- 01 편: 소개
- 02 편: 작업환경 설정
- 03 편: Node.js 맛보기
- 04 편: REPL 터미널
- 05 편: NPM
- 06 편: Callback Function 개념
- 07 편: Event Loop
- 08 편: HTTP Module

Express.JS

- 09 편: Express 프레임워크 사용해보기
- 10-1 편: Express 프레임워크 응용하기 – EJS
- 10-2 편: Express 프레임워크 응용하기 – RESTful API 편
- 10-3 편: Express 프레임워크 응용하기 – express-session 편
- 11 편: Express 와 Mongoose 를 통해 MongoDB 와 연동하여 RESTful API 만들기

React.JS

- [React.js] 강좌 목록

GULP

- 12.1 편: GULP – JavaScript 빌드 자동화툴 알아보기 + ES6 문법으로 사용해보기
- 12.2 편: GULP – 응용하기 (babel, webpack, nodemon, browser-sync)

JWT

- 토큰(Token) 기반 인증에 대한 소개
- JSON Web Token 소개 및 구조
- Express.js 서버에서 JWT 기반 회원인증 시스템 구현하기

[Node.JS] 강좌 01편: 소개

2016년 2월 7일 velopertdev.log / node.js / tech.log7 Comments

Node.js 가 뭐지?

NodeJS 는 구글 크롬의 자바스크립트 엔진 (V8 Engine) 에 기반해 만들어진 서버 사이드 플랫폼입니다. 2009 년에 Ryan Dahl 에 의해 개발되었으며 현시점 (2016-02-07) 최신 버전은 v5.5.0 입니다. NodeJS 공식 사이트에서 제공되는 정보는 다음과 같습니다.

Node.js®는 Chrome V8 JavaScript 엔진으로 빌드된 JavaScript 런타임입니다. Node.js 는 이벤트 기반, 논블로킹 I/O 모델을 사용해 가볍고 효율적입니다. Node.js 의 패키지 생태계인 npm 은 세계에서 가장 큰

오픈 소스 라이브러리이기도 합니다.
(출처: https://nodejs.org/ko/)

입문자들의 오해

Node 는 웹서버가 **아니랍니다.** Node 자체로는 아무것도 하지 않습니다 – 아파치 웹서버처럼 HTML 파일 경로를 지정해주고 서버를 열고 그런 설정이 없습니다. 단, HTTP 서버를 직접 작성해야합니다 (일부 라이브러리의 도움을 받으면서). Node.js 는 그저 코드를 실행할 수 있는 하나의 방법에 불과한 그저 JavasScript 런타임일 뿐입니다.

Node.js 의 특징

- **비동기 I/O 처리 / 이벤트 위주:** Node.js 라이브러리의 모든 API 는 비동기식입니다, 멈추지 않는다는거죠 (Non-blocking). Node.js 기반 서버는 API 가 실행되었을때, 데이터를 반환할때까지 기다리지 않고 다음 API 를 실행합니다. 그리고 이전에 실행했던 API 가 결과값을 반환할 시, NodeJS 의 이벤트 알림 메커니즘을 통해 결과값을 받아옵니다.
- **빠른 속도:** 구글 크롬의 V8 자바스크립트 엔진을 사용하여 빠른 코드 실행을 제공합니다.
- **단일 쓰레드 / 뛰어난 확장성:** Node.js 는 이벤트 루프와 함께 단일 쓰레드 모델을 사용합니다.

이벤트 메커니즘은 서버가 멈추지않고 반응하도록 해주어 서버의 확장성을 키워줍니다. 반면, 일반적인 웹서버는 (Apache) 요청을 처리하기 위하여 제한된 쓰레드를 생성합니다. Node.js 는 쓰레드를 한개만 사용하고 Apache 같은 웹서버보다 훨씬 많은 요청을 처리할 수 있습니다.

- **노 버퍼링:** Node.js 어플리케이션엔 데이터 버퍼링이 없고, 데이터를 chunk 로 출력합니다.
- **라이센스:** Node.js 는 MIT License 가 적용되어있습니다.

Node.js 는 누가쓸까?

Node.js 는 eBay, GoDaddy, Microsoft, Paypal, Yahoo! 등 많은곳에서 사용되고 있답니다.
다음은 Node.js 를 사용하는 프로젝트, 어플리케이션 및 회사의 리스트를 포함하고있는 GitHub 위키 링크입니다.

Node.js 를 사용하는 프로젝트 / 어플리케이션 / 회사

Node.js 를 어디에 쓸까?

다음과 같은 분야에 Node.js 가 사용된다면 뛰어난 효율성을 달성 할 수 있습니다.

- 입출력이 잦은 어플리케이션

- 데이터 스트리밍 어플리케이션
- 데이터를 실시간으로 다루는 어플리케이션ㅇㄷ
- JSON API 기반 어플리케이션
- 싱글페이지 어플리케이션

> Node.js 를 쓰지 말아야 할 곳?

CPU 사용률이 높은 어플리케이션에선 Node.js 사용을 권장하지 않습니다.

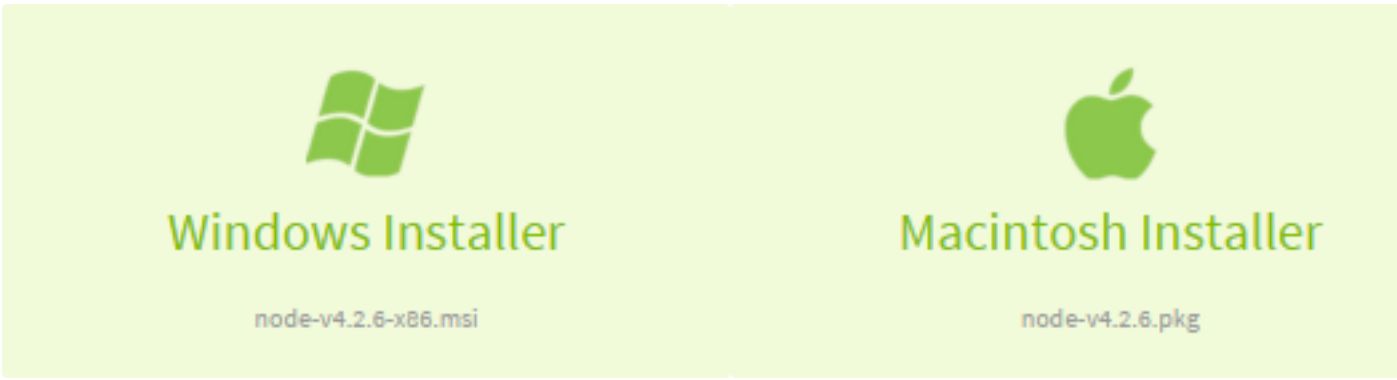
[Node.JS] 강좌 02편: 작업환경 설정

2016년 2월 8일 velopertdev.log / node.js / tech.log4 Comments

로컬 작업환경 설정

지금 사용하고 계신 PC (Windows/MAC) 이나 리눅스 서버에 Node.js 런타임을 설치합니다.
우선 개발 공부가 목적이시라면 스크롤을 아래로 내려 클라우드 IDE 사용란을 읽어주세요.

Windows / MAC



(위 이미지를 클릭하면 인스톨러 다운로드 페이지로 이동됩니다)

윈도우와 맥의 경우, 인스톨러를 통해 자동으로 작업환경을 설정 할 수 있습니다.

LINUX

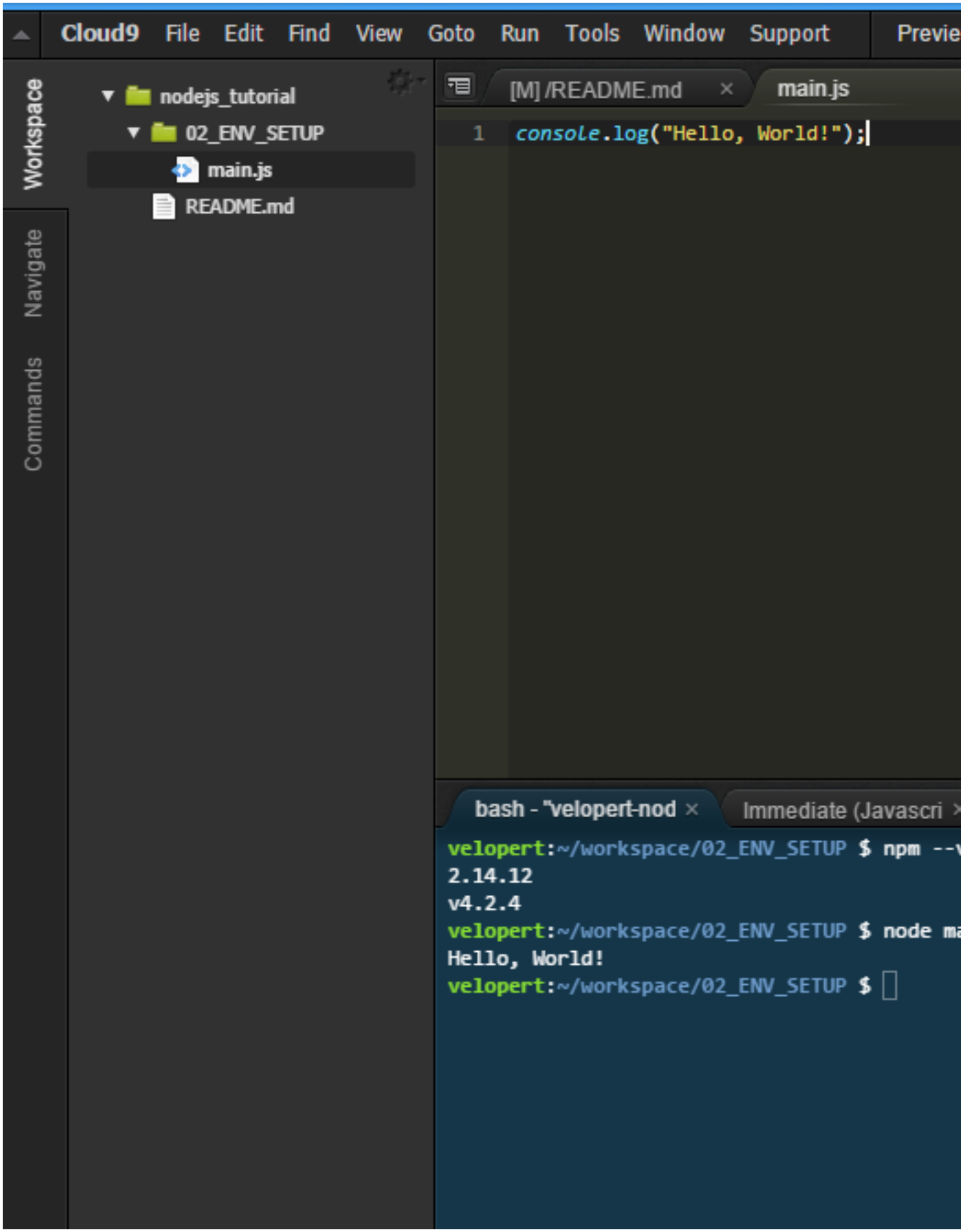
데비안 계열

```
$ sudo apt-get update

$ sudo apt-get install nodejs

$ sudo apt-get install npm

$ sudo ln -s /usr/bin/nodejs
/usr/bin/node
```

