NodeJS란?

Javascript outside browser

Chrome V8 JavaScript engine로 빌드 된 자바스크립트 런타임

브라우저 밖에서 돌아가는 JavaScript. 95년 JS가 처음 만들어졌을 때, JS는 브라우저에서만 작동. 브라우저와 좀 더 상호작용하기 위해 만들어진 언어였지만 현재 가장 많이 사용하는 언어가 되었다. 브라우저만 있다면 누구든 사용할 수 있었던 언어. 하지만 유일한 언어이기도 했다.

Ryan Dahl이 브라우저와 분리된 환경에서도 JS를 이용하기 위해 만듦

NodeJS를 이용해 React Native와 electron 사용이 가능해졌고, electron으로 만들어진 데스크톱 앱에는 VS code, slack, Figma 등이 있다. React Native 이용 앱은 Facebook, Instagram, Discord 등.

NPM이란?

JavaScript 언어를 위한 패키지 매니저로, node.js설치 시 함께 설치되며, node.js와 상호작용을 하여 이용. 꼭! 함께 써야함

여러 개발자들이 패키지를 만들어 공유 할 수 있고, 다운로드 받아 다른 개발자들이 사용 가능

SET UP

1. 폴더 생성 후 package.json 생성.

package.json이란? 프로그래머가 파일에 정보를 저장하기 위해 만든 방식으로, 가장 처음으로만들어야 하는 파일.node.js에서 이름은 꼭 package.json이어야만 한다. 대소문자 구분하니 꼭 소문자로 작성!

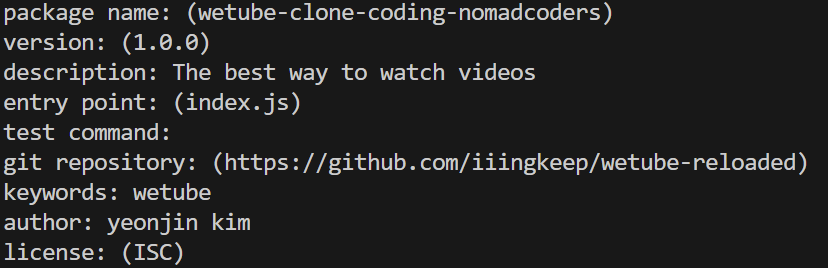
이 package.json은 직접 작성하면 실수가 있을 수 있으니, 나중에 도움을 받아 설치

2. git 등록

-폴더에서 git init으로 repository 생성

-깃허브에 repository를 생성 후, git remote add origin 깃헙url 등록

-npm init 입력 시 package.json의 내용에 담을 항목들이 나온다. 정보를 입력해주면 된다.



모두 입력 한 뒤, package.json 파일이 자동 생성되며 안에 입력한 항목이 담겨있다.

{

  "name": "wetube-clone-coding-nomadcoders",

  "version": "1.0.0",

  "description": "The best way to watch videos",

  "main": "index.js",

  "scripts": {

    "test": "echo \"Error: no test specified\" && exit 1"

  },

  "repository": {

    "type": "git",

    "url": "(https://github.com/iiingkeep/wetube-reloaded)"

  },

  "keywords": [

    "wetube"

  ],

  "author": "yeonjin kim",

  "license": "ISC"

}

Main은 main file이 되는 것을 말하며, index.js라고 써져 있으므로 index.js파일을 생성한다. Main이 필수는 아니니 지워줘도 된다.

시작은 이렇게, package.json파일 + index.js 두개의 파일로!

3. node.js 실행

첫 번째: 터미널에 node index.js 처럼 node 파일명 을 입력하면 실행

두 번째: package.json이 실행하도록 설정

"scripts": {

    "win": "node index.js"

  },

위와 같이 입력한다면 npm run win 입력시 node index.js를 입력하는 것과 같아 node.js 실행

이 명령어는 package.json이 있는 디렉터리에서만 작동!

4. npm i

프롬프트에 npm i 설치할 패키지이름 을 입력시, package-lock.json파일과 node\_modules 폴더가 생성 된다. Node\_modules 폴더에는 npm으로 설치한 모든 패키지 저장. 하나의 package를 설치 시, 다른 여러가지 패키지들이 함께 필요하며 각각의 package.json 파일을 가지고 있다.

또한, package.json의 dependencies부분에 설치한 패키지가 자동으로 등록된다.

 "dependencies": {

    "express": "^4.19.2"

  }

이 dependencies는 필요한 패키지들을 담아뒀다가 node\_modules폴더 또는 package-lock.json이 삭제됐더라도 npm i 만 입력하면 자동으로 설치되도록 한다.

Package.json이 열려있을 때 npm i를 하다가 버전이 충돌/변경 될 수 있으니 꼭 package.json파일은 닫고 npm i를 하도록 한다

5. package-lock.json

패키지들을 안전하게 관리. 즉, 다른 사람이 내 파일을 받아 npm i를 한다면 패키지의 버전을 모두 동일하게 맞춰주어 다른 컴퓨터에서도 잘 작동하게 한다

Babel이란?

JavaScript compiler 즉, JavaScript는 계속 업데이트 되는데, nodejs가 이해하지 못하는 JS언어가 생길 수 있다. 이 때 사용할 수 있는 것이 바로 babel. 우리가 작성한 최신 자바스크립트를 컴파일 해줌으로써 nodejs가 자바스크립트를 문제 없이 이해할 수 있도록 한다

<https://babeljs.io/>

1. 설치를 해준다: npm install --save-dev @babel/core

Package.json을 열어 보면 다음과 같이 추가되어 있다.

"devDependencies": {

    "@babel/core": "^7.24.3"

  }

dependencies와 devDependencies의 차이는?  
dependencies에는 프로젝트에 필요한 패키지를 나열하고, devDependencies에는 개발자에게 필요한 패키지를 나열한다. 이것은 마치 운전을 하려면 차에 연료가 필요한 것(dependencies)과 같고, 내가 운전을 더 잘 하려면 음악이 필요한 것(devDependencies)과 같다.

2. 설정 파일 생성

맥과 리눅스에서는 프롬프트에 babel.config.json 입력하면 파일이 자동 생성

윈도우는 수동으로 위의 파일명을 입력하여 파일을 생성해줘야한다.

생성된 파일에 사이트에 나와있는대로 입력

{

  "presets": ["@babel/preset-env"]

}

마지막으로 한 가지 더 설치: npm install @babel/preset-env --save-dev

Package.json에 보면 추가 되어 있다

"devDependencies": {

    "@babel/core": "^7.24.3",

    "@babel/preset-env": "^7.24.3"

  }

최신 문법을 이해하기 위해, babel이 알아서 babel.config.json파일을 찾아 내용을 들여다 본다.

특히 presets은 babel을 위한 거대한 플러그인으로, 다양한 preset들이 있는데, 우리는 최신 문법 사용을 위해 preset-env을 사용

Babel 홈페이지의 usage 부분을 보면 transform “code”부분에 코드를 넣으면 ugly 코드(예전 문법)으로 변환을 해준다. 그런데 이것을 일일이 하는 것은 비효율적! 그래서

3. Package.json에 Babel로 컴파일하는 script를 생성



npm install @babel/core @babel/node --save-dev 를 통해 @babel/node 설치해준다.

이를 package.json의 scripts에 추가하여 실행할 때 마다 자동으로 컴파일 할 수 있도록 다음과 같이 추가해준다

"scripts": {

    "dev": "babel-node index.js"

  },

Npm run dev를 입력할 시, babel-node가 자동으로 실행 될 것이고, 최신 문법을 오랜 JS 문법으로 자동으로 바꿔 줄 것이다.

const express = require("express");

import express from "express";

이제 아래 문장으로 입력해도(최신 버전) babel이 알아서 위의 문장으로(옛날 버전) 번역 해 준다.

4. nodemon 설치, 적용

지금까지는 코드를 수정하면 다시 서버를 실행해야 하는 불편함이 있었다. Nodemon은 실행 중 코드가 수정되면 이를 감지하고, 변경된 부분을 반영하여 자동으로 재시작 해준다.

Npm I nodemon 입력 후 package.json의 scripts 부분을 다음과 같이 변경

"scripts": {

    "dev": "nodemon --exec babel-node index.js"

  },

\*\*\*실행 정리\*\*\*

1. node index.js: node 환경에서 index.js 실행

2. babel-node index.js: babel-node 이용해 최신 문법을 자동으로 컴파일하며 index.js 실행

3. nodemon –exec babel-node index.js: 코드 수정 시 즉각 반영하여 서버 재실행

Your First Server: 첫번째 서버 만들기

서버란?

인터넷과 연결되어 있는 하나의 컴퓨터로, 항상 request를 listening 하고 있다.

Request란 주소창에 google.com 또는 google.com/login을 입력해 이동한다고 할 때 google.com에 request를 보낸 것이고, google.com/login에 request를 보내는 것.

import express from "express";

import ~ from ~ 구분으로 필요한 패키지를 불러올 시, npm과 Nodejs가 node\_modules폴더에서 express 패키지를 찾아 그 패키지의 index.js를 불러온다. Wow

express 사용하기

1. express application 생성: const app = express();

import express from "express";

const app = express();

app이 아닌 어떤 변수가 들어가도 상관 없지만, app이 관습적이므로 그대로 사용

express()함수는 express application을 생성해준다.

2. app이 포트를 listen 하도록 한다. 포트 번호는 4000번으로 임의 지정

const handleListening = () => console.log("Server listening on port 4000");

app.listen(4000, handleListening)

3. 하드코딩 방지! 다음과 같이 코드를 정리 해 준다

import express from "express";

const PORT = 4000;

const app = express();

const handleListening = () => console.log(`Server listening on port ${PORT}`);

app.listen(PORT, handleListening)

\*\*\*결과: 주소창에 localhost:4000 입력 시 Cannot GET / 을 표시하는 페이지로 이동한다. 이는 우리가 만든 서버에서 온 응답! 서버 만들기 성공!

서버가 request에 response 하도록 만들기

Request란?

먼저, 위에서 응답으로 받았던 Cannot GET / 이 무엇인지 파악해보자.

/ 슬래시: root, 첫 페이지를 의미.

GET: HTTP method

HTTP란? 우리가 서버와 / 또는 서버와 서버가 의사소통 하는 방식으로, 처음 사용된 방식이라 가장 오래된 방식이지만 가장 안정적인 방식.

http request: 웹사이트에 접속하고 서버에 정보를 보내는 방법.

우리가 웹사이트에 접속하려고 하면, 브라우저는 자동으로 request하고, 우리에게 웹사이트를 get 해주려고 한다. 우리가 요청한 어떤 특정 페이지를 가져다(get) 주는 것이다.

Request: 유저가 뭔가를 요청하거나, 보내는 등의 행동

Request는 다음과 같이 다룰 수 있다.

app.get("/", () => console.log("sombody is trying to go home"))

=

const handleHome = console.log("sombody is trying to go home")

app.get("/", handleHome);

위의 코드는 서버가 자신이 request를 다룰 수 있다는 것을 알게 하고, 요청이 들어오면 브라우저가 /(root)로 가려는구나 하며 callback함수를 실행시킨다.

하지만 callback 함수를 실행시키기만 할 뿐, 응답은 주지 않아 브라우저는 계속 응답을 기다리는 상태가 된다.

Response란?

Request가 오면 express는 Callback 함수에 req, res object를 넣어준다.

응답 하는 방법

1. return res.end(); 응답 종료

const handleHome = (req, res) => {

  return res.end()

}

그래서, 위와 같이 callback 함수에 req, res를 넣어준 뒤 return res.end() 를 입력해주면 브라우저가 응답으로 end를 받아 종료시킨다. 즉, 브라우저는 더 이상 응답을 기다리지 않고 빈화면만을 출력한다. 서버가 request를 종료시키는 응답을 했기 때문! Cannot GET / 문구도 더 이상 뜨지 않는다.

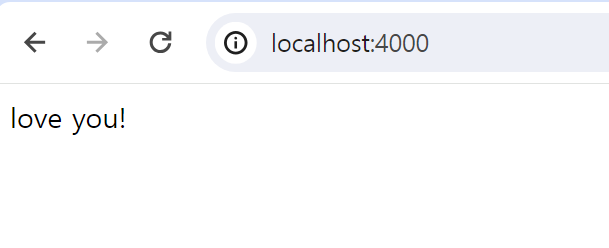
2. return res.send(“message”); 메시지 반환

const handleHome = (req, res) => {

  return res.send("love you!")

}

app.get("/", handleHome);



Send 함수를 이용하면 브라우저에 메시지를 띄울 수 있다. 응답을 잘 보내는 지 확인할 때 사용하면 좋을 것 같다.

Route 하나 더 추가해보기: /login

const handleHome = (req, res) => {

  return res.send("love you!")

}

const handleLogin = (req, res) => {

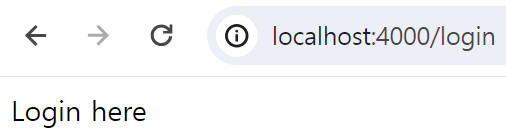
  return res.send("Login here");

}

app.get("/", handleHome);

app.get('/login', handleLogin);

위와 같이 /login에 요청이 온다면 handleLogin 함수를 실행하여 응답으로 Login here 메시지를 반환한다.



응답의 방식은 html, json, status code 등 다양하다. 나중에 차차 알아보자.

**Middlewares**

Middleware란?

Request와 response의 중간에 있는 소프트웨어로, 모든 middleware는 handler이고 모든 handler는 middleware이다. 또한 이는 controller라고도 불린다(MVC)

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

  next();

}

const handleHome = (req, res) => {

  return res.send("love you!")

}

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

위와 같이 gossipMiddleware라는 미들웨어 하나를 추가해 준다.

미들웨어 추가 시, app.get 부분에도 넣어줘야한다.

미들웨어는 req, res에 추가로 next argument를 받으며, 이는 next(); 코드를 이용해 미들웨어에서 실행해야 할 코드를 실행 뒤, 다음 컨트롤러로(handleLogin) 역할을 넘긴다.

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

  console.log("i'm middleware!");

  next();

}

const handleHome = (req, res) => {

  return res.send("i love middlewares!")

}

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

위와 같은 코드가 있다면 서버는 /(root)로 get request가 들어온 것을 인지하고 gossipMiddleware 미들웨어를 실행시킨다. 이 함수는 콘솔에 메시지를 출력한 뒤 next(); 함수를 이용해 다음 컨트롤러인 handleHome을 실행시켜 브라우저에 I love middlewares! 메시지를 출력.

req.url 사용하기

req.url은 요청이 온 url을 기억하고 있다. 위의 코드에서 미들웨어를 다음과 같이 수정한다면

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

  console.log(`someone is going to: ${req.url}`);

  next();

}

콘솔에는 다음의 메시지가 출력된다: someone is going to: /

App.use();

: 어느 URL에도 작동하는 global middleware를 만들 수 있게 해주는 함수. App.use()는 사용하는 순서가 중요한데, app.use가 먼저 오고 그 다음 app.get 이 와야한다.

app.use(gossipMiddleware);

app.get("/", handleHome);

만약 app.use가 app.get보다 나중에 오도록 순서를 바꾼다면

app.get("/", handleHome);

app.use(gossipMiddleware);

/(root)로 이동을 해도 gossipMiddleware는 작동하지 않으니 모든 route에 적용시키려면 맨 위에 둬야한다.

미들웨어를 하나 더 추가 해 본다면

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

  console.log(`${req.method} ${req.url}`);

  next();

}

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

  const url = req.url;

  if(url === "/protected") {

    return res.send("<h1>Not Allowed</h1>");

  }

  console.log("Allowed, you may continue.");

  next();

}

const handleHome = (req, res) => {

  return res.send("i love middlewares!")

}

const handleProtected = (req, res) => {

  return res.send("welcome");

}

app.use(logger);

app.use(privateMiddleware);

app.get("/", handleHome);

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

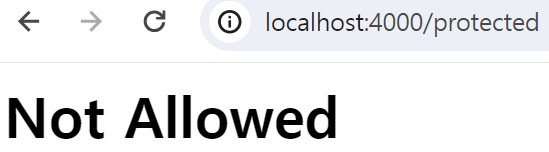
/(root) 의 요청을 받은 서버는 모든 route에 사용된 미들웨어 두 개를 거쳐 콘솔에 메세지 출력

GET /

Allowed, you may continue.

그 이후에야 handleHome 함수를 실행시켜 화면에는 I love middlewares!를 반환.

반면에, /protected 요청을 받은 서버는 logger 미들웨어에 의해 콘솔에 GET /protected 메시지를 출력하지만 privateMiddleware의 if구문에서 return 후 응답을 종료한다. 이것이 미들웨어의 힘!



\*\*\*정리\*\*\*

\*어떤 controller든 Middleware가 될 수 있으며 (req, res, next)를 포함하고 있다.

\*App.get(‘/route’, middleware, controller) 의 순서로, middleware함수에서 실행해야 할 부분을 실행 뒤 next(); 함수로 controller를 실행시키도록 넘어간다. middleware함수에서 return을 하면 다음 controller는 실행되지 않고 응답을 중지한다. 또한 if구문으로 경우를 나누어 다음 controller로 넘어가도록 하거나 응답을 중지하기도 한다. 응답하는 마지막 controller는 next() 사용 X

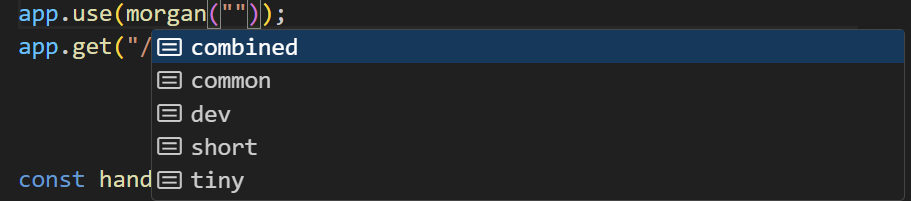
\*app.use()안에 middleware를 넣어 app.use(middleware)로 사용한다면 이 middleware는 어느 route에서든 작동되도록 한다. App.use는 app.get보다 윗부분에 있어야 작동한다.

\*req.method: GET, POST, PUT, DELETE 등 어떤 method인지 알려준다

\*req.url 또는 req.path: /, /login, /mypage와 같이

추가 Middleware:: morgan

: HTTP request logger middleware for node.js (npm 공식문서)



다섯가지 옵션(combined, common, dev, short, tiny)이 있는데 그 중 dev 사용

import express from "express";

import morgan from "morgan";

const PORT = 4000;

const app = express();

const logger = morgan("dev")

const handleHome = (req, res) => {

  return res.send("i love middlewares!")

}

app.use(logger);

app.get("/", handleHome);

const handleListening = () => console.log(`Server listening on port ${PORT}`);

app.listen(PORT, handleListening)

npm i morgan으로 설치 후 위와 같이 코드 작성.

/(root)로 요청이 들어오면 logger middleware가 작동하는데 이는 morgan(“dev”)를 실행 시키고 next()로 handleHome을 실행시킨다. 지금까지 직접 만든 middleware들을 사용해봤는데, morgan을 설치해 사용하는 이유는 morgan이 좀 더 세부적이기 때문.

/(root)요청 후 console에는 GET / 304 0.520 ms - - 메시지가 출력된다. 이는 method, path, status code, 응답까지 걸린 시간 정보를 반환해주는 것. Morgan에 사용하는 옵션마다 반환하는 정보가 다르다.

#4. Routers

: Simple middleware-style router(npm 공식문서) = controller와 URL 관리를 쉽게 도와줌

라우터 만들기 플랜

프로젝트에서 가장 먼저 생각할 것은 데이터. 어떤 종류의 데이터를 이용할 것인가?

우리의 경우, 1.비디오 2.유저

Wetube Reloaded 작성

/ (Home)

/join (Join)

/login (Login)

/search (Search)

/edit-user (Edit User)

/delete-user (Delete User)

/watch-video (Watch Video)

/edit-video (Edit Video)

/delete-video (Delete Video)

이렇게 라우터를 설정할 수 있다. 하지만 이는 다음과 같이 정리할 수 있다.

\*\*\*\*\*\*\*\*\*\*Global router\*\*\*\*\*\*\*\*\*\*\*

/ (Home)

/join (Join)

/login (Login)

/search (Search)

\*\*\*\*\*\*\*\*\*\*Users router\*\*\*\*\*\*\*\*\*\*\*

/users/edit (Edit User)

/users/delete (Delete User)

\*\*\*\*\*\*\*\*\*\*Videos router\*\*\*\*\*\*\*\*\*\*\*

/videos/watch (Watch Video)

/videos/edit (Edit Video)

/videos/delete (Delete Video)

/videos/comments (Comment on a video)

/videos/comments/delete (Delete A Comment of a Video)

즉, 라우터는 작업중인 주제를 기반으로 URL을 그룹화(global / users / videos)

Router 만들기

1. app.use(‘/nameof’, nameofRouter); 로 express를 이용해 url과 라우터 이름 매치

Ec) ‘ / ‘ 이 url은 globalRouter라고 할거야~

app.use('/', globalRouter);

app.use('/videos', videoRouter);

app.use('/users', userRouter);

2. nameofRouter.express.Router(); 로 각 대표 라우터를 Router로 등록

Ex ) globalRouter, userRouter, videoRouter를 Router로 등록

응답 함수를 설정해준 뒤 각 nameofRouter.get(‘url’, cb); 로 Router의 콜백함수로 넣어준다.

const globalRouter = express.Router();

const handleHome = (req, res) => res.send('Home');

globalRouter.get('/', handleHome);

const userRouter = express.Router();

const handleEditUser = (req, res) => res.send('Edit user');

userRouter.get('/edit', handleEditUser);

const videoRouter = express.Router();

const handleWatchVideo = (req, res) => res.send('Watch Video');

videoRouter.get('/watch', handleWatchVideo);

app.use('/', globalRouter);

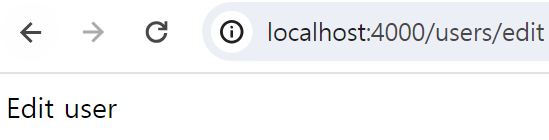
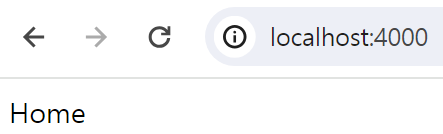
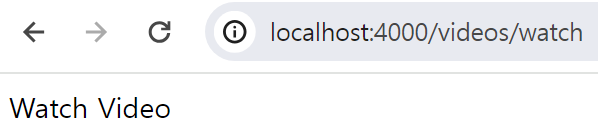
app.use('/videos', videoRouter);

app.use('/users', userRouter);

위와 같이 설정하면 유저가 /router로 시작하는 url에 접근을 시도할 시 express는 해당 대표 라우터로 들어가 그 안에 있는 세부 url(controller)을 찾아내 해당 함수를 실행한다.

