# Express – 1

안녕하세요. Express강의를 맡게 된 HokeyDokey입니다. 자스민에서 처음으로 서버라는 방대하고 중요한 개념을 설명하게 되어 나름의 뿌듯함과 중압감이 동시에 밀려옵니다. 그렇기에 이번 강의는 저에게 허락된 최대한의 긴 주차로 강의를 설정하여 진행하여 약 4주간의 강의를 기획하게 되었습니다. 제가 강의하게 될 주차 별 세부계획은 다음과 같습니다.

1일차

1. Node.js 소개 및 동작 메커니즘
2. Node.js와 http모듈만 활용하여 서버 만들기
3. URL과 Rest API 소개
4. Router와 middleware란
5. Express 설치
6. Express를 활용하여 html 및 CSS, static문서 전송

2일차

1. http request message 설명
2. client to server data transfer 구성하기(fetch 함수 및 json 설명)
3. my sql설치 및 연결
4. 전달 받은 data 저장 하기
5. Server to client data transfer 구성하기 (글 가져오기)

3일차

1. Pug template란
2. Join/login 구현하기
3. next()
4. cookie로 회원만 data에 접근하도록 하기.

4일차

1. data 검색 미들웨어 작성
2. data 수정 미들웨어 작성
3. data 삭제 미들웨어 작성

시간이 부족하게 되는 경우 제가 미리 작성한 문서를 깃에 자스민 저장소에 업로드 하도록 하겠습니다.

마지막으로 여러분 중에는 서버를 이미 기획하고 구현까지 해본 인원도 있지만 이번 기회를 통해 서버를 처음 접하시는 분들도 꽤나 있는 것으로 생각됩니다. 그렇기에 이번 강의는 어느 정도의 수준을 위해 Node.js로 서버의 설계 및 구현에 필요한 필수 개념들만 설명을 하고 주로 서버와 클라이언트의 상호작용에 대한 실습을 하며 진행하도록 하겠습니다.

## Node.js

먼저 Node.js는 서버가 아니라 자바스크립트 파일을 실행케 하는 런타임 환경입니다. 기존 자바스크립트는 브라우저에서만 작동하는 언어로써 DOM을 변경하는 언어로써 웹에서 간단한 연산을 돕기 위한 언어에 불과하였습니다. 그러나 구글 크롬의 V8엔진을 사용한 Node.js가 2009년에 발표되면서 그저 DOM을 변경시키는데 사용되던 언어가 http 서버 구성, js lint, bundler 등과 같은 역할을 할 수 있게 되어 각광 받게 되었습니다.

먼저 Node.js의 메커니즘을 살펴보면 Node.js 는 event driven, Non-blocking I/O , single Thread event loop의 처리 방식을 가지고 있습니다.

이벤트 기반(Event drive)이란 클릭과 네트워크 요청과 같은 이벤트가 발생할 때 미리 지정해둔 작업을 수행하는 방식을 의미합니다. 즉 이벤트 리스너에 콜백 함수를 등록한다고 표현합니다. 예를 들어 우리가 브라우저에서 버튼을 클릭하게 되면 경고 창이 뜨게 클릭 이벤트를 설정하였을 때 브라우저는 이 이벤트가 종료가 되면 해당 문서를 닫고 프로세스를 종료시키는 것이 아니라 다음 이벤트가 일어날 때까지 문서를 유지하며 다음 이벤트가 발생할 때 까지 대기 합니다. 노드도 이벤트 기반으로 동작하므로 이벤트가 발생하면 이벤트 리스너에 등록한 콜백 함수를 실행 합니다. 발생한 이벤트가 없거나 발생했던 이벤트를 다 처리하면 노드는 다음 이벤트가 발생할 때 까지 대기합니다.

다음으로 Non-blocking I/O를 설명하기 전에 예제를 먼저 보고 설명하도록 하겠습니다.

function run(){

console.log("3초 후 실행")

}

console.log("시작");

setTimeout(run,3000);

console.log("종료");

위의 코드 흐름은 시작 -> 종료 -> 3초 후 실행입니다.

