Node js

리터럴 – 데이터를 표현하는 방식

Var a = 1 – 정수를 표현하는 방식

Var a = “1” – 문자를 표현하는 방식.

javascript에서 ‘내용’ + 변수 +’내용 \n’ 등의 표현은 번거롭기 때문에

template literal 을 사용한다

~칸의 ` 을 사용

`내용 ${변수}

내용‘ 으로 같은 결과를 얻는다. ${1+1}등도 가능

http – 서버와의 통신 방법, 프로토콜

opentutorials.org – 인터넷에 연결되어 있는 컴퓨터, host,domain

3000 – 포트

Main – 경로

?id=HTML&page=12 – 웹서버에 전달할 수 있는 정보 ?로 시작하며 값과 값은 &로 연결 값의 이름과 값은 =로 연결



Fs, http 같은건 모듈임 비슷한기능끼리 묶어둔 것.

Require(‘url’) - nodejs에게 url이라는 모듈을 사용할 것임을 말하는 것.

Fs = 파일 시스템

fs.readFile('/etc/passwd', (err, data) => {

if (err) throw err;

console.log(data);

});

최신기능?

fs.readFile('sample.txt', ‘utf8’ function(err, data) {

console.log(data);

});

구버전 같은 효과

이렇게 함으로서 다른 파일의 내용을 불러올 수 있게 된다.

Nodejs 파일은 수정될 때 마다 재시작 해야 하지만.

Nodejs 에서 불러온 파일이 수정된 경우 새로고침만 해도 바로 적용된다.

콘솔에서의 입력

Var args = process.argv;

args라는 배열에 콘솔에서 입력값을 문자로 받아온다.

콘솔에서 node name.js 123 이라고 입력하면

process.argv 가 ‘123’을 입력으로 받아

args[2]에 저장한다.

Response.writeHead(200) – 전송이 성공적이다.

Response.writeHead(404) – 파일을 찾을 수 없다.

Var \_url=request.url;

Url.parse(\_url,ture)는 url로부터 여러가지 정보를 가져온다.

Var pathname= Url.parse(\_url,ture).pathname;

Pathname==’/’라면 가장 최상위의 루트를 의미하며

Pathname은 / 이후의 쿼리 스트링은 포함하지 않는다.

따라서 쿼리스트링을 통해 페이지를 나누는 경우 pathname이 루트인지 아닌지에 따라 유효한 페이지인지 아닌지를 구분할 수 있다.

루트의 메인 페이지의 경우 /로만 구분되었기 때문에 하위 페이지와는 구분이 안되어있다.

따라서 queryData.id 가 undifined라면 메인페이지 임을 알 수 있으므로 하위페이지의 처리와 동일하게 조건문을 통해 구분하여 메인페이지를 구성할 수 있다.

Nodejs에서 파일 목록 불러오기

Var testFolder = ‘실행 위치기준에서의 해당 폴더 경로’;

Var fs = require(‘fs’);

Fs.readdir(testFolder,function(error,filelist){

Console.log(filelist);

})

라고 하면 해당 폴더의 파일들의 목록을 배열상태로 가져온다.

Node js상의 함수

Function 함수명(){

}

호출 함수명;

Math라는 객체에 정리되어있는 반올림 함수 round

호출 Math.round(1.6)

=2

함수 입출력

Function sum(first,second){//파라미터/parameter

Return first+second;

}

Sum(2,4); //아귀먼트/argument

출력 = 6

동기 – 비효율적

비동기 – 효율적, 병렬처리

패키지 매니저 – 여러가지 추가 소프트웨어

Pm2 – nodejs가 꺼지면 자동으로 다시 켜주고, 수정사항 발생시 자동으로 껏다 켜준다.

설치

Npm install pm2 -g

실행

Pm2 start main.js

Pm2로 실행중인 프로그램들 모니터링

Pm2 monit

Pm2로 실행중인 목록

Pm2 list

종료

Pm2 stop main.js

수정시 자동 껏다킴//웹페이지 자체는 새로고침 해야함

Pm2 start main.js –watch

로그/문제사항 발생시 확인

Pm2 log

Form은 그 안의 각각의 컨트롤에 사용자가 입력한 정보를 submit버튼을 눌렀을 때

액션 속성이 가리키는 서버로 쿼리스트링 형태로 데이터를 전송한다.

이는get 방식으로 적절하지 못하다

메소드 속성을 추가하여 post 방식으로 전송해야 서버만 쿼리스트링에 추가된다.