Ex) 유저가 /video 로 시작하는 url에 접근을 시도하면 express는 videoRouter 안에 있는 /watch를 찾아 /video/watch 라는 url을 만든다. 그 후 handleWatchVideo 함수 실행

각각의 url로 접속해보면 해당 url에 설정한 응답이 잘 출력되는 것을 볼 수 있다.

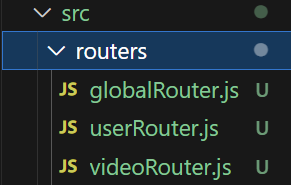


# 4.2 Cleaning the Code, 코드 정리하기

Server.js에 라우터 설정을 해 주었다. 개발자들의 bible인 ‘클린 코드’책에는 이렇게 나와 있다.

“일단 생각 나는대로 코드를 작성해라. 지저분해도 좋다! 이후에 같은 시간을 들여 고민해서 코드를 깔끔하게 정돈해라!”

먼저 Server.js에 작성한 라우터 관련 코드들을 정리하기 위해 src폴더에 routers 폴더를 만든 뒤, 각각의 대표 라우터의 JS파일을 생성



각각의 Router 관련 코드들을 해당 폴더로 옮긴다.

이 때 중요한 점은, 모든 JS파일은 각각이 모듈이며 private한 성질을 가진다. 그러므로 새로 생성한 파일에서 express를 사용하기 위해 express를 import 해줘야 한다.

또한 Server.js에서는 app.use로 각각의 Router를 사용하겠다고 했지만 Router에 대한 정의가 없어졌기 때문에 다음과 같이 에러가 뜬다



새로 생성한 Router 파일에서 정의된 Router 변수들을 server.js로 export 한 뒤, server.js에서 import 해줘야 한다.

Src > routers > globalRouter.js

import express from "express";

const globalRouter = express.Router();

const handleHome = (req, res) => res.send('Home');

globalRouter.get('/', handleHome);

export default globalRouter;

server.js

import express from "express";

import morgan from "morgan";

import globalRouter from "./routers/globalRouter";

import userRouter from "./routers/userRouter";

import videoRouter from "./routers/videoRouter";

const PORT = 4000;

const app = express();

const logger = morgan("dev")

app.use(logger);

app.use('/', globalRouter);

app.use('/videos', videoRouter);

app.use('/users', userRouter);

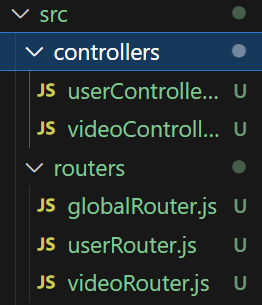
const handleListening = () => console.log(`Server listening on port ${PORT}`);

app.listen(PORT, handleListening)

각각의 ~Router.js 파일에는 앞으로 만들 페이지들에 따라 더 많은 url 코드를 생성할 것이다.

그런만큼 cb함수도 같이 늘어날 것. 함수는 controller다.

이 파일들은 router를 위한 ~Router.js 파일이지, router&controller.js 파일이 아니다. 이 controller들도 분리를 해 주는 것이 클린코드 하는 것! 그러니 controllers폴더도 만들어준다.

■ controllers폴더에 globalController.js가 없는 이유?

메인(‘ / ‘)페이지엔 video들을 출력할 것이므로 videoController.js에 작성,

/join 은 user가 이용할 것이므로 userController.js에 작성하기 때문!

각각의 ~Controller.js에 controller관련 코드를 작성, controller들을 export 해 주고, ~Router.js에서 해당 코드를 삭제 후 controller들을 import 해 준다.

videoControllers.js

export const trending = (req,res) => res.send('Home Page Videos');

export const watch = (req, res) => res.send('Watch');

export const edit = (req,res) => res.send(' Edit');

videoRouter.js

import express from "express";

import {watch, edit} from "../controllers/videoController"

const videoRouter = express.Router();

videoRouter.get('/watch', watch);

videoRouter.get('/edit', edit);

export default videoRouter;

■ export vs default export

Default export: 하나의 파일에서 하나의 변수나 함수 만을 export 해 줄 수 있다. 대신 그 이름을 기억해 다른 파일에서 다른 이름으로 변경해 import 할 수 있다. 하지만 헷갈려서 보통 같은 이름으로 사용

Export: 하나의 파일에서 여러 개의 변수 또는 함수를 export할 때 사용한다. 대신 이름을 기억하지 못하기 때문에 이름을 그대로 사용해줘야 한다. 또한 { }(object) 안에 넣어 import 해줘야 한다.

#4.7 URL Parameters part One

:id 는 무엇일까?

Parameter. 즉, url 안에 변수를 포함시킬 수 있게 해준다.

: (콜론)은 express에게 이게 변수라는 것을 알리기 위해 사용하는 기호.

videoRouter.j에 다음과 같이 입력되어 있고

videoRouter.get('/upload', upload);

videoRouter.get('/:id', see);

videoRouter.get("/:id/edit", edit);

videoRouter.get('/:id/delete', deleteVideo);

videoController.js에 다음과 같이 입력

export const see = (req, res) => {

  console.log(req.params);

  return res.send('Watch');

}

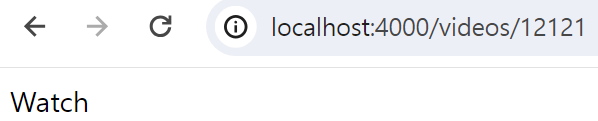
export const edit = (req,res) => {

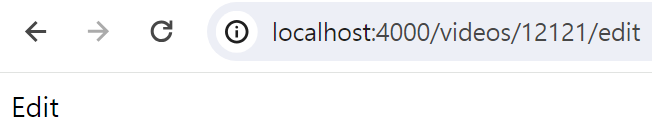
  console.log(req.params);

  return res.send('Edit');

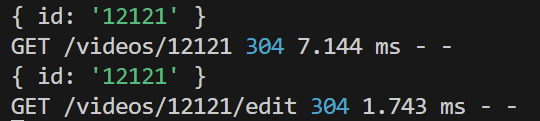
}

URL에 아무런 id가 될만한 텍스트를 입력해보면



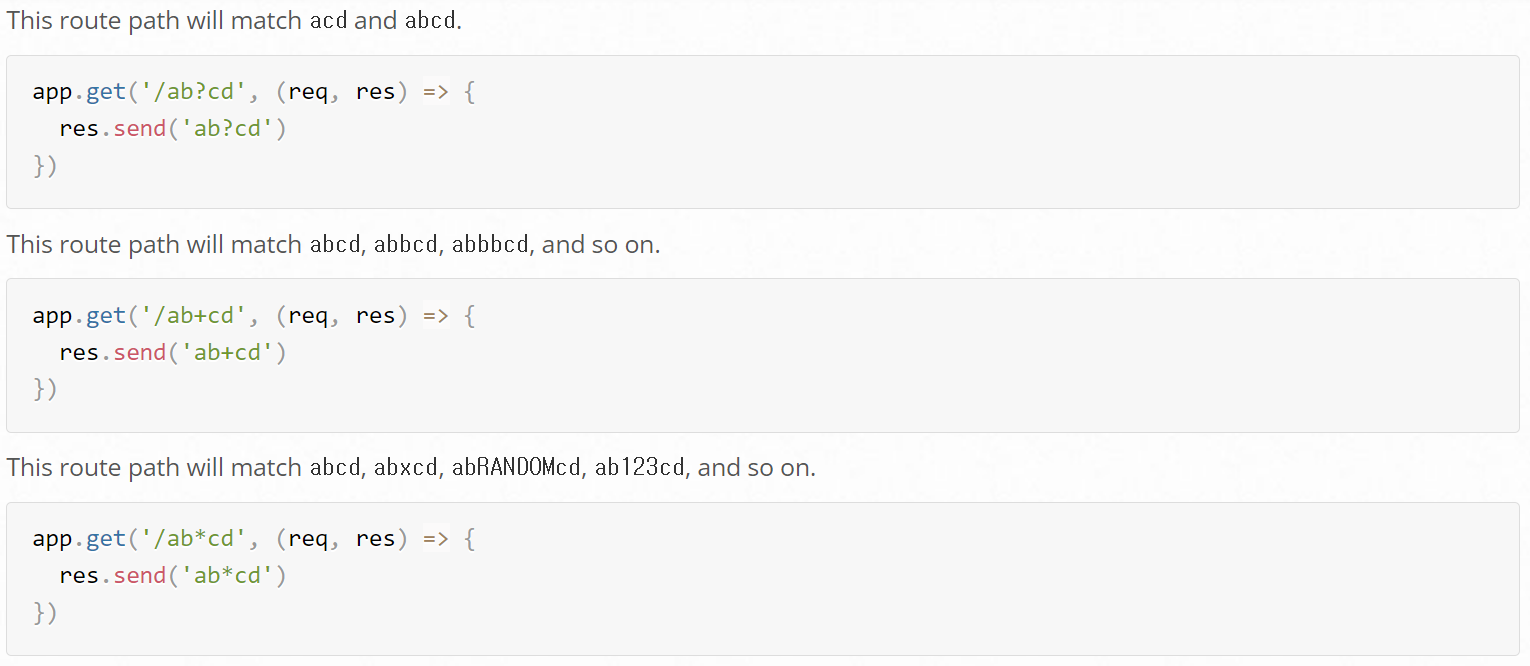


이런식으로 응답을 받을 수 있고 콘솔에서도 이를 인식해 id를 알려준다.



#4.8 URL Parameters part Two

정규식(regular expression)을 알아보자



? : 바로 앞의 글자가 있을 수도 있고, 없을 수도 있다

+ : 바로 앞의 글자가 몇 개든지 올 수 있다.

\* : \*의 자리에는 어떤 문자든 삽입 될 수 있다.

자바스크립트에 사용하기

videoRouter.get('/upload', upload);

videoRouter.get('/:id(\\d+)', see);

videoRouter.get('/:id(\\d+)/edit', edit);

videoRouter.get('/:id(\\d+)/delete', deleteVideo);

우리가 지정한 :id 파라미터 뒤에 ([\\d](file:///\\d)+) 코드를 추가해 숫자만 가능하다는 것을 나타낸다.

이렇게 정규식으로 지정해주면 더 이상 upload가 id랑 혼동되지 않아 코드가 맨 위에 있지 않아도 식별이 되어 /upload url로 이동한다.

#5 TEMPLATES

#5.1 Configuring Pug

Pug란? 사용자에게 뷰를 좀 더 간편하게 제공하기 위한 패키지. Res.render( ) 안에 html 태그를 일일이 넣어 뷰를 표시하려면 같은 코드의 발생이 너무나 많아지고 길어지기 때문에 이를 간단하게 할 수 있는 Pug를 사용.

<https://www.npmjs.com/package/pug>

사용법을 보면, 태그 없이 간단하게 코드를 작성하면 html 문법으로 바꿔준다.





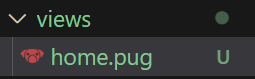
Npm I pug 로 설치해준 뒤, express에 view engine으로 pug를 사용할 것을 알린다.

app.set('view engine', 'pug');

pug를 view engine으로 사용하려면 views 폴더를 만들어줘야 한다.



Src > views 폴더 만들어주고, home.pug 파일을 먼저 만들어주자.



doctype html

html(lang='ko')

  head

    title Wetube

  body

    h1 Welcom to Wetube

    footer &copy; 2021 Wetube

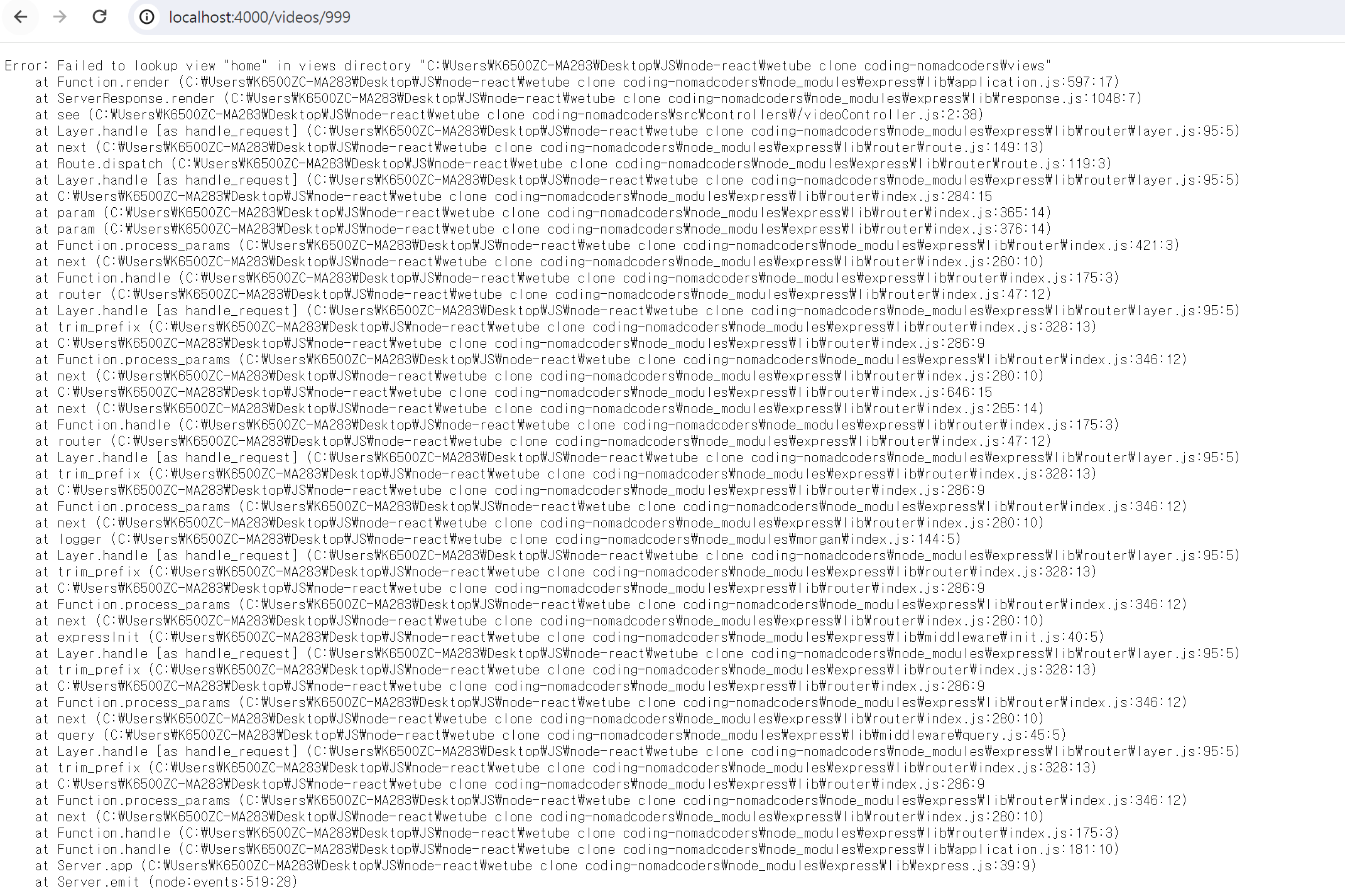
위와 같이 코드를 입력하면 pug가 렌더링해 html 문법으로 만들어주고, 그것을 express가 인식.

export const see = (req, res) => res.render('home');

위와 같이 render( ) 안에 pug 파일명을 담아주면 그 view를 보여준다.

설정 끝!

url을 입력하여 아무 페이지나 가 보면 당황스럽게도 에러를 마주한다.



그 이유는, express 사이트에 나와있는 표를 다시 보면,



/views 폴더는 cwd(current working directory)에 위치해야 하기 때문.

Cwd란 node.js가 실행되는 위치, 즉, package.json이 위치한 폴더의 위치가 된다. 현재 나로서는 wetube clone coding-nomadcoders 폴더가 cwd이다.

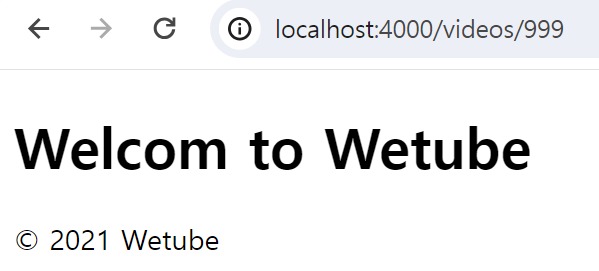
원래대로라면, wetube clone coding-nomadcoders 폴더 바로 안에 /views 폴더를 위치시켜야 home.pug 파일을 찾을 수 있다. 하지만 우린 src 폴더에 모든 코드를 넣을 거기 때문에 이 방법은 좋지 않다.

다른 방법 2. App.set을 이용하기

app.set('view engine', 'pug');

app.set('views', process.cwd() + '/src/views');

위와같이 입력해 default 경로를 바꿔준다. 다시 url을 입력해 렌더링해보면 에러가 뜨지 않는다.



Watch 페이지 view 만들기

/views/watch.pug 파일 생성 후 코드 입력. js코드는 #{ } 안에 넣어주면 된다.

doctype html

html(lang='ko')

  head

    title Wetube

  body

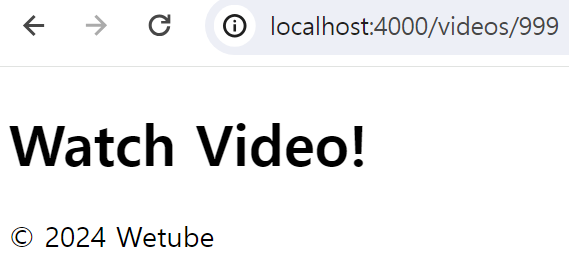
    h1 Watch Video!

    footer &copy; #{new Date().getFullYear()} Wetube

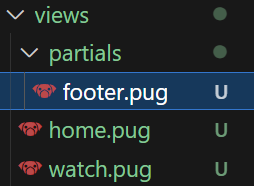
videoController.js에서 render ( ) 안을 view 이름으로 변경(소문자여야만 한다!)

export const see = (req, res) => res.render('watch');

결과



하지만 이럴 경우, 페이지마다 같은 코드가 계속 반복되므로 똑같이 반복되는 코드, 예를들어 footer부분은 partial 폴더를 만들어 따로 관리.



Footer.pug에 반복되던 footer부분의 코드를 입력한 후

footer &copy; #{new Date().getFullYear()} Wetube

include 문법을 이용해 삽입해준다.

doctype html

html(lang='ko')

  head

    title Wetube

  body

    h1 Welcom to Wetube

  include partials/footer

이렇게 하면, footer의 코드의 어떤 부분을 변경해야 할 때 일일이 하나하나 변경하지 않고 footer.pug 파일에서만 변경해주면 된다.

**Pug의 Inheritance / Extends / Block**

하지만 여전히 같은 코드가 모든 페이지마다 반복되는 문제가 계속 된다.

views폴더에 base.pug 파일 생성, 반복되는 기본 코드들을 작성해준다.

doctype html

html(lang='ko')

  head

    title Wetube

  body

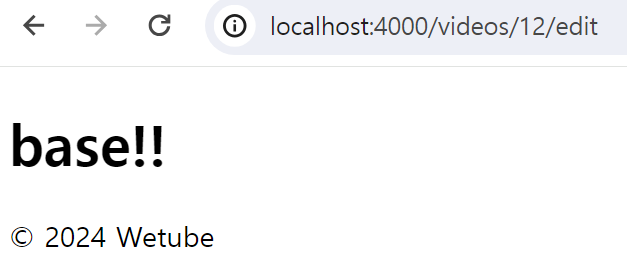
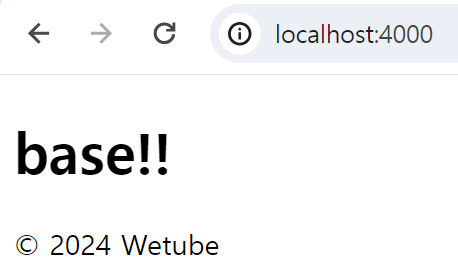
    h1 base!!

  include partials/footer.pug

그리고 home.pug, edit.pug, watch.pug에 extends 문법으로 base.pug의 내용을 가져온다

extends base.pug

결과

모두 같은 결과를 반환.

이제 여기서, body 부분에 각각의 페이지에 맞는 다른 내용을 넣기 위해, 이 부분만 다른 코드를 넣어줘야 하는데, block 문법을 이용한다.

Base.pug

doctype html

html(lang='ko')

  head

    title Wetube

  body

    block content

  include partials/footer.pug

edit.pug

extends base.pug

block content

  h1 Edit!

이런 식으로 block name 을 정해준 후 다른 파일에서 그 block에 들어갈 내용을 삽입.

extends base.pug

block content

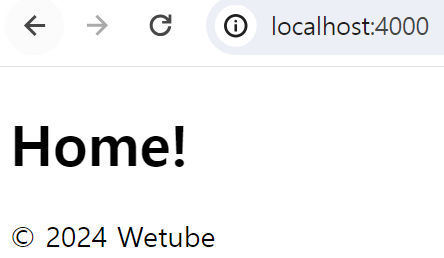
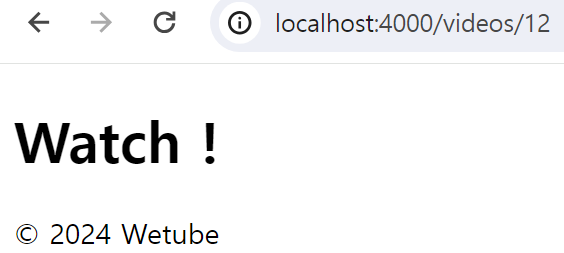
  h1 Home!

extends base.pug

block content

  h1 Watch !

