테스트주도개발(TDD)로 만드는 NodeJS API 서버

목차

[1. Node.js 배우기 3](#_Toc517972583)

[2. 개발 환경 구성 3](#_Toc517972584)

[A. Node.js 터미널 실행하여 “Hello World” 출력하기 3](#_Toc517972585)

[3. NodeJS 기초 3](#_Toc517972586)

[A. 노드JS 특징 3](#_Toc517972587)

[B. 비동기 세계 5](#_Toc517972588)

[4.Node로 만나는 Hello World 5](#_Toc517972589)

[A. 헬로월드 코드읽기 5](#_Toc517972590)

[B. 라우팅 추가하기 6](#_Toc517972591)

[5. 익스프레스 (ExpressJS) 기초 7](#_Toc517972592)

[A. 어플리케이션 7](#_Toc517972593)

[B. 미들웨어 7](#_Toc517972594)

[C. 라우팅 10](#_Toc517972595)

[D. 요청객체 10](#_Toc517972596)

[E. 응답객체 10](#_Toc517972597)

[F. Hello world 익스프레스 버전 10](#_Toc517972598)

[6. npm에 대해 더 알아보기 11](#_Toc517972599)

[7. REST API란 13](#_Toc517972600)

[A. 요청 형식 13](#_Toc517972601)

[B. 응답 형식 13](#_Toc517972602)

[C. 첫 번째 API 만들기: 사용자 목록 조회 API 14](#_Toc517972603)

[8. 테스트 주도 개발(TDD; Test-Driven Development) 15](#_Toc517972604)

[A. 모카(mocha) 15](#_Toc517972605)

[B. 슈드(should) 17](#_Toc517972606)

[C. 슈퍼테스트(superTest) 18](#_Toc517972607)

[9. TDD로 하는 API 서버 개발 19](#_Toc517972608)

[A. 사용자 목록 조회 API 테스트 코드 만들기 19](#_Toc517972609)

[B. 사용자 조회 (get) 27](#_Toc517972610)

[C. 사용자 삭제 (delete) 30](#_Toc517972611)

[D. 사용자 추가 (post) 32](#_Toc517972612)

[E. 사용자 수정 (put) 34](#_Toc517972613)

[10. 라우터 클래스 37](#_Toc517972614)

[A. 컨트롤러 함수로 분리 40](#_Toc517972615)

[B. 테스트 코드 이동 42](#_Toc517972616)

[C. 테스트 환경 개선 42](#_Toc517972617)

[11. 데이터베이스 44](#_Toc517972618)

[A. 데이터베이스 소개 44](#_Toc517972619)

[B. ORM 소개 45](#_Toc517972620)

[C. 노드의 ORM 시퀄라이져 45](#_Toc517972621)

[D. 모델 정의 45](#_Toc517972622)

[E. 데이터베이스-ORM 동기화 46](#_Toc517972623)

[F. 데이터베이스와 index 컨트롤러 연동 47](#_Toc517972624)

[G. 데이터베이스와 show컨트롤러 연동 50](#_Toc517972625)

[H. 데이터베이스와 destroy컨트롤러 연동 51](#_Toc517972626)

[I. 데이터베이스와 create컨트롤러 연동 52](#_Toc517972627)

[J. 데이터베이스와 update컨트롤러 연동 53](#_Toc517972628)

[12. 마무리(포스트맨으로 실제 서버 API 검증해보기) 53](#_Toc517972629)

# 1. Node.js 배우기

(인프런-김정환[우아한형제들 개발자])

\*배우는 것들

-테스트주도개발 방법

-NodeJS 기초

-ExpressJS 기초

-Rest API 서버 개발

# 2. 개발 환경 구성

-노드 설치(Node.js)

-에디터 설치(Visual Studio Code)

-터미널 설치(iterm2 는 Mac 용 이므로 cmder사용)

## A. Node.js 터미널 실행하여 “Hello World” 출력하기

> console.log('Hello World');

Hello World

undefined

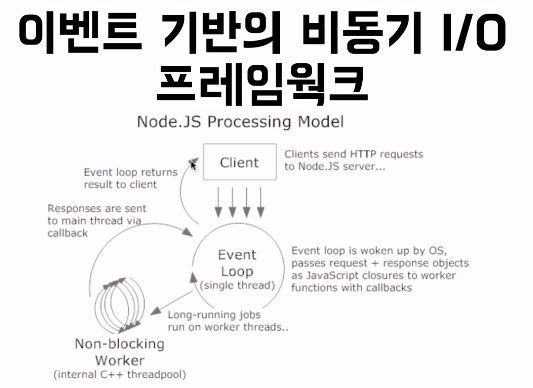
# 3. NodeJS 기초

## A. 노드JS 특징

\*브라우저 밖에서 자바스크립트 코드를 실행할 수 있다.

\*크롬에서 사용하는 V8 엔진을 사용한다. (V8은 해석기라고 생각하면 된다)

\*이벤트 기반의 비동기 I/O 프레임웍크다.

🡺 클라이언트가 Node.JS 서버로 요청을 보내면 (1), 이벤트 루프(싱글쓰레드)가 OS에 의해 활성화 되고(2), 각 요청을 이벤트 큐로 할당해서 (FIFO, 3), 요청에 대한 응답을 하나씩 처리하게 된다(4)

만약, 무거운 Job(큰 data loading 등..) 은 이벤트 루프에서 처리하지 않고 워커 쓰레드(내부 C++ 쓰레드풀)에서 처리하고, 그 결과를 다시 이벤트 루프로 큐로서 돌려주어 클라이언트로 응답처리한다(비동기 처리).

\*CommonJS를 구현한 모듈 시스템이 있다.

-기본 모듈

-써드파티 모듈

-사용자 정의 모듈

🡺(not Node.js)기존의 브라우저에서는 윈도우 컨텍스트를 사용하거나, RequireJS같은 의존성 로더를 사용했다.

🡺노드는 파일형태로 모듈을 관리할 수 있는 CommonJS로 구현

node-api라는 폴더(윈도우)를 만들고,

>mkdir node-api

해당 경로로 이동한 후,

>cd node-api

vscode를 실행하면 해당 경로에서 편집 창을 열 수 있다.

>code .

vscode 편집창이 열리면,

const http = require(‘http’);

🡺위와 같이 입력하면 ‘http’라는 모듈을 가져와서(require) http 라는 변수에 할당하면 이후에는 http 모듈이 제공하는 여러가지 메소드들을 사용할 수 있다.

ex) 사용자 정의 모듈 (a+b function 만들기)

vscode에서 index.js와 math.js 파일을 각각 다음과 같이 만든다.

\*\*index.js\*\*

|  |
| --- |
| const math = require('./math.js');  const result = math.sum(1, 2);  console.log(result); |

\*\*math.js\*\*

|  |
| --- |
| function sum (a, b) {  return a + b;  }  module.exports = {  sum: sum  }; |

🡺math.js에서 생성한 함수(sum)을 선언하여 exports 후 index.js에서 사용자 정의 모듈을 호출하여 사용할 수 있다.

## B. 비동기 세계

-노드는 기본적으로 비동기로 동작함; 노드를 사용하려면 비동기 동작방식에 익숙해져야 함

ex) readFile vs readFileSync

|  |
| --- |
| const fs = require('fs');  //const data = fs.readFileSync('./data.txt', 'utf8'); //동기방식  const data = fs.readFile('./data.txt', 'utf8', function(err, data) {  console.log(data); //비동기방식  });  console.log(data); |

🡺동기방식의 경우 readFileSync의 명령어의 응답(data.txt 파일을 모두 읽은 후 data 변수에 리턴)이 끝날 때까지 다음 명령어가 실행될 수 없다.

하지만, 비동기방식의 경우 readFile의 명령어의 응답을 기다리지 않고 다음 명령어가 수행될 수 있으며, 응답이 가능한 경우가 됐을 때 data라는 변수에 결과를 저장하여 응답처리한다.

# 4.Node로 만나는 Hello World

## A. 헬로월드 코드읽기

|  |
| --- |
| const http = require('http');  const hostname = '127.0.0.1';  const port = 3000;  const server = http.createServer((req, res) => {  res.statusCode = 200; //성공 시 리턴 값  res.setHeader('Content-Type', 'text/plain');  res.end('Hello World!\n');  });  server.listen(port, hostname, () => {  console.log(`Server running at http://${hostname}:${port}/`);  }); |

🡺코드 설명:

‘http’ 모듈을 받는 변수를 선언하고, http 모듈의 메소드 중 하나인 createServer 기능을 server라는 변수로 선언한다. () 내에 콜백 함수를 새로 만들어 선언.

‘=>’ es6 문법 중 하나로 ‘(req, res) =>’는 ‘function(req, res)’ 로 변경하여 사용할 수 있다.

마지막으로 server의 메소드 중 listen 메소드는 서버를 요청 대기 상태로 만들어주는 기능을 한다.

터미널에서

curl -X GET localhost:3000

명령어를 수행하게 되면 콘솔에

Hello World! 라는 문구를 찍는다.

## B. 라우팅 추가하기

4-A에서 생성한 서버는 root 경로로 접속했을 때만 응답을 하는 서버이다.

경로에 맞는 응답을 새로 추가할 수 있다.

4-A의 소스 중 server에 들어가는 부분을 아래와 같이 수정할 수 있다.

|  |
| --- |
| const server = http.createServer((req, res) => {  if (req.url === '/') {  res.statusCode = 200; //성공 시 리턴 값  res.setHeader('Content-Type', 'text/plain');  res.end('Hello World!\n');  } else if (req.url === '/users') {  res.statusCode = 200;  res.setHeader('Content-Type', 'text/plain');  res.end('User list');  } else {  res.statusCode = 404;  res.end('Not Found');  }  }); |

분기를 두어서 req.url이 /(root)일 때 Hello World!를 출력하고, /users 일 때는 User list를 출력하게 할 수 있다.

하지만 이렇게 되면 분기가 많아지는 경우 코드가 무거워지고 중복코드가 많아 가독성이 떨어진다.

🡺Node.js의 프레임워크 중 Express.js 를 사용하여 해결할 수 있다.

