# Express - 2

안녕하세요 . Express 강의를 맡게 된 hokeydokey입니다. 지난 주에 서버의 기본 구성을 이루는 middle ware와 router를 직접 함수로 만들어보고, express generator를 설치해 static한 파일을 전달하는 것까지 해보았습니다 . 이번에는 서버에서 클라이언트로 데이터를 전달하는 것이 아닌 클라이언트에서 데이터를 전달하고 데이터를 직접 서버에 저장 까지 해보도록 하겠습니다.

## http request message

우리가 누군가에게 특정 행동을 요구하는 것을 요청이라고 합니다. http request message는 http 프로토콜에서 사용되는 요청에 필요한 내용을 담고 있는 메세지 입니다.

http request message는 총 4개의 line으로 이루어져 있습니다. 첫번째 request line은 http request method , 요청 url , version으로 이루어져 있습니다. http method는 종류가 다양하지만 필요할 때를 제외하고 restful api로 URI를 구성하는 것이 일반적이기에 GET, POST, PUT, DELETE 메서드를 주로 사용 합니다.

각 method의 의미를 살짝 집어 보도록 각 method별로 어떤 요청을 하는 건지 맞춰 볼까요 ?

URL은 우리가 알고 있는 host,port 혹은 domain , path로 이루어진 문자열입니다. Version은 1.0 , 2.0이 존재하며, 제가 예전에 3.0 내용을 올린 것을 보시면 아시겠지만 1.0과 2.0은 TCP/IP로 통신하며, 3.0버전은 UDP통신을 지원합니다. 이번 강의는 네트워크에 관한 내용이 아니기 때문에 여기까지 하고 Head line으로 넘어가겠습니다.

Head line은 해당 메세지에 대한 정보를 작성합니다. Body는 어떤 포맷으로 요청을 하는지 , 인코딩은 어떻게 되어 있는지, 현재 어느 프로세스에서 메세지를 보냈는지에 관한 내용들이 작성되어 있습니다. 우리가 지난번 html파일을 전달 했을 때도 contents-*type : text*/html을 작성 한 적 있듯이 메세지의 내용을 정의하기도 합니다.

Body line은 실질적인 메세지의 내용을 담고 있는 부분입니다. 우리가 id와 password같은 내용을 전달하여 해당 데이터가 유효성 검사를 거치고 서버로부터 접속 허가를 줄 수 있도록 하는 데이터를 담을 수 있는 부분입니다. 이 body line은 모든 메서드에서 data를 담을 수 있는 것은 아니고 POST와 PUT메서드만 body에 내용을 담을 수 있습니다.

마지막 blank line은 body와 header라인의 경계를 의미하기에 개행을 통해 바디와 헤더라인의 경계를 나누는 역할을 합니다.

우리가 브라우저를 통신하거나 curl 명령어로 내용을 가져오는 행위는 get에 해당합니다.

## Client to server data transfer

C to S를 하기 위해 간단하게 회원가입 창을 새로 만들도록 하겠습니다. 이번 페이지의 목적은 mpa를 구성하는 것이 목적이기에 spa처럼 만들지 않고 html페이지를 새롭게 만들어 보내도록 하겠습니다.

먼저 login.html에서 회원가입 버튼에 a태그를 입혀주는 것으로 회원가입 페이지를 get해봅니다.

<a href="http://localhost:3000/service/join"><button class="login-tool-button login-input-box input">회원가입</button></a>

이제 join.html을 만들어주도록 하겠습니다.

Join.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">

<script src="http://localhost:3000/main/join.js"></script>

</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" id="login-id" placeholder="ID를 입력해주세요.">

</div>

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

<input class="input" type="text" id="login-email" placeholder="이메일을 입력해주세요">

</div>

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

<input class="input" type="password" id="login-pw-1" placeholder="Password를 입력해주세요.">

<input class="input" type="password" id="login-pw-2" placeholder="Password를 재확인 해주세요">

</div>

</div>

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

<button class="login-tool-button login-input-box login-button input" onclick="join()">회원가입</button>

<a href="http://localhost:3000/"><button class="login-tool-button login-input-box input">취소</button></a>

</div>

</div>

</div>

</body>

</html>

그리고 join.js파일을 하나 만들어 간단한 회원가입을 위한 유효성검사를 할 수 있도록 합시다.