결과

View에 변수를 설정하고, controller에서 변수의 값을 받아 오기

Base.pug

doctype html

html(lang='ko')

  head

    title #{pageTitle} | Wetube

    link(rel="stylesheet" href="https://unpkg.com/mvp.css")

  body

    main

     block content

  include partials/footer.pug

오타 또는 복붙을 방지하기 위해 변수로 지정해야 할 부분을 변수로 만들어준다 = pageTitle

videoController.js

export const trending = (req,res) => res.render('home', {pageTitle: 'Home'});

res.render안에 뷰로 지정할 파일 이름과 변수에 들어갈 데이터 값을 입력

res.render(뷰 파일이름, { 변수명: 넣을 데이터 });

또한 너무 기본적인 css가 예쁘지 않다면 mvp.css를 이용해 자동으로 약간의 css를 적용할 수 있다. Link를 사용해 가져오면 된다.

#5.7 Conditionals

#{pageTitle} : pageTitle이라는 변수이지만, 여러 개의 값을 갖는다

=pageTitle : pagaeTitle이라는 변수이며 오직 한 가지 값을 갖는다

Conditioin: if / else if / else

fakeUser의 loggedIn 상태에 따라 로그인 또는 로그아웃 버튼이 나오도록 뷰를 표현하기

videoController.js

const fakeUser = {

  username: "yj",

  loggedIn: false,

}

export const trending = (req,res) => res.render('home', {pageTitle: 'Home', fakeUser: fakeUser });

base.pug

doctype html

html(lang='ko')

  head

    title #{pageTitle} | Wetube

    link(rel="stylesheet" href="https://unpkg.com/mvp.css")

  body

    header

      if fakeUser.loggedIn

        small Hello #{fakeUser.username}

      nav

        ul

          if fakeUser.loggedIn

            li

              a(href='/logout') logout

          else

            li

              a(href='/login') login

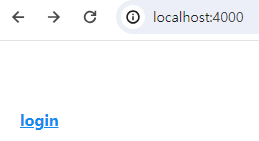
    main

     block content

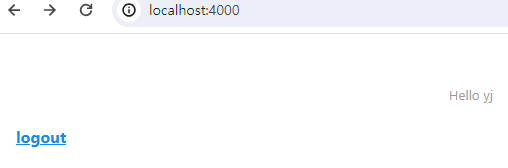
  include partials/footer.pug

결과

1. loggedIn: false 일 때



2. loggedIn: true 일 때



#5.8 Iteration

Iteration: elements의 list를 보여주는 것. 배열 또는 객체 가능

문법: each thing in things

things에는 videoController.js에서 이미 명시되어 있고, home.pug로 전달해주는 변수가 들어가야 한다. 만약 변수가 정의 되어 있지 않거나, 빈 array이거나 length=0일 경우 다음의 에러 발생.

Cannot read properties of undefined (reading 'length')

Home.pug

extends base.pug

block content

  h2 Welcome here

  each video in videos

    li=video

  else

    li Sorry nothing found

결과

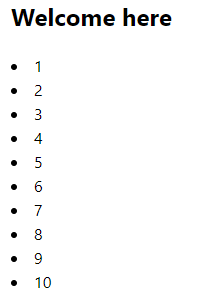
1. videos의 배열에 값이 있는 경우 (videoController.js)

export const trending = (req,res) => {

  const videos = [1,2,3,4,5,6,7,8,9,10]

  res.render('home', {pageTitle: 'Home', fakeUser: fakeUser, videos });

}



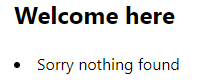
2. videos의 배열에 값이 없는 경우 (videoController.js)

export const trending = (req,res) => {

  const videos = []

  res.render('home', {pageTitle: 'Home', fakeUser: fakeUser, videos });

}



#5.9 Mixins

Mixins: 데이터를 받을 수 있는 partial. Smart partial.

Html을 return 해주는 일종의 function이다.

Mixins 폴더 생성 후 video.pug파일 생성.

Minxin 이름은 video, variable은 info로 설정.

mixin video(info)

  div

      h4=info.title

      ul

        li #{info.rating}/5

        li #{info.comments} comments

        li Posted #{info.createdAt}

        li #{info.views} views

home.pug

include를 이용해 mixin을 포함시킨 뒤 + 문법으로 video mixin 넣어준다

extends base

include mixins/video

block content

  h2 Welcome here

  each x in videos

    +video(x)

  else

    li Sorry nothing found

+video() 의 ()안에는 argument를 넣어준다. 여기에선 videos 배열의 각각의 객체 x를 argument로 넣어주고, 이 argument는 video mixin파일의 info와 같다.

아래의 정해놓은 html블럭 안에서 x.title / x.ratings / x.comments와 같이 각 비디오의 키에 따른 값을 가져올 수 있다.

#6.0

Fake database 만들기

videoController.js

let videos = [

  {

    title: 'First Video',

    rating: 5,

    comments: 2,

    createdAt: "2 minutes ago",

    views: 1,

    id: 1,

  },

  {

    title: 'Second Video',

    rating: 5,

    comments: 2,

    createdAt: "2 minutes ago",

    views: 100,

    id: 2,

  },

  {

    title: 'Third Video',

    rating: 5,

    comments: 2,

    createdAt: "2 minutes ago",

    views: 49,

    id: 3,

  },

]

export const trending = (req,res) => {

  return res.render('home', {pageTitle: 'Home', videos });

}

Trending에 있던 videos를 블록 바깥으로 빼 모든 변수가 이용 가능할 수 있도록 한다.

Home.pug를 렌더링 하는데, pageTitle과 videos 값을 넘겨준다.

mixin video(video)

  div

      h4

        a(href=`/videos/${video.id}`)=video.title

      ul

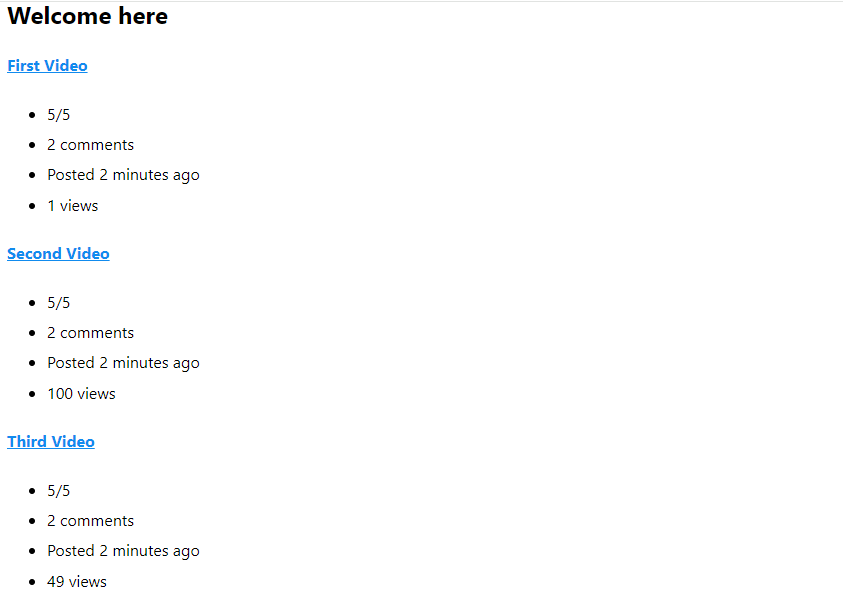
        li #{video.rating}/5

        li #{video.comments} comments

        li Posted #{video.createdAt}

        li #{video.views} views

video.pug에서는 각각의 비디오의 제목을 누르면 위의 경로로 페이지를 이동하도록 설정.



위의 경로는 watch 템플릿과 연결된다. Watch.pug

extends base.pug

block content

  h3 #{video.views} #{video.views === 1 ? 'view' : 'views'}

  a(href='/edit') Edit Video &rarr;

watch 템플릿에는 조회수와 Edit 페이지로 이동할 수 있는 버튼 출력

href 이용 시 **absolute url vs relative url**

ex) localhost4000/profile/profile-edit/password 이 페이지에서 다음의 링크 클릭 시

absolute url: href=’/yoyo -> localhost4000/yoyo 로 이동

relative url: href=’yoyo’ -> localhost4000/profile/profile-edit/yoyo 로 이동

‘/’ 유무에 따라 이동하는 경로가 다르다.

각 id에 해당하는 비디오 수정 페이지로 이동해야 하므로

extends base.pug

block content

  h3 #{video.views} #{video.views === 1 ? 'view' : 'views'}

  a(href=`${video.id}/edit`) Edit Video &rarr;

위와 같이 변경해준다. 앞에 / 를 붙여주지 않아야 videos/id/edit 경로로 이동할 수 있다.

일일이 앞의 url을 전부 적어주지 않아도 되어 유용하다.

#6.2 Edit Video part One

Edit 페이지 내용 설정하기

videoController.js

export const getEdit = (req,res) => {

  const {id} = req.params;

  const video = videos[id-1];

  return res.render('edit', {pageTitle: `Editing: ${video.title}`, video});

}

watch에서 했던 것과 같이, video의 id를 받고, 또한 id를 이용해 각각의 video 설정.

Edit.pug

extends base.pug

block content

  h4 Change Title of video

  form(method='POST')

    input(name='title', placeholder='Video Title', value=video.title, required)

    input(value='save', type='submit')

Edit에선 영상의 제목을 수정할 수 있는 폼을 생성한다.

수정한 내용을 제출해야 하는데, form의 action과 method를 지정함으로써 가능.

Action=/edit-content 와 같이 url을 넣어주면 데이터가 그 url로 이동한다.

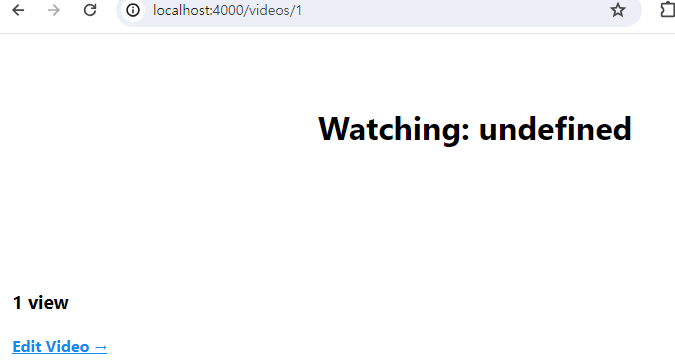
같은 url로 데이터를 받는다면 action 속성을 사용하지 않으면 된다.

Method: form과 backend 사이에서 정보 전송 방식 설정을 위한 속성

GET, POST 등을 지정해 줄 수 있는데, 데이터를 변경하기 위해서는 POST를 이용.

Method의 기본값은 GET으로, 따로 지정해주지 않으면 GET method가 되며 GET method를 이용할 시 그 내용(input의 name)이 url에 추가 된다.

Name: 아래 나오는 req.body를 사용하기 위해선 input의 속성으로 name을 꼭 지정해줘야 한다. 만약 name속성을 지정하지 않는다면 req.body는 undefined가 된다.



videoController.js

export const postEdit = (req,res) => {}

videoRouter.js

videoRouter.post('/:id(\\d+/edit', postEdit);

router에게 post request를 어떻게 이용할지 알려주기 위해 코드를 추가(postEdit)

한 url에 get과 post를 동시에 한다면,

videoRouter.get('/:id(\\d+)/edit', getEdit);

videoRouter.post('/:id(\\d+/edit', postEdit);

이 두 코드를 합쳐 다음과 같이 만들 수 있다.

videoRouter.route('/:id(\\d+)/edit').get(getEdit).post(postEdit);

express.urlencoded: Express의 middleware 중 하나로, body-parser모듈이 내장되어 있어 form 요소에서 submit을 할 때 전달된 데이터를 body 안으로 자동 파싱 해준다. 이는 router 이전에 위치해야 유효하다.

(req.body를 사용하기 위해서 express.urlencoded 사용)

form으로 제출되는 값은 x-www-form-urlencoded 형태로 들어오기 때문에 express.json() 설정으로는 값을 해석할 수 없어 해당 미들웨어를 사용해야한다.

옵션인 extended: true / false 설정을 꼭 해줘야 에러가 나지 않는데,

True: 해석하려는 데이터 방식이 x-www-form-urlencoded가 아닌 경우. qs 모듈 사용

False: 해석하려는 데이터 방식이 x-www-form-urlencoded인 경우. querystring모듈 사용

server.js

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

express app이 form의 value들을 이해해 자바스크립트 형식으로 변형해준다.

videoController.js

export const postEdit = (req,res) => {

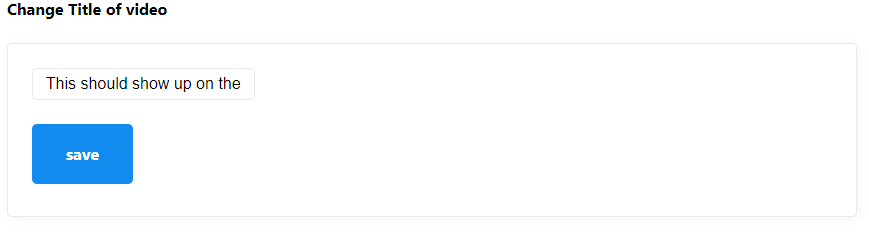
  const {id} = req.params;

  console.log(req.body);

  return res.redirect(`/videos/${id}`);

}

콘솔로 req.body값을 찍어보면, urlencoded를 사용하지 않았을 때엔 express가 form의 value를이해하지 못해 undefined로 나왔지만 urlencoded 사용 시 다음과 같이 찍힌다.



위와 같이 입력 후 save를 눌러주면 입력값이 콘솔에 찍힌다. Title은 input의 name



videoController.js를 다음과 같이 수정

export const postEdit = (req,res) => {

  const {id} = req.params;

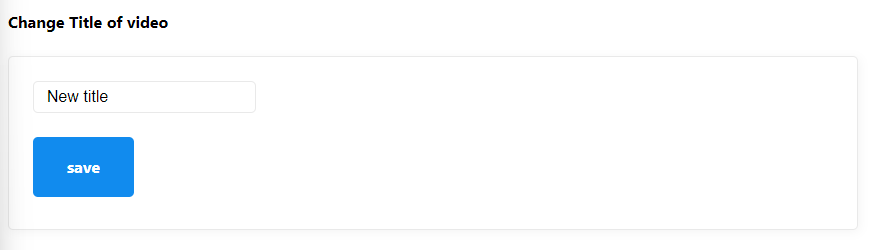
  const {title} = req.body;

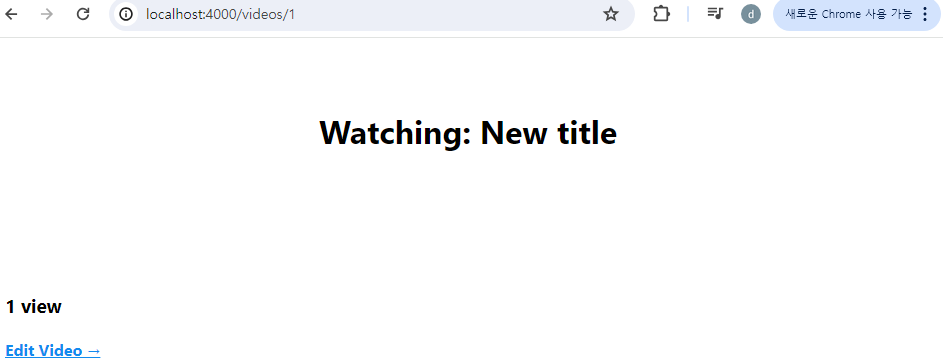
  videos[id-1].title = title;

  return res.redirect(`/videos/${id}`);

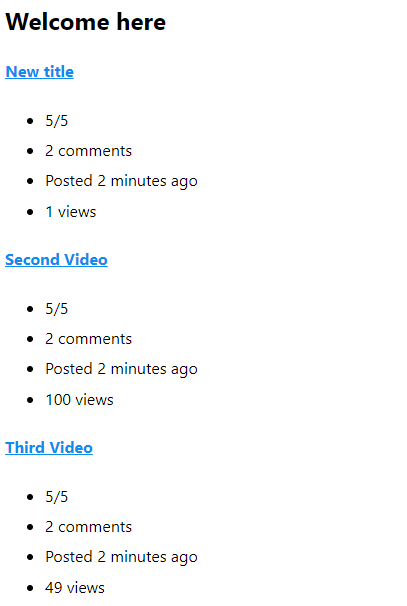
}

이후에 edit 페이지에서 새로운 title로 name을 변경하고 save를 클릭해보면





Title이 내가 입력한 New title로 바뀐 것을 볼 수 있다. 메인 페이지로 가 봐도 First Video였던 title이 New title로 변경되었다!



#6.5 More Practice part One

#6.7 Introduction to MongoDB

MongoDB: general purpose, document-based, distributed database

Document-based: 다른 DB들은 sql-based인 것 과는 다르게 json-like-document

#6.8 Connecting to Mongo

Mongoose: elegant mongodb object modeling for node.js

즉, node,js와 mongoDB를 잇는 다리 역할

MongoDB Community Server Download 진행(무료버전)

<https://www.mongodb.com/try/download/community>

version / platform / package를 설정한 뒤 다운로드 해 준다.

(나는 7.0.9(current) / Windows x64 / msi로 했다.) 설치된 msi 파일을 실행시킨 뒤 complete 버전으로 설치 완료

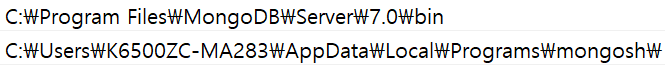
Mongo Shell(Mongosh) Download 진행

<https://www.mongodb.com/ko-kr/docs/mongodb-shell/install/>

위와 같이 진행.

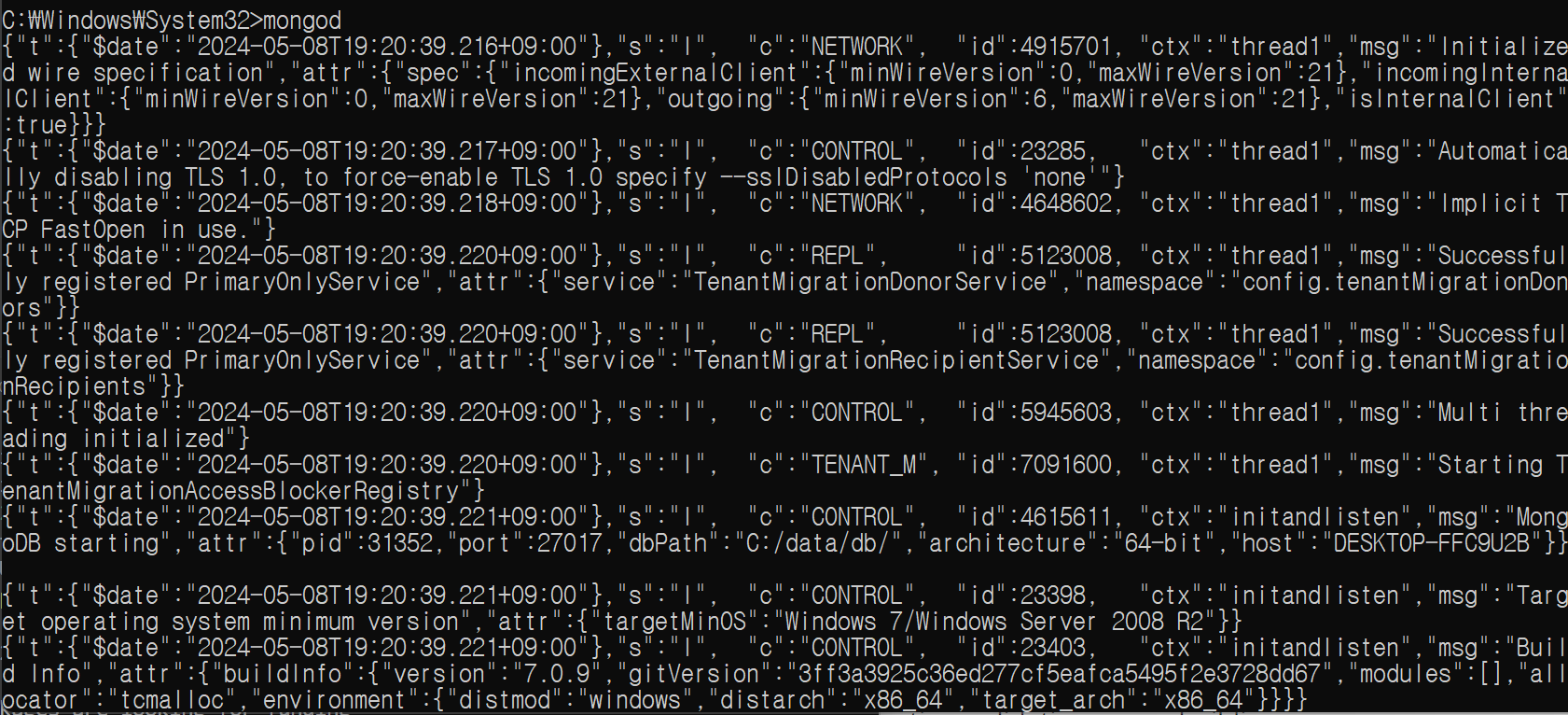
설치한 mongodb와 mongosh를 터미널로 이용할 수 있으려면 시스템 환경 변수에 추가해줘야 한다.