# 5. 익스프레스 (ExpressJS) 기초

A~E까지의 다섯가지 개념을 이해해야 제대로 사용할 수 있다.

## A. 어플리케이션

익스프레스 인스턴스를 어플리케이션이라고 한다.

서버에 필요한 기능인 미들웨어를 어플리케이션에 추가한다.

라우팅 설정을 할 수 있다.

서버를 요청 대기 상태로 만들 수 있다.

|  |
| --- |
| const express = require('express');  const app = express();  app.listen(3000, function(){  console.log('Server is running');  }) |

## B. 미들웨어

미들웨어는 함수들의 연속이다.

로깅 미들웨어를 만들어 보자.

써드파티 미들웨어를 사용해 보자.

일반 미들웨어 vs 에러 미들웨어

404, 500 에러 미들웨어를 만들어 보자.

|  |
| --- |
| const express = require('express');  const app = express();  function logger(req, res, next){  console.log('i am logger');  next();  }  app.use(logger);  app.listen(3000, function(){  console.log('Server is running');  }) |

🡺미들웨어를 추가할 때는 use라는 메소드를 사용한다. logger라는 함수를 호출하는 미들웨어를 추가한 뒤(function을 선언할 때 next(); 구문을 쓰지 않으면 명령어 수행이 종료되지 않고 대기하게 되니 꼭 써줘야 한다),

터미널에서 express를 설치부터 하고

npm install express

server를 활성화한 뒤,

node index.js

다른 터미널에서 요청을 보내보면 된다.

curl -X GET localhost:3000

응답은

서버(i am logger)

클라이언트(<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>Error</title>

</head>

<body>

<pre>Cannot GET /</pre>

</body>

</html>)

\*미들웨어 실행 순서

|  |
| --- |
| const express = require('express');  const app = express();  function logger(req, res, next){  console.log('i am logger');  next();  }  function logger2(req, res, next){  console.log('i am logger2');  next();  }  app.use(logger);  app.use(logger2);  app.listen(3000, function(){  console.log('Server is running');  }) |

🡺logger2를 호출하는 새 미들웨어를 선언하면 클라이언트 호출 시 동작 순서는,

logger 라는 미들웨어를 먼저 호출하여 logger 함수를 실행하게 되고 next()라는 명령어를 만나 미들웨어 수행이 종료된다.

이후 logger2 미들웨어를 수행하게 된다.

\*다른 개발자가 만든 미들웨어 사용하기 (써드파티 미들웨어)

npm install express 라는 명령어를 통해, 기존에 만들어져 있는 익스프레스 프레임웍을 설치했듯이,

<https://www.npmjs.com>

사이트를 접속해 필요한 미들웨어를 찾아 명령어로 설치하여 사용할 수 있다.

npm install morgan 🡸로거를 사용할 수 있는 써드파티 미들웨어

|  |
| --- |
| const express = require('express');  const morgan = require('morgan'); //모르간 모듈 사용 선언  const app = express();  function logger(req, res, next){  console.log('i am logger');  next();  }  function logger2(req, res, next){  console.log('i am logger2');  next();  }  app.use(logger);  app.use(logger2);  app.use(morgan('dev')); //모르간 써드파티 미들웨어 실행  app.listen(3000, function(){  console.log('Server is running');  }) |

\*에러 미들웨어

|  |
| --- |
| const express = require('express');  const app = express();  function commonmw(req, res, next){  console.log('commonmw');  next(new Error('error ouccered'));  }  function errormw(err, req, res, next){  console.log(err.message);  // 에러를 처리하거나  next();  }  app.use(commonmw);  app.use(errormw);  app.listen(3000, function(){  console.log('Server is running');  }) |

일반 미들웨어가 아닌 에러를 처리하는 미들웨어를 선언하여 사용할 수 있다.

## C. 라우팅

요청 url에 대해 적절한 핸들러 함수로 연결해 주는 기능을 라우팅이라고 부른다.

어플리케이션의 get(), post() 메소드로 구현할 수 있다.

라우팅을 위한 전용 Router 클래스를 사용할 수도 있다.

## D. 요청객체

클라이언트의 요청 정보를 담은 객체를 요청(Request)객체라고 한다.

http 모듈의 request 객체를 래핑한 것이다.

req.params(), req.query(), req.body() 메소드를 주로 사용한다.

## E. 응답객체

클라이언트의 응답 정보를 담은 객체를 응답(Response)객체라고 한다.

http 모듈의 response 객체를 래핑한 것이다.

res.send(), res.status(), res.json() 메소드를 주로 사용한다.

## F. Hello world 익스프레스 버전

index.js를 다음과 같이 바꿀 수 있다.

|  |
| --- |
| var express = require('express');  var app = express();  app.get('/', function (req, res) {  res.send('Hello World!');  });  app.listen(3000, function () {  console.log('Example app listening on port 3000!');  }); |

익스프레스 모듈을 호출하고 res 래핑 객체의 send 메소드를 통해 Hello World! 문구를 출력하는 소스이다.

# 6. npm에 대해 더 알아보기

npm을 통해 익스프레스나 모르간 모듈을 설치하는 명령어를 통해

node\_modules 라는 폴더가 생성되고 그 아래 모듈에서 사용할 api가 생성된다.

하지만, 이렇게 되면 익스프레스나 모르간 모듈을 사용한 소스는 내 컴퓨터에서는 돌아가지만,

해당 모듈을 사용하고자 하는 다른 환경의 OS에서는 모듈이 존재하지 않는 이상 실행할 수 없게된다.

이 때 터미널 프로그램에서 npm init 명령어를 통해

λ npm init

This utility will walk you through creating a package.json file.

It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields

and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and

save it as a dependency in the package.json file.

Press ^C at any time to quit.

package name: (api) node-api

version: (1.0.0)

description:

entry point: (index.js)

test command:

git repository: (https://github.com/bogurs/nodejs.git)

keywords:

author:

license: (ISC)

About to write to D:\Song\Work\gogogo\nodejs\node-api\package.json:

{

"name": "node-api",

"version": "1.0.0",

"description": "",

"main": "index.js",

"dependencies": {

"express": "^4.16.3",

"morgan": "^1.9.0"

},

"devDependencies": {},

"scripts": {

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

},

"repository": {

"type": "git",

"url": "git+https://github.com/bogurs/nodejs.git"

},

"author": "",

"license": "ISC",

"bugs": {

"url": "https://github.com/bogurs/nodejs/issues"

},

"homepage": "https://github.com/bogurs/nodejs#readme"

}

Is this ok? (yes)

package.json 파일을 생성하게 되면 그 안에 내용은 다음과 같다.

|  |
| --- |
| {  "name": "node-api",  "version": "1.0.0",  "description": "",  "main": "index.js",  "dependencies": {  "express": "^4.16.3",  "morgan": "^1.9.0"  },  "devDependencies": {},  "scripts": {  "test": "echo \"Error: no test specified\" && exit 1"  },  "repository": {  "type": "git",  "url": "git+https://github.com/bogurs/nodejs.git"  },  "author": "",  "license": "ISC",  "bugs": {  "url": "https://github.com/bogurs/nodejs/issues"  },  "homepage": "https://github.com/bogurs/nodejs#readme"  } |

dependencies에 기재된 express나 morgan 을 통해 다른 pc에서 npm install을 하게 되면 express와 morgan 모듈을 모두 선행 설치하게 된다.

npm version 5 이상 부터는 –save옵션이 default

npm install 시 "–save-dev" or "no-save" 옵션을 안주면 –save(–save-prod) 가 default 로 입력 된다. 따라서 굳이 –save 옵션을 주지 않아도 package.json 에 dependency가 추가 된다.

package.json 파일에 scripts 부분에 다음과 같이 추가하여 js파일을 실행하게 할 수 있다.

|  |
| --- |
| "scripts": {  "test": "echo \"Error: no test specified\" && exit 1",  "start": "node index.js"  }, |

터미널 프로그램에서

npm start

라는 명령어를 입력하면

λ npm start

> node-api@1.0.0 start D:\Song\Work\gogogo\nodejs\node-api

> node index.js

Example app listening on port 3000!

와 같이 index.js를 실행한다.

# 7. REST API란

## A. 요청 형식

HTTP 메소드: 서버 자원에 대한 행동을 나타낸다. (동사로 표현)

-GET: 자원을 조회

-POST: 자원을 생성

-PUT: 자원을 갱신

-DELETE: 자원을 삭제

이는 익스프레스 어플리케이션의 메소드로 구현되어 있다.

app.get(), app.post() 등..

## B. 응답 형식

HTTP 상태코드(3자리의 정수로 표현. 200~ 300~ 대의 응답 코드를 많이 사용한다.)

-1XX: 아직 처리중

-2XX: 자, 여기있어!

-3XX: 잘 가~

-4XX: 니가 문제임

-5XX: 내가 문제임

-200: 성공(success), GET, PUT

-201: 작성됨(created), POST

-204: 내용 없음(No Content), DELETE

-400: 잘못된 요청 (Bad Request)

-401: 권한 없음(Unauthorized)

-404: 찾을 수 없음(Not Found)

-409: 충돌(Conflict)

-500: 서버 에러(Internal server error) ; 자세한 에러 내용은 응답 메시지로 주지 않는다.

## C. 첫 번째 API 만들기: 사용자 목록 조회 API

-GET /users

-사용자 목록을 조회하는 기능을 구현해보자!

|  |
| --- |
| var express = require('express');  var app = express();  var morgan = require('morgan');  var users = [  {id: 1, name: 'alice'},  {id: 2, name: 'bek'},  {id: 3, name: 'chris'},  ];  app.use(morgan('dev'));  app.get('/users', function (req, res) {  res.json(users);  });  app.listen(3000, function () {  console.log('Example app listening on port 3000!');  }); |

위와 같이 index.js를 작성한 후 (get(‘/users’))

명령어를 통해 서버 및 클라이언트 요청/응답은 다음과 같다.

서버(

npm start

> node-api@1.0.0 start D:\Song\Work\gogogo\nodejs\node-api

> node index.js

Example app listening on port 3000!

GET /users 200 4.770 ms – 71

)

클라이언트(

curl -X GET localhost:3000/users -v

Note: Unnecessary use of -X or --request, GET is already inferred.

