<공부방법>

1. 결과물을 보고 일단 따라 만들어보려고 시도해본다.
2. 막히는 부분이 생기면 완성된 코드를 본다.
3. 코드를 봐도 알 수 없는 부분들은 강의를 다시 본다.
4. 강의를 봐도 알 수 없는 부분들은 구글에 검색해서 찾는다.
5. 알게 된 것들을 여기 메모해 둔다(그 상황만을 해결하는 방법보다는 일반화된 해결 방법론을 메모하는 것이 좋다). 잘못 메모해둔 부분은 그때그때 고친다.
6. 결과물을 다 따라 만들었다면, 막히는 부분이 없을 때까지 1~5를 반복한다(이때 메모해둔 것들을 참고해도 된다).
7. 나중에 공부가 부족하거나 빠진 부분이 있다고 판단되면, 결과물의 일부분만 따라해본 것은 아닌지 체크해본다. 만약 그렇다면 결과물에 다른 부분의 코드를 쭉 읽어보면서 모르는 부분을 체크하고 찾아본다.
8. 그래도 공부가 부족하면 모든 부분을 따라 만든다(1~6).

<개발환경설정>

* NodeJS 설치 -> 2.3 (npm도 자동설치됨.)
* Express 설치 ->3.2
* Babel 설치 -> 3.5
* Nodemon 설치 -> 3.5
* Morgan 설치 -> 3.7
* Helmet 설치 -> 3.7
* Body-parser 설치 -> 3.7
* Cookie-parser 설치 -> 3.7
* Pug 설치 -> 3.13
* mongoDB 설치 -> 4.0
* mongoose 설치 -> 4.0
* dotenv 설치 -> 4.1
* multer 설치 -> 4.6
* esLint 설치 -> 4.11

※babel – 우리가 쓴 최신 코드를 컴퓨터에서 오래된 코드로 바꿔 줌.(최신 문법을 사용할 때 필요) (‘node 서버가 담긴 자바스크립트 이름’=>‘babel-node 서버가 담긴 자바스크립트 이름’으로 npm start해야 함)

Nodemon – 서버를 이용할 때 우리가 npm start->ctrl-c를 일일이 안해도, 우리가 코드를 변경하면 자동으로 해줌.( ‘babel-node 서버가 담긴 자바스크립트 이름’=>’ nodemon --exec babel-node 서버가 담긴 자바스크립트 이름 --delay 2’으로 npm start해야 함)

Package.json 속 Dependencies와 devDependencies의 의미

Dependencies : 코드를 실행시키는 데 반드시 필요한 속성

devDependencies : 우리가 코딩을 더 편하게 하기 위해서 필요한 속성

morgan : logging에 도움을 주는 middleware. //logging : 어디서 무슨 일이 일어났는지를 기록하는 것.

Helmet :보안용 middleware.

Body-parser : form을 가지고 있는 req obj에 접근하기 위한 middleware.

Cookie-parser : session을 다루기 위해 cookie에 유저 정보를 넣기 위한 middleware. (아직은 잘 모르겠음)

<github에 업데이트한 내용 올리는 방법>(2.3에 자세히 나와있음.)

* + - 1. 업데이트 내용을 git에 기록시킨다.
* (git add .)
  + - 1. 업데이트한 내용을 저장한다.
* (git commit –m “저장할 제목 이름”) ex. git commit -m “first draft"
  + - 1. push해서 github에 업데이트한다.
* (git push origin “branch이름”) ex. Git push origin master

<서버의 개요>

POST request => 정보를 전달

GET request => URL에 접근

- URL을 쓴다(youtube.com) => 브라우저가 GET method를 실행한다 => 브라우저가 URL의 페이지를 읽어온다

- 웹사이트에 Login 한다 => 브라우저가 POST method 실행한다. => 브라우저가 웹사이트에 정보를 전달한다.

- req obj : 누군가 정보를 얻고자 페이지를 요청했을때, 어떤 종류의 데이터가 페이지로 전송됐는지 확인할때 request object를 이용한다. 예를 들어 이 URL에 POST 방식으로 아이디와 패스워드를 전송한다면, 이렇게 app.post로 아이디랑 패스워드를 보낸다면 서버에는 request object로 정보를 얻을 수 있다.