시스템 변수(S)의 Path 선택 후 편집을 누른 뒤 새로만들기 후 mongodb와 mongosh 경로 추가

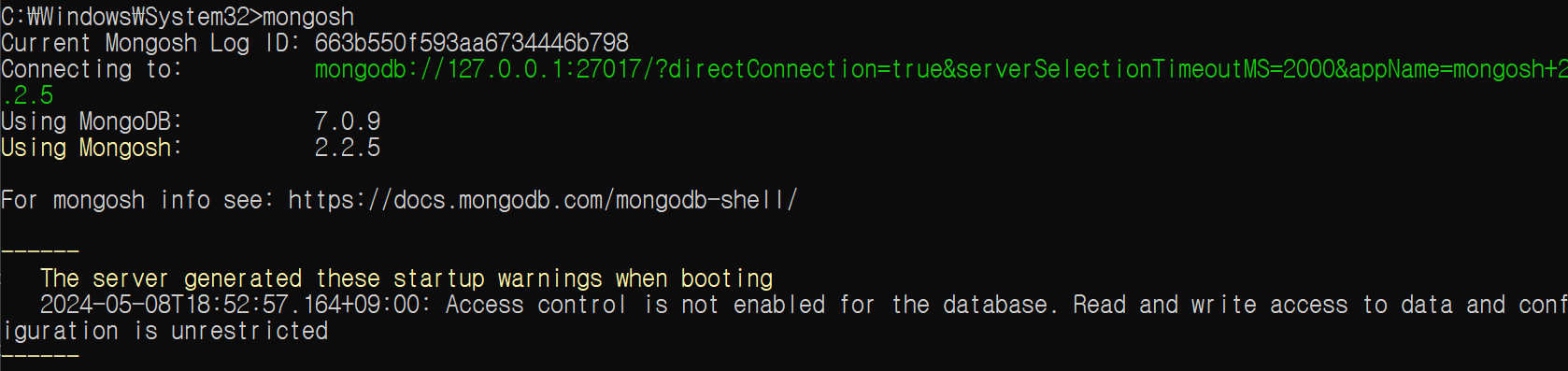


설치가 잘 되고, 환경변수도 잘 등록이 되었다면 관리자모드로 터미널을 열어 확인 가능하다.

-Mongod 명령어 입력해 내용이 나오는지 확인



-Mongosh 명령어 입력해 내용이 나오는지 확인



이후에 Mongodb://127.0.0.1:27017 이 url을 복사해서 사용해야 한다.

Mongoose 설치

Npm i mongoose 로 mongoose를 설치

Src 폴더에 db.js 파일을 생성한 후 다음의 코드 작성

import mongoose from 'mongoose';

mongoose.connect('mongodb://127.0.0.1:27017/wetube');

27017포트 뒤에 db 이름을 꼭 명시해 줘야 한다(wetube)

이 파일 자체를 server.js에서 import 해 주어 mongodb에 연결

import './db';

db.js 파일에 몇 줄의 코드를 추가하여 db가 잘 연결이 되었는지, 이벤트 발생시 잘 진행이 되는지 확인하기 위해 콘솔에 출력해준다.

import mongoose from 'mongoose';

mongoose.connect('mongodb://127.0.0.1:27017/wetube');

const db = mongoose.connection;

const handleOpen = () => console.log('✅ Connected to DB');

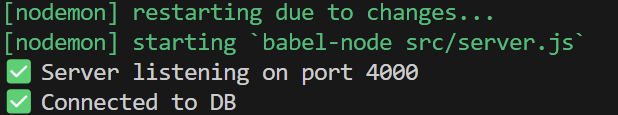
const handleError = (error) => console.log('❌ DB Error', error);

db.on('errer', handleError);

db.once('open', handleOpen);

우리는 server.js에 이 파일을 import 해줌으로써, 서버가 실행되면 자동으로 db를 연결시켜 준다.

그래서 서버가 실행될 때 터미널에 다음과 같은 구문이 출력된다



#6.9 CRUD Introduction

C: Create

R: Read

U: Update

D: Delete

Mongoose가 이해할 수 있도록 db 모델을 만들어 줘야 한다.

src폴더 안에 models 폴더를 생성 후 Video.js 파일 생성.

이 Video.js에서 영상이 가지는 데이터 종류와(제목, 작성자, 댓글 등) 타입(숫자, 문자 등)을 정의

import mongoose from 'mongoose';

const videoSchema = new mongoose.Schema({

  title: String,

  description: String,

  createdAt: Date,

  hashtags: [{ type: String }],

  meta: {

    views: Number,

    rating: Number,

  }

});

const Video = mongoose.model('Video', videoSchema);

export default Video;

------------파일 분리-----------------

Server.js: Express된 것과 server의 configuration 관련 코드들 처리

// express된 것과 server의 configuration 관련된 코드 처리

import express from "express";

import morgan from "morgan";

import globalRouter from "./routers/globalRouter";

import userRouter from "./routers/userRouter";

import videoRouter from "./routers/videoRouter";

const app = express();

const logger = morgan("dev")

app.use(logger);

app.set('view engine', 'pug');

app.set('views', process.cwd() + '/src/views');

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

app.use('/', globalRouter);

app.use('/videos', videoRouter);

app.use('/users', userRouter);

export default app;

Init.js: import된 것 모두 확인 후 서버 실행

// 필요로하는 정보들은 먼저 import 한 뒤 서버를 실행하는 파일

import './db';

import './models/Video';

import app from './server';

const PORT = 4000;

const handleListening = () => console.log(`✅ Server listening on port ${PORT}`);

app.listen(PORT, handleListening);

앞으로 둘을 나누어 파일을 관리한다.

Video 모델을 init에 import 해줌으로써 함께 import한 db와 연결이 된다.

Mongoose의 queries

<https://mongoosejs.com/docs/queries.html>

mongoose가 지원하는 Model.find()함수를 이용, videoController.js Video 모델을 찾는다

import Video from '../models/Video';

export const home = (req,res) => {

  Video.find({})

  .then((videos) => {

    console.log('videos', videos);

    return res.render('home', {pageTitle: 'Home', videos: [] });

  })

  .catch((error) => {

    console.log('error', error);

  })

}

Localhost4000에 접속해보면 콘솔에 아래의 내용이 찍힌다.



아무런 에러 없이 javascript를 통해 database와 연결 완료!

**※ Callback 함수와 promise 차이점**

**Callback 함수를 사용했을 때의 실행 순서**

export const home =(req,res) => {

  console.log('i start');

  Video.find({}, (error, videos) => {

    console.log('i finish');

  if(error) {

  return res.render('server-error');

  }

    return res.render('home', {pageTitle: 'Home', videos});

})

console.log('i finish first');

  };

1. i start

2. i finish first

3. I finish

4. render

=> JS는 위에서 아래로 순차적으로 코드가 실행되는데, Vedeo.find() 함수로 videos를 찾는 동안의 시간을 기다려 주지 않고 다음의 코드가 먼저 실행된다. 그래서 I finish first가 먼저 출력되고, videos를 찾고난 이후에 I finish 가 출력되고 페이지가 렌더링 되는 것.

**장점: 에러를 위한 추가적인 구문 없이 바로 에러 확인 가능**

**단점: 코드를 적은 순서와 실행 순서가 다름**

**Promise로 실행했을 때의 실행 순서**

export const home = async(req,res) => {

  try{

    console.log('i start');

    const videos = await Video.find({})

    console.log(videos);

    console.log('i finish');

    return res.render('home', {pageTitle: 'Home', videos});

  } catch(error) {

    return res.render('error page', {error});

  }

}

1. I start

2. [ ] (=videos)

3. I finish

4. render

Promise 방식을 사용할 경우 await 구문에서 Video.find()함수가 문서를 찾을 때까지 기다린 이후에 다음 줄의 코드로 넘어갈 수 있다. 그렇기 때문에 videos를 찾는 데 까지 시간이 걸리지만, 그동안 기다렸다가 다음 줄의 코드를 출력하기 시작. Videos는 위에서 이미 찾았으므로 바로 출력이 가능하게 되고 그 다음인 I finish가 출력된다.

만약 try 구문에서 에러 발생 시, try 구문을 실행하지 않고 바로 catch 블록으로 넘어가 에러 출력 코드 실행.

**장점: 코드를 적은 순서대로 코드 실행 가능. 직관적! 코드를 이해하기 쉬움**

**단점: 에러를 확인하기 위해 try – catch 구문을 따로 사용해야 함**

# 6.15 Creating a Video part One

우리가 만든 Schema에 맞게 postUpload 부분의 코드를 수정해 document를 만들어 준다.

videoController.js

export const postUpload = (req,res) => {

  const {title, description, hashtags} = req.body;

  const video = new Video({

    title,

    description,

    createdAt: Date.now(),

    hashtags: hashtags.split(',').map((word) => `#${word}`),

    meta: {

      views:0,

      rating:0,

    }

  });

  console.log(video);

  return res.redirect('/');

}

유저에게 데이터를 받을 항목들도 새로 생성해준다.

Upload.pug

extends base

block content

  form(method='POST')

    input(name='title', placeholder='Title', required, type='text')

    input(name='description', placeholder='Description', required, type='text')

    input(name='hashtags', placeholder='Hashtags, separated by comma', required, type='text')

    input(type='submit', value='Upload Video')

여기에서 hashtag를 형성하는 방법!

**Split() 과 map() 함수**

Split(‘separator’): 한 문자열 안에서 특정 separator(구분자) 만나면 문자열을 나누어 배열로 저장

이것은 해쉬태그를 쉼표로 구분해 작성하면 각각의 해쉬태그를 배열로 저장해준다

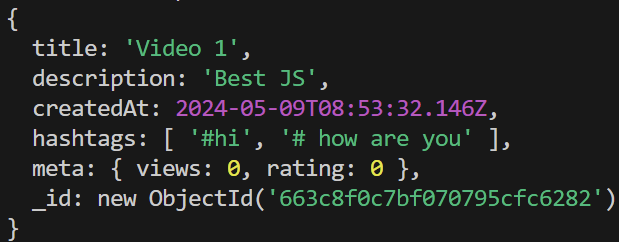


Map(each things => each things you want): 배열 안 elements를 순회하며 원하는 모양으로 만듦.

작성한 해쉬태그 단어들 앞에 #를 자동으로 붙여주는 작업을 쉽게 해줄 수 있다.



위와 같이 설정 후 새로고침 해 보면 콘솔에 video의 내용이 출력된다.



우리가 설정한 것 외에 mongoose가 임의로 부여한 id도 존재한다.

이 값은 JS에 저장되어 있지만, database에는 저장되지 않았다.

Database에 저장하기

만약 우리가 모델로 설정해놓은 데이터 형식이 아닌 다른 형식의 데이터를 넣으려고 한다면?

예를 들어, 다음과 같이 views 항목에 Number가 아닌 String을 입력한다면

const video = new Video({

    title,

    description,

    createdAt: Date.now(),

    hashtags: hashtags.split(',').map((word) => `#${word}`),

    meta: {

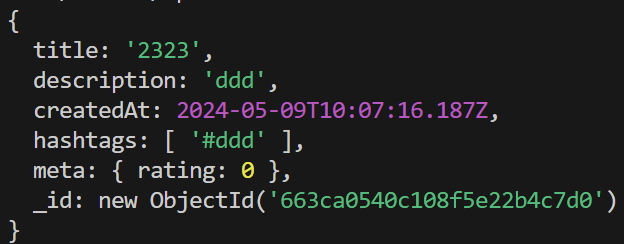
      views:'dd',

      rating:0,

    }

  });

아래와 같이 views의 정보가 나오지 않는다. 즉, mongoose가 데이터 타입의 유효성 검사를 하여 document에 유효하지 않은 값이 들어가는 것을 막아준다.



유효한 정보로만 db에 값을 넣는다면? 이제 정말 db에 저장할 수 있게 해보자

VideoController.js

export const postUpload = async(req,res) => {

  const {title, description, hashtags} = req.body;

  const video = new Video({

    title,

    description,

    createdAt: Date.now(),

    hashtags: hashtags.split(',').map((word) => `#${word}`),

    meta: {

      views:0,

      rating:0,

    }

  });

  const dbVideo = await video.save();

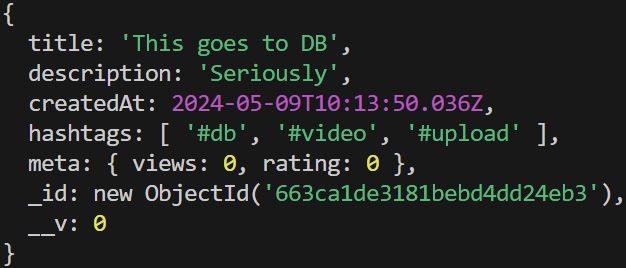
  console.log(dbVideo)

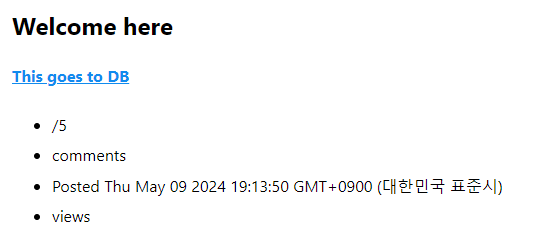
  return res.redirect('/');

}

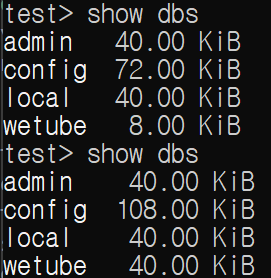
Mongoose에서 제공하는 video.save()함수를 이용해 db에 저장. 단, db에 데이터가 저장되는 데에는 시간이 걸리기 때문에 async – await 구문을 이용하여 저장하는 것을 기다렸다 다음 코드를 실행할 수 있도록 해준다.

새로고침 후 입력값들을 알맞게 넣어주면 콘솔에 다음과 같이 찍히고, 드디어 데이터가 저장되어 홈으로 돌아가보면 저장된 데이터에 대한 정보가 출력된다.





이 데이터는 database에 저장되었으므로 서버를 다시 실행해도 사라지지 않는다! 감격!

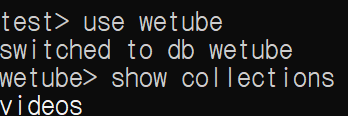


또한, show dbs 명령어로 mongosh에서 확인을 해 보면, 데이터 저장 이전에 8KiB였던 용량이 40KiB로 증가한 것을 볼 수 있다. 데이터가 저장되어 용량이 늘어난 것.

Use wetube 명령어로 사용 database를 wetube로 바꾼 뒤 show collections 명령어를 입력하면 현재

Show collections 명령어를 실행해보면

Collection: group of documents. Mongoose는 모델을 찾아 **모델의 이름을 소문자로 변경+s(복수형)을 붙여 collection 생성**. 그러므로 모델 이름은 Video이지만 컬랙션의 이름은 videos가 된다.



**Video.create()**

위에서 new Object … await Model.save() 를 해 준 것과 같은 효과.

즉 앞에서는 new Object로 class를 생성 후 마지막에 따로 save를 해 줘야 했다면 creaete()를 이용하면 생성과 저장을 한 번에 해결해준다. 더 효율적인 코드가 되는 것.(await와 함께 사용)

export const postUpload = async(req,res) => {

  const {title, description, hashtags} = req.body;

  await Video.create({

    title,

    description,

    createdAt: Date.now(),

    hashtags: hashtags.split(',').map((word) => `#${word}`),

    meta: {

      views:0,

      rating:0,

    }

  });

  return res.redirect('/');

}

또한 데이터가 형식에 맞지 않거나 required 설정을 했는데 데이터를 빼먹은 경우 등 다양한 에러가 발생할 수 있는 상황에 대비해 에러를 catch해야한다.

Await 구문에서 에러가 발생하는데 에러를 catch 하는 구간이 없다면 영원히 await에 머무르며 기다린다. 어떤 에러가 발생 했는지도 알 수 없다. 그러므로 try – catch 구문을 넣어준다

export const postUpload = async(req,res) => {

  const {title, description, hashtags} = req.body;

  try{

    await Video.create({

      title,

      description,

      createdAt: Date.now(),

      hashtags: hashtags.split(',').map((word) => `#${word}`),

      meta: {

        views:0,

        rating:0,

      }

    });

    return res.redirect('/');

  } catch(error) {

    console.log(error);

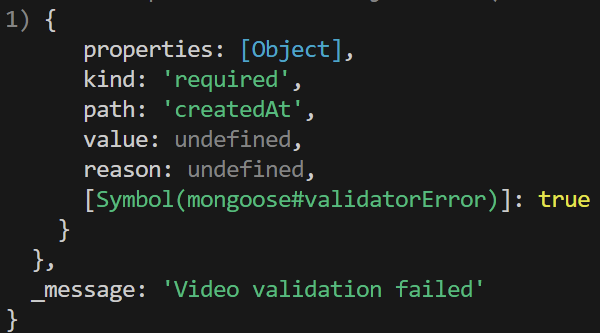
    return res.render('upload', {pageTitle: "Upload Video"});

  }

}

createdAt 부분에 required 설정을 해 놨는데, 고의로 createdAt 부분을 빼먹어봤더니 콘솔에 에러가 출력되고 /upload 페이지를 렌더링 해준다.





또한 유저에게 에러가 났음을 알리기 위해 에러메세지를 출력해준다.

catch(error) {

    return res.render('upload', {

      pageTitle: "Upload Video",

      errorMessage: error.\_message,});

  }

이 부분에 errorMessage도 함께 전달 => \_message는 mongoose에서 준 message인데, 이를 이용해서 똑같이 message를 출력하도록 이용했다.

Upload.pug 에 에러메세지를 출력하도록 설정

block content

  if errorMessage

    span=errorMessage

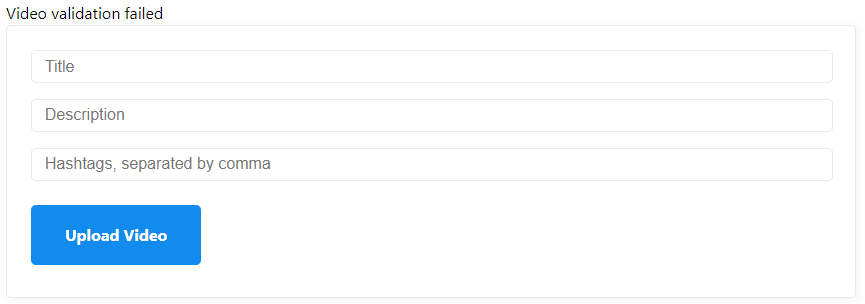
  form(method='POST')

    input(name='title', placeholder='Title', required, type='text')

    input(name='description', placeholder='Description', required, type='text')

    input(name='hashtags', placeholder='Hashtags, separated by comma', required, type='text')

    input(type='submit', value='Upload Video')



에러를 내자 위에 ‘Video validation failed’ 문구가 출력.

모델에 Default 설정 하기

Video 모델에는 createdAt(생성일)이 생성한 그 시간으로 부여되도록 코드를 설정했었다.

그런데 이런 자동 생성되는 항목은 따로 videoController의 new Video 항목에 적을 필요 없이 모델 설정에서 한번에 해결할 수 있다.

Models 폴더의 Video.js

const videoSchema = new mongoose.Schema({

  title: String,

  description: String,

  createdAt: {type: Date, required: true, default: Date.now},

  hashtags: [{ type: String }],

  meta: {

    views: Number,

    rating: Number,

  }

});

videoController.js 에서는 createdAt 항목을 빼 줄 수 있다. Default로 설정했으므로 영상을 업로드 할 때 자동으로 데이터가 들어가진다.

await Video.create({

      title,

      description,

      hashtags: hashtags.split(',').map((word) => `#${word}`),

      meta: {

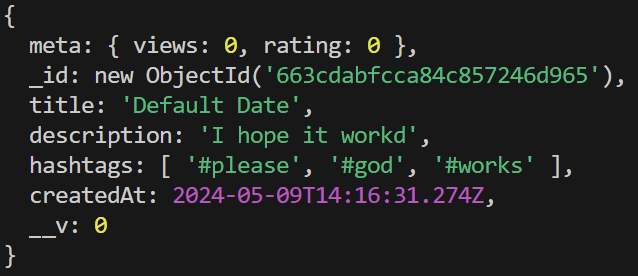
        views:0,

        rating:0,

      }

    });

콘솔을 확인해보면 에러 없이 생성일이 잘 들어가는 것을 볼 수 있다.



Required, default 외의 다른 옵션

String 타입에는 maxLength와 minLength 설정 가능. Html의 input에도 maxlength와 minlength를 설정할 수 있다. Html과 database 모두 이 설정을 하는 것이 좋다.

html에서 1차적으로 길이 제한을 받아 보안을 할 수 있지만, 사이트를 해킹하려는 사람이나 콘솔을 열어 html을 삭제 또는 수정하여 데이터를 넣으려는 사람이 있을 수 있다. 이 때 2차적으로 database측에 걸어놓은 길이 제한으로 이를 막을 수 있다.

모델 설정 시에는 필요한 옵션을 구체적으로 많이 적어줄수록 보안 등 여러가지 측면에서 도음을 받을 수 있으니 웬만하면 많이 적어준다.

Video.js

const videoSchema = new mongoose.Schema({

  title: {type: String, required: true, trim:true, maxLength: 80},

  description: {type: String, required: true, trim:true, minLength: 20},

  createdAt: {type: Date, required: true, default: Date.now},

  hashtags: [{ type: String, trim:true }],

  meta: {

    views: {type: Number, default: 0, required: true},

    rating: {type: Number, default: 0, required: true},

  }

});

라우터의 정규표현식 변경해주기

mongoose에서 각 Video documents에 부여해주는 id는 24자의 16진수이다(0~9 + a~f).

우리는 이전에 url에 사용되는 id의 값을 정수만 가능하도록 제한했었는데 이를 24자 16진수에 맞춰 바꿔주어야 한다.

videoRouter.js

videoRouter.get('/:id([0-9a-f]{24})', watch);

videoRouter.route('/:id(\\d+)/edit').get(getEdit).post(postEdit);

id를 이용해 해당 video 찾기

위와 같이 라우터에서 설정한 :id 가 있기 때문에 이를 이용해 찾을 수 있다.

해당 비디오 페이지로 이동하기 위해 누르면 비디오 id가 url에 표시되는데, 이 id가 req.params에 전송이 되고 video 검색. Await 하며 mongoose 지원 함수인 findById()로 해당 id를 찾고, 검색이 끝나면 video가 불려진다.