레드햇 계열

```
$ sudo yum install epel-release

$ sudo yum install nodejs

$ sudo yum install npm
```

클라우드 IDE 작업환경 설정

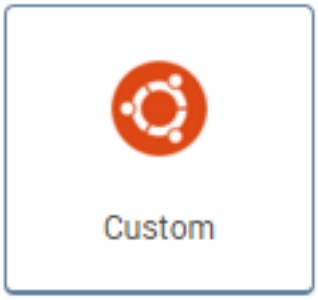

Cloud9 File Edit Find View Goto Run Tools Window Support
Workspace
Navigate
Commands
nodejs_tutorial
02_ENV_SETUP
main.js
README.md
[M] /README.md
main.js
1 console.log("Hello, World!");
bash - "velopert-nod
Immediate (Javascri
velopert:~/workspace/02_ENV_SETUP $ npm --
2.14.12
v4.2.4
velopert:~/workspace/02_ENV_SETUP $ node ma
Hello, World!
velopert:~/workspace/02_ENV_SETUP $

유명한 Cloud IDE 중 하나인 Cloud9 을 사용하시면 별도의 작업환경 설정이 필요 없이 바로 코딩을 시작하실 수 있습니다.

Cloud9 (http://c9.io)

클라우드 9 가입 / Workspace 생성하는 과정은 생략하도록 하겠습니다.

Workspace 를 만들때 template 는 Custom 으로 설정해주세요.



작업환경 설정 완료: 파일 실행 해보기

작업환경 설정이 완료되었다면 파일을 실행해봅시다.

main.js 파일을 생성해주세요.

```
/* Hello World in NodeJS */

console.log("Hello, World!");
```

그 다음, 콘솔에서 다음 명령어를 입력하여 main.js 를 실행해보세요.

```
$ node main.js
```

작업환경이 성공적으로 설정되었다면 Hello World! 이 출력됩니다.

[Node.JS] 강좌 03편: Node.js 맛보기

2016년 2월 8일 velopertdev.log / node.js / tech.log3 Comments

Node.js Application 만들기

1단계: 필요한 모듈 import 하기

어플리케이션에 필요한 모듈을
불러올땐 **require** 명령을 사용합니다.
다음 코드는 HTTP 모듈을불러오고 반환되는 HTTP 인스턴스를 http 변수에 저장합니다.

```
var http = require("http");
```

2단계: 서버 생성하기

이번 단계에선, 1 단계에서 만들은 http 인스턴스를 사용하여 **http.createServer()** 메소드를 실행합니다.
그리고 **listen** 메소드를 사용하여 포트 8081 과 bind 해줍니다.

http.createServer() 의 매개변수로는 request 와 response 를 매개변수로 가지고있는 함수를 넣어줍니다.
다음 코드는 언제나 “Hello World” 를 리턴하는 포트 8081 의 웹서버를 생성해줍니다.

```
http.createServer(function(request, response){

    /*

        HTTP 헤더 전송

        HTTP Status: 200 : OK

        Content Type: text/plain

    */

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



    /*

        Response Body 를 "Hello World" 로 설정

    */

    response.end("Hello World\n");
```

```
}).listen(8081);
```

3단계: 서버 테스트 해보기

1 단계와 2 단계를 파일 main.js 에 합쳐서 작성해보세요

```
var http = require("http");


http.createServer(function(request, response){
   /*
      HTTP 헤더 전송
      HTTP Status: 200 : OK
      Content Type: text/plain
   */
   response.writeHead(200, {'Content-Type': 'text/plain'});

   /*
      Response Body 를 "Hello World" 로 설정
```

```
    */

    response.end("Hello World\n");

}).listen(8081);
```

```
console.log("Server running at http://127.0.0.1:8081");
```

서버를 실행해봅시다.

```
$ node main.js
```

서버가 성공적으로 실행됐다면 다음 텍스트가 출력됩니다.

```
Server running at http://127.0.0.1:8081/
```

브라우저에서 http://127.0.0.1:8081/ 을 열으면 다음과 같은 결과를 확인 할 수 있습니다.

[Node.JS] 강좌 04편: REPL 터미널

2016년 2월 8일 velopertdev.log / node.js / tech.log0 Comments

REPL 은 **R**ead **E**val **P**rint **L**oop 의 약자입니다. 이는 윈도우 커맨드, 혹은 UNIX/LINUX Shell 처럼 사용자가 커맨드를 입력하면 시스템이 값을 반환하는 환경을 가르킵니다.
Node.js 는 REPL 환경과 함께 제공되며 다음과 같은 기능을 수행 할 수 있습니다.

- **Read** – 유저의 값을 입력 받아 JavaScript 데이터 구조로 메모리에 저장합니다.
- **Eval** – 데이터를 처리(Evaluate) 합니다.
- **Print** – 결과값을 출력합니다.
- **Loop** – Read, Eval, Print 를 유저가 Ctrl+C 를 두번 눌러 종료할때까지 반복합니다.

Node.js 의 REPL 환경은 자바스크립트 코드를 테스팅 및 디버깅할때 유용하게 사용됩니다.

REPL 시작하기

REPL 은 쉘/콘솔에 파라미터 없이 node 를 실행하여 실행 할 수 있습니다.

```
$ node
```

```
>
```

간단한 표현식 사용

Node.js REPL 커맨드 프롬프트에 간단한 연산자를 사용해봅시다.

```
$ node

> 1 + 5

6

> 1 + ( 6 / 2 ) - 3

1

>
```

변수 사용

다른 스크립트처럼, 변수에 값을 저장하고 나중에 다시 출력 할 수 있습니다.

만약 **var** 키워드를 사용하면 명령어를 입력했을때 변수의 값이 출력되지 않고, **var** 키워드를 사용하지 않으면 값이 출력됩니다.
또한, console.log() 를 통해 출력 할 수 있습니다.

```
$ node

> x = 10

10
```

```
> var y = 5

undefined

> x + y

15

> console.log("Value is " + ( x + y ))

Value is 15

undefined
```

Multi-Line 표현식 사용

do-while 루프를 REPL 에서 실행해봅시다.

```
$ node

> var x = 0

undefined

> do {

... x++;

... console.log("x: " + x);

... } while ( x < 3 );

x: 1
```

```
x: 2
x: 3
undefined
>
```

Underscore (_) 변수

밑줄 _ 변수는 최근 결과값을 지칭합니다.

```
$ node
> var x = 10;
undefined
> var y = 5;
undefined
> x + y;
15
> var sum = _
undefined
> console.log(sum)
15
```

```
undefined
>
```

REPL 커맨드

- **Ctrl+C** – 현재 명령어를 종료합니다.
- **Ctrl+C (2 번)** – Node REPL 을 종료합니다.
- **Ctrl+D** – Node REPL 을 종료합니다.
- **위/아래 키** – 명령어 히스토리를 탐색하고 이전 명령어를 수정합니다.
- **Tab** – 현재 입력란에 쓴 값으로 시작하는 명령어 / 변수 목록을 확인합니다.
- **.help** – 모든 커맨드 목록을 확인합니다.
- .**break** – 멀티 라인 표현식 입력 도중 입력을 종료합니다.
- **.clear** – .break 와 같습니다.
- **.save filename** – 현재 Node REPL 세션을 파일로 저장합니다.
- **.load filename** – Node REPL 세션을 파일에서 불러옵니다.

[Node.JS] 강좌 05편: NPM

2016년 2월 8일 velopertdev.log / node.js / tech.log4 Comments

Node Package Manager (NPM) 은 두가지의 주요 기능을 지니고 있습니다.