Input text 속성으로 placeholder=”내용”

하면 입력 전에는 내용이 표시되고 입력하면 내용이 사라지고 입력한 내용만 표기된다.

Post 방식의 데이터 받기

request.on('data', function (data) {

body += data;

});

데이터가 들어올 때 마다 fuction(data) 콜백함수를 호출함 data라는 인자를 통해 데이터를 줌

상단 추가 var qs = require('querystring');

request.on('end', function () {

var post = qs.parse(body);

var title = post.title;

var description = post.description;

});

모든 수신이 끝난경우 변수에 배열 형태로 저장한 뒤 이용할 수 있게 된다.

이후 파일을 생성하여 저장하려면

Fs.writeFile(`data/${title}`,description,’utf8’,function(err){

/\*에러 처리시의 내용을 넣어야 하는데 현재는 다루지 않는다\*/

response.writeHead(200);

response.end('success');

})

를 통해 title의 제목으로 description을 내용으로 하는 파일을 만들 수 있다.

현재는 파일을 생성하는데 성공하면 success만 뜨고 다시 주소를 수정해서 되돌아가야 한다.

이때 필요한 것이 redirection이다

Response.writeHead(302,{Location:`/?id=${title}`});

Response.end();

라고 하면 302는 리다이렉션을 의미하고 location으로 자동으로 이동된다.

현 실습 예제에서 update구현을 위해 어떤 내용을 업데이트 할 지에 대해 링크에 쿼리스트링을 붙여서 전송하였다.

<a href="/update?id=${title}">update</a>

사용자가 제목을 수정하게 될 경우 기존의 ${title}을 값으로 사용하면 수정된 이후에 data폴더에 존재하지 않을 수 있기 때문에 제목이 수정되더라도 보존되는 데이터가 필요하다.

따라서

<input type=”hidden” name=”id” value=”${title}”> 를 추가하여 미리 ${title}값을 꺼낸다.

이름의 바뀐 경우에 이름을 바꾸는 방법

Fs.rename(oldpath,newpath,callback(err))

이므로

현재 예제의 경우에는 에러 처리를 따로 안하고 있으므로

fs.rename(`data/${id}`,`data/${title}`,function(error){}); 로 하면 된다.

 fs.rename(`data/${id}`,`data/${title}`,function(error){

        //위의 create에서의 기능과 같이 이름을 바꾸고 내용을 바꾼다.

        fs.writeFile(`data/${title}`, description, 'utf8', function (err) {

          response.writeHead(302, { Location: `/?id=${title}` });

          response.end();

        })

      });

이와 같이 업데이트의 내용을 중간에 삽입함으로서 이름과 내용을 바꿀 수 있게 된다.

문서 삭제기능을 a href로 하게되면 정보가 get방식으로 전송되므로 반드시 form으로 감싸서 post방식으로 할 것

<form action = "/delete\_process" method="post" onsubmit="return del\_check();">

              <input type="hidden" name="id" value=${title}>

              <input type="submit" value="delete">

</form>

예제에서는 다음과같이 hidden으로 사라질 제목을 미리 알아낸 뒤 해당 절차로 넘어가는데, onsubmit의 내용은 내가 직접 추가한 함수로 확인을 한 뒤 true, false를 반환한다.

fs.unlink(`data/${id}`, function (error) {

        response.writeHead(302, { Location: `/` });

        response.end();

      })

삭제의 경우

Fs.unlink(path,callback)으로 이루어진다.

Onsubmit 에 커스텀 함수 넣으면 동작 안함 이유 불명 일단 confirm으로 대체함.

배열이 순서가 있는 정보를 정리하는 수단이라면

객체는 순서가 없는 정보들을 정리하는 수단.

var members = ['egoing', 'k8805', 'hoya'];

console.log(members[1]); // k8805

var roles = {

'programmer':'egoing',

'designer' : 'k8805',

'manager' : 'hoya'

}

console.log(roles.designer); //k8805

객체를 이용하여 저장한 예시

중괄호를 사용함에 유의

객체는 키와 값으로 저장된다

객체를 반복문에 사용할 때 .키도 가능하지만 [키] 도 가능하다

첫번째 var 값으로는 키를 받는다.



함수 또한 이러한 방식으로 객체에 넣고 사용할 수 있다.



객체 내에서 자기 자신을 참조 하는 경우 this를 사용한다.