만약, 존재하지 않는 id를 입력하여 페이지에 접속한다면?

존재하지 않는 id 입력하여 페이지 접속 => findById()함수가 해당 id를 검색하지 못 함 => 찾을 수 없는 video이 값은 null이 됨 => watch 템플릿에서는 video.title을 요구하고 있는데 해당 video는 null이 되었고, 그로 인해 title을 찾을 수 없어 에러 발생.

해결: if구문을 추가해 비디오를 찾을 수 없으면 404템플릿을 반환하도록 설정

export const watch = async(req, res) => {

  const {id} = req.params;

  const video = await Video.findById(id);

  if(video) {

    return res.render('watch', {pageTitle: video.title, video});

  }

  return res.render('404', {pageTitle: 'Video not found'});

}

Edit.pug 의 해시태그 포맷 설정하기

extends base.pug

block content

  h4 Change Title of video

  form(method='POST')

    input(name='title', placeholder='Title', value=video.title, required, type='text', maxlength=80)

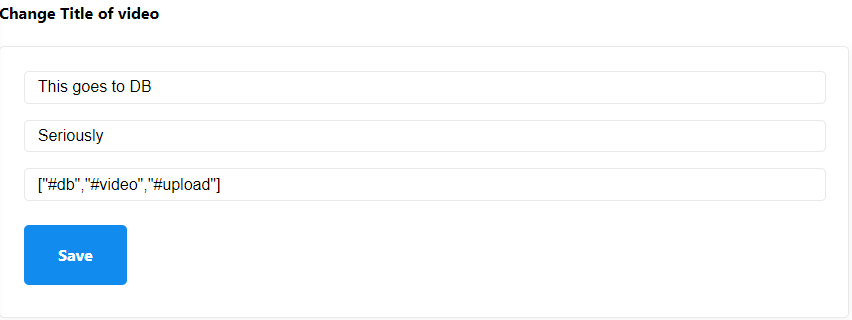
    input(name='description', placeholder='Description', value=video.description, required, type='text', minlength=20)

    input(name='hashtags', placeholder='Hashtags, separated by comma', value=video.hashtags, required, type='text')

    input(value='Save', type='submit')

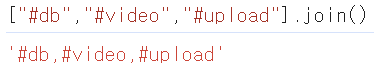
위와 같이 수정 페이지에서는 이전에 저장된 데이터를 불러와 value로 출력한다.

하지만 해시태그의 경우 array로 저장되도록 했었다. 그래서 다음과 같이 출력된다



유저가 보기에 좋지 않으므로 배열을 없애야 하는데 그 때 join()함수를 사용.

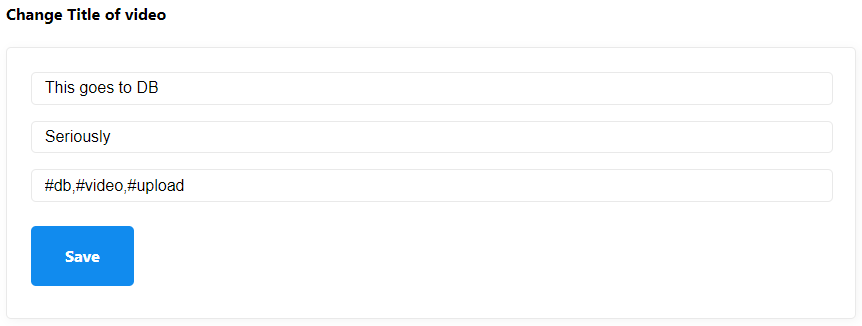
배열을 하나의 string으로 만들어준다.



이를 활용해 hashtags 부분의 value 값을 다음과 같이 변경

input(name='hashtags', placeholder='Hashtags, separated by comma', value=video.hashtags.join(), required, type='text')

이후에 페이지를 새로고침하면 다음과 같이 배열 없이 잘 나온다.



업데이트시 반환할 해시태그 함수 변경하기

현재 해시태그를 받아내는 함수

video.hashtags = hashtags.split(',').map(word => `#${word}`);

하지만 이대로 업데이트 페이지에서 해시태그를 위의 방법으로 받는다면 해시태그마다 #이 하나씩 더 증가하게 될 것(#맛집 -> ##맛집)

그러므로 단어마다 앞에 해시태그가 이미 있으면 word를 그대로 반환하고, 없다면 앞에 #을 붙이는 함수로 변경

video.hashtags = hashtags.split(',').map((word) => word.startsWith('#') ? word : `#${word}`);

postEdit은 다음과 같이 설정할 수 있다

export const postEdit = async(req,res) => {

  const {id} = req.params;

  const {title, description, hashtags} = req.body;

  const video = await Video.findById(id);

  if(!video) {

    return res.render('404', {pageTitle: 'Video not found'});

  }

  video.title = title;

  video.description = description;

  video.hashtags = hashtags.split(',').map((word) => word.startsWith('#') ? word : `#${word}`);

  await video.save();

  return res.redirect(`/videos/${id}`);

}

Req.params와 req.body를 통해 필요한 데이터를 찾고, id를 통해 해당 video를 찾은 뒤 비디오 항목마다 새롭게 바뀐 데이터로 변경해준 뒤, video를 database에 다시 저장하는 과정을 거친다.

이 과정이 많이 반복되거나, 데이터 항목이 많거나, 그렇지 않아도 더 짧은 코드로 작성할 수 있기 때문에 이는 다음과 같은 코드로 짧게 변경할 수 있다

findByIdAndUpdate() 함수 사용

export const postEdit = async(req,res) => {

  const {id} = req.params;

  const {title, description, hashtags} = req.body;

  const video = await Video.findById(id);

  if(!video) {

    return res.render('404', {pageTitle: 'Video not found'});

  }

  await Video.findByIdAndUpdate(id, {

    title,

    description,

    hashtags: hashtags.split(',').map((word) => word.startsWith('#') ? word : `#${word}`),

  })

  return res.redirect(`/videos/${id}`);

}

Video.title = title 과 같이 길게 표현되던 부분들을 title 과 같이 짧게 표현 가능하며 save()를 굳이 써 줄 필요가 없어진다.

**Video.exists( filter )**

위의 코드에서는 먼저 id를 통해 video object를 찾고, 이후에 업데이트를 진행한다.

하지만 이 postEdit에서는 video object를 필요로 하지 않는다(이후 코드에서 쓰이지 않음)

그래서 첫번째 Video.findById(id)를 Video.exists({\_id: id})로 변경해준다.

Video.exists(property=filter) 는 원하는 데이터로 검색을 해서(filter) 그 video가 존재하는지 true/false로 반환해준다. 활용 시 제목 또는 원하는 항목으로 video를 검색할 수 있게 된다.

export const postEdit = async(req,res) => {

  const {id} = req.params;

  const {title, description, hashtags} = req.body;

  const video = await Video.exists({\_id: id});

  if(!video) {

    return res.render('404', {pageTitle: 'Video not found'});

  }

  await Video.findByIdAndUpdate(id, {

    title,

    description,

    hashtags: hashtags.split(',').map((word) => word.startsWith('#') ? word : `#${word}`),

  })

  return res.redirect(`/videos/${id}`);

}

#6.23 Middlewares

Express의 middleware: request 중간에 가로채서 뭔가를 하고 이어서 진행. 흐름 방해

Mongoose의 middleware: 무언가 하기 전, 후로 뭔가를 하고 이어서 진행. 흐름을 방해하지 않음

미들웨어는 모델 생성 전에 만들어져야 한다.

Video.js 에 다음 코드 추가

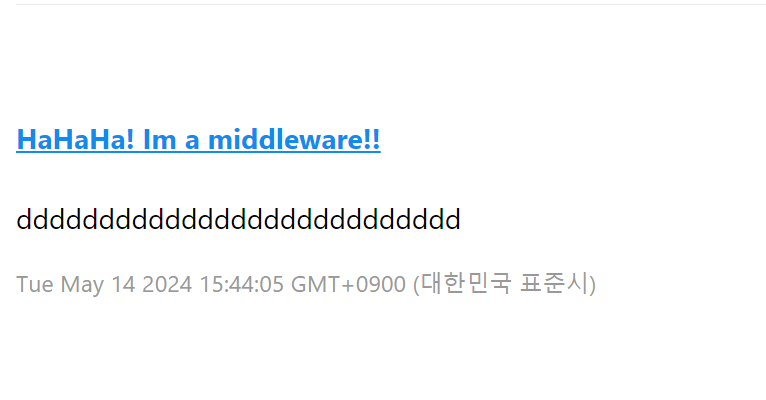
videoSchema.pre('save', async function() {

  console.log('We are about to save: ',this);

  this.title = 'HaHaHa! Im a middleware!!'

})

유저가 save를 눌러 database에 데이터를 저장하기 직전, 위의 미들웨어로 인해 그 title이 유저가 설정한대로 저장되지 않고 위에 입력한대로 바뀌어 제목이 저장된다.



다른 예

videoSchema.pre('save', async function() {

  this.hashtags = this.hashtags[0].split(',').map(word => word.startsWith('#') ? word : `#${word}`)

})

이렇게 지정해 준다면 hashtag가 저장되기 전에 우리가 지정한 대로 저장이 된다.

하지만 이 방법은 비디오를 업로드 할 때(save 시)는 적용이 가능하지만 수정 후 업데이트 할 때엔 접근할 수 없다. 그래서 미들웨어를 사용하는대신 다른 방법으로 해시태그를 적용해보자

첫번째 방법. formatHashtags 함수를 만들어 적용하기

Video.js에 함수를 만든 뒤 export, VideoController.js에 import 하여 함수 사용

export const formatHashtags = (hashtags) => hashtags.split(',').map((word) => (word.startsWith('#') ? word : `#${word}`));

import Video, { formatHashtags } from '../models/Video';

export const postEdit = async(req,res) => {

  const {id} = req.params;

  const {title, description, hashtags} = req.body;

  const video = await Video.exists({\_id: id});

  if(!video) {

    return res.render('404', {pageTitle: 'Video not found'});

  }

  await Video.findByIdAndUpdate(id, {

    title,

    description,

    hashtags: formatHashtags(hashtags),

  })

  return res.redirect(`/videos/${id}`);

}

두번째 방법. mongoose의 static을 이용해 모델 관련 함수 만들기

Video.js에 static을 확용해 hashtags 관련 함수를 생성한 뒤 videocontroller.js에서 사용

videoSchema.static('formatHashtags', function(hashtags) {

  return hashtags.split(',').map((word) => (word.startsWith('#') ? word : `#${word}`));

})

export const postEdit = async(req,res) => {

  const {id} = req.params;

  const {title, description, hashtags} = req.body;

  const video = await Video.exists({\_id: id});

  if(!video) {

    return res.render('404', {pageTitle: 'Video not found'});

  }

  await Video.findByIdAndUpdate(id, {

    title,

    description,

    hashtags: Video.formatHashtags(hashtags),

  })

  return res.redirect(`/videos/${id}`);

}

장점은 static을 이용할 경우 모델을 객체로 하여 내가 원하는 함수를 얼마든지 생성할 수 있고, 다른 파일에 import 혹은 export 할 필요가 없다.

#6.25 Delete Video

Router와 Controller를 만들어 준다.

videoRouter.route('/:id([0-9a-f]{24})/delete').get(deleteVideo);

export const deleteVideo = async(req,res) => {

  const {id} = req.params;

  await Video.findByIdAndDelete(id);

  return res.redirect('/');

}

Model.findByIdAndDelete(id) 는 findOneAndDelete({ \_id: id })의 약어이다.

원래 mongoose는 id를 \_id형식으로 표시하며 데이터타입이 객체인데, findOneAndDelete를 사용하면 \_id: id 로 객체를 다시 String으로 변환해주는 작업을 해야한다.

번거로움을 덜기 위해 findByIdAndDelete(id)로 줄여서 표현.

~~또한 비슷한 remove가 있는데, remove는 웬만해선 이용하지 말라고 mongoose에 나와있다.~~

Remove는 mongoose 공식 문서에서 아예 사라졌다..!

#6.26 Search part One

Home에 있는 비디오들을 정렬하기

export const home = async(req,res) => {

  const videos = await Video.find({}).sort({createdAt: 'desc'})

    return res.render('home', {pageTitle: 'Home', videos});

}

Video.find({}).sort({원하는항목: ‘desc’ 또는 1, ‘asc’또는 -1}) 으로 정렬 가능

위의 경우 최신순으로 정렬하기 위해 createdAt: ‘desc’ 로 작성

검색을 하기 위한 폼 생성

globalRouter.js에 추가

globalRouter.get('/search', search);

videoController.js에 일단 다음 코드 추가

export const search = (req,res) => {

  console.log(req.query);

  return res.render('search', {pageTitle:'Search'});

}

Search.pug 생성

extends base

block content

  form(method='GET')

    input(name='keyword', placeholder='Search by title', type='text')

    input(type='submit', value='Search now')

검색을 위한 폼 생성.

Input에 name을 입력해주면 그 값이 url로 들어가게 된다. 이 정보를 통해 req.query로 접근 할 수 있다.

예를 들어, 검색을 위한 input name을 keyword라고 위와 같이 설정한다면 검색어 입력 후 ‘Search now’ 버튼을 클릭했을 때 내가 입력한 검색어 ‘videos’가 url에 나오는 것을 볼 수 있고, 콘솔로 찍어보면 req.query 값은 다음과 같이 나온다.

 / 

여기에서, 검색페이지로 막 들어갔을 때의 keyword는 undefined가 되고, 검색어를 입력 후 버튼을 눌렀을 때만 keyword가 부여가 된다. 이런 점을 이용해 검색 기능을 만들어보자

videoController.js

export const search = async(req,res) => {

  const {keyword} = req.query;

  let videos = [];

  if (keyword) {

    videos = await Video.find({

      title: keyword,

    });

  }

  return res.render('search', {pageTitle:'Search', videos});

}

if문 안의 videos에는 const 설정을 하면 안 된다! 그렇게 되면 해당 블록에서만 작용하는 videos가 되고, search 페이지를 처음 들어갔을 때 videos는 undefined가 되어 에러가 난다.

처음에는 let으로 videos를 빈 배열로 선언하여 search 페이지에 처음 들어갔을 때 videos가 없어도 오류가 안 나도록 하고 검색된 keyword에 따라 비디오의 배열이 바뀌어도 상관 없도록 한다.

Search.pug : Mixin을 활용해 검색된 video를 반환 할 수 있게 한다.

extends base

include mixins/video

block content

  form(method='GET')

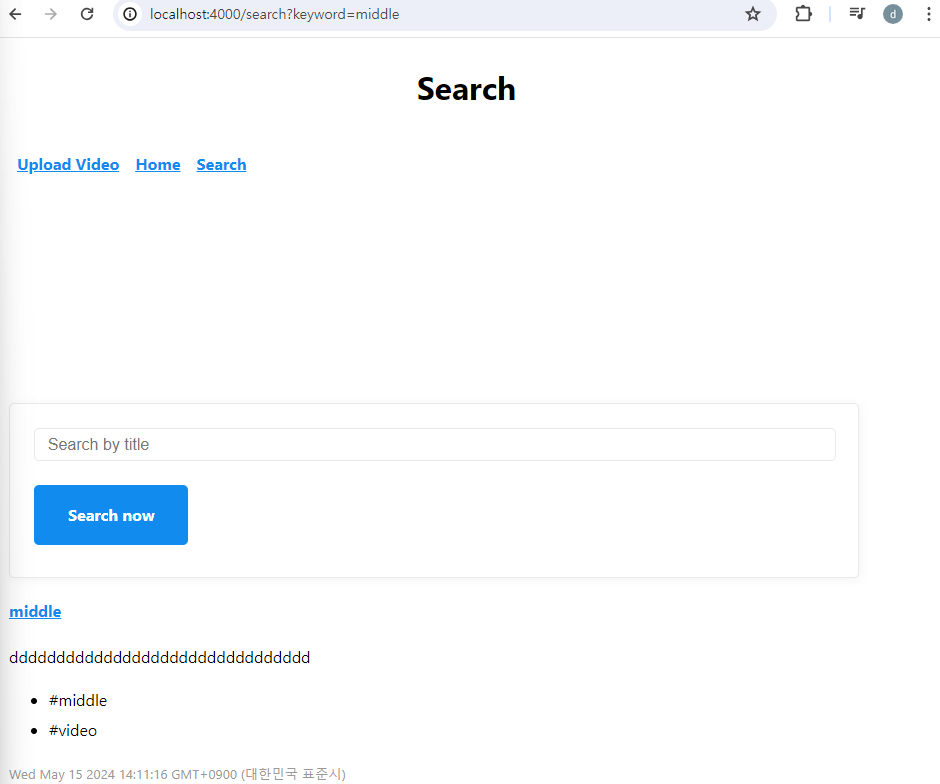
    input(name='keyword', placeholder='Search by title', type='text')

    input(type='submit', value='Search now')

  each video in videos

    +video(video)

‘middle’ title로 검색했더니 검색한 middle 영상만 잘 나온다.



일부의 단어만 일치해도 검색이 될 수 있도록 검색 옵션 지정하기

mongoDB의 다양한 operators 사용

(<https://www.mongodb.com/docs/manual/reference/operator/query/regex/> )

I: ignore case(대소문자 무시)

g: global

welcome$: welcome이 마지막에 오는 단어만 검색

^welcome: welcome으로 시작하는 단어만 검색

$Regex: regular expression의 줄임말

$gt: greater than => gt3: 3보다 큰

export const search = async(req,res) => {

  const {keyword} = req.query;

  let videos = [];

  if (keyword) {

    videos = await Video.find({

      title: {

        $regex: new RegExp(keyword, 'i') //keyword를 포함하는 title 검색

      },

    });

  }

  return res.render('search', {pageTitle:'Search', videos});

}

Keyword로 시작하는 title 검색: $regex: new RegExp(`^${keyword}, 'i')

Keyword로 끝나는 title 검색: $regex: new RegExp(`${keyword}$`, 'i')

#7.0 Create Account part One

Join 페이지 만들기

models폴더에 User.js 파일 생성

import mongoose from 'mongoose';

const userSchema = new mongoose.Schema({

  email: {type: String, required: true, unique: true},

  username: {type: String, required: true, unique: true},

  password: {type: String, required: true},

  name: {type: String, required: true},

  location: String,

});

const User = mongoose.model('User', userSchema);

export default User;

rootRouter.js

rootRouter.route('/join').get(getJoin).post(postJoin);

userController.js

export const getJoin = (req,res) => res.render('Join', {pageTitle: 'Create Account'});

export const postJoin = (req,res) => {

  console.log(req.body);

  res.end();

}

Join.pug

extends base

block content

  form(method='POST')

    input(name='name', placeholder='name', type='text', required)

    input(name='email', placeholder='email', type='email', required)

    input(name='username', placeholder='username', type='text', required)

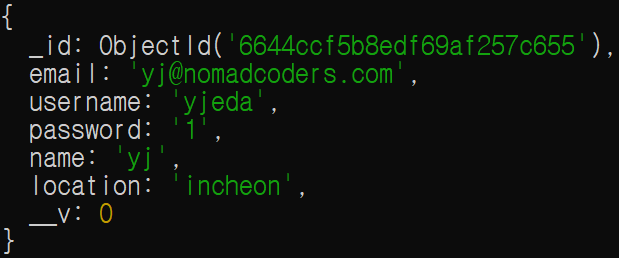
    input(name='password', placeholder='password', type='password', required)

    input(name='location', placeholder='location', type='text', required)

    input(type='submit', value='join')

join 페이지에서 정보를 입력 후 버튼을 클릭하면 db에 유저 정보가 저장된다.

하지만 db.user.find() 로 users에 저장된 데이터를 보면 password가 그대로 노출된 것 확인.



이 때, 비밀번호를 db에 저장하기 전 미들웨어를 이용해 비밀번호 해싱 후 저장되도록 설정

Password 암호화 중 hash(해시화): 일방향 함수로, 문자열을 특정 길이의 문자열로 변환해준다.

노마드코더 해시함수 5분 설명 => (<https://www.youtube.com/watch?v=67UwxR3ts2E>)

일방향 함수: 입력값으로 출력값을 알 수 있지만, 출력값으로 입력값을 알 수 없다. 또한 같은 입력값은 항상 같은 출력값(해시값)이 나온다.(=deterministic function, 결정적 함수)

bcrypt 사용: 해싱된 패스워드로 해커가 패스워드를 해킹할 수 있는 레인보우 테이블 공격법을 막아주는 라이브러리.

npm I bcrypt 로 설치

bcrypt.hash(plainPassword, saltRounds, function(err, hash) { }

saltRounds: 해싱을 한 결과값을 가지고 또 해싱하기를 반복하여 해커가 비밀번호를 유추하기 더 어렵게 만든다. 해싱할 횟수 입력

User.js에 미들웨어 생성

// this는 Users가 create될 때 유저가 입력한 value

userSchema.pre('save', async function() {

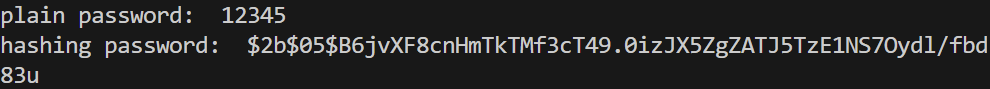
  console.log('plain password: ', this.password);

  this.password = await bcrypt.hash(this.password, 5);

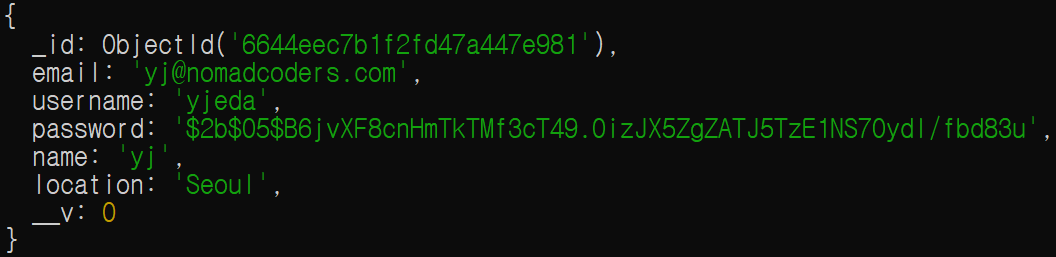
  console.log('hashing password: ', this.password);

})

이후 유저 정보를 입력하고 join을 한 뒤 콘솔을 확인해보면 해싱 전 후 바뀐 것을 볼 수 있다.