- NPMSearch 에서 탐색 가능한 Node.js 패키지/모듈 저장소
- Node.js 패키지 설치 및 버전 / 호환성 관리를 할 수 있는 커맨드라인 유틸리티

npm 이 제대로 설치되있는지 확인하려면 다음 명령어를 입력하세요:

(현재 버전은 8.2 22/1/21 일 체크)

```
$ npm --version

3.7.1
```

npm 버전이 구버전이라면 다음 명령어로 쉽게 최신버전으로 업데이트 할 수 있습니다:

```
$ sudo npm install npm -g

npm http GET https://registry.npmjs.org/npm
npm http 200 https://registry.npmjs.org/npm
npm http GET https://registry.npmjs.org/npm/-/npm-3.7.1.tgz
npm http 200 https://registry.npmjs.org/npm/-/npm-3.7.1.tgz
/usr/local/bin/npm -> /usr/local/lib/node_modules/npm/bin/npm-cli.js
npm@3.7.1 /usr/local/lib/node_modules/npm
```

NPM 에서 일부 패키지를 설치 할 때 python 을 요구하므로 호환성을 맞추기 위하여 파이썬 런타임도 설치하도록 합시다.

NPM 을 사용하여 모듈 설치하기

```
npm install <모듈 이름>
```

예를들어 유명한 Node.js 웹 프레임워크중 하나인 **express** 를 설치한다면 다음 명령어를 입력하면됩니다.

```
$ npm install express
```

설치하면 여러분의 js 에서 이렇게 모듈을 사용 할 수 있습니다.

```
var express = require('express');
```

글로벌 vs 로컬 모듈 설치

기본적으로는, npm 은 모듈을 로컬모드로 설치합니다. 로컬모드란건, 패키지를 명령어를 실행한 디렉토리안에 있는 node_modules 에 설치하는것을 의미합니다.

글로벌 설치는 시스템 디렉토리에 설치하는것을 의미합니다. 한번 express 를 글로벌 모드로 설치해볼까요?

```
$ sudo npm install express -g

/usr/lib
└─┬ express@4.13.4
  ├─┬ accepts@1.2.13
  │ ├─┬ mime-types@2.1.9
  │ │ └── mime-db@1.21.0
  │ └── negotiator@0.5.3
 .... 길어서 생략....
  │ └── statuses@1.2.1
  ├── utils-merge@1.0.0
```

```
└── vary@1.0.1
```

보이다시피, 현재 경로가 아닌 /usr/lib/node_modules 에 모듈을 설치합니다.
시스템에 저장하므로, 루트 계정이 아니라면 앞에 sudo 를 붙여주어야합니다.
글로벌 모드로 설치하였을때는, node 어플리케이션에서 바로 require 할 수는 없습니다.
단, 다음처럼 npm link 명령어를 입력하고나면 해당 모듈을 불러올 수 있습니다.

```
$ npm install -g express

$ cd [local path]/project

$ npm link express
```

package.json

package.json 은 노드 어플리케이션 / 모듈의 경로에 위치해 있으며 패키지의 속성을 정의합니다.

다음은 express 로 프로젝트를 생성했을때 생성되는 package.json 입니다.

```
{

  "name": "myapp",

  "version": "0.0.0",

  "private": true,

  "scripts": {
```

```
    "start": "node ./bin/www"
  },
  "dependencies": {
    "body-parser": "~1.13.2",
    "cookie-parser": "~1.3.5",
    "debug": "~2.2.0",
    "express": "~4.13.1",
    "jade": "~1.11.0",
    "morgan": "~1.6.1",
    "serve-favicon": "~2.3.0"
  }
}
```

보시다시피 이 파일은 프로젝트가 의존하는 모듈과 모듈버전의 정보를 담고있습니다.

package.json 에 관한 자세한 내용은 감성 프로그래밍 블로그에서 읽어보실 수 있습니다.

모듈 제거

다음 명령어로 설치된 모듈을 제거 할 수 있습니다.

```
$ npm uninstall express
```

모듈 업데이트

다음 명령어로 모듈을 업데이트 할 수 있습니다.

```
$ npm update express
```

모듈 검색

다음 명령어로 모듈을 검색 할 수 있습니다.

```
$ npm search express
```

이 명령어는 처음 이용 할 때 메모리를 굉장히 많이 잡아먹습니다.

클라우드 IDE 를 사용하거나 서버에 램이 1G 정도라면 매우 오래걸리거나 에러가 납니다.

그럴 경우엔 NPMSearch 에서 검색을 하면 됩니다.

[Node.JS] 강좌 06편: Callback Function 개념

2016년 2월 8일 velopertdev.log / node.js / tech.log6 Comments

Callback Function 이 뭘까?

자바스크립트에서는, 함수(function)는 일급 객체입니다. 즉, 함수는 *Object* 타입이며 다른 일급객체와 똑같이 사용 될 수 있습니다. (String, Array, Number, 등등..) function 자체가 객체이므로 변수안에 담을 수도 있고 인수로서 다른 함수에 전달 해 줄수도있고, 함수에서 만들어질수도있고 반환 될수도있습니다. Callback function 은, 특정 함수에 매개변수로서 전달된 함수를 지칭합니다.

그리고 그 Callback function 은 그 함수를 전달받은 함수 안에서 호출되게 됩니다.

이해를 돕기 위하여 jQuery 에서 사용된 callback function 예제를 살펴봅시다.

```
// 보시면, click 메소드의 인수가 변수가 아니라 함수이죠?

// click 메소드의 인수가 바로 Callback 함수입니다.

$("#btn_1").click(function() {
```

```
  alert("Btn 1 Clicked");

});
```

설명: click 메소드에 이름이 없는 callback function 을 인수로 전달해줍니다.
그리고 jQuery 안의 click 메소드에서는, 마우스 클릭이 있으면 callback function 을 호출하게 설정을 하지요.

> 흠.. 이걸 근데 대체 왜 쓸까?

Node.js 에선 Callback 함수가 매우 많이 사용된답니다. 콜백의 개념이 어느정도 이해가 됐다면 Node.js 에서의 예제를 한번 살펴보겠습니다.

Blocking Code

첫번째 예제는 Callback 함수가 *사용되지 않는*, Blocking Code 예제입니다.
말그대로 어떤 작업을 실행하고 기다리면서 코드가 "막히"게 됩니다.

우선, input.txt 라는 텍스트파일을 생성해줍니다.

```
Let's understand what is a callback
function.

What the HELL is it?
```

그 다음, main.js 를 작성하세요.

```
var fs = require("fs");
```

```
var data = fs.readFileSync('input.txt');


console.log(data.toString());

console.log("Program has ended");
```

이제 결과값을 확인해볼까요?

```
$ node main.js

Let's understand what is a callback
function.

What the HELL is it?

Program has ended
```

보다시피, 텍스트를 출력하고나서 프로그램이 종료되었다는 문구를 출력합니다.

Non-Blocking Code

두번째 예제는 Callback 함수가 사용된 Non-Blocking Code 예제입니다.
위 예제와는 반대로 함수가 실행될 때, 프로그램이 함수가 끝날때까지 기다리지않고
바로 그 아래에있는 코드들을 실행하게 되지요. 그 다음에 함수에있던 작업이 다 끝나면 콜백함수를 호출합니다.

input.txt 는 위 예제에서 사용한 똑같은 파일을 사용합니다.

main.js 를 이렇게 수정해보세요.

```
var fs = require("fs");


fs.readFile('input.txt', function (err, data) {

    if (err) return console.error(err);

    console.log(data.toString());

});


console.log("Program has ended");
```

모든 Node 어플리케이션의 비동기식 함수에서는 첫번째 매개변수로는 error 를, 마지막 매개변수로는 callback 함수를 받습니다.
fs.readFile() 함수는 비동기식으로 파일을 읽는 함수이고, 도중에 에러가 발생하면 err 객체에 에러 내용을 담고
그렇지 않을 시에는 파일 내용을 다 읽고 출력합니다.

결과는?

```
Program has ended
```

Let's understand what is a callback function.

What the HELL is it?

readFile() 메소드가 실행 된 후, 프로그램이 메소드가 끝날때까지 대기하지 않고 곧바로 다음 명령어로 진행하였기 때문에,
프로그램이 끝났다는 메시지를 출력 한 후에, 텍스트 내용을 출력했습니다.

그렇다고 해서 프로그램이 끝나고나서 텍스트 내용을 출력한것은 아닙니다.
프로그램이 실질적으로 끝난건 텍스트가 출력된 후입니다.

만약에 readFile() 다음에 실행되는 코드가 그냥 console.out() 이 아니라
readFile() 보다 작업시간이 오래걸리는 코드였다면 텍스트 출력을 먼저 하게되겠죠?

**callback 함수를 사용하여 이렇게 프로그램의 흐름을 끊지 않음으로서,
Non-Blocking 코드를 사용하는 서버는 Blocking 코드를 사용하는 서버보다
더 많은 양의 요청을 빠르게 처리 할 수 있게됩니다.**