모듈



서로 다른 js파일에서 모듈을 생성하고 외부사용을가능하게하고(왼쪽)

같은 디렉토리 내의 외부의 모듈을 가져와서 사용하는 방법이다(오른쪽)

Mudule.exports = {

V:’v’,

F:function(){

Console.log(this.v);

}

}

처럼 exports문을 객체 이름대신 넣어도 가능하다.

보안 – 오염된정보 입력

쿼리스트링의 경로를 통한 침입의 문제를 해결하기 위한 간단한 방법 으로 path 모듈을 사용한다.

url에 경로를 입력하여 들어오는 침입등은 외부로의 침입이므로 예제에서 외부에서 url의 입력이 들어오는 readfile들을 수정한다. 그외에 입력 수정 생성 등에도 적용한다.

예시

Var path = require(‘path’);

var filteredID = path.parse(queryData.id).base;

fs.readFile(`data/${filteredID}`, 'utf8', function (err, description) {

오염된 정보 출력.

페이지 생성등에서 html이나 스크립트를 입력하여 오염된 정보가 출력되는 경우

예시) 이상한 페이지를 로드, 이상한 창을 생성 등.

Npm을 이용해서 Sanitize-html을 사용할것임.

먼저 npm init을 cmd에 친다.

이렇게하면 내 프로젝트를 npm을 통해 패키지로서 관리하겠다는 의미이다.

Npm install -S sanitize-html 을 입력한다.

Sanitize-html 모듈을 사용한다는 것인데, 실제 사용할 때에는 인터넷에 npm sanitize 를 검색해서 모듈을 사용할 것, Sanitize-html은 공식 npm페이지에서 평판이 좋았음.

-g는 글로벌로 모든 pc위치에서 사용가능한 독립 프로그램으로서

-S는 해당 프로젝트에서 만 사용가능한 부품으로서 설치함을 의미한다.

설치가 끝난 뒤 json파일에 dependencies에 추가가 되었는지 확인한다.

Sanitize 모듈을 불러와서 사용한다.

Var sanitizeHtml = require(‘sanitize-html’) 로 모듈을 불러오고

Var sanitizedDescription = sanitizeHtml(description); 으로 필터를 적용시키면 된다

이후 description을 사용한 자리에 sanitizedDescription을 사용하면 끝난다.

만약 몇몇 태그를 추가하고 싶은 경우

Var sanitizedDescription = sanitizeHtml(description,

{allowedTags:[‘h1’]}

);

이런식으로 객체로 넣어주면 된다.

Pm2 사용법/추가

Pm2 kill – pm2로 실행한 모든 프로그램 종료

Pm2 start main.js --watch --ignore-watch=”data/\* sessions/\*” --no-daemon

-no-daemon 옵션은 바로 log기능을 수행하는 것이고

–ignore-watch=”data/\* 는 data하위폴더의 변경사항에 의해 껏다켜짐을 방지하고 한칸 띄고 sessions/\*”를 넣음으로서 2폴더를 지정할 수 있다.

nodejs에서 mysql 접근

npm install -S mysql 실행

mysql 관련 패키지 설치

var mysql = require('mysql'); //mysql이라는 변수로 mysql 모듈 사용

// 비밀번호는 별도의 파일로 분리해서 버전관리에 포함시키지 않아야 합니다.

var connection = mysql.createConnection({//접속

host : 'localhost',

user : 'root',

password : 'root’,

database : 'opentutorials'

});

connection.connect();

connection.query('SELECT \* FROM topic', function (error, results, fields) {

if (error) { console.log(error);

}

console.log(results);

});//쿼리문, 에러처리,결과

connection.end();//접속종료

에러처리

If(err)

{

Throw err;

}

에러 발생시 다음 명령어를 수행하지 않고 앱을 바로 종료시킨다.

페이지 표현

쿼리문을 통해 where로 지정하여 받아올 때 반환되는 정보가 배열값일 때 주의할 것.

예) topic.title 이 아닌 topic[0].title

보안

`select \* from topic where id=queryData.id` 보다는

`select \* from topic where id=?`, [queryData.id] 가 좋은데

[쿼리 아이디]가 ?로 자동으로 치환되어 외부의 입력에서의 공격을 방지할 수 있다.

문서추가(Create\_process)\_

db.query(`insert into topic (title,description,created,author\_id) values(?,?,now(),?)`,[post.title, post.description, 1], function (err, result) {

if (err) {

throw err;

}

response.writeHead(302, { Location: `/?id=${result.insertId}` });

response.end();

})

});

원래는

Insert into topic (title,description,created,author\_id) values(제목,내용,now(),아이디);

이렇게 작성되어야 하는 쿼리문이

insert into topic (title,description,created,author\_id) values(?,?,now(),?);

실제 입력값이 어떻게 들어오는지 모르므로 ?(와일드카드)처리되었다.