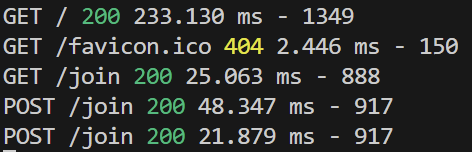
Db를 확인 해 봤을 때도 해싱된 비밀번호로 저장되어 있는 것 확인



#7.4 Status Codes

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#client_error_responses>

지금까지 페이지가 새로 로드되거나 이벤트가 발생 할 경우 logger에 의해 메시지를 받아왔다.



여기에 나오는 숫자들(200, 404 등)이 의미하는 것?

상태 코드(HTTP response Status Code): 특정 HTTP request가 성공적으로 완료되었는지 판단하기 위해 사용되는 코드. 이는 응답 형태에 따라 5가지 그룹으로 나뉜다.

|  |  |
| --- | --- |
| 1XX (Informational, 정보제공) | 임시 응답. 현재 클라이언트의 요청까지는 처리되었으니 계속 진행하라는 의미 |
| 2XX (Success, 성공) | 클라이언트의 요청이 서버에서 성공적으로 처리됨 |
| 3XX (Redirection, 리다이랙션) | 완전한 처리를 위해 추가 동작이 필요한 경우. 주로 서버의 주소 또는 요청한 uri의 웹 문서가 이동되었으니 그 주소로 다시 시도하라는 의미 |
| 4XX (Client Error, 클라이언트 에러) | 없는 페이지를 요청하는 등 클라이언트의 요청 메시지 내용이 잘못된 경우 의미 |
| 5XX (Server Error, 서버 에러) | 서버 사정으로 메시지 처리에 문제 발생. 서버 부하, DB 처리 과정 오류, 서버 인셉션 발생 등 |

200: OK

구글 브라우저는 id와 password가 들어간 form을 제출하면 방금 제출한 id와 password를 저장할 것이냐는 메시지를 띄운다. 하지만 틀린 id와 password를 입력해도 id와 password를 저장할 것이냐는 메시지를 띄우는데 (현재는 뜨지 않는 것 같다) 이는 상태코드 때문.

Res.render로 페이지를 렌더링 할 경우 상태코드 200을 응답으로 받는다. 그러므로

Form을 제출하면 잘못 된 정보라도 상태코드 200을 응답으로 받게된다. 200은 OK 라는 의미를 지니기 때문에 브라우저는 계정이 잘 생성된 줄 알고 메시지를 보내는 것. 그러므로 계정 생성에 실패했을 때는 다른 상태코드를 응답으로 받아야 브라우저도 이를 알고 메시지를 보내지 않을 것.

400: Bad Request

브라우저는 잘 실행됐다고 판단하는 2XX 상태코드를 받으면 url 히스토리에 남기고, 그렇지 않으면 히스토리에 남기지 않는다. 그러므로 정확한 상태코드를 반환 해 주는 것이 필요하다.

#7.5 Login part One

getLogin과 postLogin 생성

userController.js

export const getLogin = (req,res) => res.render('login', {pageTitle: 'Login'});

login.pug

extends base

block content

  if errorMessage

    span=errorMessage

  form(method='POST')

    input(name='username', placeholder='username', type='text', required)

    input(name='password', placeholder='password', type='password', required)

    input(type='submit', value='login')

  hr

  div

    span Don't have an account?

    a(href='/join') Create one now &rarr;

postLogin의 경우 확인해야할 것 2가지

1. username이 존재하는가? – 존재하지 않으면 에러 띄우기

=> const exists = await User.exists({username}) 으로 true/false 값 저장,

=> if (!exists) { } 로 존재하지 않는다면 400 상태코드와 에러메세지 전달.

2. password가 그 username이 등록한 password와 일치하는가? – 일치하지 않으면 에러

=> db에 등록된 password는 해싱 된 상태이고, hash는 일방향 함수라 유저가 입력한 원래 password값을 알 수 없다. 어떻게 비교하나?

=> 유저가 password 입력 후 로그인 할 시 그 password를 바로 해싱해 db에 등록된 해싱된 패스워드값과 비교. 일치하면 로그인을 성공시키고, 불일치하면 에러메세지를 띄우면 된다.

=> bcrypt.compare(plainPassword, hash).then(function(Result) {}); 이용. Bcrypt가 알아서 form에서 받은 password를 해싱한 뒤 그 값을 db에 있는 해싱된 password값과 비교 후 true/false 반환

export const postLogin = async(req,res) => {

  const { username, password } = req.body;

  const pageTitle = 'Login'

  // 입력한 username을 가지는 데이터가 있는지 db에서 확인

  const user = await User.findOne({username});

  if(!user) {

    return res.status(400).render('login', {

      pageTitle,

      errorMessage: 'An account with this username does not exists'

    });

  }

  // 유저가 form에 입력한 password를 bcrypt.compare를 이용해 db에 저장된 해당 username이 등록한 password와 일치하는지 확인

  const ok = await bcrypt.compare(password, user.password);

  if(!ok) {

    return res.status(400).render('login', {

      pageTitle,

      errorMessage: 'Wrong password',

    });

  }

  console.log('LOG USER IN! COMING SOON!')

  return res.redirect('/');

};

#7.7 Sessions and Cookies part One

브라우저가 유저를 기억하는 방법: Cookie 보내주기

Session: 백엔드와 브라우저 간 어떤 활동을 했는지 기억하는 것으로, 현재 사용하는 브라우저와 내가 만든 백엔드 사이에 세션 존재. Like memory or history

Stateless

로그인 후 다른 페이지로 이동 시, GET요청을 보내는데, 백엔드가 HTML을 render 후 연결이 끝난다. 서버와 브라우저 모두 누가 연결을 보냈는지 잊음. 이 상태를 stateless 라 부른다.

즉, 브라우저와 서버 사이에 state가 없는 상태.

유저가 백엔드에 무언가 요청 시 누가 요청하는지 알 수 있도록 만들어야 한다. 유저가 로그인 할 때 이를 위한 장치를 설정해 놓아야 함!

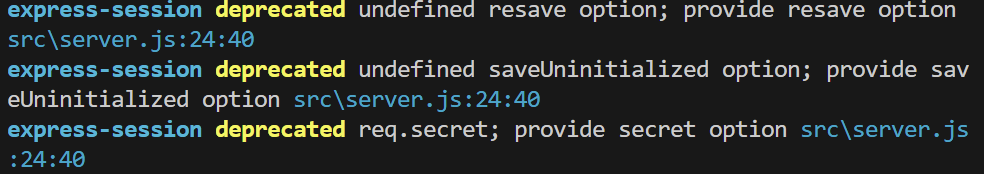
express-session 미들웨어 이용: express에서 세션을 처리할 수 있도록 해줌.

Express는 session을 메모리에 저장해 서버를 재시작하면 세션이 사라진다. 계속 잊음!

Npm I express-session 설치 후 server.js에 import한다.

import session from 'express-session';

그러면 터미널에 다음과 같은 문구가 뜬다



Express-session 사용하기

app.use(session({

  secret: 'Hello!',

  resave: true,

  saveUninitialized: true,

}));

Secret: 필수 옵션. 세션 id 쿠키에 서명하는 데 사용되는 비밀 코드. 보통 환경 변수를 사용해 로컬 환경에 숨겨 사용.(하드코딩 X). 주기적으로 변경하는 것이 좋다. 이 비밀코드를 여러 비밀코드가 들어간 배열로 사용하면 비밀코드 변경 시 세션이 무효처리 되지 않고 업데이트 될 수 있다.

Resave

true => 세션정보가 변경되지 않아도 무조건 세션 정보를 다시 저장

false => 세션정보가 변경되었을 경우에만 저장

saveUninitialized: 세션초기화 설정(true)/미설정(false)

세션 초기화: 세션 식별자를 생성하고 서버 측에서 세션 데이터를 저장하기 위한 저장소 설정 등의 작업 수행. 시간이 걸리고 가볍지만은 않은 작업

true =>

1. 클라이언트가 서버에 최초의 요청을 보낸다.

2. 서버는 세션 식별자를 생성하여 클라이언트에게 세션 쿠키를 전송한다.

3. 클라이언트는 세션 쿠키를 저장하고, 이후의 요청에서 세션 식별자를 함께 전송한다.

4. 서버는 세션 식별자를 기반으로 세션 데이터를 초기화하고 저장소에 저장한다.

False =>

1. 클라이언트가 서버에 최초의 요청을 보낸다.

2. 서버는 세션을 초기화한다.

세션을 초기화하기 전에는 세션쿠키를 포함하지 않은채 클라이언트에게 응답을 보낸다.

3. 초기화 작업이 끝난 후 요청이 들어오면 세션 쿠키(세션 식별자를 포함한)를 클라이언트에게 전송된다.

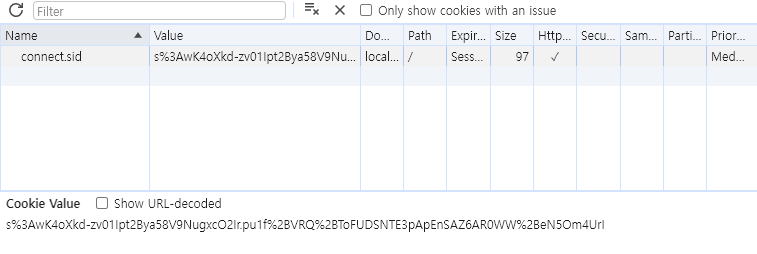
4. 클라이언트는 세션 쿠키를 저장하고, 이후의 요청에서 세션 식별자를 함께 전송한다.

5. 서버는 세션 식별자를 기반으로 클라이언트의 세션을 식별하고 세션 데이터를 저장소에서 가져온다.

6. 요청과 응답 사이에서 세션 데이터를 사용하여 작업을 수행하거나 상태를 유지한다.

쿠키 보기

개발자도구 – Application – cookies – <http://localhost:4000>



이런식으로 저장되어있는 쿠키를 볼 수 있다.

Server.js 에 다음의 코드로 sessions를 보면

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

  req.sessionStore.all((error, sessions) => {

    console.log(sessions);

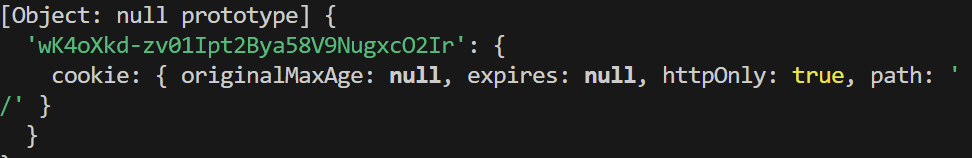
    next();

  })

})

콘솔에 다음과 같이 찍힌다(sessions)

Session id와 cookie 객체가 들어있다.



저장되어있는 쿠키의 Value값의 5번째 값부터 아래 콘솔에 찍힌 값이 동일한 것을 알 수 있다.

즉, 백엔드 – 브라우저 간 세션/쿠키 작동 방식은 다음과 같다.

1. 최초 요청 시 session id(connect.sid)와 개체를 생성 후 브라우저에 응답. 하지만 아직 쿠키가 없어 session DB가 생성되었어도 session id와 개체를 저장하지 않음.

2. 응답 받은 브라우저는 session id가 있을 시 자동으로 쿠키 생성, 쿠키 정보를 헤더에 추가.

3. 브라우저에서 다음의 요청 시 헤더에 추가된 쿠키가 서버로 전달

4. 세션 미들웨어가 브라우저에서 받은 쿠키 정보를 가공해 session 개체에 추가, 핸들러에 제공. 핸들러는 세션 개체에 데이터를 추가하거나 쿠키 구성을 수정할 수 있다. 모든 핸들러가 동작한 이후에 수정된 세션 개체를 세션id와 함께 세션 db에 저장하고 응답 마무리.

4. 세번째 요청부터는 전달받은 쿠키 정보를 세션 DB에 있는 id와 비교해 저장된 세션 개체가 있다면 해당 개체를 핸들러에 제공하고 이후 두번째 요청과 같은 프로세스로 진행

postLogin에 다음의 코드를 추가하면

req.session.loggedIn = true;

req.session.user = user;

로그인 후 home, join, search 등 다른 페이지를 여기저기 옮겨가도 다음의 같은 정보를 받을 수 있다.



로그인하지 않은 상태라면 loggedIn과 user의 정보를 받지 않고 id와 cookie정보만 받으므로, 어떤 브라우저가 로그인을 했는지 안 했는지 구분할 수 있다.

#7.10 Logged In User part Two

pug에서 session의 정보를 바로 이용할 수 없다. Pug와 express는 서로 locals를 공유할 수 있다.

Locals = res 의 한 속성. 다음으로 res를 콘솔에 찍어보면 다양한 정보가 나오는데.

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

  console.log(res);

  req.sessionStore.all((error, sessions) => {

    console.log(sessions);

    next();

  })

});

콘솔에서 locals를 찾아보면 다음과 같이 빈 값을 반환한다.



하지만 이런식으로 locals에 데이터를 추가한다면

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

  res.locals.sexy = you

  req.sessionStore.all((error, sessions) => {

    console.log(sessions);

    next();

  })

});

Pug 템플릿에서 #{sexy} 로 ‘you’라는 데이터를 불러와 사용할 수 있다.

Locals object를 이용하면 pug 템플릿에 변수를 전역적으로 보낼 수 있어 모든 pug 파일에서 사용 가능하다.

미들웨어를 사용하기 위해 middlewares.js 파일을 생성 후 코드 작성

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

  res.locals.loggedIn = Boolean(req.session.loggedIn);

  res.locals.siteName = 'Wetube';

  console.log(res.locals);

  next();

}

Server.js에 코드 작성 (주의: session 설정 코드 이후에 localsMiddleware 사용 코드가 와야한다.)

import { localsMiddleware } from "./middlewares";

// 다른 코드들

app.use(localsMiddleware);

base.pug

doctype html

html(lang='ko')

  head

    title #{pageTitle} | #{siteName}

    link(rel="stylesheet" href="https://unpkg.com/mvp.css")

  body

    header

      h1=pageTitle

      nav

        ul

          li

            a(href='/videos/upload') Upload Video

          li

            a(href='/') Home

          if loggedIn

            li

              a(href='/logout') Log out

          else

            li

              a(href='/join') Join

            li

              a(href='/login') Login

          li

            a(href='/search') Search

    main

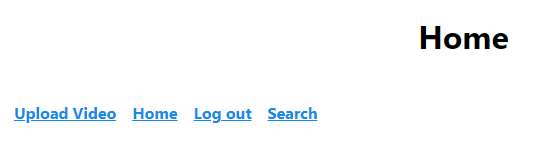
     block content

  include partials/footer

위와 같이 if 구문을 이용해 로그인 되었을 때와 되지 않았을 때의 템플릿을 구분해 작성 가능!!

로그인을 해 보면,





1. 로그인

2. postLogin에 의해 session.loggedIn = true 할당

3. locals.loggedIn = true 할당

4. pug에서 loggedIn에 할당된 데이터를 불러와 if문에 사용(if loggedIn / if #{loggedIn})

즉, 뷰 템플릿에서 session의 데이터를 사용하기 위한 연결 다리의 역할로 locals를 사용!

여기까진 서버를 재시작하면 session 데이터가 사라짐!

이를 방지하기 위해 세션을 DB에 저장. mongoDB에 저장하기 위해 connect-mongo 사용

<https://www.npmjs.com/package/connect-mongo>

Npm i connect-mongo 후 다음과 같이 사용



import MongoStore from 'connect-mongo';

// 다른 코드들

app.use(session({

  secret: 'Hello!',

  resave: true,

  saveUninitialized: true,

  store: MongoStore.create({mongoUrl: 'mongodb://127.0.0.1:27017/wetube'}),

}));

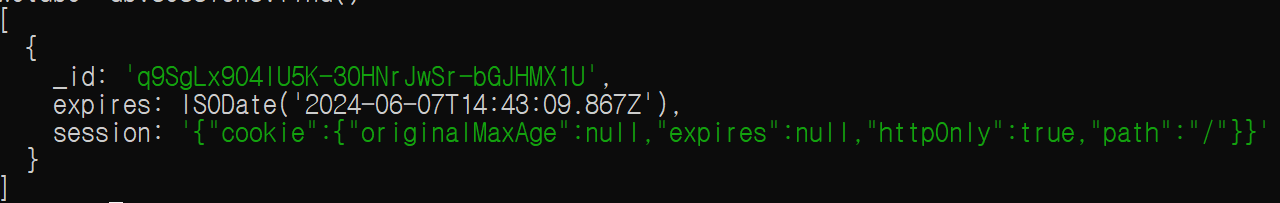
options에는 mongoUrl을 넣어주면 된다.

이미 mongoose가 mongoDB와 연결되어있기 때문에 mongoUrl을 다시 넣어주는 대신 다음과 같이 입력할 수도 있다.

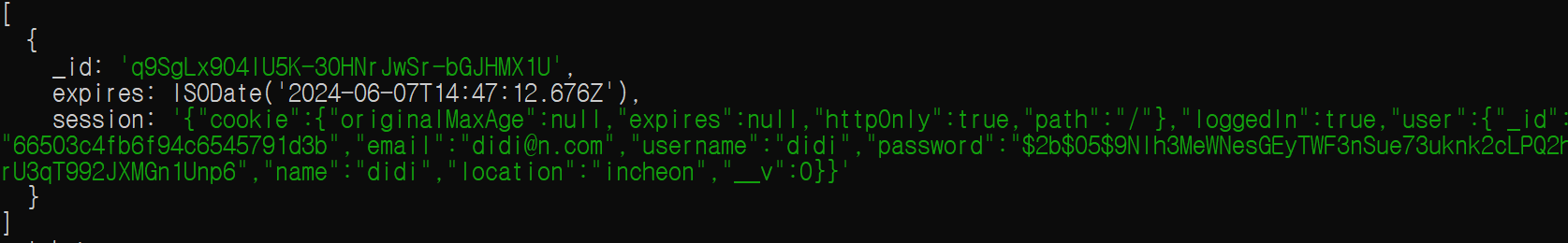
store: MongoStore.create({ client: connection.client })

그러고 mongoDB의 show collections명령어 입력 시 sessions가 생긴 것을 볼 수 있다.

Db.sessions.find() 명령어 입력 시 처음엔 아무것도 없지만, 새로고침 해 브라우저가 백엔드에 요청을 보내면 sessions 컬렉션에 새로운 데이터가 생긴다.



또한 로그인 후 다시 db.sessions.find() 명령어를 입력 해 보면



위와 같이 추가적인 데이터가 더 표시된다. 또한 위 정보가 DB에 저장되어있기 때문에 서버를 재시작해도 서버에서 잊혀지지 않는다.

Connect-mongo 사용 전: session이 서버 메모리(RAM)에 저장 => 서버 재시작 시 잊음

Connect-mongo 사용 후: session이 mongoDB(보조기억장치)에 저장 => 서버 재시작해도 기억

#7.13 Uninitialized Sessions

브라우저를 방문한 모든 사용자에 대해 쿠키를 주고, session 데이터를 DB에 저장한다면??

=> DB에 방대한 양의 데이터 저장. 용량과 비용 문제가 발생.

saveUninitialized: uninitialized = 세션이 만들어졌지만 수정되지 않은 초기의 상태

즉, saveUninitialized = 세션을 수정할 때만 세션을 DB에 저장하고 쿠키를 넘겨주는 설정

우리는 login했을 때 session을 수정하도록 설정했다.

  req.session.loggedIn = true;

  req.session.user = user;

=> 로그인한 사용자에게만 쿠키를 주고, 익명의 사용자에게는 쿠키를 주지 않음

saveUninitialized: false로 설정한 뒤 페이지 새로고침을 하니 실제로 브라우저에 쿠키가 생성되지 않고, DB에도 데이터가 저장되지 않는다.

로그인을 하니 그제서야 쿠키를 생성하고, DB에 데이터가 저장되었다.

Resave: 모든 request마다 세션의 변경사항이 있든 없든 세션을 다시 저장

True => 스토어에서 세션 만료일자를 업데이트 해주는 기능이 따로 없으면 true로 설정해 매 request마다 세션 업데이트

False => 변경사항이 없는데도 세션을 저장하면 비효율적! 동작 효율을 높이기 위해 사용.

각각 다른 변경사항을 요구하는 request를 동시에 처리할 때 세션 저장 과정에서 충돌이 발생할 수 있는데 이를 방지하기 위해 사용

7.14 Expiration and Secrets

쿠키의 속성들

Name

Value

Domain: 쿠키를 만든 백엔드가 누구인지 알려줌 => localhost

Path: 단순 url

Expires: 만료날짜가 명시되지 않았을 시 Session cookie로 설정되며, 클라이언트가 브라우저를 닫거나 컴퓨터를 재시작하면 session cookie 만료.

Max-age: 세션이 만료되는 날짜

Size

HttpOnly

Secure

SameSite

Priority

Max-Age

.env 파일: 코드를 숨기고자 환경변수를 사용 할 때 사용한다 (API key, url 등)

1. .env파일 생성

2. .gitignore 파일에 .env 추가

3. 환경변수와 값을 설정

Secret: 백엔드가 쿠키를 줬다는걸 보여주기 위해 쿠키에 sign할 때 사용하는 string

Session hijeck을 통해 쿠키를 해킹하려는 시도를 막기위해

mongoUrl

.env 파일

COOKIE\_SECRET=kdnfi9u4j3309mg29jnfniwe112ggsgdnlkjhr

DB\_URL=mongodb://127.0.0.1:27017/wetube

4. 환경변수가 들어가야 할 코드에 process.env.환경변수 형태로 삽입

app.use(session({

  secret: process.env.COOKIE\_SECRET,

  resave: false,

  saveUninitialized: false,

  cookie: {

    maxAge: 20000, // milisecond로 설정

  },

  store: MongoStore.create({mongoUrl: process.env.DB\_URL}),

}));

5. npm I dotenv로 dotenv 설치 후 import => require('dotenv').config() / import ‘dotenv/config’

Import 시 주의할 점: 가장 처음에 실행되는 파일에 import 해야한다. 즉, 우리의 폴더구조에 있어서는 init.js가 가장 먼저 실행되는 파일이므로 init.js에 import 해주어야 한다. 그래야 .env 안에 지정한 환경변수가 모두 정의된다.