\* Trying ::1...

\* TCP\_NODELAY set

\* Connected to localhost (::1) port 3000 (#0)

> GET /users HTTP/1.1

> Host: localhost:3000

> User-Agent: curl/7.55.1

> Accept: \*/\*

>

< HTTP/1.1 200 OK

< X-Powered-By: Express

< Content-Type: application/json; charset=utf-8

< Content-Length: 71

< ETag: W/"47-AswcodMnmv7+S9S8PW5fCVVRgCc"

< Date: Sun, 17 Jun 2018 13:14:28 GMT

< Connection: keep-alive

<

[{"id":1,"name":"alice"},{"id":2,"name":"bek"},{"id":3,"name":"chris"}]\* Connection #0 to host localhost left intact

)

# 8. 테스트 주도 개발(TDD; Test-Driven Development)

-TDD로 개발하자!

-mocha, should, superTest

테스트를 하나씩 통과하게 하는 개발 방식. 그냥 개발하는 것 보다 시간이 오래 걸리지만, 유지보수 시 시간을 많이 줄일 수 있는 장점이 있다.

## A. 모카(mocha)

-모카(mocha)는 테스트 코드를 돌려주는 테스트 러너

-테스트 수트: 테스트 환경으로 모카에서는 describe()으로 구현한다.

-테스트 케이스: 실제 테스트를 말하며 모카에서는 it()으로 구현한다.

먼저, mocha를 npm 명령어로 설치한다.

npm i mocha --save-dev 🡸dev용도로만 사용하는 모듈이다. package.json에는 다음과 같이 기재된다.

|  |
| --- |
| "devDependencies": {  "mocha": "^5.2.0"  }, |

이후 다음과 같은 내용의 파일을 만든다.

\*\*utils.js\*\*

|  |
| --- |
| function capitalize(str) {  return str;  }  module.exports = {  capitalize: capitalize  }; |

\*\*utils.spec.js\*\* (spec은 테스트를 위해 작성하는 파일이라고 보면 된다.)

|  |
| --- |
| const utils = require('./utils.js');  const assert = require('assert');  describe('utils.js모듈의 capitalize() 함수는', () => {  it('문자열의 첫번째 문자를 대문자로 반환한다', ()=>{  const result = utils.capitalize('hello');  assert.equal(result, 'Hello');  })  }) |

🡺현재는 function capitalize가 대문자로 반환하는 로직을 구현하지 않았기 때문에 명령어를 통해 수행하면 다음과 같은 결과가 나온다.

|  |
| --- |
| λ node\_modules\.bin\mocha utils.spec.js  utils.js모듈의 capitalize() 함수는  1) 문자열의 첫번째 문자를 대문자로 반환한다  0 passing (13ms)  1 failing  1) utils.js모듈의 capitalize() 함수는  문자열의 첫번째 문자를 대문자로 반환한다:  AssertionError [ERR\_ASSERTION]: 'hello' == 'Hello'  + expected - actual  -hello  +Hello  at Context.it (utils.spec.js:7:16) |

테스트를 성공시키기 위해 utils.js 내용을 다음과 같이 바꾼다.

\*\*utils.js\*\*

|  |
| --- |
| function capitalize(str) {  return str.charAt(0).toUpperCase() + str.slice(1);  }  module.exports = {  capitalize: capitalize  }; |

이후 명령어를 재수행 하면 다음과 같이 출력된다.

|  |
| --- |
| λ node\_modules\.bin\mocha utils.spec.js  utils.js모듈의 capitalize() 함수는  √ 문자열의 첫번째 문자를 대문자로 반환한다  1 passing (14ms) |

describe() 메소드를 통해 테스트 케이스를 작성하고,

it() 메소드를 통해 테스트 상세 로직을 작성하면 mocha를 사용하여 테스트 케이스를 작성할 수 있다.

## B. 슈드(should)

- “노드 assert 말고 서드파티 라이브러리를 사용하라”

- 슈드(should)는 검증(assertion) 라이브러리다.

- 가독성 높은 테스트 코드를 만들 수 있다.

먼저, npm 명령어로 should 를 개발용으로 설치한다.

npm i should --save-dev 🡨 package.json 파일에는 다음과 같이 기재된다.

|  |
| --- |
| "devDependencies": {  "mocha": "^5.2.0",  "should": "^13.2.1"  }, |

이후 utils.spec.js 내용을 다음과 같이 변경한다.

|  |
| --- |
| const utils = require('./utils.js');  const should = require('should');  const assert = require('assert');  describe('utils.js모듈의 capitalize() 함수는', () => {  it('문자열의 첫번째 문자를 대문자로 반환한다', ()=>{  const result = utils.capitalize('hello');  //assert.equal(result, 'Hello'); //아래와 같이 should 구문을 써서 가독성을 높일 수 있다(should는 영문법 같이 보여 해석이 용이하다)  result.should.be.equal('Hello');  })  }) |

이후 명령어를 수행하면 다음과 같은 결과가 나온다.

|  |
| --- |
| λ node\_modules\.bin\mocha utils.spec.js  utils.js모듈의 capitalize() 함수는  √ 문자열의 첫번째 문자를 대문자로 반환한다  1 passing (11ms) |

## C. 슈퍼테스트(superTest)

-지금까지의 테스트 방법은 단위 테스트: 함수의 기능 테스트(모카, 슈드)

-슈퍼테스트는 통합 테스트: API의 기능 테스트

-슈퍼테스트는 익스프레스 통합 테스트용 라이브러리다.

-내부적으로 익스프레스 서버를 구동시켜 실제 요청을 보낸 뒤 결과를 검증한다.

먼저, npm 명령어로 supertest를 설치한다.

npm i supertest --save-dev

이후 package.json 파일에 다음과 같이 supertest가 추가된다.

|  |
| --- |
| "devDependencies": {  "mocha": "^5.2.0",  "should": "^13.2.1",  "supertest": "^3.1.0"  }, |

이후 index.js를 테스트 하기 위한 index.spec.js 파일을 생성하여 다음과 같이 작성한다.

|  |
| --- |
| const app = require('./index');  const request = require('supertest');  describe('GET /users는', ()=>{  it('...', (done)=>{  request(app)  .get('/users')  .end((err, res) => {  console.log(res.body);  done();  })  })  }) |

🡺위 소스는 비동기로 동작하기 때문에 done을 호출하지 않으면 테스트가 종료되지 않는다.

또한 아래와 같이 index.js 마지막 줄에 구문을 추가한다.

|  |
| --- |
| module.exports = app; |

모카 테스트를 재수행하여 결과를 확인한다.

|  |
| --- |
| λ node\_modules\.bin\mocha index.spec.js  Example app listening on port 3000!  GET /users는  GET /users 200 3.863 ms - 71  [ { id: 1, name: 'alice' },  { id: 2, name: 'bek' },  { id: 3, name: 'chris' } ]  √ ... (96ms)  1 passing (107ms) |

# 9. TDD로 하는 API 서버 개발

## A. 사용자 목록 조회 API 테스트 코드 만들기

\*\*시나리오\*\*

-성공

--유저 객체를 담은 배열로 응답한다.

--최대 limit 개수만큼 응답한다.

-실패

--limit이 숫자형이 아니면 400을 응답한다.

--offset이 숫자형이 아니면 400을 응답한다.

\*limit: 응답 받을 데이터의 총 최대 길이

\*offset: 데이터가 많을 경우에 잘라서 보내는데 skip 한 변수가 담긴다. ?

index.spec.js 파일을 다음과 같이 수정한다.

|  |
| --- |
| const request = require('supertest');  const should = require('should');  const app = require('./index');  describe('GET /users는', ()=>{  describe('성공시', ()=>{  it('유저 객체를 담은 배열로 응답한 ', (done)=> {  request(app)  .get('/users')  .end((err, res) => {  res.body.should.be.instanceOf(Array);  done();  })  })  })  }) |

이후 모카 테스트를 재수행 한다.

|  |
| --- |
| λ node\_modules\.bin\mocha index.spec.js  Example app listening on port 3000!  GET /users는  성공시  GET /users 200 3.996 ms - 71  √ 유저 객체를 담은 배열로 응답한 (42ms) |

\*\*npm 테스트 스크립트\*\*

긴 명령어를 통해서 모카 테스트를 하기 보다는 스크립트로 등록해서 짧은 명령어로 테스트를 하고자 할 때 package.json 파일을 아래와 같이 수정한다.

|  |
| --- |
| "scripts": {  "test": "mocha index.spec.js",  "start": "node index.js"  }, |

이후 아래와 같은 명령어로 테스트를 재수행할 수 있다.

npm test

|  |
| --- |
| λ npm test  > node-api@1.0.0 test D:\Song\Work\gogogo\nodejs\node-api  > mocha index.spec.js  Example app listening on port 3000!  GET /users는  성공시  GET /users 200 5.101 ms - 71  √ 유저 객체를 담은 배열로 응답한 (38ms)  1 passing (63ms) |

성공 시 2번(최대 limit 개수만큼 응답한다)를 만들기 위해 아래와 같이 테스트 코드를 작성한다.

\*\*index.spen.js\*\*

|  |
| --- |
| const request = require('supertest');  const should = require('should');  const app = require('./index');  describe('GET /users는', ()=>{  describe('성공시', ()=>{  it('유저 객체를 담은 배열로 응답한다 ', (done)=> {  request(app)  .get('/users')  .end((err, res) => {  res.body.should.be.instanceOf(Array);  done();  });  });  it('최대 limit 갯수만큼 응답한다 ', (done)=> {  request(app)  .get('/users?limit=2')  .end((err, res) => {  res.body.should.have.lengthOf(2);  done();  });  });  });  }); |

이후 테스트를 재수행한다.

|  |
| --- |
| λ npm t  > node-api@1.0.0 test D:\Song\Work\gogogo\nodejs\node-api  > mocha index.spec.js  Example app listening on port 3000!  GET /users는  성공시  GET /users 200 4.056 ms - 71  √ 유저 객체를 담은 배열로 응답한  GET /users?limits=2 200 0.314 ms - 71  1) 최대 limit 갯수만큼 응답한다  1 passing (77ms)  1 failing  1) GET /users는  성공시  최대 limit 갯수만큼 응답한다 :  Uncaught AssertionError: expected Array [  Object { id: 1, name: 'alice' },  Object { id: 2, name: 'bek' },  Object { id: 3, name: 'chris' }  ] to have property length of 2 (got 3)  at Assertion.fail (node\_modules\should\cjs\should.js:275:17)  at Assertion.value (node\_modules\should\cjs\should.js:356:19)  at Test.request.get.end (index.spec.js:20:38)  at Test.assert (node\_modules\supertest\lib\test.js:181:6)  at Server.assert (node\_modules\supertest\lib\test.js:131:12)  at emitCloseNT (net.js:1659:8)  at \_combinedTickCallback (internal/process/next\_tick.js:135:11)  at process.\_tickCallback (internal/process/next\_tick.js:180:9) |

🡪우리가 index.js에 정의한 배열은 3개를 받아오므로 테스트는 성공한 것이다.