이후 ?에 대한 값으로 배열형식으로 읽어들인 값을 넣는다. 저자아이디는 임의입력.

자동입력될 id는 현재 모르므로 (자동삽입되어 입력하지 않으므로)

삽입된 데이터의 id를 알아내기 위해 result.insertId를 사용한다

문서 갱신

db.query(`select \* from topic`, function (err, topics) {

      if (err) {

        throw err;

      }

      db.query('select \* from topic where id = ?', [queryData.id], function (err2, topic) {

        if (err2) {

          throw err2;

        }

        var list = template.List(topics);

        //제목이 바뀔 경우를 대비하여 id로 제목값을 따로 저장

        //사용자와 상관없는 내용이므로 hidden을 이용하여 숨긴다.

        var HTML = template.HTML(topic[0].title, list,

          `

         <form action="/update\_process" method="post">

         <input type="hidden" name="id" value="${topic[0].id}">

        <p><input type="text" name="title" placeholder="제목" value="${topic[0].title}"></p>

        <p>

          <textarea name="description" placeholder="본문">${topic[0].description}</textarea>

        </p>

        <p><input type="submit"></p>

      </form>`,

          `<a href="/create">creat</a> <a href="/update?id=${topic[0].id}">update</a>`);

        response.writeHead(200);

        response.end(HTML);

      });

    });

리스트 목록 생성을 위해 전체를 불러온다.

업데이트를 위해 선택된 id를 모르므로 쿼리스트링으로부터 선택되어있는 id를 알아낸다.

각각의 값들을 선택된 id의 값으로 바꿔준다.

request.on('end', function () {

      var post = qs.parse(body);

      db.query(`update topic set title = ?,description = ?, author\_id = 1 where id = ?`, [post.title, post.description, post.id], function (err2, topic) {

        if (err2) {

          throw err2;

        }

        response.writeHead(302, { Location: `/?id=${post.id}` });

        response.end();

      });

    });

수정만 따로 진행되므로 수정 쿼리문을 수행한다.

배열로 입력받은 post값들을 이용하여 값을 수정한 뒤

수정된 페이지로 리다이렉트한다.

db.query('delete from topic where id = ?',[post.id],function(err,topics){

        if(err)

        {

          throw err;

        }

        response.writeHead(302, { Location: `/` });

        response.end();

      });

삭제문은 간단하게 해당 아이디에 대한 삭제만을 수행하면 처리된다.

상세보기/하위페이지 join적용

db.query(`select \* from topic left join author on topic.author\_id = author.id where topic.id=?`, [queryData.id], function (err2, topic) {

          if (err2) {

            throw err2;

          }

          console.log(topic);

          var title = topic[0].title;

          var description = topic[0].description;

          var list = template.List(topics);

          var HTML = template.HTML(title, list,

            `<h2>${title}</h2>${description}

            <p>by ${topic[0].name}<p>`,

            `<a href="/create">creat</a>

             <a href="/update?id=${queryData.id}">update</a>

             <form action = "/delete\_process" method="post" onsubmit="return confirm('정말로 삭제하시겠습니까?')">

              <input type="hidden" name="id" value=${queryData.id}>

              <input type="submit" value="delete">

             </form>`

          );

          response.writeHead(200);

          response.end(HTML);

        });

간단하게 쿼리문만 고치고 원하는 위치에 원하는 내용에 대해 삽입하면 된다.

문서생성/Create 에 join적용해서 다른 테이블 정보 넣기

db.query('select \* from topic', function (err, topics) {

      if (err) {

        throw err;

      }

      db.query('select \* from author', function (err2, authors) {

        if (err2) {

          throw err2;

        }

        var title = "WEB - create";

        var list = template.List(topics);

        var HTML = template.HTML(title, list,

          `<form action="/create\_process" method="post">

          <p><input type="text" name="title" placeholder="제목"></p>

          <p>

            <textarea name="description" placeholder="본문"></textarea>

          </p>

          <p>

          ${template.authorSelect(authors)}

          </p>

          <p><input type="submit"></p>

        </form>

          `, `<a href="/create">create</a>`);

        response.writeHead(200);

        response.end(HTML);

      });

    });

기존의 topic테이블 불러오는 명령문 안에 author테이블을 불러오는 명령어를 넣고 기존의 하위 명령어를 author명령어 밑에 둔다.