#7.16 Github Login part One

흐름(<https://docs.github.com/en/apps/oauth-apps/building-oauth-apps/authorizing-oauth-apps#web-application-flow>)

1. 깃허브 로그인을 위해 깃허브 페이지로 유저를 리다이렉트

2. 깃허브 이메일과 패스워드를 이용해 로그인하면 승인

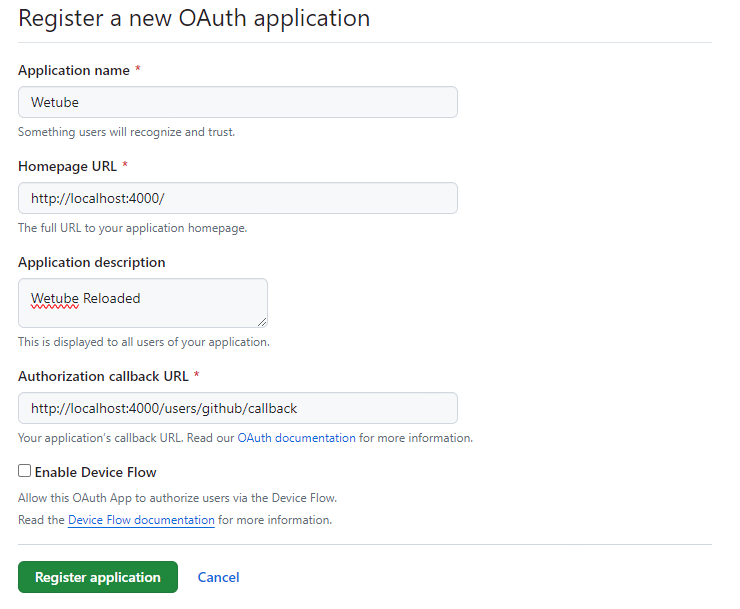
3. 유저를 token과 함께 다시 우리의 웹사이트로 리다이렉트

Token: 유저의 정보가 담겨있으며 매우 빠르게 만료됨

OAuth 생성하기

1. OAuth 등록

Github 로그인 -> Settings -> Developer Settings – Oauth Apps – Register a new application



2. 간편로그인 시 깃허브로 리다이렉트 해 줄 주소 입력

Login.pug

a(href="https://github.com/login/oauth/authorize?client\_id=Ov23li6TU0YsMgOuDFHb") Continue with Github &rarr;

<https://~authorize>까지는 깃헙 공식 문서에 나와있으며 그 뒤 파라미터를 입력해주어야 한다.

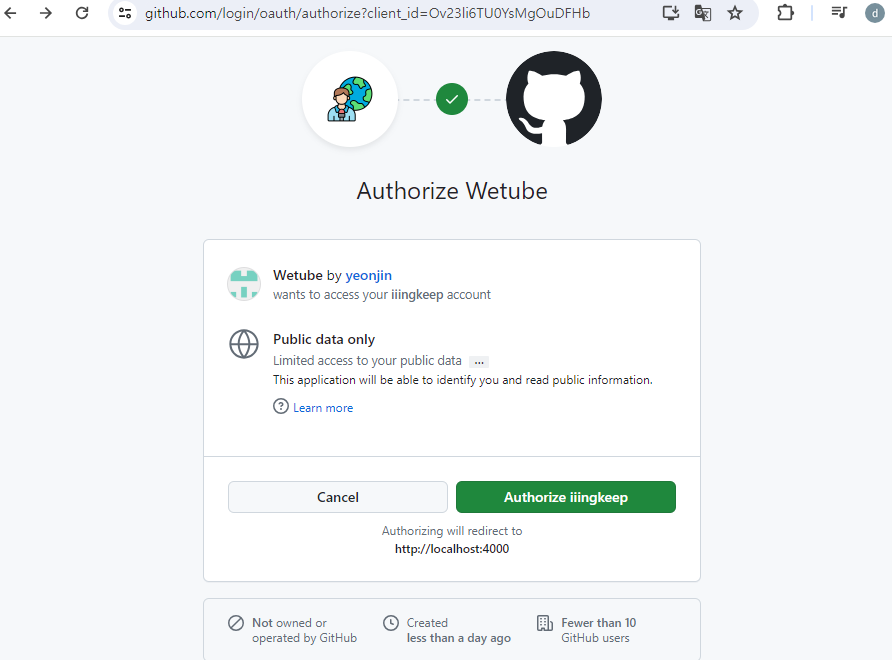
**필수적인 파라미터: client\_id**

파라미터로 사용할 수 있는 여러 정보들 중, OAuth를 만들었을 때 생성된 client\_id를 이용하면 위와 같은 url을 만들 수 있고, 해당 url로 리다이렉트 해주면 다음과 같은 페이지가 나온다.

\*\*url 생성 시

?는 url에 추가적인 정보를 더 넣을 때 사용하며 url 당 단 한번만 사용된다.

&은 그 정보가 여러가지일 때 사용하며 정보를 추가할 때 마다 사용



추가적 파라미터

Allow\_signup: true – 깃헙으로 리다이렉트 시 깃헙에서 로그인하도록 하며 계정이 없다면 계정 생성까지도 지원

False – 깃헙으로 리다이렉트 시 깃헙에서 로그인하도록 하지만 깃헙 계정을 만드는 것까지는 X

Scope: 유저에게서 얼마나 많이 정보를 읽어내고 어떤 정보를 가져올 것인지. 설정하지 않는다면 Public data only

Ex) 프로필 읽기 권한 / 유저 이메일 요청 등

하지만 파라미터를 추가할수록 url이 길어지고 한 눈에 쉽게 읽히지 않기 때문에 새로운 router 생성 => startGithubLogin

userRouter.get('/github/start', startGithubLogin);

a(href="users/github/start") Continue with Github &rarr;

startGithubLogin

export const startGithubLogin = (req,res) => {

  const baseUrl = 'https://github.com/login/oauth/authorize'

  const config = {

    client\_id: 'Ov23li6TU0YsMgOuDFHb',

    allow\_signup: false,

    scope: 'read:user user:email',

  };

  const params = new URLSearchParams(config).toString();

  const finalUrl = `${baseUrl}?${params}`;

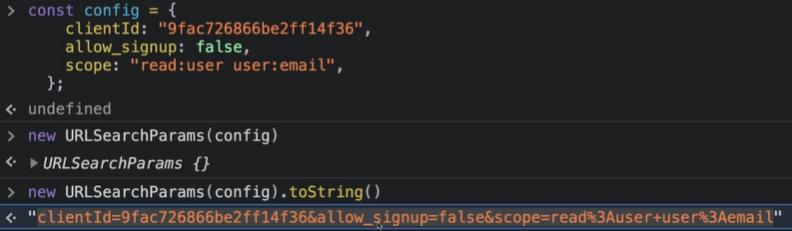
  return res.redirect(finalUrl);

};

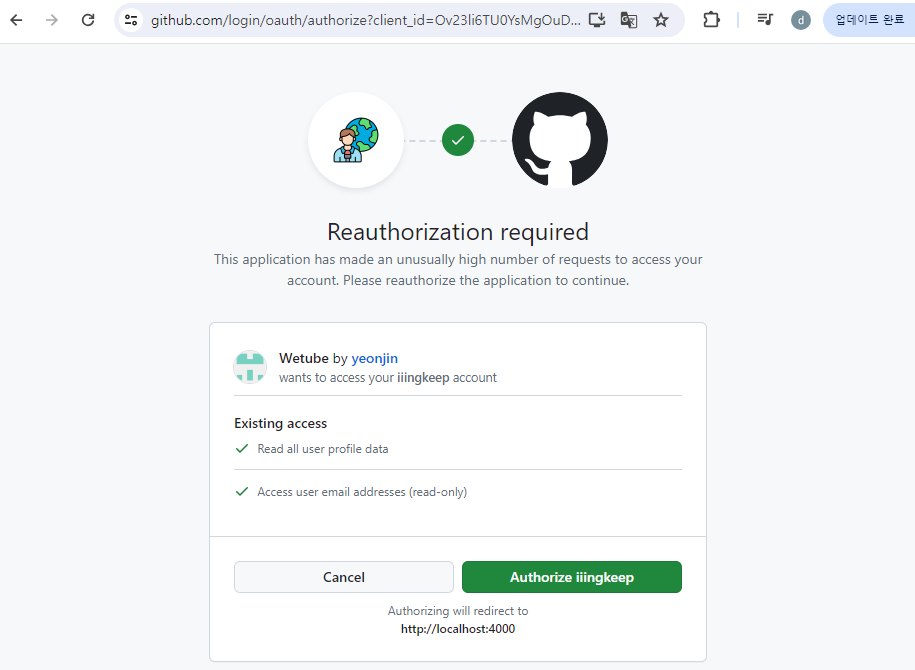
baseUrl을 토대로 config에 각각의 파라미터들을 적은 뒤 URLSearchParams와 toString()을 이용해 finalUrl을 생성하는 코드.

URLSearchParams: URL의 쿼리 문자열에 대해 작업할 수 있는 유틸리티 메서드 정의

toString(): URLSearchParams 인터페이스의 메소드로, URL에서 사용할 수 있는 쿼리 문자열 리턴. 우리가 설정한 코드에서는 아래 결과와 같다. ?는 붙지 않고 쿼리 문자열만 리턴하므로 ?는 코드에 직접 작성해줘야 한다.



그런 다음 다시 깃헙으로 로그인하기를 클릭하면



우리가 작성한 finalUrl로 리다이렉트 되고 요청 정보도 잘 담겨있는 것을 볼 수 있다.

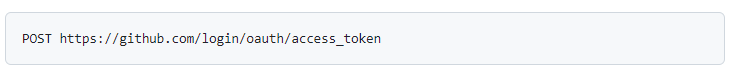
여기에서 Authorize iiingkeep 버튼을 누르면 우리가 설정한 곳으로 다시 리다이렉트 되는데, 깃헙에서 주는 code와 함께 리다이렉트 된다.

3. 깃허브에서 주는 code를 access\_token으로 변경

내가 설정한 리다이렉트 url.

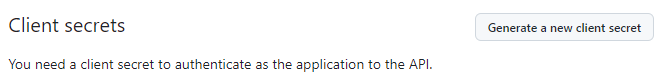
userRouter.get('/github/finish', finishGithubLogin);

깃헙에서 안내하는대로 finishGithubLogin을 통해 리다이렉트 url을 완성시켜준다.



필수 파라미터: client\_id, client\_secret, code

Client secrets 값은 우리가 생성한 oauth 페이지에서 받아 .env에 저장



userController.js의 finishGithubLogin

export const finishGithubLogin = async(req,res) => {

  const baseUrl = 'https://github.com/login/oauth/access\_token';

  const config = {

    client\_id: process.env.GH\_CLIENT,

    client\_secret: process.env.GH\_SECRET,

    code: req.query.code,

  };

  const params = new URLSearchParams(config).toString();

  const finalUrl = `${baseUrl}?${params}`;

  try {

    const data = await fetch(finalUrl, {

      method: 'POST',

      headers: {

        Accept: 'application/json',

      }

    });

    const json = await data.json();

    console.log(json);

    res.send(JSON.stringify(json));

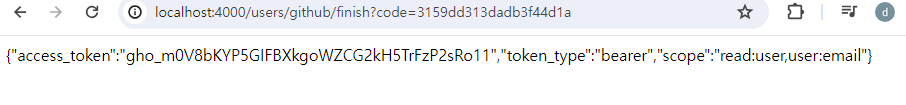
  } catch (error) {

    console.error('Error fetching data:', error);

  }

}

깃허브가 안내한대로 url을 만들어 post 요청을 보내보면 다음과 같이 access\_token을 알 수 있다.



4. 받은 access\_token을 인증을 위해 보내줌

export const finishGithubLogin = async(req,res) => {

  const baseUrl = 'https://github.com/login/oauth/access\_token';

  const config = {

    client\_id: process.env.GH\_CLIENT,

    client\_secret: process.env.GH\_SECRET,

    code: req.query.code,

  };

  const params = new URLSearchParams(config).toString();

  const finalUrl = `${baseUrl}?${params}`;

  try {

    const tokenRequest = await (

      await fetch(finalUrl, {

      method: 'POST',

      headers: {

        Accept: 'application/json',

      },

    })

    ).json();

    if('access\_token' in tokenRequest) {

      const {access\_token} = tokenRequest;

      const userData = await (

        await fetch('https://api.github.com/user', {

        headers: {

          Authorization: `token ${access\_token}`,

        }

      })

    ).json();

    console.log(userData);

    }

    else {

      return res.redirect('/login')

    }

  } catch (error) {

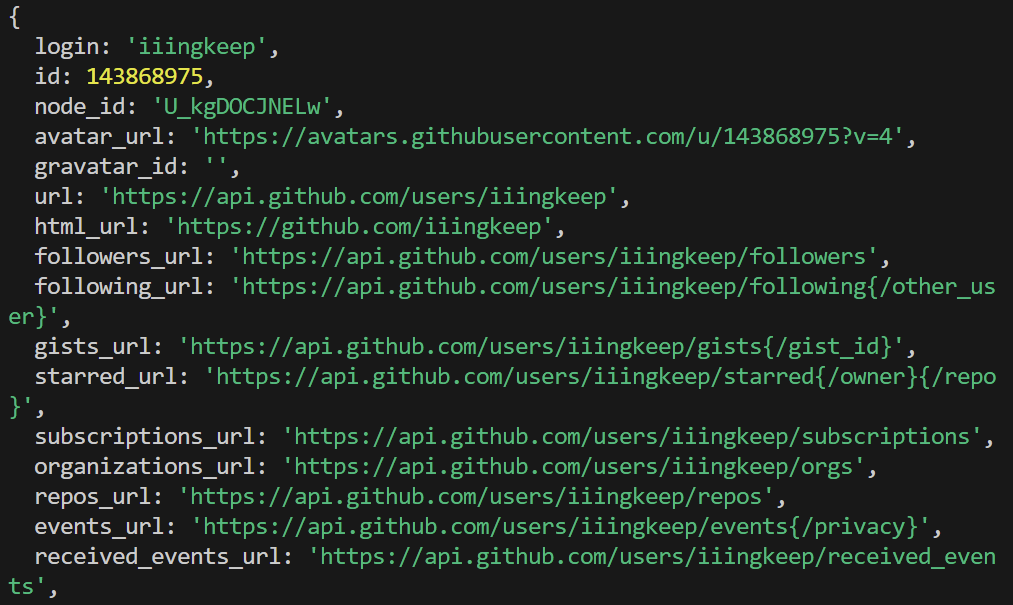
    console.error('Error fetching data:', error);

  }

}

json으로 받은 데이터에 access\_token이 있다면 그 토큰을 헤더에 넣어준다.

그 후 fetch된 url에 의해 userData가 콘솔에 찍히는 것을 볼 수 있다.(많은 데이터가 들어있다)



우리가 scope로 요청했던 read:user 데이터인데, email:user의 데이터는 null인 경우가 있다.

github에서 이메일을 private로 생성 시 email:user의 데이터가 null이 된다. private으로 설정한 유저의 이메일도 가져오기 위해 emailData를 json으로 받아볼 수 있도록 코드 작성

사용자 API를 사용하면 인증된 사용자에 대한 공개 및 비공개 정보를 얻을 수 있다.

GITHUB REST API endpoints for email addresses=>

<https://docs.github.com/en/rest/users/emails?apiVersion=2022-11-28#list-email-addresses-for-the-authenticated-user>

export const finishGithubLogin = async(req,res) => {

  const baseUrl = 'https://github.com/login/oauth/access\_token';

  const config = {

    client\_id: process.env.GH\_CLIENT,

    client\_secret: process.env.GH\_SECRET,

    code: req.query.code,

  };

  const params = new URLSearchParams(config).toString();

  const finalUrl = `${baseUrl}?${params}`;

  try {

    const tokenRequest = await (

      await fetch(finalUrl, {

      method: 'POST',

      headers: {

        Accept: 'application/json',

      },

    })

    ).json();

    if('access\_token' in tokenRequest) {

      const {access\_token} = tokenRequest;

      const apiUrl = 'https://api.github.com'

      const userData = await (

        await fetch(`${apiUrl}/user`, {

        headers: {

          Authorization: `token ${access\_token}`,

        }

      })

    ).json();

    console.log(userData);

    const emailData = await (

      await fetch(`${apiUrl}/user/emails`, {

        headers: {

          Authorization: `token ${access\_token}`,

        }

      })

    ).json();

    console.log(emailData);

    }

    else {

      return res.redirect('/login')

    }

  } catch (error) {

    console.error('Error fetching data:', error);

  }

}

그러면 콘솔에 emailData가 출력된다.



두 가지가 나오는데, email들 중 verified이면서 primary인 것을 찾는다.

이 email 정보를 가지고 아래의 사항을 해결할 수 있다.

깃허브 로그인과 웹페이지 로그인의 이메일이 충돌할 경우?

유저는 자신이 깃허브로 로그인해왔는지, 직접 이 웹페이지에서 join하여 로그인해왔는지 잊을 수 있다. 그럴 때 이미 해당 이메일이 존재한다면 로그인을 시켜주고, 아니라면 새로운 유저로 등록하여 로그인 시켜줄 수 있다.

export const finishGithubLogin = async (req, res) => {

  const baseUrl = 'https://github.com/login/oauth/access\_token';

  const config = {

    client\_id: process.env.GH\_CLIENT,

    client\_secret: process.env.GH\_SECRET,

    code: req.query.code,

  };

  const params = new URLSearchParams(config).toString();

  const finalUrl = `${baseUrl}?${params}`;

  const tokenRequest = await (

    await fetch(finalUrl, {

      method: 'POST',

      headers: {

        Accept: 'application/json',

      },

    })

  ).json();

  if ('access\_token' in tokenRequest) {

    const { access\_token } = tokenRequest;

    const apiUrl = 'https://api.github.com';

    const userData = await (

      await fetch(`${apiUrl}/user`, {

        headers: {

          Authorization: `token ${access\_token}`,

        },

      })

    ).json();

    const emailData = await (

      await fetch(`${apiUrl}/user/emails`, {

        headers: {

          Authorization: `token ${access\_token}`,

        },

      })

    ).json();

    const emailObj = emailData.find(

      (email) => email.primary === true && email.verified === true

    );

    if (!emailObj) {

      // set notification

      return res.redirect('/login');

    }

    // 이미 해당 email을 가진 유저가 존재한다면 로그인 시켜주고 홈으로 리다이렉트

    // 이메일이 없다면 계정 생성. 소셜로그인 했음을 알기 위해 socialOnly: true 추가

    let user = await User.findOne({ email: emailObj.email });

    if (!user) {

      user = await User.create({

        avatarUrl: userData.avatar\_url,

        name: userData.name,

        username: userData.login,

        email: emailObj.email,

        password: '',

        socialOnly: true,

        location: userData.location,

      });

    }

    req.session.loggedIn = true;

    req.session.user = user;

    return res.redirect('/');

  } else {

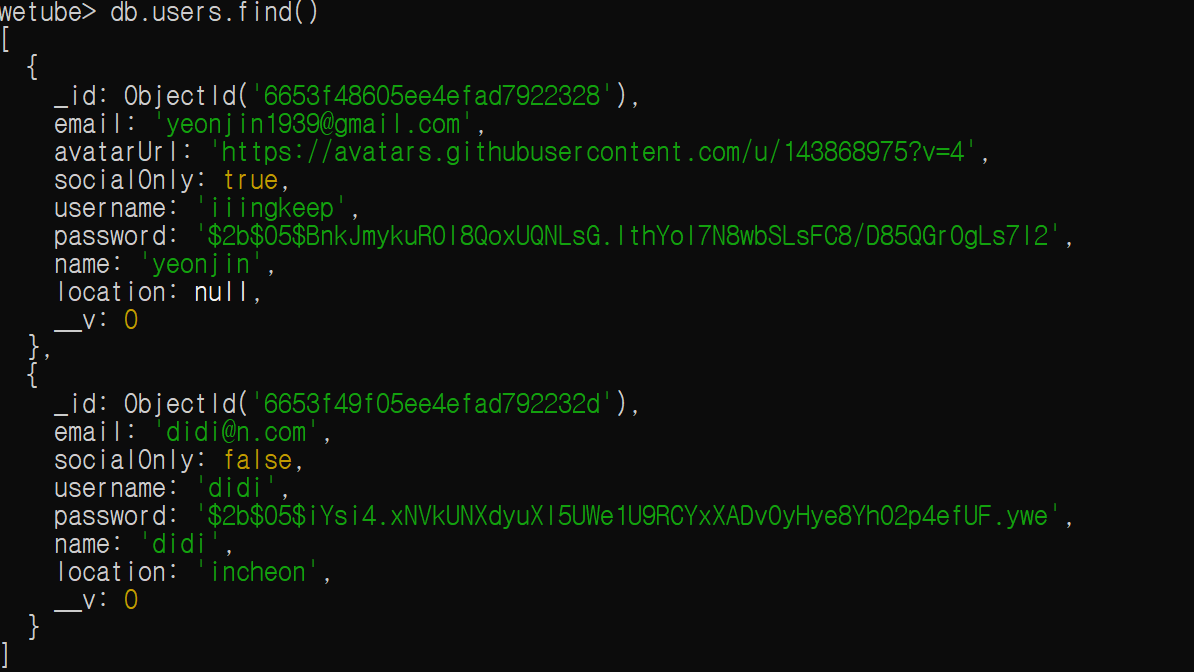
    return res.redirect('/login');

  }

};

User.js의 mongoDB 유저 모델에 avatarUrl과 socialOnly 항목을 추가했고, password가 require 이었는데 해제했다. 이를 통해 소셜 로그인을 한 유저인지 아닌지 구분이 가능

실제로 깃허브로 로그인한 유저와 아닌 유저의 정보를 비교해보니 avatarUrl과 socialOnly 항목에 차이가 있다



avatarUrl: 깃허브 프로필에 저장한 프로필사진 경로

\*\*정리\*\*

1. 깃허브로 로그인을 클릭하면 이동할 수 있는 라우터 생성

2. 그 라우터에서는 깃허브가 요구하는대로 파라미터들을 담아 finalUrl을 만들어 리다이렉트 해줌

3. 유저가 접근 권한을 확인 후 승인을 누름

4. 깃허브가 finalUrl에 담긴 정보를 받아 필요한 정보를 볼 수 있도록 access\_token을 돌려줌

5. 깃허브가 요구하는대로 파라미터들을 담아 finalUrl을 만들어 json으로 받아보면, access\_token을 확인할 수 있고, 그 access\_token으로 깃허브 API와 상호작용하여 내가 얻고 싶었던 scope를 확인 가능(access\_token을 headers에 넣어 보내줘야한다)

**로그아웃**

userController.js

export const logout = (req,res) => {

  req.session.destroy();

  return res.redirect('/')

};

아주 간단히, 로그아웃을 위해 req.session.destroy() 사용

#8.0 Edit Profile GET

컨트롤러를 통해 Edit Profile을 관리하는 변수를 만들어 준다 => getEdit

정보 수정 페이지이므로 이미 등록되어있는 정보들이 나타나야 한다.