function emailValidationCheck(email){

const emailRegexp = /^[0-9a-zA-Z]([-\_\.]?[0-9a-zA-Z])\*@[0-9a-zA-Z]([-\_\.]?[0-9a-zA-Z])\*\.[a-zA-Z]{2,3}$/i;

return emailRegexp.test(email);

}

const join = () => {

const id = document.getElementById("login-id").value;

const email = document.getElementById("login-email").value;

const pw\_1 = document.getElementById("login-pw-1").value;

const pw\_2 = document.getElementById("login-pw-2").value;

function \_join(id,email,password\_1,password\_2){

if(id.length <5){

alert("아이디는 5글자 이상 되어야 합니다.");

return;

}

if(!emailValidationCheck(email)){

alert("이메일을 확인해주세요.");

return;

}

if(!password\_1 || !password\_2){

alert("비밀번호를 입력해주세요.");

return;

}

if(password\_1 !== password\_2){

alert("비밀번호가 일치 하지 않습니다.");

return

}

alert("good");

}

\_join(id,email,pw\_1,pw\_2);

}

여기까지 테스트를 종료했으면 이제 /service/join router를 구성하도록 하겠습니다.

Service.js

var express = require('express');

var router = express.Router();

const fs = require('fs');

*/\** GET home page. *\*/*

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

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

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

if(err){

next(err);

return;

}

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

res.end(data);

})

});

module.exports = router;

그리고 app.js에 해당 라우터를 추가해주도록 하겠습니다.

var serviceRouter = require('./routes/service')

…

app.use('/service',serviceRouter);

그럼 이제 로그인 페이지에서 회원가입 페이지로 작동하며 스크립트 파일도 제대로 작동되는 것을 확인할 수 있습니다 .

## Post method 사용

원래는 fetch api를 사용하려 했으나 모든 리퀘스트를 작성해야하는 불편함이 너무 커서 이미 잘 만들어진 라이브러리를 쓰지 뭐하러 fetch api를 쓸까 생각이 들어서 axios라는 널리 사용되는 통신라이브러리를 사용하도록 하겠습니다.

https://developer.mozilla.org/ko/docs/Web/API/Fetch\_API/Fetch의\_사용법

<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>

인터넷으로 axios를 검색하여 cdn script를 얻어오도록 합시다. 다음 join.html에 입력합니다.

그런다음 \_join()에서 axios를 콘솔로그로 확인해보니 정상적으로 axios가 전역적으로 입력된 것을 볼 수 있습니다. Axios는 get,post,put,delete 의 메서드를 가지고 있습니다. 각 메서드는 http 메서드에 맞는 타입으로 http request method를 구성합니다. 이제 axios를 사용해보도록 하겠습니다.

Axios의 get과 delete메서드는 바디가 없기 때문에 첫번째인자로 요청 메시지를 보낼 url을 문자열 형태로 작성합니다.

Post와 put 메서드는 바디가 존재하기 때문에 2번째 인자로 요청메세지 바디에 담을 data를 구성합니다. 여기서 axios의 장점이 나타나는데, fetch api페이지를 보면 바디에 담은 데이터 타입에 따라 여타 다른 헤더를 구성해야 하는데, axios는 이런 헤더를 body에 담긴 데이터 타입 헤더와 통신에 필요한 각종 헤더를 구성해줍니다. 우리가 사용할 데이터 타입은 json으로 json은 문자열로 객체를 표현하여 사람과 기계가 읽기 용이한 데이터 타입입니다.

Json의 시작은 중괄호로 시작하며 프로퍼티와 값은 모두 큰따옴표로 감싸줍니다. Key와 밸류 사이의 콜론은 큰 따옴표를 적지 않습니다.

{  
    "name": "식빵",  
    "family": "웰시코기",  
    "age": 1,  
    "weight": 2.14  
}

배열 또한 표현이 가능합니다.

"dog": [  
    {"name": "식빵", "family": "웰시코기", "age": 1, "weight": 2.14},  
    {"name": "콩콩", "family": "포메라니안", "age": 3, "weight": 2.5},  
    {"name": "젤리", "family": "푸들", "age": 7, "weight": 3.1}  
]

Fetch api나 xml을 이용한 통신에서는 객체를 스트링으로 변환하는 함수를 사용해서 넣어야 하지만 axios는 객체를 직접 인자로 주었을 때 열거 가능한 데이터들만 property 로 만들어 JSON을 구성하고 head에 데이터 타입을 json으로 변경 해줍니다.