위와 같은 상황에서 정상적인 테스트를 하기 위해 index.js를 다음과 같이 수정한다.

\*\*index.js\*\*

|  |
| --- |
| var express = require('express');  var app = express();  var morgan = require('morgan');  var users = [  {id: 1, name: 'alice'},  {id: 2, name: 'bek'},  {id: 3, name: 'chris'},  ];  app.use(morgan('dev'));  app.get('/users', function (req, res) {  const limit = req.query.limit;  res.json(users.slice(0, limit));  });  app.listen(3000, function () {  console.log('Example app listening on port 3000!');  });  module.exports = app; |

이후 테스트를 재수행한다.

|  |
| --- |
| λ npm t  > node-api@1.0.0 test D:\Song\Work\gogogo\nodejs\node-api  > mocha index.spec.js  Example app listening on port 3000!  GET /users는  성공시  GET /users 200 2.963 ms - 71  √ 유저 객체를 담은 배열로 응답한 (38ms)  GET /users?limit=2 200 0.568 ms - 47  √ 최대 limit 갯수만큼 응답한다  2 passing (73ms) |

실패 시 1번(limit이 숫자형이 아니면 400을 응답한다.)을 만들기 위해 테스트 코드를 작성한다.

\*\*index.spec.js\*\*

|  |
| --- |
| const request = require('supertest');  const should = require('should');  const app = require('./index');  describe('GET /users는', ()=>{  describe('성공시', ()=>{  it('유저 객체를 담은 배열로 응답한다 ', (done)=> {  request(app)  .get('/users')  .end((err, res) => {  res.body.should.be.instanceOf(Array);  done();  });  });  it('최대 limit 갯수만큼 응답한다 ', (done)=> {  request(app)  .get('/users?limit=2')  .end((err, res) => {  res.body.should.have.lengthOf(2);  done();  });  });  });  describe('실패시', ()=>{  it('limit이 숫자형이 아니면 400을 응답한다 ', (done)=> {  request(app)  .get('/users?limit=two')  .expect(400)  .end(done);  });  });  }); |

이후 테스트를 재수행한다.

|  |
| --- |
| λ npm t  > node-api@1.0.0 test D:\Song\Work\gogogo\nodejs\node-api  > mocha index.spec.js  Example app listening on port 3000!  GET /users는  성공시  GET /users 200 6.850 ms - 71  √ 유저 객체를 담은 배열로 응답한다 (48ms)  GET /users?limit=2 200 0.578 ms - 47  √ 최대 limit 갯수만큼 응답한다  실패시  GET /users?limit=two 200 0.255 ms - 2  1) limit이 숫자형이 아니면 400을 응답한다  2 passing (95ms)  1 failing  1) GET /users는  실패시  limit이 숫자형이 아니면 400을 응답한다 :  Error: expected 400 "Bad Request", got 200 "OK"  at Test.\_assertStatus (node\_modules\supertest\lib\test.js:268:12)  at Test.\_assertFunction (node\_modules\supertest\lib\test.js:283:11)  at Test.assert (node\_modules\supertest\lib\test.js:173:18)  at Server.assert (node\_modules\supertest\lib\test.js:131:12)  at emitCloseNT (net.js:1659:8)  at \_combinedTickCallback (internal/process/next\_tick.js:135:11)  at process.\_tickCallback (internal/process/next\_tick.js:180:9) |

🡪 숫자형이 아님에도 불구하고 200을 리턴 받아 테스트에 성공하지 못했다.

제대로 된 테스트를 위해 index.js 를 다음과 같이 변경한다.

\*\*index.js\*\*

|  |
| --- |
| var express = require('express');  var app = express();  var morgan = require('morgan');  var users = [  {id: 1, name: 'alice'},  {id: 2, name: 'bek'},  {id: 3, name: 'chris'},  ];  app.use(morgan('dev'));  app.get('/users', function (req, res) {  const limit = parseInt(req.query.limit, 10);  if(Number.isNaN(limit)){  return res.status(400).end();  }  res.json(users.slice(0, limit));  });  app.listen(3000, function () {  console.log('Example app listening on port 3000!');  });  module.exports = app; |

🡪request의 limit 변수를 10진수로 받아 isNaN(숫자가 아니면 true) 분기를 추가하여 400을 리턴하도록 구현하여 테스트를 재수행한다.

|  |
| --- |
| λ npm t  > node-api@1.0.0 test D:\Song\Work\gogogo\nodejs\node-api  > mocha index.spec.js  Example app listening on port 3000!  GET /users는  성공시  GET /users 400 1.460 ms - -  1) 유저 객체를 담은 배열로 응답한다  GET /users?limit=2 200 2.761 ms - 47  √ 최대 limit 갯수만큼 응답한다  실패시  GET /users?limit=two 400 0.138 ms - -  √ limit이 숫자형이 아니면 400을 응답한다  2 passing (102ms)  1 failing  1) GET /users는  성공시  유저 객체를 담은 배열로 응답한다 :  Uncaught AssertionError: expected Object {} to be an instance of Array  at Assertion.fail (node\_modules\should\cjs\should.js:275:17)  at Assertion.value (node\_modules\should\cjs\should.js:356:19)  at Test.request.get.end (index.spec.js:11:36)  at Test.assert (node\_modules\supertest\lib\test.js:181:6)  at Server.assert (node\_modules\supertest\lib\test.js:131:12)  at emitCloseNT (net.js:1659:8)  at \_combinedTickCallback (internal/process/next\_tick.js:135:11)  at process.\_tickCallback (internal/process/next\_tick.js:180:9) |

🡪실패 시 1번 항목은 테스트가 성공했지만, 성공 시 1번에 대한 테스트가 실패났다. 원인은 index.js에서 limit을 받아 올 때 성공 시 1번은 limit을 null로 넘겨주지 않고 있어서 isNaN 분기에 true를 리턴하였고 그 결과 400을 리턴 받아 실패한 경우이다.

위와 같은 상황을 방지하기 위해 index.js를 다음과 같이 수정한다.

\*\*index.js\*\*

|  |
| --- |
| var express = require('express');  var app = express();  var morgan = require('morgan');  var users = [  {id: 1, name: 'alice'},  {id: 2, name: 'bek'},  {id: 3, name: 'chris'},  ];  app.use(morgan('dev'));  app.get('/users', function (req, res) {  req.query.limit = req.query.limit || 10;  const limit = parseInt(req.query.limit, 10);  if(Number.isNaN(limit)){  return res.status(400).end();  }  res.json(users.slice(0, limit));  });  app.listen(3000, function () {  console.log('Example app listening on port 3000!');  });  module.exports = app; |

🡪req.query.limit = req.query.limit || 10; // request의 limit 변수를 받아오면 그 값 그대로 저장하고, 아니면 10을 default 값으로 할당한다.

이후 테스트를 재수행하면 다음과 같다.

|  |
| --- |
| λ npm t  > node-api@1.0.0 test D:\Song\Work\gogogo\nodejs\node-api  > mocha index.spec.js  Example app listening on port 3000!  GET /users는  성공시  GET /users 200 10.071 ms - 71  √ 유저 객체를 담은 배열로 응답한다 (42ms)  GET /users?limit=2 200 0.454 ms - 47  √ 최대 limit 갯수만큼 응답한다  실패시  GET /users?limit=two 400 0.123 ms - -  √ limit이 숫자형이 아니면 400을 응답한다  3 passing (91ms) |

모든 테스트 케이스가 성공한 것을 확인할 수 있다.

## B. 사용자 조회 (get)

아래와 같이 index.spec.js 를 수정하여 “users/:id” 에 대한 테스트를 준비한다.

\*\*index.spec.js\*\*

|  |
| --- |
| … 위 생략 …  describe('GET /users/1은', ()=>{  describe('성공시', ()=>{  it('id가 1인 유저 객체를 반환한다 ', (done)=> {  request(app)  .get('/users/1')  .end((err, res) => {  res.body.should.have.property('id', 1);  done();  });  });  });  }); |

테스트를 수행하면 아래와 같다.

|  |
| --- |
| GET /users/1 404 1.359 ms - 146  1) id가 1인 유저 객체를 반환한다  3 passing (179ms)  1 failing  1) GET /users/1은  성공시  id가 1인 유저 객체를 반환한다 :  Uncaught AssertionError: expected Object {} to have property id  at Assertion.fail (node\_modules\should\cjs\should.js:275:17)  at Assertion.value [as property] (node\_modules\should\cjs\should.js:356:19)  at Test.request.get.end (index.spec.js:41:38)  at Test.assert (node\_modules\supertest\lib\test.js:181:6)  at Server.assert (node\_modules\supertest\lib\test.js:131:12)  at emitCloseNT (net.js:1659:8)  at \_combinedTickCallback (internal/process/next\_tick.js:135:11)  at process.\_tickCallback (internal/process/next\_tick.js:180:9) |

🡪index.js에 /users/1 을 처리하는 로직이 없다.

index.js를 아래와 같이 수정해야 한다.