★서버의 기본적 작동 방식 :

1. 서버가 생성되고 get request를 기다린다.

2. get request를 받으면, 브라우저의 주소에 따라 route가 생성된다.

3. 함수를 통해 특정 route에 응답한다.

<require>

Import와 비슷한 개념임. 특정 폴더명을 가진 폴더를 가져와 변수에 저장한다. 만약 폴더를 찾지 못했을 경우에는 node.js의 경우 node\_modules에서 찾아옴.

문법 : const 변수명 = require('폴더명');

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

★Babel이 있으면 import라고 쓰면 된다.

문법 : imort 변수명 from “폴더명”;

Ex) import express from "express";

<import와 export>

Export, morgan, bodyParser 같은 import폴더들은 node\_modules 내부에 있음. 하지만 그 전에 우리가 만든 파일에도 있는지를 먼저 확인한다.

만약 우리가 만들 파일에서 가져오게 하고 싶을 때는, 앞에 ./ 따위를 붙인다.

※알파벳 순으로 import하는 게 좋은 습관임.

문법 : import 외부 파일 from "./외부 파일";

Ex) import app from "./app";

이때 default로 export한 게 아닐 때는 중괄호가 필요하다.

문법 : import {외부 파일 내 객체} from "./외부 파일";

Ex) import { userRouter } from "./router";

반대로, 이렇게 내가 만든 객체를 다른 파일에서 import해서 사용할 수 있게 하려면, 그 객체가 있는 파일 내에서 export를 해 주어야 하는데, 이때 default로 export할 수도 있고, default 없이 export할 수도 있다.

1. default로 할 때

문법 : (문서 최하단에) export default 파일이름;

Ex) export default app;

//default로 export한다는 건 이 파일 전체를 export한다는 뜻

2. default로 하지 않을 때

문법 : (객체를 생성할 때) export const 객체이름 = 객체선언문~~~;

Ex) export const userRouter = express.Router();

<express.js를 이용해 서버 생성하는 방법>

1. express를 가져와서 실행하여 서버를 생성한다.

문법 : const 변수명 = require('express');

const 서버변수명 = 변수명(); //변수명() 이거는 아마도 생성자..같은게 아닐까 한다.

//서버변수명에 express가 실행되어 담기게 됨.

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

const app = express();

2. listen 함수를 통해 request를 기다린다.

문법 : 서버변수명.listen(PORT번호, request를 받은 순간 실행할 함수(get과는 독립적으로 작동) 이름);

서버변수명.listen은 특정 포트번호를 가진 포트의 request을 기다리게 한다. localhost:포트번호라고 URL에 입력하면 그 포트에 request가게 된다. (PORT는 호스트 내에서 실행되는 프로세스를 구분하기 위한 숫자)

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

const app = express();

const PORT = 4000;

function whenServerRequested() {

console.log(`listening started on port ${PORT}.`);

}

app.listen(PORT, whenServerRequested);

※만든 서버 실행하는 법 : 터미널을 열어서 ‘node 서버가 담긴 자바스크립트 이름’이렇게 입력하면 서버가 생성되고, ctrl+c를 눌러서 종료시킨다. 이때 작업을 편하게 하기 위해 package.json 파일에 다음과 같이 추가해 주면 ‘npm 약자이름’만으로도 실행할 수 있다.

"scripts": {

"약자 이름": " node 서버가 담긴 자바스크립트 이름 "

}

Ex)

"scripts": {

"start": "node index.js"

}

<get request 응답하기>

문법 :