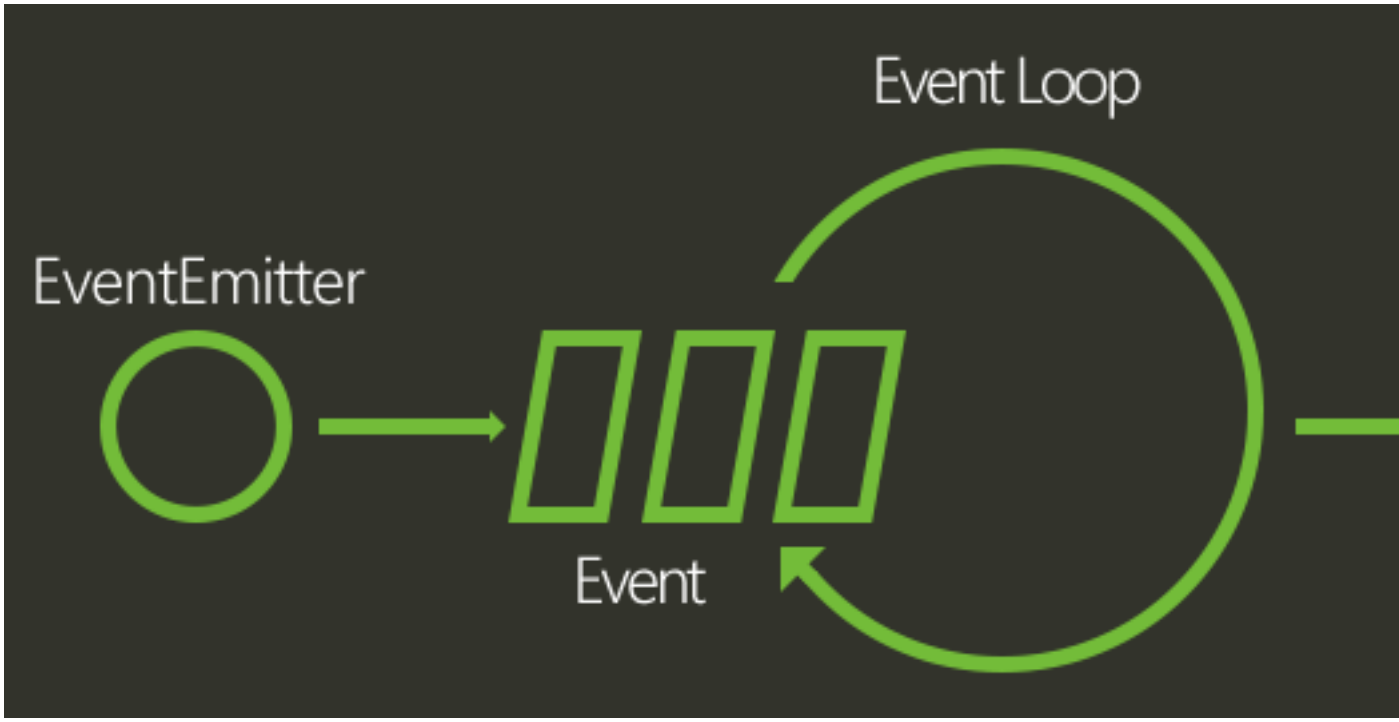
[Node.JS] 강좌 07편: Event Loop

2016년 2월 9일 velopertdev.log / node.js / tech.log2 Comments

Node.js 에선 Event 를 매우 많이 사용하고, 이 때문에 다른 비슷한 기술들보다 훨씬 빠른 속도를 자랑합니다.

Node.js 기반으로 만들어진 서버가 가동되면, 변수들을 initialize 하고, 함수를 선언하고 이벤트가 일어날때까지 기다립니다.

이벤트 위주 (Event-Driven) 어플리케이션에서는, 이벤트를 대기하는 메인 루프가 있습니다.
그리고 이벤트가 감지되었을시 Callback 함수를 호출합니다.



이벤트가 콜백과 비슷해 보일 수 도 있습니다.
차이점은, 콜백함수는 비동기식 함수에서 결과를 반환할때 호출되지만,
이벤트핸들링은 옵저버 패턴에 의해 작동됩니다.

"옵저버요?"

디자인 패턴 중 하나 입니다. 자세한 내용은 위키피디아를 참조하세요.

이벤트를 대기하는 (EventListeners) 함수들이 옵저버 역할을 합니다. 옵저버들이 이벤트를 기다리다가, 이벤트가 실행되면 이벤트를 처리하는 함수가 실행됩니다.

Node.js 에는 **events** 모듈과 **EventEmitter** 클래스가 내장되어있는데요,
이를 사용하여 이벤트와 이벤트핸들러를 연동(bind) 시킬 수 있습니다:

```
// events 모듈 사용

var events = require('events');



// EventEmitter 객체 생성

var eventEmitter = new
events.EventEmitter();
```

이벤트 핸들러와 이벤트를 연동시키는건 다음과 같습니다:

```
// event 와 EventHandler 를 연동(bind)

// eventName 은 임의로 설정 가능

eventEmitter.on('eventName',
eventHandler);
```

프로그램안에서 이벤트를 발생시킬땐 다음 코드를 사용합니다:

```
eventEmitter.emit('eventName');
```

이벤트 핸들링 예제

위에서 배운것을 토대로 이벤트를 다루는 예제를 작성해보도록 하겠습니다.

```
// events 모듈 사용

var events = require('events');



// EventEmitter 객체 생성

var eventEmitter = new
events.EventEmitter();



// EventHandler 함수 생성

var connectHandler = function
connected(){

    console.log("Connection Successful");

    

    // data_recevied 이벤트를 발생시키기

    eventEmitter.emit("data_received");

}
```

```
// connection 이벤트와 connectHandler 이벤트 핸들러를 연동

eventEmitter.on('connection', connectHandler);


// data_received 이벤트와 익명 함수와 연동

// 함수를 변수안에 담는 대신에, .on() 메소드의 인자로 직접 함수를 전달

eventEmitter.on('data_received', function(){

    console.log("Data Received");

});


// connection 이벤트 발생시키기

eventEmitter.emit('connection');


console.log("Program has ended");
```

출력물

```
$ node main.js
```

```
Connection Successful

Data Received

Program has ended
```

EventEmitter 메소드

추후 수정 예정.. EventEmitter 의 상세한 내용은 NodeJS Documentation 을 확인해주세요.

Event / Loop / Node.js / tutorial / 강좌

Post navigation

Previous Post
Previous post:

Next Post
Next Post:

[Node.JS] 강좌 08편: HTTP Module

2016년 2월 9일 velopertdev.log / node.js / tech.log7

Comments

Node.JS 강좌 03 편에서 맛보기로 Hello World 만을 리턴하는 웹서버를 만들어봤었습니다.
이번 강좌에서는 http 모듈을 이용해 더 기능이 향상된 웹서버과 웹클라이언트를 코딩해보도록 하겠습니다.

HTTP 서버 예제

우선 index.html 을 생성하세요.

```
<html>
    <head>
        <title>Sample Page</title>
    </head>
    <body>
        Hello World!
    </body>
```

```
</html>
```

다음엔 server.js 를 작성하세요.

```
var http = require('http');

var fs = require('fs');

var url = require('url');


// 서버 생성

http.createServer( function (request, response) {

   // URL 뒤에 있는 디렉토리/파일이름 파싱

   var pathname = url.parse(request.url).pathname;


   console.log("Request for " + pathname + " received.");


   // 파일 이름이 비어있다면 index.html 로 설정

   if(pathname=="/"){
```

```
      pathname = "/index.html";
   }

   // 파일을 읽기
   fs.readFile(pathname.substr(1), function (err, data) {
      if (err) {
         console.log(err);
         // 페이지를 찾을 수 없음
         // HTTP Status: 404 : NOT FOUND
         // Content Type: text/plain
         response.writeHead(404, {'Content-Type': 'text/html'});
      }else{
         // 페이지를 찾음
         // HTTP Status: 200 : OK
         // Content Type: text/plain
         response.writeHead(200, {'Content-Type': 'text/html'});
```

```
        // 파일을 읽어와서 responseBody 에 작성
        response.write(data.toString());
    }
    // responseBody 전송
    response.end();
  });
}).listen(8081);

console.log('Server running at http://127.0.0.1:8081/');
```

클라이언트에서 서버에 접속을하면 URL 에서 열고자 하는 파일을 파싱하여 열어줍니다.
파일이 존재하지 않는다면 콘솔에 에러 메시지를 출력합니다.

출력물

```
$ node server.js

Server running at http://127.0.0.1:8081/
```

```
Request for / received.
Request for /showmeerror received.
{ [Error: ENOENT: no such file or
directory, open 'showmeerror']
  errno: -2,
  code: 'ENOENT',
  syscall: 'open',
  path: 'showmeerror' }
Request for /index.html received.
```

서버를 실행하고 다음 링크들을 들어갔을때 뜨는 출력물입니다:

1. http://127.0.0.1:8081/
2. http://127.0.0.1:8081/showmeerror
3. http://127.0.0.1:8081/index.html

HTTP 클라이언트 예제

```
var http = require('http');



// HTTPRequest의 옵션 설정

var options = {

   host: 'localhost',

   port: '8081',

   path: '/index.html'
```

```
};


// 콜백 함수로 Response를 받아온다
var callback = function(response){
   // response 이벤트가 감지되면 데이터를 body에 받아온다
   var body = '';
   response.on('data', function(data) {
      body += data;
   });

   // end 이벤트가 감지되면 데이터 수신을 종료하고 내용을 출력한다
   response.on('end', function() {
      // 데이저 수신 완료
      console.log(body);
   });
}
```

```
// 서버에 HTTP Request 를 날린다.

var req = http.request(options, callback);

req.end();
```

14 번과 19 번 줄을 보면 response.on() 을 사용하죠. .on() 메소드, 익숙하지 않나요? response 는 강좌 07 편 Event Loop 에서 봤었던 EventEmitter 클래스를 상속한 객체입니다.

출력물

```
$ node client.js

<html>

    <head>

        <title>Sample Page</title>

    </head>

    <body>

        Hello World!

    </body>

</html>
```

[Node.JS] 강좌 09편: Express 프레임워크 사용해보기

2016년 2월 9일 velopertdev.log / node.js / tech.log9 Comments

Node.js 로 웹서버에 필요한 기능을 하나하나 다 짜면, 사실상 조금 귀찮은것들이 많습니다.
라우팅에, 세션관리에 이것저것 골치 아프겠죠?

NodeJS 의 웹프레임워크를 사용하면 간편하게 웹서버를 구축 할 수 있습니다.
웹프레임워크 종류는 대표적으로 Express, Koa, Hapi 등이 있는데요
이 포스트에선 Express 를 사용해보도록 하겠습니다.