Node내부에서 함수가 실행되게 되면 Node 내부에서는 이벤트 루프, 태스크 큐, 백그라운드의 상호작용으로 함수의 실행 시기가 결정되게 됩니다.

호출 스택 : 실행 컨텍스트가 변경되며, 호출 된 함수들을 관리하는 공간.

이벤트 루프 : 이벤트 발생 시 호출할 콜백 함수들을 관리하고, 호출된 콜백함수의 실행 순서를 결정하는 역할을 담당.

태스크 큐 : 이벤트 발생 후 호출되어야 할 콜백 함수들이 기다리는 공간 . 이벤트 루프가 정한 순서대로 줄을 서있으므로 콜백 큐라고도 부른다.

백그라운드 : 타이머나 I/O작업 콜백 또는 이벤트 리스너들이 대기하는 곳.

Settimeout함수가 실행되면 콜백 run은 백그라운드에 보내지게 됩니다. 그리고 setTimeout함수는 종료가 되어 호출 스택에서 제거 됩니다. 백그라운드에서 3초 후 태스크 큐로 보내지게 됩니다.

이후 호출 스택이 비워지면 run함수는 이벤트 루프에 의하여 호출스택으로 올려져서 실행되게 됩니다.

자 여기서 중요한 것은 자바스크립트는 함수들을 비동기 식으로 처리하는 것 입니다. 즉 시작 -> 3초후 실행 -> 종료 순으로 처리하는 것이 아니라, 시작 -> 종료 -> 3초 후 실행 순으로 말이죠. Non-blocking i/o는 결국 I/O를 함에 있어 I/O의 return 결과를 기다리다 다음 컨텍스트로 이동하는 것이 아니라, I/O처리를 요청하고 콜백이 백그라운드에 대기하다 해당 process에서 return 결과가 오면 task queue로 이동하고 호출 스택이 비어있을 때까지 대기하다 콜백을 실행합니다. 이런 비동기식 흐름은 함수들의 waiting time을 줄여주는 것으로 blocking i/o방식에 비해 더 짧은 시간 동안 처리할 수 있음을 알 수 있습니다. 하지만 node.js는 싱글 쓰레드라는 한계 때문에 모든 코드가 이 방식으로 시간적 이득을 볼 수 있는 것은 아닙니다.

자바스크립트에서 non-blocking 방식이 중요한 이유는 바로 node.js가 single thread process이기 때문입니다. Thread에 대해서 깊이 논할 필요는 없고 thread는 쉽게 말해 process가 컴퓨팅 작업을 처리할 수 있게끔 처리하는 일손이라고 생각하면 됩니다. 그렇기 때문에 node.js가 blocking i/o로 처리를 하게 된다면 일을 순서대로 처리할 수 있으나, 코드에 따라 대단히 느린 결과물을 보여 줄 수 있습니다. 그렇다면 당연히 점원 수를 늘리면 좋지 않겠냐 라는 생각이 떠올릴 수가 있습니다. 그렇다면 node.js 애초에 multi Thread방식이면 여러 요청에 대해 더 빠르게 응답할 수 있지 않겠느냐 라는 생각이 듭니다. 맞습니다. Multi Thread방식이라면 여러 요청에 대해 더 빠르게 대처 할 수 있습니다. 그러나 Thread는 컴퓨터에서 희소하고 귀중한 자원이기 때문에 만약 요청이 오지 않게 된다면 희소하고 귀중한 자원이 아무것도 안하고 대기만 하는 일이 발생합니다. 그렇다면 요청의 개수에 따라 쓰레드를 증가시키고 감소 시키면 되지 않느냐라는 물음도 생깁니다. 여기서 발생하는 문제는 새로운 쓰레드를 발생시키는데 비용이 비싸다는 것입니다. 즉 요청 한개 당 응답시간에 쓰레드 증가 시간이 추가적으로 발생하게 됩니다. 마지막으로 여러 쓰레드를 관리하는 것은 매우 어렵습니다. 그렇기에 node.js는 서버 프로그래밍에 익숙하지 않은 사람도 쉽게 입문할 수 있습니다.