그럼 이제 axios.post를 이용해서 post를 준비해보도록 하겠습니다.

axios.post('http://localhost:3000/service/join',{

id,

email,

password\_1

})

Join.js에 입력을 마쳤다면 service router로 돌아와서 post메서드를 받을 준비를 하도록 하겠습니다.

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

console.log(req.body);

});

이제 회원가입페이지에서 대충 양식을 적고 회원가입버튼을 눌러보도록 하겠습니다. 이제 콘솔창을 확인해보니 정상적으로 데이터가 전송된 것을 확인할 수 있습니다. 이제 잘 받았다는 표시로 OK 텍스트를 작성해서 보내고, html내부에서도 data를 확인해보도록 하겠습니다.

Post service/join

…

res.send("OK")

Join.js 변경

…

axios.post('http://localhost:3000/service/join',{

id,

email,

password\_1

}).then((res)=>{

console.log(res);

})

이제 데이터를 보내고 콘솔창에서 확인해보면 res객체 내부에 data프로퍼티에 서버에서 텍스트가 전송 된 것을 볼 수 있습니다.

## Mysql 설치 및 저장

이제 데이터를 데이터베이스에 저장하기 위해서 관계형 데이터베이스인 mysql을 설치하도록 하겠습니다. 아무래도 이 수업은 서버를 구성 함이 목적이니 어려운 cmd환경이 아닌 gui환경에서 데이터베이스를 사용하도록 하겠습니다. 이 수업에서는 데이터베이스 쿼리와 db를 구성하는 요소들에에 관한 설명은 생략하도록 하니, 이에 관한 설명은 직접 공부하시거나, 박세진 교수님의 데이터베이스 1, 예은씨의 데이터베이스 강의를 수강 하시길 바라겠습니다.

구글에 mysql workbench를 검색하셔서 다운로드를 진행 한 후 설치를 완료해주세요. 다음 mysql server를 검색해서 설치를 완료해주시기 바랍니다.

다음 스키마를 생성한 다음 어카운트 테이블을 구조를 만들어보도록 하겠습니다. 우리가 어느 데이터를 저장하려고 하면 우리가 설정한 스키마의 테이블에 그 자료가 저장되게 됩니다.

Account 테이블 생성

전자기기, 키보드, 계산기, 컴퓨터이(가) 표시된 사진

자동 생성된 설명

## Mysql연결

이제 express를 데이터 베이스를 연결하여 id 중복체크도 하고, 회원가입도 할 수 있도록 하겠습니다 .

Npm install mysql로 mysql 연결에 필요한 라이브러리를 가져옵니다. 그리고 현재 위치에서 db폴더를 만든다음 config.js파일을 만들도록 합시다.

module.exports = (function(){

return {

local : {

host : 'localhost',

port : '3306',

user : 'root',

password : 'gozld123',

database : 'jasmin\_express',

},

real : {

host : '',

port : '',

user : '',

password : '',

database : '',

},

dev : {

host : '',

port : '',

user : '',

password : '',

database : '',

}

}

})();

그런 다음 mysql 연결을 만들도록 하겠습니다.

var mysql = require('mysql');

var config = require('./config')

module.exports = (function(config){

return {

init : function(){

return mysql.createConnection({

host : config.host,

port : config.port,

user : config.user,

password : config.password,

database : config.database

})

},

test\_open : function (){

this.init().connect(function(err){

if(err){

console.log("mysql connection error" + err);

}else{

console.log("mysql is connected successfully");

}

})

}

}

})(config.local)

그리고 이제 서버가 시작하게되면 정상적으로 데이터베이스에 접속 해 있는지 app.listen메서드를 불러와서 서버 시작할 때 test를 진행 하도록 하겠습니다.

const mysqlConnect = require('./db/mysql');

//상단에 추가

app.listen(()=>{

*//시작 할때 데이터 베이스 연결 확인*

mysqlConnect.test\_open();

})

이제 서버를 껏다 켜보면 테스트가 정상적으로 실행되고 해당 설정에서 문제가 없다고 출력되는 것을 볼 수 있습니다 .

이제 해당 /serive/join router에서 다음과 같이 구성해봅니다.