1. 디렉토리 구조 이해하기

```
express_tutorial/

├── package.json

├── public
```

```
|     └── css
|     └── style.css
├── router
|     └── main.js
├── server.js
└── views
  ├── about.html
  └── index.html
```

혹시.. '엥? 내가 이 파일들을 모두 만들어야되나?' 하고 머릿속에서 생각하셨나요?
그렇다면 제가 드릴 답은... 네니오 입니다. 일부는 자동으로 생성되고 나머지는 강좌를 진행하면서 차근차근 같이 만들어갈꺼니까 미리 만들지는 마세요 ~

2. package.json 파일 생성

이 파일, 강좌 05 편: NPM 에서 본적있죠?
프로젝트의 이름, 버전, 의존패키지 리스트 등 정보들에 대한 정보를 담고있는 파일입니다.

```
{
```

```
  "name": "express-tutorial",
  "version": "1.0.0",
  "dependencies":
  {
    "express": "~4.13.1",
    "ejs": "~2.4.1"
  }
}
```

2.1 NPM 으로 Dependency (의존 패키지) 설치

package.json 을 생성 하셨으면 다음 명령어로 의존패키지들을 설치하세요.

```
$ npm install
```

3. Express 서버 생성

저희는 package.json 파일을 생성했고 의존 패키지들도 모두 설치하였습니다. 이제 서버를 만들어볼 차례입니다.
server.js 파일을 생성하고 다음 코드를 입력하세요.

```
var express = require('express');
```

```
var app = express();

var server = app.listen(3000, function(){

    console.log("Express server has started on port 3000")

})
```

아무것도 하지 않는 웹서버 입니다.(아직은 에러가 발생한다. 아래에 추가 코드 필요)

```
$ node server.js
```

를 입력하면 포트 3000 으로 웹서버를 열고, 페이지에 들어가면 Cannot GET / 이라는 텍스트가 나타납니다. 왜냐구요? Router 를 아직 정의하지 않았으니까요

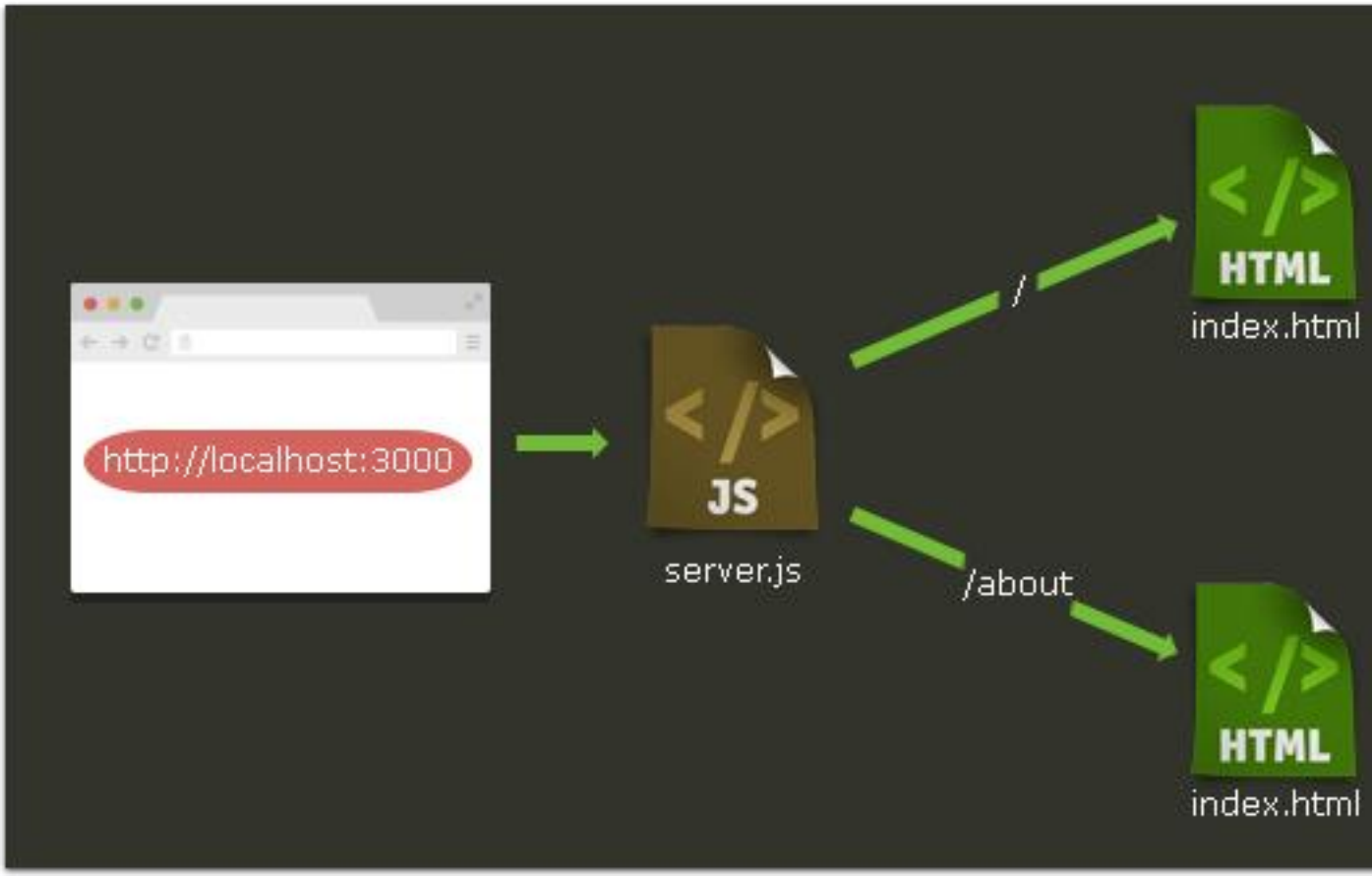
4. Router 로 Request 처리하기

현재 우리는 서버를 돌리기위해 필요한것을 모두 갖추었습니다.
이제 브라우저에서 Request 가 왔을때 서버에서 어떤 작업을 할 지 Router 를 통하여 설정해주어야합니다.

자 간단한 router 를 작성해봅시다.

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

    res.send('Hello World');

});
```

이 코드를 추가해주고 server.js 를 재실행하시면, http://localhost:3000/ 으로 접속하였을 때, Hello World 를 반환합니다.
Hello World 지긋지긋하죠? 이건 그냥 라우터 예제일 뿐이였으니까 한번 확인해보고 지우세요.
진짜 라우터를 작성해볼 차례입니다.



라우터 코드와 서버 코드는 다른 파일에 작성하는것이 좋은 코딩 습관입니다.

router 라는 폴더를 만들고 그 안에 main.js 를 생성해주세요.

```
module.exports = function(app)
```

```
{
    app.get('/',function(req,res){
        res.render('index.html')
    });
    app.get('/about',function(req,res){
        res.render('about.html');
    });
}
```

파일을 저장하고 아직 코드를 실행하진 마세요.
module.exports 는 우리가 router 코드를 따로 작성했기에, server.js 에서 모듈로서 불러올 수 있도록 사용된답니다.

5. HTML 페이지를 띄우기

HTML 페이지를 띄우기 위해선 우선 html 파일이 있어야겠지요?

views/ 디렉토리를 만들고, 그 안에 index.html 과 about.html 을 생성해주세요.

```
<html>
  <head>
```

```
    <title>Main</title>

    <link rel="stylesheet" type="text/css" href="css/style.css">

  </head>

  <body>

    Hey, this is index page

  </body>

</html>
```

```
<html>

  <head>

    <title>About</title>

    <link rel="stylesheet" type="text/css" href="css/style.css">

  </head>

  <body>

    About... what?

  </body>
```

```
</html>
```

(style.css 는 아직 만들지 않았지만, 이 포스트의 아랫부분에서 다루게 됩니다)

그 후, 다시 server.js 를 업데이트 해봅시다.

```
var express = require('express');

var app = express();

var router =
require('./router/main')(app);



app.set('views', __dirname + '/views');

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

app.engine('html',
require('ejs').renderFile);



var server = app.listen(3000, function(){

    console.log("Express server has
started on port 3000")

});
```

3 번째 줄 : 라우터 모듈인 main.js 를 불러와서 app 에 전달해줍니다.
5 번째 줄 : 서버가 읽을 수 있도록 HTML 의 위치를 정의해줍니다.
6 번째, 7 번째 줄: 서버가 HTML 렌더링을 할 때, EJS 엔진을 사용하도록 설정합니다.
* EJS 엔진에 대해선 다음 강좌에서 살펴보도록 하겠습니다.

정적 파일 (Static files) 다루기

정적 파일이란? HTML 에서 사용되는 .js 파일, css 파일, image 파일 등을 가르킵니다.
서버에서 정적파일을 다루기 위해선, express.static() 메소드를 사용하면 됩니다.

public/css 디렉토리를 만드시고 그 안에 style.css 파일을 생성해주세요.