이러한 non-blocking i/o, single thread 방식으로 인하여 node.js는 i/o작업이 많은 작업에 적합합니다. 이와 반대로 node는 cpu연산이 많이 필요한 작업에 적합하지 않습니다. 이유는 single thread 특성 상 컴퓨터가 어느 연산을 하고 있을 경우 다른 연산 작업을 할 수 없기 때문입니다.

## Node.js 설치 간단한 http서버 만들기

Node.js설치가 안되신 분들은 공식 홈페이지를 통하여 최신버전을 설치 후 터미널 혹은 CMD창에서 node -v , npm -v 로 설치를 확인해주시기 바랍니다.

이후 폴더와 index.js를 만들어 주시기를 바랍니다. 저희가 만들 것은 저희 로컬에서 브라우저를 통해 저희 서버로 접속 하였을 경우 상대방에게 hello world의 메세지를 전달하는 서버를 만들 것 입니다.

Node.js에는 암호화 모듈, OS 모듈, Console 모듈 등 몇 가지의 기본 내장 모듈이 존재하는데 그중 http모듈을 사용하여 서버를 구성 해보도록 하겠습니다.

const http = require('http'); *// 노드 모듈을 가져온다*

const hostname = '127.0.0.1'; *// 사용할 서버 호스트네임*

const port = 3000; *// 사용할 서버 포트*

*// 서버를 만든다*

const server = http.createServer((req, res) => { *// 요청이 오면 실행되는 콜백 함수*

console.log("누군가 접속하였음");

res.statusCode = 200; *// 응답 상태값 설정*

res.setHeader('Content-Type', 'text/plain'); *// 응답 헤더 중 Content-Type 설정*

res.end('Hello, World!\n'); *// 응답 데이터 전송*

});

*// 서버를 요청 대기 상태로 만든다*

server.listen(port, hostname, () => { *// 요청 대기가 완료되면 실행되는 콜백 함수*

*// 터미널에 로그를 기록한다*

console.log(`Server running at http://${hostname}:${port}/`);

});

이후 브라우저를 작동시켜 127.0.0.1:3000을 입력해보면 저희가 작성한 응답 메시지의 내용이 화면에 표시되는 것을 볼 수 있습니다. 서버의 로그에는 누군가 접속하였다는 메세지가 표시됩니다.

터미널을 사용하여 curl 127.0.0.1:3000을 작성하게 되어도 동일하고 -v옵션을 주어 http request response 메시지의 내용을 확인 할 수도 있습니다.

## URL과 RESTful API

이번 강의는 서버를 구성하며 node.js부터 express까지 간단한 예제를 만들어 보는 것이 목적이기에 네트워크에 관한 전반적인 이론을 공부할 이유는 없습니다. 그렇지만 간단하게 서버를 구성하기 위해 필수적으로 등장해야 하는 개념들을 두고 커다란 개념인 서버를 설명하기란 무리가 존재합니다. 그렇기에 이번에는 간단하게 URL과 RESTful api의 기본적인 구성과 개념을 설명하고 넘어가고자 합니다.

URL은 User Resource Locator의 약자로 네트워크 상에서 자원이 어디 있는지 알려주기 위한 규약입니다. 즉 다른 컴퓨터에 접속하여 특정 파일 혹은 정보를 얻고자 위치와 통신 방법을 적어놓은 string입니다. 제 블로그 주소를 보며 URL이 어떻게 구성되어 있는지 확인 해봅시다.

url은 프로토콜, IP, port , path로 이루어져 있습니다. 그 중 IP, port는 사용자의 편의를 위해 string으로 변환하여 domain으로 쉽게 사용할 수 있습니다.