\*\*index.js\*\*

|  |
| --- |
| … 앞 생략 …  app.get('/users/:id', function (req, res) {  const id = parseInt(req.params.id, 10);  const user = users.filter((user) => user.id === id)[0];  res.json(user);  });  … 뒤 생략 … |

이후 테스트를 재수행하면 다음과 같다.

|  |
| --- |
| GET /users/1은  성공시  GET /users/1 200 0.762 ms - 23  √ id가 1인 유저 객체를 반환한다 |

실패 시 1(id가 숫자가 아닐 경우 400 응답)과 2(id로 유저를 찾을 수 없을 경우 404 응답)를 작성하기 위해 index.spec.js 를 다음과 같이 수정한다.

|  |
| --- |
| … 앞 생략 …  describe('실패시', ()=>{  it('id가 숫자가 아닐경우 400으로 응답한다 ', (done)=> {  request(app)  .get('/users/one')  .expect(400)  .end(done);  });  it('id로 유저를 찾을 수 없을 경우 404로 응답한다 ', (done)=> {  request(app)  .get('/users/999')  .expect(404)  .end(done);  });  });  … 뒤 생략 … |

이후 테스트를 재수행해보면,

|  |
| --- |
| GET /users/one 200 0.156 ms - -  1) id가 숫자가 아닐경우 400으로 응답한다  GET /users/999 200 0.121 ms - -  2) id로 유저를 찾을 수 없을 경우 404로 응답한다  4 passing (114ms)  2 failing  1) GET /users/1은  실패시  id가 숫자가 아닐경우 400으로 응답한다 :  Error: expected 400 "Bad Request", got 200 "OK"  at Test.\_assertStatus (node\_modules\supertest\lib\test.js:268:12)  at Test.\_assertFunction (node\_modules\supertest\lib\test.js:283:11)  at Test.assert (node\_modules\supertest\lib\test.js:173:18)  at Server.assert (node\_modules\supertest\lib\test.js:131:12)  at emitCloseNT (net.js:1659:8)  at \_combinedTickCallback (internal/process/next\_tick.js:135:11)  at process.\_tickCallback (internal/process/next\_tick.js:180:9)  2) GET /users/1은  실패시  id로 유저를 찾을 수 없을 경우 404로 응답한다 :  Error: expected 404 "Not Found", got 200 "OK"  at Test.\_assertStatus (node\_modules\supertest\lib\test.js:268:12)  at Test.\_assertFunction (node\_modules\supertest\lib\test.js:283:11)  at Test.assert (node\_modules\supertest\lib\test.js:173:18)  at Server.assert (node\_modules\supertest\lib\test.js:131:12)  at emitCloseNT (net.js:1659:8)  at \_combinedTickCallback (internal/process/next\_tick.js:135:11)  at process.\_tickCallback (internal/process/next\_tick.js:180:9) |

🡪index.js에 실패 시에 대한 로직이 없기 때문에 테스트가 실패한다. index.js를 다음과 같이 변경한다.

\*\*index.js\*\*

|  |
| --- |
| app.get('/users/:id', function (req, res) {  const id = parseInt(req.params.id, 10);  if(Number.isNaN(id)) return res.status(400).end();  const user = users.filter((user) => user.id === id)[0];  if(!user) return res.status(404).end();  res.json(user);  }); |

🡪 id값이 숫자가 아닌 경우 (isNaN) 400 리턴, id 값이 없는 경우 404 리턴.

테스트를 재수행해보면 다음과 같다.

|  |
| --- |
| 실패시  GET /users/one 400 0.067 ms - -  √ id가 숫자가 아닐경우 400으로 응답한다  GET /users/999 404 0.173 ms - -  √ id로 유저를 찾을 수 없을 경우 404로 응답한다 |

## C. 사용자 삭제 (delete)

사용자 삭제 테스트를 위해 index.spec.js에 다음과 같이 추가한다.

|  |
| --- |
| … 앞 생략 …  describe('DELETE /users/1', ()=>{  describe('성공시', ()=>{  it('204를 응답한다 ', (done)=> {  request(app)  .delete('/users/1')  .expect(204)  .end(done);  });  });  });  … 뒤 생략 … |

이후 테스트를 재수행한다.

|  |
| --- |
| 1) DELETE /users/1  성공시  204를 응답한다 :  Error: expected 204 "No Content", got 404 "Not Found"  at Test.\_assertStatus (node\_modules\supertest\lib\test.js:268:12)  at Test.\_assertFunction (node\_modules\supertest\lib\test.js:283:11)  at Test.assert (node\_modules\supertest\lib\test.js:173:18)  at Server.assert (node\_modules\supertest\lib\test.js:131:12)  at emitCloseNT (net.js:1659:8)  at \_combinedTickCallback (internal/process/next\_tick.js:135:11)  at process.\_tickCallback (internal/process/next\_tick.js:180:9) |

🡪index.js에 삭제(delete)에 대한 express app에 대한 정의가 없으므로 index.js를 다음과 같이 수정한다.

\*\*index.js\*\*

|  |
| --- |
| … 앞 생략 …  app.delete('/users/:id', (req, res) => {  const id = parseInt(req.params.id, 10);  users = users.filter(user=> user.id !== id);  res.status(204).end();  });  … 뒤 생략 … |

🡪 users(array)의 filter 기능을 활용해 받아온 id를 제외한 배열을 users 배열에 재할당한 후 204를 리턴한다.

이후 테스트를 재수행한다.

|  |
| --- |
| DELETE /users/1  성공시  DELETE /users/1 204 0.185 ms - -  √ 204를 응답한다 |

실패 시 (id가 숫자가 아닐경우 400으로 응답) 경우를 작성하기 위해 index.spec.js와 index.js를 다음과 같이 수정한다.

\*\*index.spec.js\*\*

|  |
| --- |
| … 앞 생략 …  describe('실패시', ()=>{  it('id가 숫자가 아닐경우 400으로 응답한다 ', (done)=> {  request(app)  .delete('/users/one')  .expect(400)  .end(done);  });  });  … 뒤 생략 … |

\*\*index.js\*\*

|  |
| --- |
| … 앞 생략 …  app.delete('/users/:id', (req, res) => {  const id = parseInt(req.params.id, 10);  if(Number.isNaN(id)) return res.status(400).end();  users = users.filter(user=> user.id !== id);  res.status(204).end();  });  … 뒤 생략 … |

테스트를 재수행하면 다음과 같다.

|  |
| --- |
| DELETE /users/1  성공시  DELETE /users/1 204 0.247 ms - -  √ 204를 응답한다  실패시  DELETE /users/one 400 0.237 ms - -  √ id가 숫자가 아닐경우 400으로 응답한다 |

## D. 사용자 추가 (post)

사용자 추가에 대한 성공 시 1(생성된 유저 객체를 반환한다)과 2(입력한 name을 반환한다) supertest 테스트 케이스를 만들기 위해 다음과 같이 index.spec.js와 index.js 를 작성한다.

\*\*index.spec.js\*\*

|  |
| --- |
| … 앞 생략 …  describe('POST /users', ()=>{  describe('성공시', ()=>{  let name = 'daniel',  body;  before(done=>{  request(app)  .post('/users')  .send({name})  .expect(201)  .end((err, res) => {  body = res.body;  done();  });  });  it('생성된 유저 객체를 반환한다 ', ()=> {  body.should.have.property('id');  });  it('입력한 name을 반환한다 ', ()=> {  body.should.have.property('name', name);  });  });  }); |

\*\*index.js\*\*

|  |
| --- |
| var bodyParser = require('body-parser');  … 생략 …  app.use(bodyParser.json()); // for parsing application/json  app.use(bodyParser.urlencoded({ extended: true })); // for parsing application/x-www-form-urlencoded  … 생략 …  app.post('/users', (req, res) => {  const name = req.body.name;  const id = Date.now();  const user = {id, name};  users.push(user);  res.status(201).json(user);  }); |

🡪post 방식은 body에 데이터가 넘어오므로 body를 받을 수 있는 express npm 중 body-parser를 install 해야 한다. 따라서 다음과 같은 명령어를 실행하여 body-parser를 설치한다.

npm i body-parser --save

이후 테스트를 재 수행한다.

|  |
| --- |
| POST /users  성공시  POST /users 201 87.226 ms - 36  √ 생성된 유저 객체를 반환한다  √ 입력한 name을 반환한다 |

실패 시 1(name 파라미터 누락 시 400을 반환)과 2(name이 중복일 경우 409를 반환) supertest 테스트 케이스를 작성하기 위해 index.spec.js와 index.js를 다음과 같이 수정한다.

\*\*index.spec.js\*\*

|  |
| --- |
| describe('실패시', ()=>{  it('name 파라미터 누락시 400을 반환한다 ', (done)=> {  request(app)  .post('/users')  .send({})  .expect(400)  .end(done);  });  it('name이 중복일 경우 409를 반환한다 ', (done)=> {  request(app)  .post('/users')  .send({name: 'daniel'})  .expect(409)  .end(done);  });  }); |

\*\*index.js\*\*

|  |
| --- |
| app.post('/users', (req, res) => {  const name = req.body.name;  if(!name) return res.status(400).end();  const isConflict = users.filter(user => user.name === name).length;  if(isConflict) return res.status(409).end();  const id = Date.now();  const user = {id, name};  users.push(user);  res.status(201).json(user);  }); |

🡪body에 name 값이 없는 경우 400을 반환. name이 users 배열에 존재하지 않으면 409를 반환.

테스트를 재 수행하면 다음과 같다.

|  |
| --- |
| 실패시  POST /users 400 0.577 ms - -  √ name 파라미터 누락시 400을 반환한다  POST /users 409 0.241 ms - -  √ name이 중복일 경우 409를 반환한다 |

## E. 사용자 수정 (put)

사용자 수정 supertest를 위해 index.spec.js와 index.js를 다음과 같이 작성한다.

\*\*index.spec.js\*\*

|  |
| --- |
| describe('PUT /users/:id', ()=>{  describe('성공시', ()=>{  it('변경된 name을 응답한다 ', (done)=> {  const name = 'chally';  request(app)  .put('/users/3')  .send({name})  .end((err, res) =>{  res.body.should.have.property('name', name);  done();  })  });  });  }); |

\*\*index.js\*\*

|  |
| --- |
| app.put('/users/:id', (req, res) => {  const id = parseInt(req.params.id, 10);  const name = req.body.name;  const user = users.filter(user=> user.id === id)[0];  user.name = name;  res.json(user);  }); |

테스트를 재 수행하면 다음과 같다.

|  |
| --- |
| PUT /users/:id  성공시  PUT /users/3 200 0.496 ms - 24  √ 변경된 name을 응답한다 |

실패 시 1(정수가 아닌 id일 경우 400을 응답), 2(name이 없을 경우 400을 응답), 3(없는 유저일 경우 404를 응답), 4(이름이 중복일 경우 409를 응답) supertest 테스트 케이스를 생성하기 위해 index.spec.js와 index.js에 다음과 같은 코드를 추가 및 수정한다.