입력시에 저자를 선택하게 하기 위한 코드를 삽입하는데, 기존 템플릿에 객체로 추가하여 해당 기능을 따로 구현한다.

authorSelect: function(authors){

        var tag ='';

        var i=0;

        while(i<authors.length){

          tag +=`<option value="${authors[i].id}">${authors[i].name}</option>`;

          i++;

        }

        return `<select name="author">${tag}</select>`;

    }

Create process에서는 단순히 post.author로 값을 불러와 쓰기만 하면 된다.

한가지 기능만 외부로 사용하게 할 경우

Module.exports = (내보낼 기능의 )이름 이지만

여러가지 기능을 모듈 하나에 넣을 경우에는

Exports.이름 = 기능

Exports.이름 = 기능

Exports.이름 = 기능

으로 각각의 기능마다 앞에 exports.을 붙인다.

보안

기존의 값을 ? 로 두고 다음 값으로 배열을 통해 값을 치환시키는 것은 보안상의 이유였다.

예를들어 링크를 눌러 쿼리문이 실행될 때

Select \* from topic where id = ? 의 경우 ?부터 쿼리 스트링 전체가 ‘’로 묶이기 때문에

누군가 쿼리스트링을 통해 데이터 삭제를 시도하더라도

Select \* from topic where id = ‘1; drop table topic;’ 으로 처리되어 실패한다.

또한 ?를 사용하지 않더라도, db.query 자체적으로 한번에 하나 이상의 처리를 금지하고 있기 때문에 db.js의 데이터베이스 설정에 multipleStatments:true 설정만 하지 않으면 예방 가능하다.

만일 설정을 해야 하는 경우 ?를 제외한 방법으로는

Select \* from topic where id = ${queryData.id} 를 Select \* from topic where id = ${db.escape(queryData.id)}로 처리하면 ?를 사용한것과 같은 효과가 나온다.

Express 설치

Npm Install express –save

const express = require('express') //모듈을 불러오는 코드

const app = express(); //app변수에 express 함수의 반환값 -> 애플리케이션

app.get('/',(req,res)=>res.send('Hello world')) //get메소드는 첫자리에 경로, 두번째자리에는 경로로 들어왔을 때 수행할 콜백 함수

app.get('/',function(req,res){

return res.send('hello world')

})

//이것과 똑같이 동작한다.

app.listen(3000,()=>console.log('example app listening on port 3000'))

//listen 메소드가 실행될 때 서버가 실행되고 3000번 포트를 리스닝하게 되고 성공하면 콜벡 함수가 수행된다.

Return 은 없어도 된다.

get메소드는 route, routing 이라고 하는데 접속 경로에 따라 다른 서비스를 한다

기존의 if로 페이지 구분한것과 동일

기존의

response.writeHead(200);

response.end(HTML);

는 성공함과 성공이후에 보내어질 결과를 전송하는데 express에서는

Response.send(html)로 한번에 처리한다.

  app.get('/page/:pageid',function(request,response){ //라우팅 방식

    response.send(request.params);

  })

/page/:pageid 로 접속하면 response의 객체 params에 :pageid위치에 실제로 들어간 값이 pageid값에 저장된다.

즉 localhost:3000/page/HTML 로 접속하면 params객체의 pageid 값은 HTML이 된다

출력시 {"pageid":"HTML"}로 출력이 된다.

기존의 querrtData.id 는 request.params.pageId

기존의 /?id=${} 으로 쿼리스트링으로 받아오던 값을

/page/${filelist[i]} 로 바꾸면 패스 파라미터 형식으로 바꿀 수 있다.

리다이렉트 방식

response.redirect(302, `/page/${title}`);

혹은

response.redirect(`/page/${title}`);

삭제 구현중

Form action=”delete\_process” 하면

/page/delete\_process로 처리된다.

Form action=”/delete\_process” 하면

바로 /delete\_process 즉 최상위에서 바로수행된다.