Protocol은 요청하고자 하는 컴퓨터와의 대화 방법을 정의합니다. 여러 protocol이 존재하지만 우리가 주로 사용하는 protocol은 http와 https입니다. http는 hyper text transfer protocol의 약자로 우리가 주로 웹 문서를 얻기 위해 사용하는 방법입니다. Domain은 우리가 접속하고자 하는 특정 컴퓨터를 지칭합니다. 그리고 path인데 이 곳은 정보 혹은 파일이 저장되어 있는 위치를 표시합니다. 디렉토리 구조와 비슷하게 이루어진 것이 특징입니다. 조금 더 domain을 자세히 살펴보면 domain은 sub 도메인과 host로 이루어져 있는데, host는 특정 그룹 혹은 컴퓨터를 지칭하는 ip와 특정 process를 칭하는 port의 조합입니다. 그렇기에 sub domain은 없어도 상관이 없고 subdomain이 여러 있는 café.naver 같은 조합은 naver.com과 ip가 다릅니다.

우리가 눈 여겨 보아야 할 부분은 path부분 입니다. 이 path로 인하여 우리는 자원의 위치를 특정함으로써 정보를 얻을 수 있고, 정보를 보낼 수 도 있습니다. 그렇지만 이 path가 엉망이게 되어버리면 사용자는 url을 통하여 내가 원하는 action의 결과를 측정하기 어려워집니다. 예를 들어 다음과 같이 path가 이루어졌다면 어느 분류에 있는지 알기 어려워 집니다. 그러나 아래와 같이 path를 자원의 분류에 대하여 세분화 한다면 URL을 가지고 어떤 정보를 얻을 수 있는지 파악이 가능합니다.

이렇게 URL의 path를 설계하고 서버의 구조를 만드는 방법으로 RESTful API설계 방법이 있습니다. 이번 강의에서는 RESTful API를 깊게 설명하는 것이 아니라 용어와 rest api의 소개를 하도록 하겠습니다. Rest api는 쉽게 말하여 자원 , 행위, 표현으로 구성되어 URI는 정보의 자원을 표현해야 하며, 자원에 대한 행위는 HTTP method (get delete put delete)로 표현 하도록 구성하면 됩니다.

자원을 요청하는 형태의 방법을 표현하는 method는 다음과 같이 많습니다. 또한 서버는 한개의 method에 대하여 한개의 처리 방법을 구성할 수 있는데, 한개의 메서드에서 모든 수정 삭제 삽입 검색처리를 개별적으로 구성하는 것은 비효율 적이기 때문에 자원의 상태로 표현 할 수 있는 4개의 메서드를 통하여 URI를 구성하는 것이 효율적이고, URI를 잘 설명 할 수 있는 방법입니다.

4개의 메서드는 GET, POST,PUT,DELETE메서드로 이루어져 있습니다. 각 메서드는 다음과 같은 자원의 상태를 표현합니다. GET은 자원 검색, POST는 자원의 추가, PUT은 자원의 수정, DELETE는 자원의 삭제를 의미합니다. 그래서 URI를 구성 할 때는 명사형으로 자원을 표현해야 하며, 자원의 상태는 메서드로만 표현해야 합니다.

## Router와 middleware

라우터는 네트워크 기기의 이름이기도 한데, 클라이언트의 요청을 요청하고자 하는 컴퓨터를 찾아 경로를 지정하고 보내는 물류 허브와 같은 역할의 기기입니다. 소프트웨어 서버에서의 라우터란 요청하는 URI 즉 path에 해당하는 처리를 담당하는 코드입니다. 한개의 요청은 서버의 router로 연결되며 서버에서 응답을 보내는 것으로 한개의 요청을 종료할 수 있습니다.

미들웨어란 라우터에서 인증, 글 추가, 로그 기록과 같은 한개의 요청에 대하여 처리해야하는 모든 작업들을 말합니다. 그래서 라우터는 처리에 따라 여러 미들웨어로 이루어질 수 있고, 한개의 미들웨어로만 이루어질 수 있습니다. 또한 미들웨어는 router없이 요청에 대한 필수적인 처리 로직들을 수행 하다 해당 요청에 대한 라우터에서 응답을 보내는 것으로 어떤 미들웨어가 작동하지 않을 수 있습니다.

