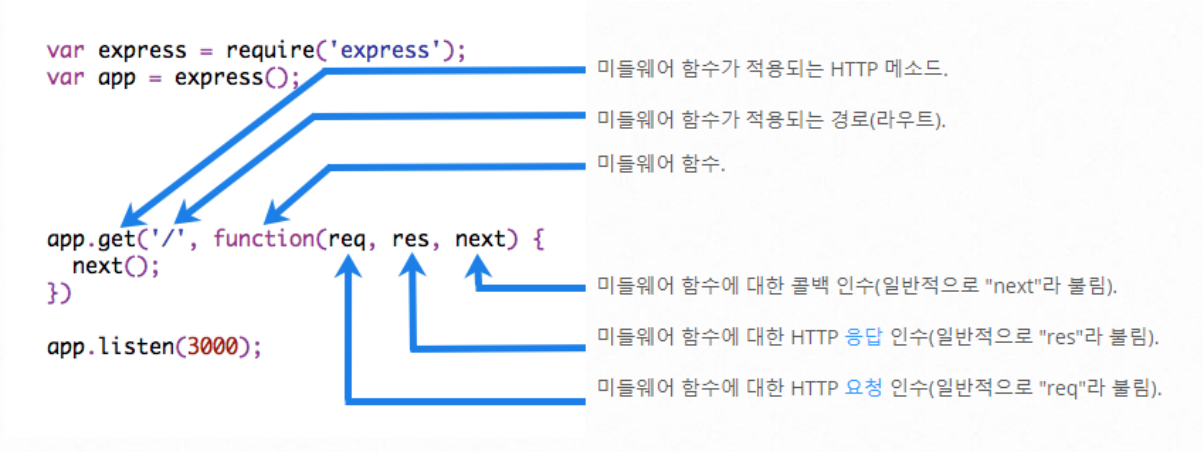
```
//public 디렉토리 안에서 static 파일을 찾겠다는 뜻
app.use(express.static('public'))
```

   - 예제

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

app.get('/', (request, response) => {

  const fileList = request.list;
  console.log(fileList);

  const title = 'Welcome';
  const description = `Hello Node.js!

                      `;
  const list = template.list(fileList);
  const html = template.html(title, list, description,
    `<a href="/create">Create</a>
    <img src="/images/11.png"> //추가된 img태그
    `);

  response.send(html);
})
```

5. 써드파티 미들웨어(Third-party Middleware)
    - 다른 사람들이 제작한 미들웨어
    - 예로 `body-parser`나 `compression`, `cookie-parser` 등 수많은 서드파티 미들웨어가 존재하고, 우리는 그것들을 다양하게 활용함으로써 좀더 간편하고, 완성도가 높은 어플리케이션을 개발할 수 있는 것이다.

Middleware 사용



미들웨어의 구조

- Middleware 함수 구현

```
//미들웨어 함수 구현
var functionName = function (req, res, next) {
//do something
next();
};
//미들웨어 실행
app.use('/', functionName);
```

```
//()=> arrow function을 사용하여 구현한 미들웨어
let functionName = (request, response, next) => {
//do something
next();
}
//미들웨어 실행
app.use('/', functionName)
```

- `app.method`

    - `app.method`를 사용해 엔드포인트 별 응답 처리
    - HTTP 요청 메소드
        - CONNECT.
        - DELETE.
        - GET.
        - HEAD.
        - OPTIONS.
        - PATCH.
        - POST.
        - PUT.
    - 다만 특정 method 요청이 있을 경우만 작동된다는게 아래 `app.use()`와 다른 점
    - **app.get('/', handler)**는 get 요청이며 루트 path인 경우 미들웨어가 실행된다.
- `app.use()`
    - 미들웨어를 전역 처리 스택에 추가한다.
    - 모든 요청이 들어올 때마다 미들웨어가 실행되어진다.
- Middleware 사용방법

```
app.use((req, res, next)=> {
//do something
next();
};
```

    - 미들웨어 모듈 사용
        - 미리 만들어진 모듈을 `npm install`로 설치해 불러서 적용시켜줄 수 있다.
        - 공식문서: express 미들웨어 모듈

```
//미들웨어 모듈 불러서 사용
const ex = require('미들웨어');

app.use(ex);
```

    - 직접 미들웨어 정의해서 사용

- 내가 필요한 로직을 작성해 미들웨어로 적용해 줄 수 있다.

```
//사용자가 미들웨어 생성
app.use((req, res) => {
  console.log('get request!!');
})
```

URL 파라미터 사용하기

- **Route parameters**
  - ex) GET /artists/1, GET /artists/1/company/entertainment

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

router.get('/artists/:id', function (req, res) {
  console.log("id는 " + req.params.id + " 입니다")
  res.send("id : " + req.params.id)
});

// 여러개도 가능
router.get('/artists/:id/company/:company', function (req, res) {
  res.send("id : " + req.params.id + " 회사 : " + req.params.company)
});
```

- **Query string**
  - ex) GET /artists?name=hello

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

router.get('/artists', function (req, res) {
  console.log("이름은 " + req.query.name + " 입니다")
  res.send("name : " + req.query.name)
});
```