```
body{

    background-color: black;

    color: white;

}
```

그 후, server.js 의 11 번줄 아래에 해당 코드를 추가해주세요:

```
app.use(express.static('public'));
```

이제 서버를 실행하고

```
$ node server.js
```

http://localhost:3000/ 에 접속했을 때 css 가 적용된 페이지가 나타나면 성공입니다.

다음 강좌에선 Express.js 를 추가적으로 활용하는 방법에 대해서 포스트 하도록하겠습니다.

[Review] REST/HTTP API 클라이언트의 끝판왕, Postman

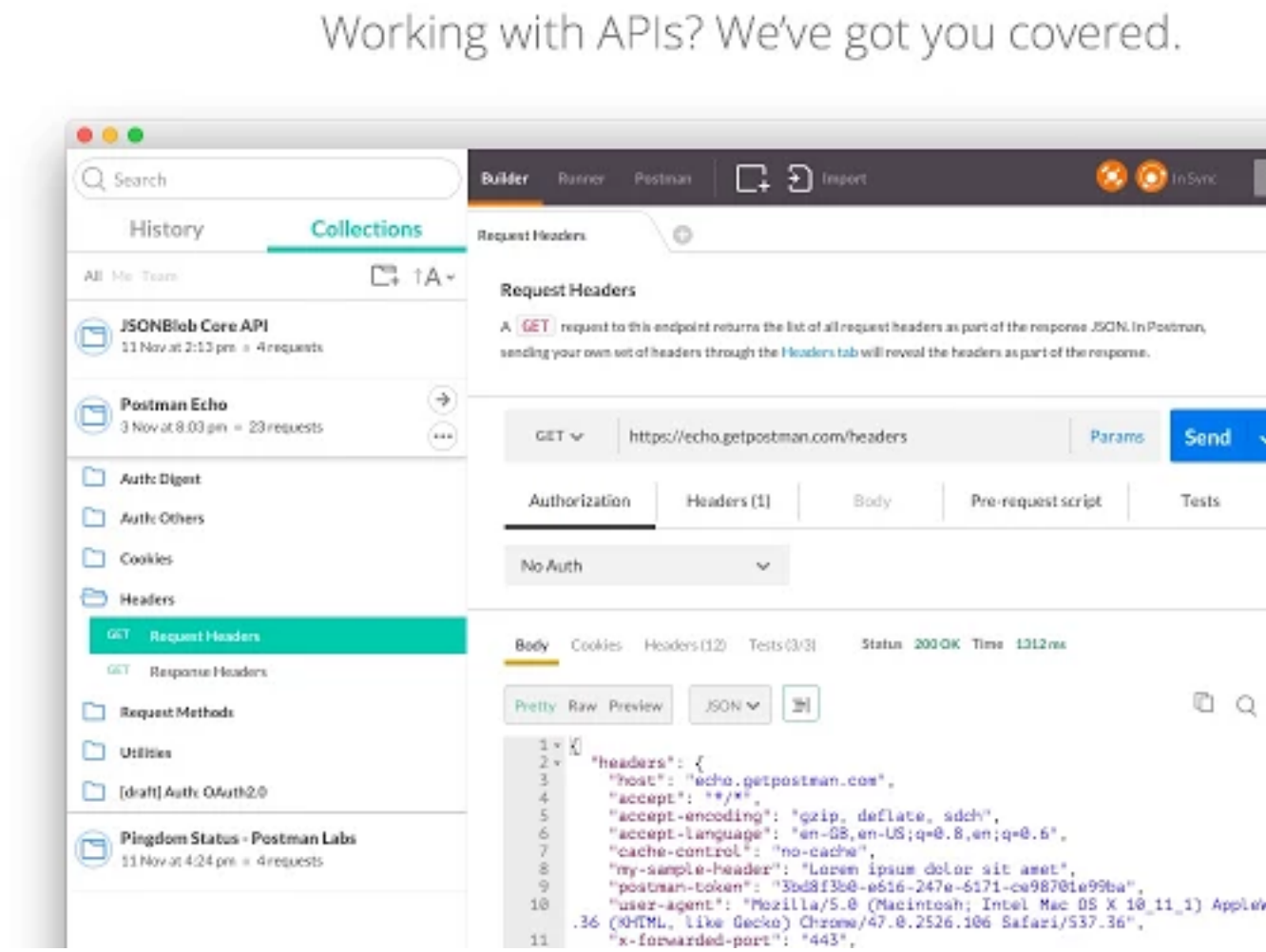
2016년 2월 12일 velopertreview.log2 Comments

크롬의 Extension 중 REST/HTTP API 를 개발할 때 유용하게 사용 할 수 있는 클라이언트 Postman,

우연히 발견했는데, 이거.. 물건이다.

난 사실 그냥 HTTP 패킷 테스트로 전송해볼 용도로만 설치했었으나

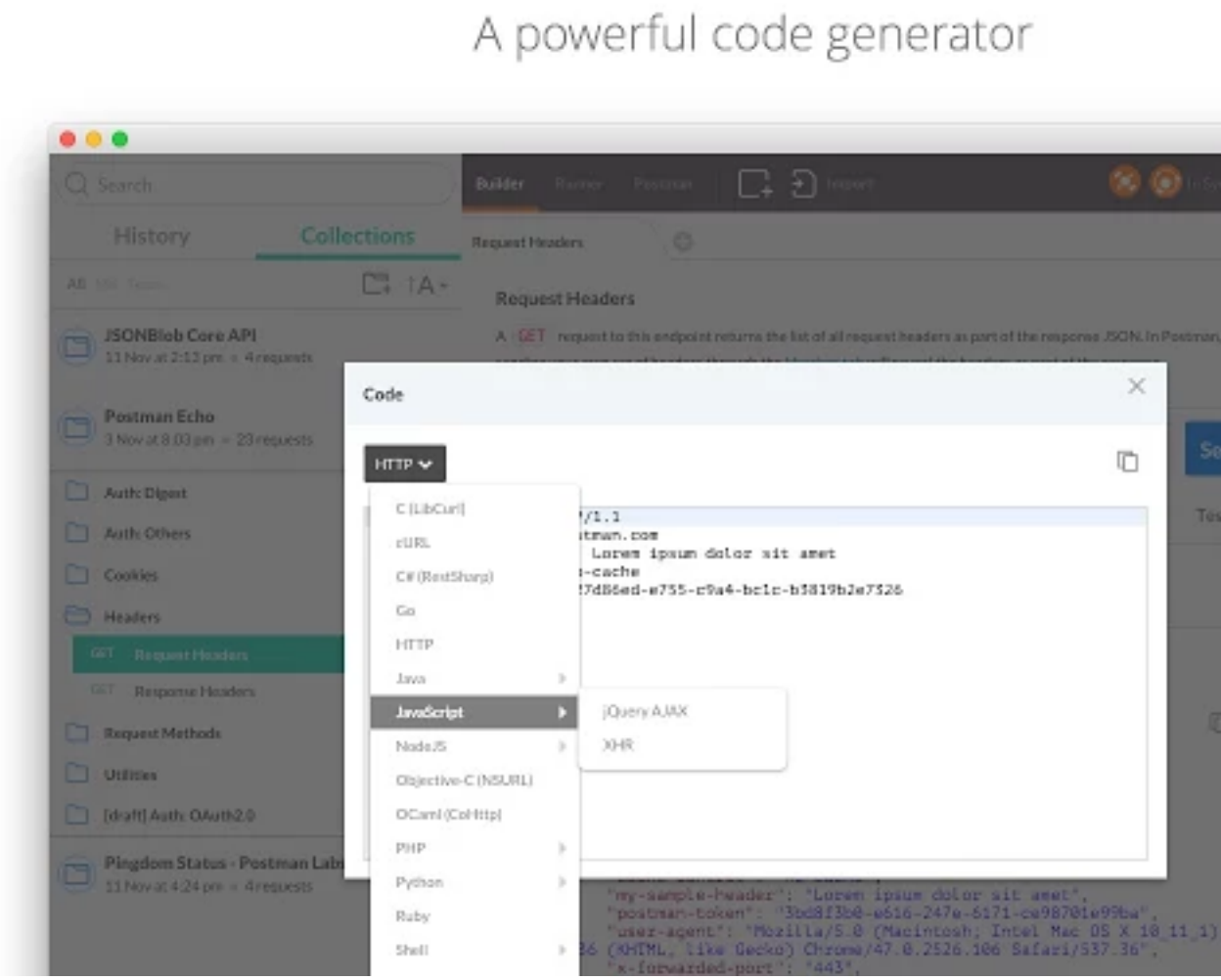
엄청나게 좋으다..



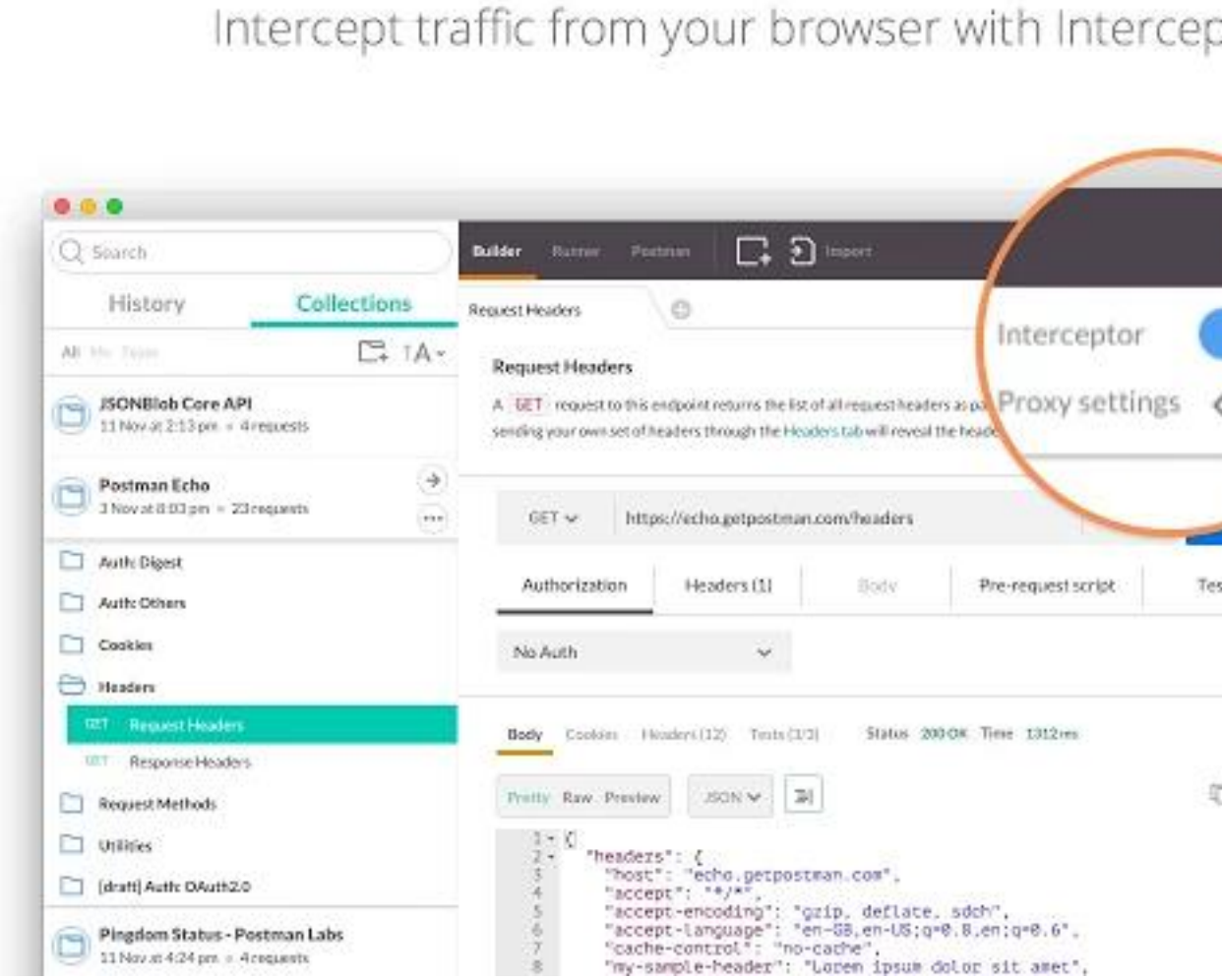
구글 크롬 앱스토어 – Postman

1. UI 가 무진장 이쁘다. 요새 윈도우 프로그램 디자인을 보고 감탄한적이 많이 없는데, 마치 "요새 윈도우 프로그램은 이렇게 디자인하는거란다" 하고 과시 하는 수준이다.
2. 본 기능에 충실하다. 인터페이스가 정말 사용하기 쉽다.

3. 다른 HTTP Client 와는 다르게, 사용할때
   회원가입을 요구하는데, (구글 계정 사용 가능)
   이를 통해 내가 했었던 것들을 History 에 저장하고
   이 데이터를 서버상에 저장하는듯 싶다.
   고로 일일히 저장 할 필요가 없으므로 엄지 똑.
4. 코드로 변환 기능: </> 버튼을 누르면 내가 선택한
   HTTP 요청을 코드로 변환해주는
   기능이있다.



5. 옛날에 C# 할때 HTTP Request 를 복붙하면 C# 코드로 변환해주는걸 만들어서 사용한적이 있었는데
   와 이거 정말 편하다.. 여러가지 언어로 제공된다.
6. 브라우저에서 요청하는 패킷들을 잡아낼 수 있다 (이는 확장프로그램 하나를 추가적으로 설치해야한다)



https 형태도 볼 수 있고, 스니퍼가 따로 필요없다.

좋으다 좋으다. 개발자라면 크롬에 설치해둬야 할 앱 !

[Node.JS] 강좌 10-1편: Express 프레임워크 응용하기 - EJS

2016년 2월 12일 velopertdev.log / node.js / tech.log12 Comments

강좌 09 편에 이어 Express 를 응용하는 방법에 대해 알아보겠습니다.
전 강좌와 같은 프로젝트를 사용하니, 강좌 09 편을 읽지 않으신분은 전 강좌부터 읽어주세요.
강좌를 작성하다가 글이 너무 길어져서 3 편으로 나누어 작성한 점 유의해주세요

10-1. EJS
10-2. Restful API
10-3. Session

---

0. 디렉토리 구조