응답 함수 선언 => function 응답함수이름(req,res){ //request object, response object.

~~~~(함수내용)

}

응답 함수 호출 => 서버변수명.get("라우팅할 주소 이름(default는 “/”)", 응답 함수 이름);

※- req,res obj : 누군가 정보를 얻고자 페이지를 요청했을때, 어떤 종류의 데이터가 페이지로 전송됐는지 확인할때 request object를 이용한다. 예를 들어 이 URL에 POST 방식으로 아이디와 패스워드를 전송한다면, 이렇게 app.post로 아이디랑 패스워드를 보낸다면 서버에는 request object로 정보를 얻을 수 있다.

Ex)

function handleProfile(req, res) {

res.send("you are on my profile");//아직 잘 모름. 서버 화면에 출력됨.

}

app.get("/profile", handleProfile);

<express의 문자열 읽는 방법>

Express는 똑똑해서, url 주소에 콜론(:)으로 시작하는 값이 있으면 변수로 인식한다.

=> "/:id" ≈ `/${id}`

<arrow function>

함수의 최신 문법이다. 원래 함수든 arrow function이든 그냥 똑 같은 거라고 생각하면 됨.

원래 함수 문법 :

Function 함수이름(변수들){

함수내용~~

}

Arrow function 문법 :

Const 함수이름 = (변수들)=>{함수내용~~};

(혹은 Const 함수이름 = (변수들)=>함수내용;)

(끝에 세미콜론 꼭 넣어야함!!)

이때 이것보다 더 줄여서, ‘이름 없는 함수’를 만들 수도 있다. 이런 경우 함수를 다른 곳에서 호출할 수는 없기 때문에, 함수가 호출될 자리에서 바로 선언해버리면 된다.

문법 : (변수들)=>{함수내용}

Ex)

원래 문법 :

const whenServerRequested = () => {

console.log("now listening...");

}

app.listen(PORT, whenServerRequested);

이름 없는 함수 문법 :

app.listen(PORT, () => console.log("now listening..."));

* arrow function에서의 implicit return : arrow function에서 중괄호가 없으면, 화살표 다음 내용에 암묵적으로 return이 포함된다.

function explicit1() {

return true;//이게 원래 문법.

}

const implicit = () => true;//중괄호가 없으면, true 앞에 암묵적으로 return이 포함된다.

//const implicitError = () =>return true; <= 에러 발생

const explicit2 = () => {

return true;//explicit1 함수와 explicit2는 완전히 동일하다. 중괄호가 있으므로 return을 붙인다.

}

<middleWare>

App(서버변수)이 get 함수를 실행할 때, 그 이전에 실행되는 함수들. Get 함수와 get이 실행하는 함수 중간에 낀 함수라서 middleWare라고 부른다.

문법 :

* 선언 :

const middleWare함수이름 = (req, res, next) => {//req,res 순서 지켜야 함.

(함수내용~~)

next();

};

여기서 next 인자는 그 다음 함수를 실행할 권한을 통제한다. next();부분이 없다면 get이 실행하는 함수나, 그 다음 middleWare함수가 실행되지 않는다.

* 사용 :

1. get에서 호출

서버변수명.get("주소", middleWare함수이름1, middleWare함수이름2,…, get이 실행하는 함수);

2. 서버변수명.use를 통해 호출(권장)

get의 위쪽에 있어야 middleware로서 작동함.

서버변수명.use(middleWare함수이름);

Ex)

const middleWareOfHome = (req, res, next) => {

console.log("middleWare function operated.");

next();

};

app.use(middleWareOfHome);

//middleWares as you want...(get의 위쪽에 있어야 middleware로서 작동함.)

app.get("/", handleHome);

//app.get("/", middleWareOfHome, handleHome);

<여러 가지 middleWare들>

* Morgan

logging에 도움을 주는 middleware. //logging : 무슨 일이 일어났는지를 기록하는 것.

Morgan 설치 후 문서 최상단에 import morgan from "morgan"; 입력 -> middleWare를 둘 곳에app.use(morgan("dev"));라고 입력.( common, combined,tiny등잉 있음. 자세한건 공식문서(morgan npm)참조.)

* Helmet