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

mysql.query(`insert into account(id,pw,email) values('${req.body.id}','${req.body.password\_1}','${req.body.email}')`,(err,result)=>{

if(err){

next(err);

return;

}

console.log(result);

res.send("OK");

});

});

이제 회원가입창으로 가서 확인해보면 해당 데이터가 저장된 것을 볼 수 있습니다. 자 이제 중복된 아이디로 회원가입을 하지 못하도록 막아야 할 필요성이 존재합니다. 그렇다면 우리가 할 수 있는 시나리오는 현재 몇가지가 있습니다. 새로운 중복 확인 검사 버튼을 만들어서 버튼을 누를때 마다 중복 검사를 하고, 중복 검사가 진행된 아이디임을 인증 받은 경우 회원가입을 진행하게 하거나 회원가입 버튼을 눌렀을 경우 id검사를 같이 진행하고, 별 탈이 없는 경우 회원가입을 진행하고 만약 id가 존재하는 경우 해당 id를 다시 작성하라는 메세지를 보내 다시 아이디를 만들고 다시 회원가입 버튼을 누르게 하는 것입니다. 웹 접근성을 생각 했을 경우 전자가 좀 더 괜찮을 겁니다. 그러나 이런 경우 어떤 유저가 id를 입력한 후 다른 작업을 하다 다른 사람이 해당 id를 선점해버리는 경우 기존 User는 다시 id를 만들어야 하므로 이 경우 최소 4번의 통신이 이루어지게 됩니다. Front입장에서는 사용자를 위해 id옆에 버튼을 만들어 움직임을 최소한으로 하여 편의성을 고려하겠지만 현재 backend를 만지는 입장에서는 전체적인 통신 횟수를 줄여 서버의 성능을 이끌어 내야하므로 귀찮음을 이렇게 아름답게 포장하여 회원가입 버튼을 눌렀을 때 id를 체크하고 이에 맞는 절차를 밟을 수 있도록 합시다.

회원가입 버튼을 클릭했을 때 발생해야 하는 서버측 시나리오는 id가 존재하는지 확인하고, 해당 id가 존재하지 않았을 때 비로소 그때 데이터베이스에 저장을 해야합니다. 그럼 코드를 한번 같이 만들어 보실까요 ?

mysql.query(`select count() as isValid from account where id = '${req.body.id}'`,(err,result)=>{

if(err){

next(err);

return

}

if(result[0].isValid){

res.send("이미 존재하는 아이디 입니다.");

return;

}

mysql.query(`insert into account(id,pw,email) values('${req.body.id}','${req.body.password\_1}','${req.body.email}')`,(err,result)=>{

if(err){

next(err);

return;

}

console.log(result);

res.send("OK");

});

})

뭐 여차 저차 작동은 잘 됩니다만 코드가 길어지고 mysql에 두번 요청을 하는 것이 매우 마음에 걸립니다. 그렇기에 이 쿼리가 한번에 데이터베이스에서 처리가 되고 우리는 요청이 잘 받아 졌는지만 확인하면 되기 때문에 procedure라는 것을 만들어 이를 처리해보도록 하겠습니다.

CREATE DEFINER=`root`@`localhost` PROCEDURE `join`(in id varchar(45),in pw varchar(45),in email varchar(45))

BEGIN

declare status tinyint(4);

set status = (select count(\*) from account where account.id = id);

if status = 0

then insert into account(id,pw,email) value(id,pw,email);

end if;

select status;

END

이 프로시듀어가 정상적으로 작동되는지 이제 확인해보겠습니다. 만약 id가 존재하는 상황이라면 1이 전달되고 아니라면 0이 전달되는 것을 확인했습니다. 이제 router의 코드를 변경 하겠습니다.

mysql.query(`call jasmin\_express.join('${req.body.id}','${req.body.password\_1}','${req.body.email}')`, (err, result) => {

if (err) {

console.log(err);

next(err);

return;

}

console.log(result);

if(result[0][0].status){

res.status(202).send("id already used");

}else{

res.status(201).send("OK");

}

});

이제 회원가입창에서 콘솔을 띄우고 데이터를 전송해보면 우리의 의도대로 작동하는 것을 알 수 있습니다. 프론트의 코드도 살짝 변경하여 좀 더 다듬어 보겠습니다.

axios.post('http://localhost:3000/service/join',{

id,

email,

password\_1

}).then((res)=>{

if(res.status === 202){

alert("이미 존재하는 id입니다.");

return;

}

alert("회원가입을 축하드립니다.");

})