\*\*index.spec.js\*\*

|  |
| --- |
| … 앞 생략 …  describe('실패시', ()=>{  it('정수가 아닌 id일 경우 400을 응답한다 ', (done)=> {  request(app)  .put('/users/one')  .expect(400)  .end(done);  });  it('name이 없을 경우 400을 응답한다 ', (done)=> {  request(app)  .put('/users/1')  .send({})  .expect(400)  .end(done);  });  it('없는 유저일 경우 404를 응답한다 ', (done)=> {  request(app)  .put('/users/999')  .send({name: 'foo'})  .expect(404)  .end(done);  });  it('이름이 중복일 경우 409를 응답 ', (done)=> {  request(app)  .put('/users/3')  .send({name: 'bek'})  .expect(409)  .end(done);  });  }); |

\*\*index.js\*\*

|  |
| --- |
| … 앞 생략 …  app.put('/users/:id', (req, res) => {  const id = parseInt(req.params.id, 10);  if(Number.isNaN(id)) return res.status(400).end();  const name = req.body.name;  if(!name) return res.status(400).end();  const isConflict = users.filter(user => user.name === name).length;  if(isConflict) return res.status(409).end();  const user = users.filter(user=> user.id === id)[0];  if(!user) return res.status(404).end();  user.name = name;  res.json(user);  });  … 뒤 생략 … |

테스트를 재 수행하면 다음과 같다.

|  |
| --- |
| 실패시  PUT /users/one 400 0.209 ms - -  √ 정수가 아닌 id일 경우 400을 응답한다  PUT /users/1 400 0.229 ms - -  √ name이 없을 경우 400을 응답한다  PUT /users/999 404 0.292 ms - -  √ 없는 유저일 경우 404를 응답한다  PUT /users/3 409 0.260 ms - -  √ 이름이 중복일 경우 409를 응답 |

# 10. 라우터 클래스

현재 index.js는 코드의 길이가 길기 때문에 폴더 및 파일을 아래와 같이 재구성한다.

\*\*./api/user/index.js\*\*

|  |
| --- |
| var users = [  {id: 1, name: 'alice'},  {id: 2, name: 'bek'},  {id: 3, name: 'chris'},  ];  app.get('/users', function (req, res) {  req.query.limit = req.query.limit || 10;  const limit = parseInt(req.query.limit, 10);  if(Number.isNaN(limit)){  return res.status(400).end();  }  res.json(users.slice(0, limit));  });    app.get('/users/:id', function (req, res) {  const id = parseInt(req.params.id, 10);  if(Number.isNaN(id)) return res.status(400).end();    const user = users.filter((user) => user.id === id)[0];  if(!user) return res.status(404).end();  res.json(user);  });    app.delete('/users/:id', (req, res) => {  const id = parseInt(req.params.id, 10);  if(Number.isNaN(id)) return res.status(400).end();    users = users.filter(user=> user.id !== id);  res.status(204).end();  });    app.post('/users', (req, res) => {  const name = req.body.name;  if(!name) return res.status(400).end();  const isConflict = users.filter(user => user.name === name).length;  if(isConflict) return res.status(409).end();    const id = Date.now();  const user = {id, name};  users.push(user);  res.status(201).json(user);  });    app.put('/users/:id', (req, res) => {  const id = parseInt(req.params.id, 10);  if(Number.isNaN(id)) return res.status(400).end();    const name = req.body.name;  if(!name) return res.status(400).end();  const isConflict = users.filter(user => user.name === name).length;  if(isConflict) return res.status(409).end();    const user = users.filter(user=> user.id === id)[0];  if(!user) return res.status(404).end();    user.name = name;    res.json(user);  }); |

위와 같은 파일을 생성한 후 (원래 만들었던) index.js에서 라우팅하여 사용하기 위해선 ./api/user/index.js 파일에 라우팅을 위한 몇 가지 선언을 해줘야 한다. 다음과 같이 수정한다.

\*\*./api/user/index.js\*\*

|  |
| --- |
| const express = require('express');  const router = express.Router();//라우터 사용을 위한 선언  var users = [  {id: 1, name: 'alice'},  {id: 2, name: 'bek'},  {id: 3, name: 'chris'},  ];  router.get('/', function (req, res) {  req.query.limit = req.query.limit || 10;  const limit = parseInt(req.query.limit, 10);  if(Number.isNaN(limit)){  return res.status(400).end();  }  res.json(users.slice(0, limit));  });    router.get('/:id', function (req, res) {  const id = parseInt(req.params.id, 10);  if(Number.isNaN(id)) return res.status(400).end();    const user = users.filter((user) => user.id === id)[0];  if(!user) return res.status(404).end();  res.json(user);  });    router.delete('/:id', (req, res) => {  const id = parseInt(req.params.id, 10);  if(Number.isNaN(id)) return res.status(400).end();    users = users.filter(user=> user.id !== id);  res.status(204).end();  });    router.post('/', (req, res) => {  const name = req.body.name;  if(!name) return res.status(400).end();  const isConflict = users.filter(user => user.name === name).length;  if(isConflict) return res.status(409).end();    const id = Date.now();  const user = {id, name};  users.push(user);  res.status(201).json(user);  });    router.put('/:id', (req, res) => {  const id = parseInt(req.params.id, 10);  if(Number.isNaN(id)) return res.status(400).end();    const name = req.body.name;  if(!name) return res.status(400).end();  const isConflict = users.filter(user => user.name === name).length;  if(isConflict) return res.status(409).end();    const user = users.filter(user=> user.id === id)[0];  if(!user) return res.status(404).end();    user.name = name;    res.json(user);  });  module.exports = router; //라우터 exports(메인 index.js에서 사용하기 위함) |

이제 메인 index.js 에서 라우터 앱을 사용하기 위해 다음과 같이 수정한다.

\*\*index.js\*\*

|  |
| --- |
| var express = require('express');  var app = express();  var morgan = require('morgan');  var bodyParser = require('body-parser');  var user = require('./api/user') //./api/user 내 파일을 로드  app.use(morgan('dev'));  app.use(bodyParser.json()); // for parsing application/json  app.use(bodyParser.urlencoded({ extended: true })); // for parsing application/x-www-form-urlencoded  app.use('/users', user); //라우터를 사용하기 위한 미들웨어 설정  app.listen(3000, function () {  console.log('Example app listening on port 3000!');  });  module.exports = app; |

이후 테스트를 재 수행하면 index.js기능을 나누기 전과 같이 테스트가 정상 진행된다.

## A. 컨트롤러 함수로 분리

컨트롤러 함수로 분리하는 것은 역할에 따라 분리한 것으로 보면 되는데, index.js에서는 요청에 대한 분기 처리, 즉 라우팅을 하는 역할이라면 user.ctrl.js는 라우팅 후 수행되는 로직, 즉 api 로직을 수행하는 역할이다. 성격이 달라서 두 파일로 분리한 것으로 이해하면 된다.

앞서 index.js를 분리했다면, 이번엔 컨트롤러 파일인 user.ctrl.js에 function들을 모두 분리해줄 수 있다. 수정한 결과 ./api/user/index.js와 ./api/user/user.ctrl.js 파일은 다음과 같다.

\*\*./api/user/index.js\*\*

|  |
| --- |
| const express = require('express');  const router = express.Router(); //라우터 사용을 위한 선언  const ctrl = require('./user.ctrl');  router.get('/', ctrl.index);  router.get('/:id', ctrl.show);  router.delete('/:id', ctrl.destroy);  router.post('/', ctrl.create);  router.put('/:id', ctrl.update);  module.exports = router; //라우터 exports(메인 index.js에서 사용하기 위함) |

\*\*./api/user/user.ctrl.js\*\*

|  |
| --- |
| var users = [  {id: 1, name: 'alice'},  {id: 2, name: 'bek'},  {id: 3, name: 'chris'},  ];  const index = function (req, res) {  req.query.limit = req.query.limit || 10;  const limit = parseInt(req.query.limit, 10);  if(Number.isNaN(limit)){  return res.status(400).end();  }  res.json(users.slice(0, limit));  };  const show = function (req, res) {  const id = parseInt(req.params.id, 10);  if(Number.isNaN(id)) return res.status(400).end();    const user = users.filter((user) => user.id === id)[0];  if(!user) return res.status(404).end();  res.json(user);  };  const destroy = (req, res) => {  const id = parseInt(req.params.id, 10);  if(Number.isNaN(id)) return res.status(400).end();    users = users.filter(user=> user.id !== id);  res.status(204).end();  };  const create = (req, res) => {  const name = req.body.name;  if(!name) return res.status(400).end();  const isConflict = users.filter(user => user.name === name).length;  if(isConflict) return res.status(409).end();    const id = Date.now();  const user = {id, name};  users.push(user);  res.status(201).json(user);  };  const update = (req, res) => {  const id = parseInt(req.params.id, 10);  if(Number.isNaN(id)) return res.status(400).end();    const name = req.body.name;  if(!name) return res.status(400).end();  const isConflict = users.filter(user => user.name === name).length;  if(isConflict) return res.status(409).end();    const user = users.filter(user=> user.id === id)[0];  if(!user) return res.status(404).end();    user.name = name;    res.json(user);  };  module.exports = {  index, //es6 문법으로 index: index를 축소해서 쓴 것이다.  show,  destroy,  create,  update  }; |

이후 테스트를 재 수행하면 정상 결과를 얻을 수 있다. TDD 기반 supertest 테스트 케이스를 작성해둔 것을 토대로 소스 리팩토링이 쉬워진 것을 느껴야 한다.