보안용 middleware. 설치 후 문서 최상단에 import helmet from "helmet"; 입력 -> middleWare를 둘 곳에 app.use(helmet());라고 입력.

* bodyParser(아직은 잘 모르겠음)

form을 가지고 있는 req obj에 접근하기 위한 middleware. 공식 문서를 보면서 특정 데이터 형식에 맞는 용법을 찾아야 함. 설치 후 문서 최상단에 import bodyParser from "body-parser";라고 입력 -> middleWare를 둘 곳에 [app.use(bodyParser.json()); (서버가 json데이터를 받게 될때, 이를 인지하게 해 줌)] 나,[ app.use(bodyParser.urlencoded({ extended: true }));(서버가 html form데이터를 받게 될때, 이를 인지하게 해 줌)] 등등을 입력.

* cookieParser

session을 다루기 위해 cookie에 유저 정보를 넣기 위한 middleware. (아직은 잘 모르겠음)

cookie 사용 미들웨어.

* local 변수들을 사용하기 위한 middleWare.(문자열 등을 pug에서 변수로서 넣기 위함) 우리가 만든거임!! 우리가 직접 만들어 써야 한다. 만드는 법은 아래에 나와 있다.

<router>

서버 내 주소들이 너무 복잡해지면 일일이 외우거나 타이핑하기 매우 번거로워지므로, router를 사용하여 파일 단위로 나눠 주소들을 할당할 수 있다. 가령 index.js와 router.js가 있다고 하자. Router.js에다가 express를 import하고, 라우터 객체를 선언한 뒤 그 라우터 객체에 하위 경로를 포함해서 get request를 받게 한 다음, App.js에서 그 라우터 객체를 호출해 주면 누군가 /user경로에 접속했을 때 라우터 객체 전체, 즉 라우터 객체 내부의 하위 경로 get request를 사용할 수 있게 된다.

문법 :

[index.js]

서버변수명.use("/상위 경로 이름", 라우터 객체 이름);

//누군가 ‘/상위경로이름’에 접속하면, 라우터 객체 전체를 사용할 수 있게 함.

[router.js]

import express from "express";//express import하기

export const 라우터 객체 이름 = express.Router();//특정 주소가 라우터로서 기능하게 함.

//이를 export해서 index.js에서 쓸 수 있게 함.

라우터 객체 이름.get("/", 응답 함수);//상위경로에서 더 들어가지 않음.

라우터 객체 이름.get("/하위 주소", 응답 함수);//상위경로/하위경로로 들어감.

//응답 함수는 이름없는함수일 수도 있고, 응답 함수 이름일 수도 있음.

(…)

Ex)

[app.js]

app.use("/user", userRouter);

[router.js]

import express from "express";

export const userRouter = express.Router();

userRouter.get("/", (req, res) => res.send('user index'));// http://localhost:4000/user/

userRouter.get("/edit", (req, res) => res.send('user edit'));// http://localhost:4000/user/edit

userRouter.get("/password", (req, res) => res.send('user password'));

<파일의 주소를 찾는 방법>

.=>위로, /=>아래로

온점(.)은 현재 자신이 있는 폴더의 상위 폴더로, 슬래시(/)는 현재 자신이 있는 폴더의 하위 폴더로 이동한다는 뜻이다.

예를 들어서, 현재 자신이 있는 폴더의 상위 폴더에서 파일을 import하고 싶다면,

import 파일이름 from "../파일이름";

이라고 쓰면 된다.

Ex) import routes from "./routes";

<MVC>

데이터를 다루는 패턴이다.

m - model => data. {database}

v - view => how does the data look {template}

c - control => function that looks for the data (data를 보여주기 위한 함수) {javascript function}

<pug>

mvc의 view를 담당해주는 템플릿 언어.

express의 view engine임.

Html과 거의 비슷하다.

<템플릿을 실행시키는 방법(res.render)>

Get request 응답 함수를 만들고, 거기에 res.render함수를 넣으면 된다.

Render 함수의 첫번째 인자는 템플릿이고, 두 번째 인자는 템플릿에 추가할 변수들이 담긴 객체임.

문법 : res.render("pug템플릿이름", { 템플릿 변수 객체 정보(key:”value”) });

Ex) res.render("home", { pageTitle: "Home", cookie:"none" });

<<pug - 문법>>

한 태그 안에 다른 태그를 집어넣으려면, tab 키를 이용해 들여쓰기를 해야 한다.