```
express_tutorial/
```

```
├── data
│   └── user.json
├── node_modules
├── package.json
├── public
│   └── css
│       └── style.css
├── router
│   └── main.js
├── server.js
└── views
    ├── body.ejs
    ├── header.ejs
    └── index.ejs
```

이번 강좌에선 data/user.json 이 추가되었고 views/ 내부 파일들이 변경되었습니다.

1. 의존 모듈 추가

저번 강좌에선 그저 페이지 라우팅만 다뤘지만, 강좌 10 편에서는 EJS 엔진과 추가적으로 RESTful API, 그리고 세션을 다룰 것이므로 넣어줘야 할 의존 모듈들이 있습니다.

강좌를 작성하다가 길이 너무 길어져서 3 편으로 나누어 작성한 점 유의해주세요

- **body-parser** – POST 데이터 처리
- **express-session** – 세션 관리

우선 전 강좌에서 작성했었던 package.json 을 업데이트 하세요.

```
{
  "name": "express-tutorial",
  "version": "1.0.0",
  "dependencies":
   {
    "express": "~4.13.1",
    "ejs": "~2.4.1"    ,
    "body-parser": "~1.14.2",
    "express-session": "~1.13.0"
   }
}
```

그 후 다음 명령어를 입력해 모듈을 설치합니다.

```
$ npm install
```

(버전이 낮아서 아래와 같이 나오면 npm audit 을 실행한다. 22/1/19 일 체크)

15 vulnerabilities (2 **low**, 2 **moderate**, 10 **high**, 1 **critical**)

To address all issues, run:
    npm audit fix –force

추가한 모듈들을 server.js 에서 불러오겠습니다.

(DemoNode 세번째 데모 폴더에 있는 파일들로 덮어쓰기를 한다. 22/2/4 일 체크)

```
var express = require('express');

var app = express();

var bodyParser = require('body-parser');

var session = require('express-session');

var fs = require("fs")


app.set('views', __dirname + '/views');
```

```
app.set('view engine', 'ejs');

app.engine('html',
require('ejs').renderFile);



var server = app.listen(3000, function(){

 console.log("Express server has started
on port 3000")

});



app.use(express.static('public'));



app.use(bodyParser.json());

app.use(bodyParser.urlencoded());

app.use(session({

 secret: '@#@$MYSIGN#@$#$',

 resave: false,

 saveUninitialized: true
```

```
}));
```

```
var router = require('./router/main')(app, fs);
```

Express 의 이전 버전에서는 cookie-parser 모듈도 불러와야했지만, 이젠 express-session 모듈이 직접 쿠키에 접근하므로 cookie-parser 를 더이상 사용 할 필요가 없습니다.

추가적으로, Node.js 에 내장되어있는 fs 모듈도 불러왔는데요, 이는 나중에 파일을 열기 위함이랍니다. 그리고 원래 상단에 있던 router 코드를 아래로 내려주세요 (Line 27) 이 코드가 bodyParser 설정 아래부분에 있다면 제대로 작동하지 않는답니다. 그리고 router 에서 fs 모듈을 사용 할 수 있도록 인자로 추가해 줍니다.

router/main.js 에서 첫번째 줄도 업데이트해주세요.

```
module.exports = function(app, fs)

// ... 생략
```

session 부분에서의 값에 대해서 알아보겠습니다:

- **secret** – 쿠키를 임의로 변조하는것을 방지하기 위한 sign 값 입니다. 원하는 값을 넣으면 됩니다.
- **resave** – 세션을 언제나 저장할 지 (변경되지 않아도) 정하는 값입니다. express-session documentation 에서는 이 값을 false 로 하는것을 권장하고 필요에 따라 true 로 설정합니다.
- **saveUninitialized** – uninitialized 세션이란 새로 생겼지만 변경되지 않은 세션을 의미합니다. Documentation 에서 이 값을 true 로 설정하는것을 권장합니다.

2. EJS 템플릿 엔진

템플릿 엔진이란, 템플릿을 읽어 엔진의 문법과 설정에 따라서 파일을 HTML 형식으로 변환시키는 모듈입니다. Express 에서 사용하는 인기있는 Jade 템플릿 엔진은 기존의 HTML 에 비해 작성법이 완전 다른데, 그에 비해 EJS 는 똑같은 HTML 에서 <% %> 를 사용하여 서버의 데이터를 사용하거나 코드를 실행할 수 있습니다.
EJS 에서는 두가지만 알면 됩니다.

1. <% 자바스크립트 코드 %>
2. <% 출력 할 자바스크립트 객체 %>

2 번에서는 자바스크립트 객체를 router 에서 받아 올 수도 있습니다.

VIEW로 데이터 넘기기

우선, 전 강좌에서 작성하였던 views/index.html 과 views/about.html 을 삭제하시고,
router/main.js 를 다음과 같이 수정하세요.

```
module.exports = function(app, fs)

{

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

        res.render('index', {

            title: "MY HOMEPAGE",

            length: 5

        })

    });

}
```

JSON 데이터를 render 메소드의 두번째 인자로 전달함으로서 페이지에서 데이터를 사용가능하게됩니다.

VIEW 에서 데이터 접근 및 루프코드 실행

이제 views/index.ejs 를 다음과 같이 만들어 주세요.