## App화

Router와 middleware를 직접 추가 하기 전에 현재 저희 서버의 구조를 조금 변경 하도록 하겠습니다. 이전에는 http모듈을 직접 불러와서 index.js에서 직접 서버에 필요한 것들을 구성하였습니다. 그렇기 때문에 http서버를 직접 구성하기 전까지의 작업을 추가하기 어렵고 http 라이브러리를 우리 마음대로 유연하게 사용하지 못한다는 단점이 존재합니다. 그렇기에 listen까지 내부적으로 담고 있는 객체로 http모듈을 우리 서버에서 커스텀 모듈화하도록 하겠습니다.

App.js를 만들고 해당 문서에서 http를 호출합니다. 그리고 Application 함수를 만든다음 객체를 리턴 하도록 합니다.

const http = require('http')

const Application = () => {

const \_server = http.createServer((req, res) => {

res.statusCode = 200

res.setHeader('Content-Type', 'text/plain')

res.end('Hello World\n')

});

const listen = (port = 3000, hostname = '127.0.0.1', fn) => {

//생소한 부분이 존재하는데, 파라메터에 변수를 추가하게 된다면 함수 호출 당시 전달인자가 없을 경우 자동으로 파라메터에서 정의한 값들이 적용된다.

\_server.listen(port, hostname, fn)

}

return {

\_server,

listen

}

}

module.exports = Application

이후 index.js에서 우리가 만든 application모듈을 불러오고 호출 하여 app변수에 담고 listen 호출해봅니다.

const app = require('./app')();

const hostname = '127.0.0.1';

const port = 3000;

app.listen(port, hostname, () => {

console.log(`Server running at http://${hostname}:${port}/`);

});

터미널에서 해당 index.js를 실행해보니 기존과 동일하게 서버가 작동하는 모습을 볼 수 있습니다.

## 미들웨어 추가 구현

다시 미들웨어를 구현하기 위해서 미들웨어는 한개의 요청과 한개의 응답에 사이에 있는 모든 과정을 의미합니다. 그렇기에 우리가 createServer를 활용했던 지점에서 콜백인자를 살펴보면 req와 res를 온전히 다음 미들웨어로 전달하고 다음 미들웨어를 실행시켜야 할 필요가 있습니다.

Middleware.js라는 파일을 생성합니다.

const Middleware = () => {

const \_middlewares = [];

*//미들웨어 배열*

const add = fn => {

\_middlewares.push(fn)

*//미들웨어 추가*

}

const run = (req, res) => {

const \_req = req;

const \_res = res;

*//크리티컬 섹션 문제로 인한 req,res 클로저 형성*

const \_run = (i, err) => {

if (i < 0 || i >= \_middlewares.length) return;

const nextMw = \_middlewares[i] *//다음 실행할 미들웨어*

const next = err => \_run(i + 1, err) *//현재 실행 중인 미들웨어가 종료되고 실행할 미들웨어*

if (err) {

*//error 미들웨어는 전달인자가 총 4개*

const isNextErrorMw = nextMw.length === 4

return isNextErrorMw ?

nextMw(err, \_req, \_res, next) :

\_run(i + 1, err)

}else{

*//err이 없다면 다음 미들웨어 실행*

nextMw(\_req, \_res, next);

}

}

\_run(0);

}

return {

run,

add,

\_middlewares,

}

}

module.exports = Middleware;

이제 app.js에서 편리하게 끔 미들웨어를 추가할 수 있도록 변경 하는 코드를 수정해보겠습니다.

const app = require('./app')();

const hostname = '127.0.0.1';

const port = 3000;

app.use((req, res,next) => {

console.log("미들웨어 1");

next();

})

app.use((req, res) => {

console.log("누군가 접속하였습니다.");

res.statusCode = 200

res.setHeader('Content-Type', 'text/plain')

res.end('Hello World\n')

})

app.listen(port, hostname, () => {

console.log(`Server running at http://${hostname}:${port}/`);

});

미들웨어를 추가 해도 동일하게 코드가 진행 되는 것을 확인 할 수 있습니다 .