## B. 테스트 코드 이동

기존 index.spec.js 테스트 파일의 내용을 ./api/user/user.spec.js로 모두 이동한다. 테스트 코드를 가리켜주는 package.json 파일을 다음과 같이 수정한다.

|  |
| --- |
| "scripts": {  "test": "mocha api/user/user.spec.js",  "start": "node index.js"  }, |

이후 테스트를 재 수행하면 정상적인 결과를 얻을 수 있다.

## C. 테스트 환경 개선

npm t 명령어를 수행했을 때 테스트 결과 출력 화면에 테스트와 관련이 없는 내용들이 출력되므로 그 내용들을 테스트인 경우에는 출력하지 않도록 설정한다.

우선 package.json 파일에 scripts;test 부분과 index.js를 다음과 같이 변경한다.

|  |
| --- |
| "scripts": {  "test": "set NODE\_ENV=test&&mocha api/user/user.spec.js",  "start": "node index.js"  }, |

\*\*index.js\*\*

|  |
| --- |
| … 앞 생략 …  if (process.env.NODE\_ENV !== 'test') {  app.use(morgan('dev'));  }  … 뒤 생략 … |

이후 테스트를 재 수행하면 다음과 같다.

|  |
| --- |
| λ npm t  > node-api@1.0.0 test D:\Song\Work\gogogo\nodejs\node-api  > set NODE\_ENV=test&&mocha api/user/user.spec.js  Example app listening on port 3000!  GET /users는  성공시  √ 유저 객체를 담은 배열로 응답한다 (130ms)  √ 최대 limit 갯수만큼 응답한다 (107ms)  실패시  √ limit이 숫자형이 아니면 400을 응답한다  GET /users/1은  성공시  √ id가 1인 유저 객체를 반환한다  실패시  √ id가 숫자가 아닐경우 400으로 응답한다  √ id로 유저를 찾을 수 없을 경우 404로 응답한다  DELETE /users/1  성공시  √ 204를 응답한다  실패시  √ id가 숫자가 아닐경우 400으로 응답한다  POST /users  성공시  √ 생성된 유저 객체를 반환한다  √ 입력한 name을 반환한다  실패시  √ name 파라미터 누락시 400을 반환한다  √ name이 중복일 경우 409를 반환한다  PUT /users/:id  성공시  √ 변경된 name을 응답한다  실패시  √ 정수가 아닌 id일 경우 400을 응답한다  √ name이 없을 경우 400을 응답한다  √ 없는 유저일 경우 404를 응답한다  √ 이름이 중복일 경우 409를 응답  17 passing (389ms) |

🡪이전보다 더 깔끔하게 출력되는 것을 볼 수 있다.

‘Example app listening on port 3000!’ 이라는 문구도 테스트 진행 시에는 필요하지 않으므로, bin이라는 폴더를 생성하여 그 안에 [www.js](http://www.js)파일을 생성하여 다음과 같이 작성한다. 또한 package.json에 scripts;start 부분도 변경하여 [www.js](http://www.js)파일을 사용할 수 있게 변경한다.

\*\*bin/www.js\*\*

|  |
| --- |
| const app = require('../index');  app.listen(3000, function () {  console.log('Example app listening on port 3000!');  }); |

\*\*package.json\*\*

|  |
| --- |
| "scripts": {  "test": "set NODE\_ENV=test&&mocha api/user/user.spec.js",  "start": "node bin/www.js"  }, |

이후 npm t를 수행하면 테스트 문구를 제외한 문구는 나오지 않고, start를 수행하면 다음과 같은 결과가 나온다.

|  |
| --- |
| λ npm start  > node-api@1.0.0 start D:\Song\Work\gogogo\nodejs\node-api  > node bin/www.js  Example app listening on port 3000! |

# 11. 데이터베이스

## A. 데이터베이스 소개

-SQL

--MySQL, PostgreSQL, Aurora, Sqlite (이걸 사용한다)

-NoSQL (JSON의 도큐먼트 형식의 DB)

--MongoDB, DynamoDB

-In Memory DB (재구동 되면 날아가는 DB이지만, 속도 성능 이점이 있어 인증과 같은 서비스에 사용한다)

--Redis, Memcashed

## B. ORM 소개

강의에서는 일반 SQL(DML)은 사용하지 않는다. 대신…

-데이터베이스를 객체로 추상화해 놓은 것을 ORM(Object Relational Mapping)이라고 한다.

-쿼리를 직접 작성하는 대신 ORM의 메소드로 데이터 관리할 수 있는 것이 장점이다.

-노드에서 SQL ORM은 시퀄라이져(Sequelize) 이다.

## C. 노드의 ORM 시퀄라이져

ex)

-insert users(‘name’) value(‘alice’);

🡪User.create({name: ‘alice’})

-select \* from users;

🡪User.findAll()

-update users set name = ‘bek’ where id = 1;

🡪User.update({name: ‘bek'}, {where: {id: 1}});

delete from users where id = 1;

🡪User.destroy({where: {id: 1}});

위 구문에서 User는 ‘모델’이라고 한다.

모델은 데이터베이스 테이블을 ORM으로 추상화한 것이다.

## D. 모델 정의

DB 모델을 정의하기 위해 models.js 파일을 생성하고 다음과 같이 작성한다. 앞서 npm 명령어를 통해 모델을 사용하기 위한 sequelize와 sqlite3를 설치한다.

npm i sequelize sqlite3 --save

\*\*models.js\*\*

|  |
| --- |
| const Sequelize = require('sequelize');  const sequelize = new Sequelize({  dialect: 'sqlite',  db: './db.sqlite'  });  //User DB 모델 생성  const User = sequelize.define('User', {  name: Sequelize.STRING  });  module.exports = {Sequelize, sequelize, User}; |

## E. 데이터베이스-ORM 동기화

[www.js](http://www.js) 수행 시 ORM 동기화를 해주기 위해서 sync-db.js라는 파일을 생성한 후 다음과 같이 작성한다.

|  |
| --- |
| const models = require('../models');  module.exports = () => {  return models.sequelize.sync({force: true}); //force: true는 기존에 DB가 있더라도 재생성하겠다는 의미이다.  } |

또한 www.js에서 sync-db.js를 사용하기 위해 다음과 같이 작성한다.

|  |
| --- |
| const app = require('../index');  const syncDb = require('./sync-db');  syncDb().then( ()=>{ //콜백함수의 파라미터로 인자를 사용하지 않겠다는 의미이며 '()' 또는 '\_'를 사용할 수 있다.  console.log('Sync database!');  app.listen(3000, function () {  console.log('Example app listening on port 3000!');  });  }); |

이후 npm start를 수행하면 로컬 경로에 db.sqlite라는 파일(2진파일)이 생성된다.

|  |
| --- |
| λ npm start  > node-api@1.0.0 start D:\Song\Work\gogogo\nodejs\node-api  > node bin/www.js  sequelize deprecated String based operators are now deprecated. Please use Symbol based operators for better security, read more at http://docs.sequelizejs.com/manual/tutorial/querying.html#operators node\_modules\sequelize\lib\sequelize.js:242:13  Executing (default): DROP TABLE IF EXISTS `Users`;  Executing (default): DROP TABLE IF EXISTS `Users`;  Executing (default): CREATE TABLE IF NOT EXISTS `Users` (`id` INTEGER PRIMARY KEY AUTOINCREMENT, `name` VARCHAR(255), `createdAt` DATETIME NOT NULL, `updatedAt` DATETIME NOT NULL);  Executing (default): PRAGMA INDEX\_LIST(`Users`)  Sync database!  Example app listening on port 3000! |

## F. 데이터베이스와 index 컨트롤러 연동

이제 데이터베이스와 index 컨트롤러를 연동해서 첫 번째 테스트 케이스를 수행해보자. 먼저 users라는 배열을 담고 있던 user.ctrl.js파일에 models를 가져올 수 있도록 선언하는 부분과 리턴 해야 하는 users를 콜백함수로 리턴하는 부분을 추가한다.

\*\*user.ctrl.js\*\*

|  |
| --- |
| const models = require('../../models');  const index = function (req, res) {  req.query.limit = req.query.limit || 10;  const limit = parseInt(req.query.limit, 10);  if(Number.isNaN(limit)){  return res.status(400).end();  }  models.User.findAll({})  .then(users => {  res.json(users);  });  };  … 뒤 생략 … |

🡪뒷 부분의 여러 테스트 케이스 중 1개의 케이스만 검증할 것이므로 index를 호출하는 부분만 수정하였다.

이후 테스트를 갖고 있는 파일인 user.spec.js 파일을 수정한다.

|  |
| --- |
| const request = require('supertest');  const should = require('should');  const app = require('../../');  const models = require('../../models');  describe('GET /users는', ()=>{  describe('성공시', ()=>{  before(()=>models.sequelize.sync({force: true}));  it.only('유저 객체를 담은 배열로 응답한다 ', (done)=> {  request(app)  .get('/users')  .end((err, res) => {  res.body.should.be.instanceOf(Array);  done();  });  }); |

🡪models라는 파일을 사용할 수 있도록 선언(const)하고, before()를 추가하여 db-sync를 맞춰준다. it.only는 모카 테스트 중 only라는 기능을 사용하여 only가 걸려있는 부분만 테스트 진행하도록 해준다.

이후 테스트를 진행하면 다음과 같은 결과를 얻을 수 있다.

|  |
| --- |
| λ npm t  > node-api@1.0.0 test D:\Song\Work\gogogo\nodejs\node-api  > set NODE\_ENV=test&&mocha api/user/user.spec.js  sequelize deprecated String based operators are now deprecated. Please use Symbol based operators for better security, read more at http://docs.sequelizejs.com/manual/tutorial/querying.html#operators node\_modules\sequelize\lib\sequelize.js:242:13  GET /users는  성공시  Executing (default): DROP TABLE IF EXISTS `Users`;  Executing (default): DROP TABLE IF EXISTS `Users`;  Executing (default): CREATE TABLE IF NOT EXISTS `Users` (`id` INTEGER PRIMARY KEY AUTOINCREMENT, `name` VARCHAR(255), `createdAt` DATETIME NOT NULL, `updatedAt` DATETIME NOT NULL);  Executing (default): PRAGMA INDEX\_LIST(`Users`)  Executing (default): SELECT `id`, `name`, `createdAt`, `updatedAt` FROM `Users` AS `User`;  √ 유저 객체를 담은 배열로 응답한다 (149ms)  1 passing (213ms) |

🡪쿼리 (SELECT ‘id’, ‘name’, ‘createdAt’, ‘updatedAt’ FROM ‘Users’ AS ‘User’;)가 수행된 것을 볼 수 있다.