미들웨어

남들이 만들어 놓은 사용할 수 있는 기능

설치 npm install body-parser --save

웹브라우저에서 요청한 정보의 본체 body

그 본체를 설명하는 부분 header

본체인 데이터를 분석해서 가공해주는게 body-parser

Use 로 사용하기 하면 매번 요청마다 수행됨에 유의/효율성

Var bodyParser = require(‘body-parser’)

Add.use(bodyParser.urlencoded({extended:false}));

Main.js가 실행될 때 마다 미들웨어가 실행됨

사용자가 전송한 post 데이터를 내부적으로 분석하여 request.body 를 만들어 정보를 넣는다.

var body = '';

  request.on('data', function (data) {

    body += data;

  });

  request.on('end', function () {

    var post = qs.parse(body);

이렇게 일일이 받아서 합치는 과정을

Var post = request.body; 로 줄일 수 있다.

압축

데이터의 양이 많아질 경우 데이터 전송의 비용이 증가함

이때 압축하여 전송해야 할 필요가 생긴다.

압축 미들웨어 설치

Npm install compression –save

Var compression = require(‘compression’); 모듈 불러옴

App.use(compression())//압축 사용 자동으로 데이터를 압축해서보내줌

미들웨어 작성/만들기/생성

var myLogger = function (req, res, next) {

console.log('LOGGED');

next();

};

app.use(myLogger);

구성은 이러하며 next는 다음에 수행되어야하는 다른 미들웨어가 들어온다.

예제에서 반복적으로 나오는 fs.readdir을 미들웨어로 만들면

app.use(function (request, response, next) {

  fs.readdir('./data', function (error, filelist) {

    request.list = filelist;

    next();

  })

})

이런식으로 작성이 가능하며 기존의 filelist를 사용하던 자리에 request.list를 사용하면

반복적으로 등장하던

fs.readdir('./data', function (error, filelist) {

})

를 쓰지 않을 수 있다.

다만 이렇게 하면 페이지 요정 중 \_process 등의 리스트를 필요로 하지 않는 요청에도 반응하게 되어 비효율 적이다.

app.get('\*',function (request, response, next) {

  fs.readdir('./data', function (error, filelist) {

    request.list = filelist;

    next();

  })

})

이렇게 하면 모든 방식이 아닌 get방식에 의한 페이지 요청 전체(‘\*’)에 대해서만 적용되는 코드가 된다.

사용중인 코드들이 이미 미들웨어인 점을 알 수 있다.

미들웨어의 사용

app.get('/user/:id', function (req, res, next) {

console.log('ID:', req.params.id);

next();

}, function (req, res, next) {

res.send('User Info');

});

미들웨어 세번제 인자 값의 next()는 다음 인자로 들어온 미들웨어를실행시키고

첫번째 경로에 결로를 넣으면 수행될 범위를 특정 경로로 한정할 수 있다.

app.get('/user/:id', function (req, res, next) {

// if the user ID is 0, skip to the next route

if (req.params.id == 0) next('route');

// otherwise pass the control to the next middleware function in this stack

else next(); //

}, function (req, res, next) {

// render a regular page

res.render('regular');

});

// handler for the /user/:id path, which renders a special page

app.get('/user/:id', function (req, res, next) {

res.render('special');

});

상단의 if 문을 보면 ‘route’ 가 인자로 오는 next의경우 다음 route 미들웨어를 수행하므로 하단의 별개의 미들웨어를 실행하게 된다.

이미지 등의 정적 파일을 사용하는 방법

Express의 내장 미들웨어인 static을 사용한다

App.use(express.static(‘public’));

Public 폴더 안에서 static파일을 찾겠다는 의미.

오류처리

app.use(function(req, res, next) {

res.status(404).send('Sorry cant find that!');

});

에러처리 상단에 해당 코드를 넣으면 404 not found에 대한 설정이 가능하다.

최하단, 리스너 위에

app.use(function(err, req, res, next) {

console.error(err.stack);

res.status(500).send('Something broke!');

});

코드를 넣으면 에러에 대한 처리가 가능한데, 단 각 미들웨어문에서 err를 처리하여 next에 err을 보내야 해당 코드가 동작한다. 또한 해당 미들웨어문에 next가 인자로 들어가는지 확인해야한다.

라우터를 파일로 만들어서 분리하기

라우트들을 별도의 js파일로 뺀 뒤에

expres모듈을 불러오고, main의 var app = express();와 다르게

var router = express.Router(); 로 라우터를 리턴한다.

그리고 app 부분을 전부 router로 교체하고 main에서 모듈을 불러오도록

마지막줄에 Module.exports = router; 해준다.

그리고 이미 main에서 /topic 의 하위임이 명시되있기 때문에 분리된 파일의 route.~경로에서 /topic은 없어도 된다.

보안

$ npm install --save helmet

var helmet = require('helmet')

app.use(helmet())

종속확인

$ npm audit