## 라우터 구현

App.use()는 현재 한개의 전달 인자로 함수만 전달 받고 있습니다. 만약 app.use가 App.use(‘/service’,fn)의 형태로 만들어져 첫번째 전달인자가 path인 경우 특정 페이지를 전달하거나, 다른 응답을 전달 하도록 만들어지면 어떨까요 ?

App.js로 돌아가서 use함수를 조금 변경해보겠습니다.

const use = (path, fn) => {

if (typeof path === 'string' && typeof fn === 'function') {

fn.\_path = path;

*//만약 path가 존재하면 fn의 프로퍼티로 path를 추가*

} else if (typeof path == 'function') {

*//첫번째 전달인자가 fn이라면 fn파라메터를 path 전달인자로 들어온 함수로 할당*

fn = path;

} else {

throw Error('Usage: use(path, fn) or use(fn)');

}

\_middleware.add(fn);

}

이제 fn은 path를 프로퍼티로 가지고 있습니다. 다시 middleware.js로 돌아가 \_run부분을 조금 고쳐줍시다.

const run = (req, res) => {

const \_req = req;

const \_res = res;

*//크리티컬 섹션 문제로 인한 req,res 클로저 형성*

const \_run = (i, err) => {

if (i < 0 || i >= \_middlewares.length) return;

const nextMw = \_middlewares[i] *//다음 실행할 미들웨어*

const next = err => \_run(i + 1, err) *//현재 실행 중인 미들웨어가 종료되고 실행할 미들웨어*

if (err) {

*//error 미들웨어는 전달인자가 총 4개*

const isNextErrorMw = nextMw.length === 4

return isNextErrorMw ?

nextMw(err, \_req, \_res, next) :

\_run(i + 1, err)

}

if(nextMw.\_path){

const pathMatched = \_req.url === nextMw.\_path;

//만약 fn에 path가 존재하지 않다면 nextMw.path는 실행되지 않을 것임

console.log(\_req.url)

return pathMatched ? nextMw(\_req, \_res, next) : \_run(i + 1)

}

nextMw(\_req, \_res, next);

}

\_run(0);

}

그리고 이제 다시 index.js로 돌아와 router두개를 추가 해봅시다.

app.use('/test',(req, res) => {

console.log("testrouet");

res.statusCode = 200

res.setHeader('Content-Type', 'text/plain')

res.end('test router')

})

app.use('/test/router',(req, res) => {

console.log("두번째 router");

res.statusCode = 200

res.setHeader('Content-Type', 'text/plain')

res.end('test/router success.')

})

이제 브라우저로 돌아가서 확인해보니, 요청 url에 따라 메세지가 변환하는 것을 볼 수 있습니다 .

여기까지 미들웨어, 라우터, createServer, listen의 구조를 살펴보며 express의 기능들을 실제로 만들어 보았습니다. Express는 http 모듈을 이용하여 우리가 직접 구성해야하는 기능들을 대신 만들어 주었기 때문에 훨씬 빠르게 서버를 구성할 수 있게 도와줍니다. 시간 상 저희는 에러페이지는 따로 작성하지 않았지만 express내부에는 각종 에러페이지에 대응하는 페이지가 따로 내장되어 있습니다. 그러니 우리는 편하게 express를 이용하며 각종 express 프레임워크를 고치며 페이지만 생성하면 됩니다.

## Express 설치

Express 설치는 간단하게 sudo npm install express-generator -g를 하게 되면 서버를 구성하는 express에 필요한 파일들을 구성해줍니다. C-R-A와 비슷합니다.. 전역 설치이기 때문에 sudo 명령어를 통하여 설치해야합니다.