클래스명은 태그 옆에 .클래스이름(온점 클래스이름)을 써서 표시한다. 여러 클래스명이 있을 땐 여러 개를 붙여 쓴다.

Div 태그는 예외적으로 생략이 가능하다.

텍스트에 자바스크립트 변수를 넣고 싶다면 #{}를 입력하고 괄호 안에 원하는 내용을 넣으면 된다(`~~~~${}~~~~~~`과 비슷). 이때, #{}는 말 그대로 text나 span 등 화면상에 글자가 출력될 곳에만 쓰는 것인 듯함(ex-a(href=routes.join) Join -> #{routes.join}이 아님).

(pug는 텍스트를 텍스트가 아닌 데이터로 바꿔줘야 하는 듯함…?아닌가?)

Ex)

footer.footer

.footer\_\_icon

i.fab.fa-youtube

span.footer\_\_text &copy; #{new Date().getFullYear()} #{siteName}

//- <footer class="footer">

//- <div class="footer\_\_icon">

//- <i class="fab fa-youtube"></i>

//- <span class="footer\_\_text">

//- &copy; 2021 Wetube

//- <!--2021이 자동을 바뀌려면 자바스크립트 필요-->

//- </span>

//- </div>

//- </footer>

<express에서 view를 pug로 설정하는 방법>

express 공식문서 - api reference 클릭 - 왼쪽 Application 클릭 - app.set 클릭하면 (https://expressjs.com/en/4x/api.html#app.set) 거기서 view engine이라는 태그를 찾을 수 있음. 그 기본값은 null인데, 이를 pug로 바꾸면 됨.

바꾸는 법 : 서버변수명.set("view engine", "pug"); (middleware 위쪽에 입력하면 됨)

Ex) app.set("view engine", "pug");

<pug-extends>

다른 곳에서 만들어진 pug 파일을 우리 파일에 똑같이 옮기고, 이 파일에서 그것을 확장한다는 뜻.

문법 : pug 파일 최상단에 extends pug파일이름

Ex) extends layouts/main

<pug-overWrite>

<pug에서 fontAwesome 사용하는 법>

1. Create your new account.

Fontawesome 에 가입하여 계정을 생성합니다.

(bokko771, aa030988)

2. Please verify your email.

이메일 인증을 합니다.

3. Click the profile icon in the top right.

로그인 후 오른쪽 상단에 표시된 동그란 프로필 아이콘을 클릭하세요.

4. Select 'Font Awesome CDN'.

클릭 시 표시된 목차 중 'Font Awesome CDN' 를 선택하세요.

5. Then you can see a free link to your account.

그러면 니콜라스님이 사용하신 동일한 형태의 내 계정의 무료링크를 확인 할 수 있습니다.

6. 그 링크를 복사해서 필요한 pug html 파일의 head 내부에 붙여넣기하면 된다.

<localsMiddleWare 만드는 법>

1. middleWares.js 파일을 만들고, 그 안에 다음과 같이 localsMiddleWare 함수를 만든다.

export const localsMiddleWare = (req, res, next) => {

res.locals.siteName = "WeTube";//전역변수처럼 작동함. siteName은 변수명 같은 것. 우리 임의로 정하면 됨. 모든 템플릿,뷰 등등에서 사용 가능.

(…)

(문법 : res.locals.전역변수이름=”전역변수에 들어갈 데이터”;)

next();

};

2. app.js에서 import하고, middleWare들이 모여 있는 곳에 추가한다.

import { localsMiddleWare } from "./middlewares";

(…)

app.use(localsMiddleWare);

<one single source of truth>

한 곳에서만 정보를 보관하는 것.

매우매우 중요한 원칙. 버그를 최소화함.

<pug – include>

일종의 html 모듈화 같은 것. 한 pug 파일에 Html의 일부분을 만들어 놓고, 그걸 다른 pug 파일에서 include하면 그 일부분이 포함된 pug 파일이 완성된다.

문법 : include 포함시킬 파일주소

Ex)

(main.pug)

doctype html

html

head

title #{siteName}

body

header

include ../partials/header -> header.pug를 포함시킴

main

block content

(header.pug)

header.header ->header가 중복됨에 주의

.header\_\_column

a(href=routes.home)

i.fab.fa-youtube

.header\_\_column

ul

li

a(href=routes.join) Join

li

a(href=routes.login) Log in