Locals middleware 사용: 전에 설정해놓은 이 미들웨어를 통해 User의 정보를 가져올 수 있다.

Locals를 사용하면 어떤 템플릿에서든 접근이 가능!

res.locals.loggedInUser = req.session.user;

Edit-profile.pug

extends base

block content

  form(method='POST')

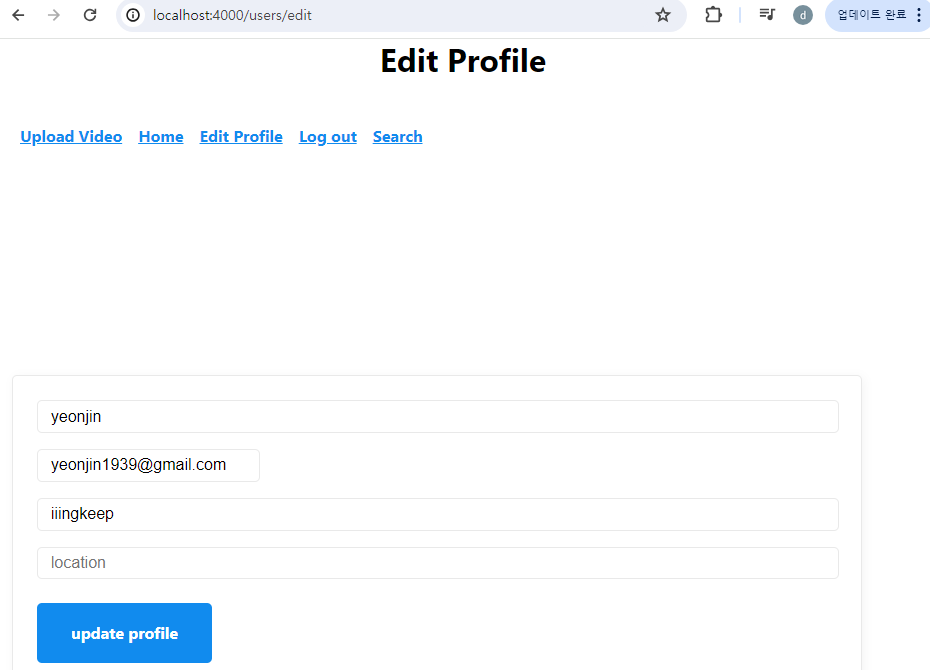
    input(name='name', placeholder='name', type='text', required, value=loggedInUser.name)

    input(name='email', placeholder='email', type='email', required, value=loggedInUser.email)

    input(name='username', placeholder='username', type='text', required, value=loggedInUser.username)

    input(name='location', placeholder='location', type='text', required, value=loggedInUser.location)

    input(type='submit', value='update profile')



고려할 사항

1. 로그인 되지 않은 채로 이 페이지에 온다면 loggedInUser가 undefined되기 때문에 에러 발생



에러를 없애기 위해 loggedInUser가 undefined 되어있는 상태어도 접근할 수 있도록 설정

res.locals.loggedInUser = req.session.user || {};

2. 로그인하지 않아도 <http://localhost:4000/users/edit> 이 url을 직접 입력해 접근할 수 있다.

로그인 유무에따른 접근가능성을 부여해주기 위한 미들웨어 설정

Middleware.js

// 유저가 로그인 되어 있다면 request를 계속하도록 하고, 아니라면 login페이지로 리다이렉트하는 미들웨어

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

  if(req.session.loggedIn) {

    return next();

  } else {

    return res.redirect('/login');

  }

};

// 유저가 로그인되어있지 않다면 request를 계속하도록 하고, 아니라면 home 페이지로 리다이렉트하는 미들웨어

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

  if(!req.session.loggedIn) {

    return next();

  } else {

    return res.render('/')

  }

};

라우터에 각각의 미들웨어 추가

userRouter.get('/logout',protectorMiddleware, logout);

userRouter.route('/edit').all(protectorMiddleware).get(getEdit).post(postEdit)

userRouter.get('/github/start', publicOnlyMiddleware ,startGithubLogin);

userRouter.get('/github/finish', publicOnlyMiddleware ,finishGithubLogin);

userRouter.get(':id', see)

여러 메서드를 함께 사용하는 라우터의 경우, all( middlewares )을 통해 모든 메서드가 동작할 때 미들웨어를 통과해야하도록 설정할 수 있다. 이는 다음의 코드보다 훨씬 효율적이다.

userRouter.route('/edit').get(protectorMiddleware, getEdit).post(protectorMiddleware, postEdit);

지금과 같이 설정해줌으로써 로그인 유무에 따라 유저의 요청을 계속 받아들이거나 요청을 거부하고 다른 url로 리다이렉트 할 수 있다.

\*\*꿀팁 한 개\*\*

VS Code에서는 내가 작성한 함수가 들어있는 파일이 열려있다면 다른 파일에서 그 함수를 이용할 때 자동으로 import문이 입력된다.

#8.2 Edit Profile POST part One

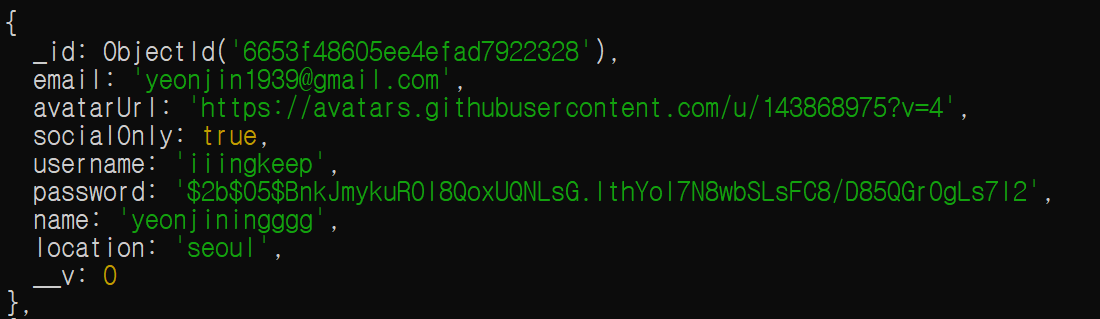
유저 정보 수정

1. 현재 로그인된 유저 정보는 세션에 저장되어 있으므로 req.session.user.\_id 값으로 mongoDB에 일치하는 \_id값을 찾은 뒤 해당 유저의 정보들을 form에 입력한 정보로 바꿔준다.

2. mongoDB와 session의 유저 정보가 일치하도록 둘 다 업데이트 해줘야 한다.

유저가 정보를 수정하고 submit하면 로그아웃이 되어 edit-profile페이지로 렌더링하지 못해 Login페이지로 가는 현상 발생(middleware 때문에)

하지만 mongoDB에는 yeonjin => yeonjininggg 으로 수정한 내용이 잘 반영되었다.



하지만 세션과 정보가 달라 로그아웃 되는 것 같다. 세션도 업데이트 해준다

export const postEdit = async(req, res) => {

  // const id = req.session.user.id;와

  // const {name, email, username, location} = req.body;

  // 위의 두 줄을 합쳐서 다음과 같이 적을 수 있다.

  const {

    session: {

      user: {\_id},

    },

    body: {

      name, email, username, location

    },

  } = req;

  const updatedUser = await User.findByIdAndUpdate(

    \_id,

    {

    name,

    email,

    username,

    location,

    },

    {new: true});

    req.session.user = updatedUser;

  return res.redirect('/users/edit');

};

세션도 업데이트가 되도록 코드를 입력해주고 나니 로그인이 유지된 채 /users/edit 페이지로 잘 리다이렉트 된다.

3. 유저가 변경한 email 또는 username이 기존에 존재한다면?

export const postEdit = async(req, res) => {

  // const id = req.session.user.id;와

  // const {name, email, username, location} = req.body;

  // 위의 두 줄을 합쳐서 다음과 같이 적을 수 있다.

  const {

    session: {

      user: {\_id, email: sessionEmail, username: sessionUsername},

    },

    body: {

      name, email, username, location

    },

  } = req;

  // 유저가 이미 등록된 email이나 username으로 업데이트를 시도하면?

  // 유저가 변경하려는 정보가 무엇인지 body의 데이터와 session의 데이터를 비교해 확인한다.

  // req.body의 정보와 mongoDB에 있는 정보를 비교한다.

  // 같은 email 또는 username이 존재하면 에러메세지와 함께 페이지를 리다이렉트한다.

  const usernameExists =

    username != sessionUsername ? await User.exists({ username }) : undefined;

  const emailExists =

    email != sessionEmail ? await User.exists({ email }) : undefined;

  if (usernameExists || emailExists) {

    return res.status(400).render("edit-profile", {

      pageTitle: "Edit Profile",

      usernameErrorMessage: usernameExists

        ? "This username is already taken"

        : 0,

      emailErrorMessage: emailExists ? "This email is already taken" : 0,

    });

  }

  // 같은 email 또는 username이 존재하지 않으면 정보 수정을 완료

  // findByIdAndUpdate함수에 아래의 세 가지 파라미터 전달.

  // 찾으려는 \_id / 업데이트 할 항목 / 업데이트 후 가장 최신 사항을 반영할 것(new: true)

  const updatedUser = await User.findByIdAndUpdate(

    \_id,

    {

    name,

    email,

    username,

    location,

    },

    {new: true});

    req.session.user = updatedUser;

  return res.redirect('/users/edit');

};

\*\* ES6 문법 꿀팁\*\*

1. 같은 req. ~~ 로 설정하는 변수 한 번에 작성하기

// const id = req.session.user.id;와

  // const {name, email, username, location} = req.body;

  // 위의 두 줄을 합쳐서 다음과 같이 적을 수 있다.

  const {

    session: {

      user: {id},

    },

    body: {

      name, email, username, location

    },

  } = req

2. 반복되는 값들은 하나만 적어주면 된다.

// await User.findByIdAndUpdate(id, {

  //   name: name,

  //   email: email,

  //   username: username,

  //   location: location,

  // })

  await User.findByIdAndUpdate(id, {

    name,

    email,

    username,

    location,

  })

#8.4 Change Password part One

강의에서는 패스워드 변경 url을 따로 만들어주었다.

// 패스워드 변경 페이지 렌더링

export const getChangePassword = (req,res) => {

  if(req.session.user.socialOnly === true) {

    return res.redirect('/');

  }

  return res.render('users/change-password', {pageTitle: 'Change Password'});

}

export const postChangePassword = async(req,res) => {

  const {

    session: {

      user: {\_id, password},

    },

    body: {

      oldPassword,

      newPassword,

      newPassword2

    },

  } = req;

  // 기존의 비밀번호가 맞는지 확인

  const oldPasswordConfirmation = await bcrypt.compare(oldPassword, password);

  if(!oldPasswordConfirmation) {

    return res.status(400).render('users/change-password', {

      pageTitle: 'Change Password',

      errorMessage: 'The current password is incorrect',

    });

  }

  // 새로 입력한 password와 password comfirmation 일치여부 확인

  if(newPassword !== newPassword2) {

    return res.status(400).render('users/change-password', {

      pageTitle: 'Change Password',

      errorMessage: 'The Password does not match',

    });

  }

  const user = await User.findById('\_id');

  user.password = newPassword;

  await user.save();

  req.session.user.password = user.password;

return res.redirect('/logout');

}

고려사항

1. 깃허브로 로그인한 유저의 경우 패스워드가 없으므로 이 url에 접근 불가

username과 password로 로그인하려는 유저의 경우 이 url에 접근 가능

2. 현재의 password가 일치하는지 여부 확인

3. 새로 변경할 password와 확인용 password의 일치 여부 확인

4. 모두 일치한다면 DB에 새로운 패스워드로 저장, 단 save()를 통해 비밀번호 해시 후 저장

5. 세션 데이터 업데이트 후 새로운 password로 로그인하도록 로그아웃 조치

#8.6 File Uploads part One

유저 프로필에 avatar를 추가하기 위해 multer 사용

<https://www.npmjs.com/package/multer>

Multer: 파일 업로드를 위해 사용하는 라이브러리

1. Edit-profile.pug. multer 사용을 위해서는 form에 enctype="multipart/form-data" 추가

또한 input에 accept 속성으로 삽입할 수 있는 파일의 종류를 정해준다.

block content

  form(method='POST', enctype='multipart/form-data')

    label(for='avatar') Avatar

    input(type='file', id='avatar', name='avatar', accept='image/\*')

2. multer를 이용한 미들웨어 생성(middlewares.js)

유저가 파일 업로드 시 해당 파일을 저장할 폴더를 만든 뒤 dest 지정

export const uploadMiddleware = multer({

  dest:'uploads/'

})

3. /edit 라우터에서 post 하기 전 실행될 수 있도록 미들웨어 설정

userRouter.route('/edit')

.all(protectorMiddleware)

.get(getEdit)

.post(uploadMiddleware.single('avatar'), postEdit);

.single(fieldname) / .array(fieldname[, maxCount]) / .array(fieldname[, maxCount]) 등의 옵션 가능

Avatar는 하나의 프로필 사진을 올리는 것이므로 single() 사용

4. postEdit 부분에 req.file 사용

const {

    session: {

      user: {\_id, email: sessionEmail, username: sessionUsername},

    },

    body: {

      name, email, username, location

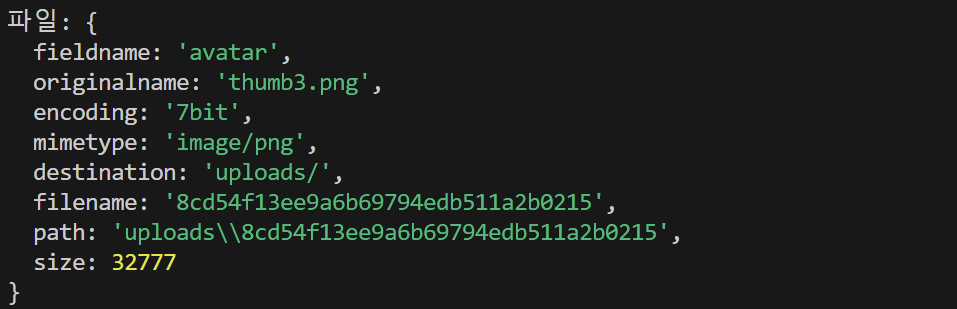
    },

    file

  } = req;

  console.log('파일:', file);

콘솔을 확인해보면 다음과 같이 파일 정보에 대해 나온다.

****

우리가 User모델에 설정한 avatarUrl에 담을 데이터가 바로 file.path

5. 유저가 정보 수정 시 avatar 업데이트 유무에 따라 다르게 처리

- 유저가 avatar 업데이트를 하기 위해 파일을 등록하면 file 정보가 생성되고, 그 중 file.path를 avatarUrl에 업데이트 한다.

- 유저가 avatar 업데이트를 하지 않고 다른 정보만 수정하면 file은 undefined가 되어 에러가 나니 기존의 avatarUrl(세션 데이터)로 그대로 업데이트 한다.

const updatedUser = await User.findByIdAndUpdate(

    \_id,

    {

    avatarUrl: file? file.path : avatarUrl,

    name,

    email,

    username,

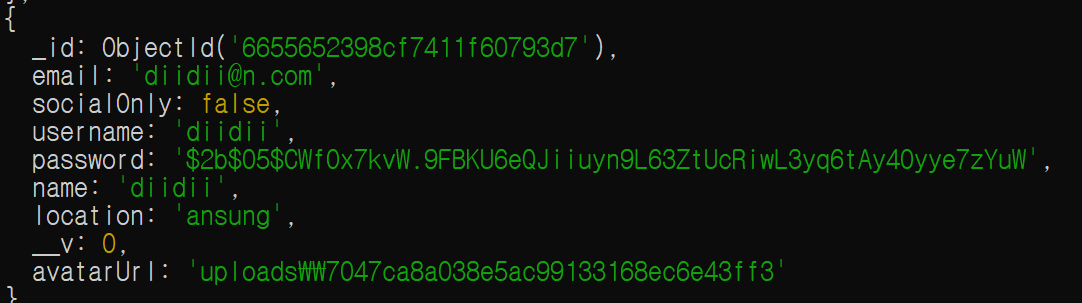
    location,

    },

    {new: true});

\*\*\*\*\* DB에는 절대 파일을 저장하지 않는다. 파일 저장을 위한 폴더를 따로 만들고, DB에는 파일의 위치만 저장한다 \*\*\*\*\*

위의 과정으로 avatar 파일을 선택 후 프로필을 update하면 DB에 avatarUrl이 생긴다.



multer과정:: form에서 파일 선택 후 post 시 uploadMiddleware에 의해 uploads 폴더에 사진 파일 저장 => 그 정보를 postEdit에 제공 => req.file 사용 가능. 해당 정보로 DB에 파일 위치 저장

내 현재 avatar 사진이 보이도록 img 태그 설정하기

Edit-profile.pug

block content

  img(src='/'+loggedInUser.avatarUrl, width='100', height='100')

src에 ‘/’를 앞에 더해주지 않으면 상대적인 url로 인식해 user/uploads/avatarUrl 이 되버린다.

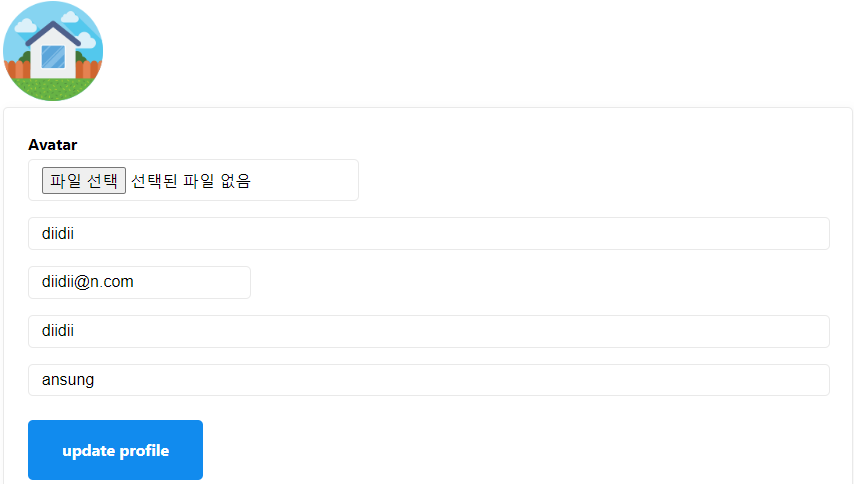
이미지를 브라우저에 띄우려면 브라우저가 uploads 폴더의 모든 파일을 볼 수 있도록 노출해 주는 작업 필요.

Server.js

app.use('/uploads', express.static('uploads'));

express.static()을 이용하여 누군가 /uploads로 가려고 한다면 uploads 폴더의 내용을 부여주도록 설정(라우터 등록)

결과



\*\*\*\*\*몽고db 관련 프롬프트 명령어\*\*\*\*\*

1. 몽고 사용하기

> mongo

2. 내가 가진 db 보기

> show dbs

3. 현재 사용 중인 db 확인

> db

4. 사용할 db 선택하기

> use dbName (현재 수업에서는 `use wetube`)

5. db 컬렉션 보기

> show collections

6. db 컬렉션 안에 documents 보기

> db.collectionName.find()

(현재 수업에서는 `db.videos.find()`)

7. db 컬렉션 안에 documents 내용 모두 제거하기

> db.collectionName.remove({}) (현재 수업에서는 `db.videos.remove({})`)

\*\*\*\*\*\*\*\*\*\*reqest 관련 함수\*\*\*\*\*\*\*\*\*\*\*\*\*\*

라우터로 지정한 :id -> req.params

pug파일에서 input으로 받은 내용 -> req.body(form이 POST일 때)

pug파일에서 input으로 받은 url내용 -> req.query (form이 GET일 때)

\*\*\*\*\*\*\*\*\*\*쿠키와 세션\*\*\*\*\*\*\*\*\*\*\*

Cookie: 주고받는 전송 수단. Session id를 저장하고 있어 백엔드에서 요청시 session id를 백엔드에 넘겨준다. Session data는 쿠키에 저장X, 서버사이드에 저장된다.

http 표준에 있어 쿠키는 자동적으로 브라우저와 백엔드간 전송하도록 설정되어 있다.

Session: 브라우저(프론트엔드)와 서버(백엔드)간의 state를 유지할 수 있도록 도와주는 미들웨어

Session store: session 저장소. 서버가 재시작되면 사라진다

Req.session.xxx 로 개체 추가: 세션에 데이터가 추가되어 어디서든 접근 가능해진다.