자 express를 설치 했다면 간단하게 동일한 폴더에 express를 만들어주시구요. 터미널에서 express폴더로 이동한다음 express . 하여 서버를 구성하기 위한 파일을 구성해보도록 하겠습니다 .

뭐가 와랄라 하며 express내에 이것저것 파일이 설치된 것을 확인 할 수 있는데,

App.js가 저희가 만든 index.js와 동일한 역할을 하는 루트 파일입니다. 이제 npm start 를 해보면 뭘 자꾸 설치하라고 하는데 npm install을 입력하는 것으로 필요한 패키지를 같이 설치하도록 합시다.

이제 npm start를 터미널에 입력해봅시다. 그럼 서버가 구동 될 겁니다. 이제 브라우저에서 localhost:3000을 입력해보면 welcome to express라며 index가 뜨게 될 겁니다.

## Html과 css파일 정적문서 전송

자 public폴더에 stati/main폴더를 만들어주시고, 해당 폴더에 login.html과 login.css를 만들어 줍니다.

login.html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Document</title>

<link rel="stylesheet" href=" http://localhost:3000/main/login.css">

//test에서는 ./login.css

</head>

<body>

<div class="main-container">

<div class="login-box">

<h1 class="login-h1">SNS</h1>

<div class="login-input-container">

<div class="login-input-box">

<input class="input" type="text" placeholder="ID를 입력해주세요.">

</div>

<div class="login-input-box">

<input class="input" type="password" placeholder="Password를 입력해주세요.">

</div>

</div>

<div class="login-tool-container login-input-container">

<button class="login-tool-button login-input-box login-button input">로그인</button>

<button class="login-tool-button login-input-box input">회원가입</button>

</div>

</div>

</div>

</body>

</html>

Login.css

.main-container{

width: 100vw;

height: 100vh;

}

.login-box{

width: 400px;

height: 400px;

position: absolute;

top: 50%;

left: 50%;

transform: translate(-50%,-50%);

border: 1px solid rgb(102, 102, 102);

background-color: aliceblue;

}

.login-h1{

text-align: center;

}

.login-input-container{

margin-top: 15px;

padding : 0px 15px 5px 15px;

}

.input{

display: block;

width: 80%;

height: 30px;

margin : 0 auto;

}

.login-input-box{

margin-top: 15px;

}

.login-tool-button{

height: 50px;

}

.login-tool-container{

width: 370px;

position: absolute;

bottom: 15px;

}

이제 routes의 index.js를 고쳐줍니다.

router.get('/', function(req, res, next) {

*// console.log(\_\_dirname)*

fs.readFile('./public/main/login.html','UTF-8',(err,data)=>{

if(err){

next(err);

return;

}

res.writeHead(200,{'Content-Type':'text/html'});

res.end(data);

})

*// res.render('index', { title: 'Express' });*

});

Node에서 file i/o를 담당하는 모듈은 fs입니다. Fs 를 불러오고 readFile메서드를 이용하여 파일을 불러오고, head에 보낼 데이터가 text/html이라는 것을 명시한 후 응답을 끝내보도록 합시다.

그런데 이상하게도 분명 link 된 url은 ./로 로컬에서 찾는것이 아니라 분명 서버를 통해 얻는데 이게 어떻게 된 일일까요 ? 그것은 바로 express가 static한 파일을 제공해주도록 app.static()이라는 메서드를 지원하기 때문입니다.

App.js로 가보면 중간 쯤에 app.use(app.static(‘\_\_dirname’,’public’))으로 된 부분이 존재합니다. 이 미들웨어는 static한 파일을 제공해주기 위한 미들웨어로

Static이라는 path가 전달되면 ‘./public/${req.url}’라는 패스를 찾아 파일을 찾아 보내줍니다. Static미들웨어가 여러개 있을경우 처음있는 디렉토리에서 먼저 찾고 다음 static미들웨어에서 찾게 됩니다.