이번엔 두번째 성공 시 및 실패 시를 모두 테스트를 진행하기 위해 user.spec.js 파일을 다음과 같이 수정한다.

|  |
| --- |
| … 앞 생략 …  describe.only('GET /users는', ()=>{  describe('성공시', ()=>{  const users = [{name: 'alice'}, {name: 'bek'}, {name: 'chris'}];  before(()=>models.sequelize.sync({force: true}));  before(()=>models.User.bulkCreate(users));  … 뒤 생략 … |

🡪GET /users라는 전체 테스트에 describe.only를 걸어두어 전체를 테스트한다. 테스트 진행 시 before()가 차례대로 수행되는데 첫번째 before()는 models을 이용해서 db-sync를 맞추는 것이고, 두번째 before()는 bulkCreate(insert)를 통해 users라는 배열을 DB에 삽입하는 과정이다.

또한, 두 번째 성공 시 테스트는 limit을 2로 했을 때 2만큼의 데이터만 가져오는 테스트인데, 이를 설정하기 위해 user.ctrl.js에 다음과 같이 설정한다.

|  |
| --- |
| models.User  .findAll({  limit: limit  })  .then(users => {  res.json(users);  }); |

이후 테스트를 재 수행하면 다음과 같다.

|  |
| --- |
| λ npm t  > node-api@1.0.0 test D:\Song\Work\gogogo\nodejs\node-api  > set NODE\_ENV=test&&mocha api/user/user.spec.js  sequelize deprecated String based operators are now deprecated. Please use Symbol based operators for better security, read more at http://docs.sequelizejs.com/manual/tutorial/querying.html#operators node\_modules\sequelize\lib\sequelize.js:242:13  GET /users는  성공시  Executing (default): DROP TABLE IF EXISTS `Users`;  Executing (default): DROP TABLE IF EXISTS `Users`;  Executing (default): CREATE TABLE IF NOT EXISTS `Users` (`id` INTEGER PRIMARY KEY AUTOINCREMENT, `name` VARCHAR(255), `createdAt` DATETIME NOT NULL, `updatedAt` DATETIME NOT NULL);  Executing (default): PRAGMA INDEX\_LIST(`Users`)  Executing (default): INSERT INTO `Users` (`id`,`name`,`createdAt`,`updatedAt`) VALUES (NULL,'alice','2018-06-28 07:38:49.352 +00:00','2018-06-28 07:38:49.352 +00:00'),(NULL,'bek','2018-06-28 07:38:49.352 +00:00','2018-06-28 07:38:49.352 +00:00'),(NULL,'chris','2018-06-28 07:38:49.352 +00:00','2018-06-28 07:38:49.352 +00:00');  Executing (default): SELECT `id`, `name`, `createdAt`, `updatedAt` FROM `Users` AS `User` LIMIT 10;  √ 유저 객체를 담은 배열로 응답한다 (137ms)  Executing (default): SELECT `id`, `name`, `createdAt`, `updatedAt` FROM `Users` AS `User` LIMIT 2;  √ 최대 limit 갯수만큼 응답한다 (117ms)  실패시  √ limit이 숫자형이 아니면 400을 응답한다 (106ms)  3 passing (458ms) |

🡪 위 결과에서 Query를 확인할 수 있다(findAll(), bulkCreate()). 하지만 테스트에서 너무 지저분하게 나오기 때문에 그 내용을 없애려면 models.js에 다음과 같이 추가해주면 된다.

|  |
| --- |
| const Sequelize = require('sequelize');  const sequelize = new Sequelize({  dialect: 'sqlite',  storage: './db.sqlite',  logging: false  }); |

이후 테스트를 재 수행하면 다음과 같다.

|  |
| --- |
| λ npm t  > node-api@1.0.0 test D:\Song\Work\gogogo\nodejs\node-api  > set NODE\_ENV=test&&mocha api/user/user.spec.js  sequelize deprecated String based operators are now deprecated. Please use Symbol based operators for better security, read more at http://docs.sequelizejs.com/manual/tutorial/querying.html#operators node\_modules\sequelize\lib\sequelize.js:242:13  GET /users는  성공시  √ 유저 객체를 담은 배열로 응답한다 (135ms)  √ 최대 limit 갯수만큼 응답한다 (112ms)  실패시  √ limit이 숫자형이 아니면 400을 응답한다 (104ms)  3 passing (425ms) |

## G. 데이터베이스와 show컨트롤러 연동

이번엔 show 테스트 케이스를 연동해보자. user.spec.js에 only를 ‘GET /users/1은’ 으로 걸도록 변경하고, user.ctrl.js 파일을 다음과 같이 수정한다.

|  |
| --- |
| … 앞 생략 …  const show = function (req, res) {  const id = parseInt(req.params.id, 10);  if(Number.isNaN(id)) return res.status(400).end();    models.User.findOne({  where: {  id  }  }).then(user => {  if(!user) return res.status(404).end();  res.json(user);  });  };  … 뒤 생략 … |

🡪 findOne이라는 models의 함수를 통해 받아 온 id를 통해 조회한 결과를 리턴하는데, user값이 없을 경우 404를 리턴하게 한다.

이후 테스트를 수행하면 되지만 모카의 기능 중 테스트를 걸어두고 수정 후 저장 시 자동으로 테스트를 재 수행해주는 기능을 설정할 수 있다. 이를 위해서 package.json 파일을 다음과 같이 수정한다.

|  |
| --- |
| … 앞 생략 …  "scripts": {  "test": "set NODE\_ENV=test&&mocha api/user/user.spec.js",  "start": "node bin/www.js"  },  … 뒤 생략 … |

이후 테스트를 재 수행하면 다음과 같다.

|  |
| --- |
| GET /users/1은  성공시  √ id가 1인 유저 객체를 반환한다 (126ms)  실패시  √ id가 숫자가 아닐경우 400으로 응답한다 (97ms)  √ id로 유저를 찾을 수 없을 경우 404로 응답한다 (111ms)  3 passing (344ms) |

## H. 데이터베이스와 destroy컨트롤러 연동

계속해서 destroy컨트롤러 연동을 위해 user.spec.js파일에 only를 ‘DELETE /users/1’에 걸어둔다. 이후 user.ctrl.js파일을 다음과 같이 수정한다.

|  |
| --- |
| … 앞 생략 …  const destroy = (req, res) => {  const id = parseInt(req.params.id, 10);  if(Number.isNaN(id)) return res.status(400).end();    models.User.destroy({  where: {id}  }).then(() => {  res.status(204).end();  });  };  … 뒤 생략 … |

저장 시 자동 테스트가 진행되며 결과는 성공적으로 출력된다.

## I. 데이터베이스와 create컨트롤러 연동

create컨트롤러 연동을 위해 user.spec.js파일에 only를 ‘POST /users’에 걸어둔다. 또한 매 테스트 케이스마다 DB를 초기화하는 부분이 필요하므로 users와 before() 두 부분을 매 describe에 추가한다.

또한, name값은 중복을 허용하지 않도록 하기 위해 models.js에 다음과 같이 수정한다.

|  |
| --- |
| … 앞 생략 …  //User DB 모델 생성  const User = sequelize.define('User', {  name: {  type: Sequelize.STRING,  unique: true  }  });  … 뒤 생략 … |

이후 user.ctrl.js파일을 다음과 같이 수정한다.

|  |
| --- |
| … 앞 생략 …  const create = (req, res) => {  const name = req.body.name;  if(!name) return res.status(400).end();  models.User.create({name})  .then(user => {  res.status(201).json(user);  })  .catch(err => {  if(err.name === 'SequelizeUniqueConstraintError') {  return res.status(409).end();  }  res.status(500).end();  });  };  … 뒤 생략 … |

🡪catch 부분의 err를 잡는 방법은 먼저 console.log(err);를 먼저 추가하여 저장 후 에러 name을 출력해서 확인한다. 해당 name 값을 가져와서 에러처리 부분을 구현해주면 된다.

이후 테스트를 재 수행하면 정상적인 테스트가 진행된다.

## J. 데이터베이스와 update컨트롤러 연동

update컨트롤러 연동을 위해 user.spec.js파일에 only를 ‘PUT /users/:id’에 걸어둔다. 이후 user.ctrl.js파일을 다음과 같이 수정한다.

|  |
| --- |
| … 앞 생략 …  const update = (req, res) => {  const id = parseInt(req.params.id, 10);  if(Number.isNaN(id)) return res.status(400).end();    const name = req.body.name;  if(!name) return res.status(400).end();  models.User.findOne({where: {id}})  .then(user => {  if(!user) return res.status(404).end();  user.name = name;  user.save()  .then(\_ => {  res.json(user);  })  .catch(err =>{  if(err.name === 'SequelizeUniqueConstraintError') {  return res.status(409).end();  }  res.status(500).end();  });  });  };  … 뒤 생략 … |

이후 테스트를 재 수행하면 정상적인 테스트가 진행된다.

# 12. 마무리(포스트맨으로 실제 서버 API 검증해보기)

앞에까지 구현한 내용을 실제 사용해보기 위해 포스트맨이라는 앱을 설치하여 실행한다.

npm start로 서버를 구동하고 GET, POST를 수행한 뒤 서버를 재시작하면 DB가 전부 초기화되어 있는 것을 볼 수 있는데, 테스트 시에만 초기화를 하고 실제 서버 구동시에는 기존의 DB와 동기화를 하도록 설정하기 위해 sync-db.js를 다음과 같이 수정한다.

|  |
| --- |
| const models = require('../models');  module.exports = () => {  const options = {  force: process.env.NODE\_ENV === 'test' ? true : false  };  return models.sequelize.sync(options); //force: true는 기존에 DB가 있더라도 재생성하겠다는 의미이다.  } |

위와 같이 설정하고 서버를 재구동하여 포스트맨에서 모든 케이스를 진행하여 검증하면 완료!