```
<html>

  <head>

  <title><%= title %></title>

    <link rel="stylesheet" type="text/css"
href="css/style.css">

  </head>

  <body>

    <h1>Loop it!</h1>

    <ul>

        <% for(var i=0; i<length; i++){ %>

            <li>

                <%= "LOOP" + i %>

            </li>

        <% } %>

    </ul>

  </body>
```

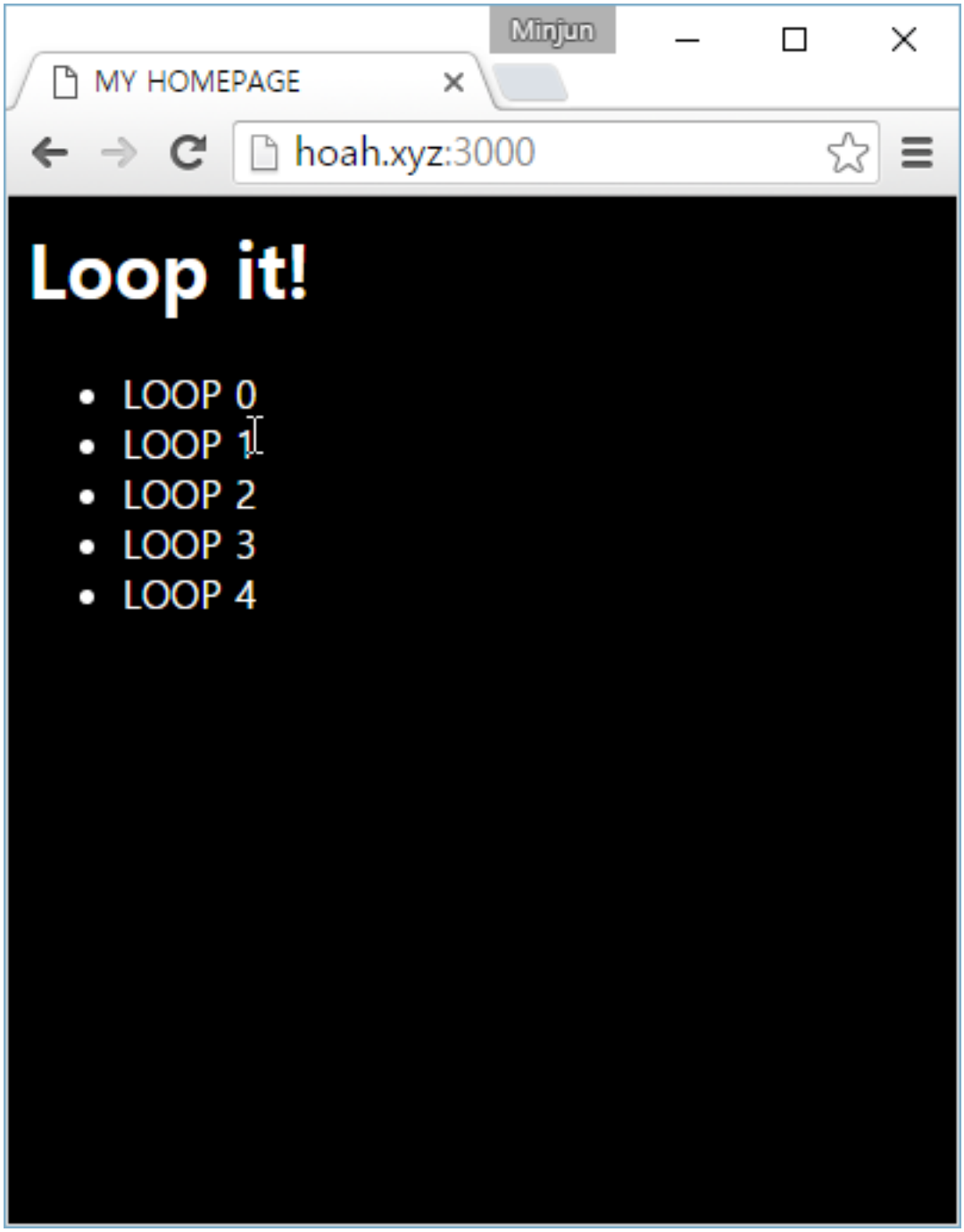
```
</html>
```

Line 3 : 라우터에서 title 받아와서 출력합니다.
Line 9~13: 루프문입니다.

출력

서버를 실행하고 http:/localhost:3000/ 에 접속해보세요.

```
$ node server.js
```



성공하셨나요? 자, 이제 view 코드를 여러 파일로 분리해봅시다.

EJS 분할하기

PHP 나 Rails 에서 처럼, EJS 에서도 코드를 여러 파일로 분리하고 불러와서 사용 할 수 있답니다. 파일을 불러올땐 다음 코드를 사용합니다.

```
<% include FILENAME %>
```

index.ejs 파일의 head 와 body 를 따로 파일로 저장해서 불러와보겠습니다.

header.ejs 파일과 body.ejs 파일:

```
 <title>

    <%= title %>

 </title>

 <link rel="stylesheet" type="text/css"
href="css/style.css">

 <script>

   console.log("HelloWorld");

 </script>

<h1>Loop it!</h1>

<ul>

   <% for(var i=0; i<length; i++){ %>
```

```
        <li>
            <%= "LOOP" + i %>
        </li>
    <% } %>
</ul>
```

이렇게 파일이 준비됐다면, index.ejs 를 다음과 같이 수정하면 됩니다.

```
<html>
  <head>
    <% include ./header.ejs %>
  </head>
  <body>
    <% include ./body.ejs %>
  </body>
</html>
```

이번 강좌에서는 EJS 의 기본적인 사용법에 대해 배웠습니다.

다음 강좌에서는 RESTful API 를 구현해보도록 하겠습니다.

[Node.JS] 강좌 10-2편: Express 프레임워크 응용하기 – RESTful API 편

2016년 2월 12일 velopertdev.log / node.js / tech.log24 Comments

이 강좌는 강좌 10-1 편과 이어지는 강좌입니다.
강좌를 작성하다가 글이 너무 길어져서 3 편으로 나누어 작성한 점 유의해주세요
10-1. EJS
10-2. Restful API
10-3. Session

---

3. RESTful API

REST 는 **R**epresentational **S**tate **T**ransfer 의 약자로서, 월드와이드웹(www) 와 같은 하이퍼미디어 시스템을 위한 소프트웨어 아키텍쳐 중 하나의 형식입니다. REST 서버는 클라이언트로 하여금 HTTP 프로토콜을 사용해 서버의 정보에 접근 및 변경을 가능케 합니다. 여기서 정보는 text, xml, json 등 형식으로 제공되는데, 요즘 트렌드는 json 이죠.

HTTP 메소드

HTTP/1.1 에서 제공되는 메소드는 여러개가 있는데요, REST 기반 아키텍쳐에서 자주 사용되는 4 가지 메소드는 다음과 같습니다.

1. **GET** – 조회
2. **PUT** – 생성 및 업데이트
3. **DELETE** – 제거
4. **POST** – 생성

여기서 잠깐, POST 와 PUT 좀 헷갈리지 않나요? 둘다 생성을 한다면..
어느 상황에 무엇을 써야하는거지? PUT vs POST, REST API (1ambda Blog) 여기서 궁금증을 해소하세요!

데이터베이스 생성

JSON 기반의 사용자 데이터베이스를 만들어보겠습니다.

Node.js 와 궁합이 잘 맞는 MongoDB 를 사용했더라면 좋았겠지만
이 포스트의 초점은 Express 이므로 다음으로 미루도록 하겠습니다.

2016/3/3 EDITED: 미룬 강좌가 작성되었습니다.
[Node.JS] 강좌 11 편: Express 와 Mongoose 를 통해 MongoDB 와 연동하여 RESTful API 만들기
data 폴더를 만들고 그 안에 user.json 파일을 생성해주세요:

```
{
    "first_user": {
        "password": "first_pass",
        "name": "abet"
    },
    "second_user":{
        "password": "second_pass",
        "name": "betty"
    }
}
```

(보안을 생각한다면 패스워드는 Encrypt 를 하거나 Hash 를 쓰는게 좋겠지만 이건 예제니까 PASS! 그 부분은나중에 한번 다루도록 하겠습니다.)

첫번째 API: GET /list

모든 유저 리스트를 출력하는 GET API 를 작성해보겠습니다.

우선, user.json 파일을 읽어야 하므로, fs 모듈을 사용하겠지요?

```
module.exports = function(app, fs)

{


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

        res.render('index', {

            title: "MY HOMEPAGE",

            length: 5

        })

    });
```

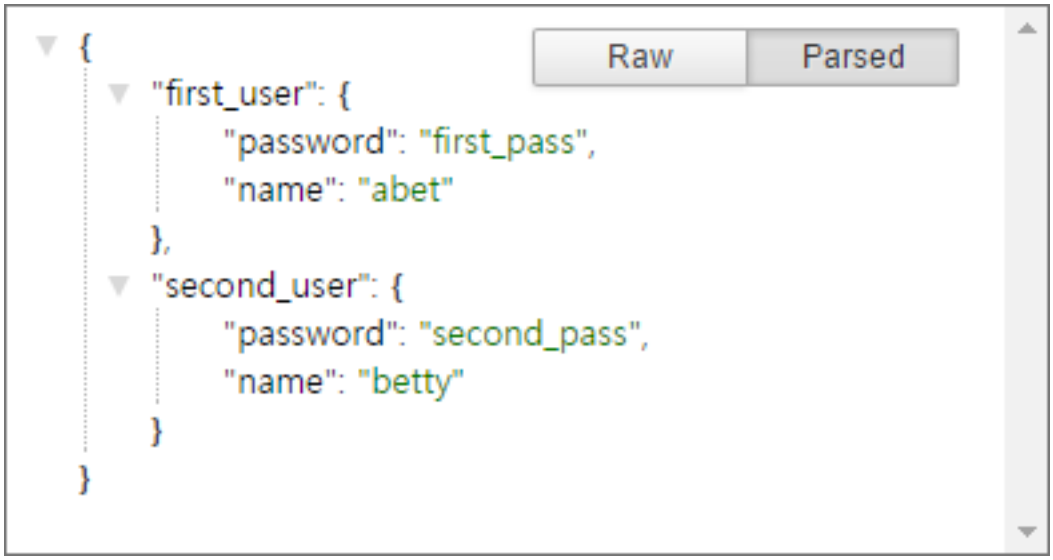
```
    app.get('/list', function (req, res) {

        fs.readFile( __dirname +
"/../data/" + "user.json", 'utf8',
function (err, data) {

            console.log( data );
```

```
            res.end( data );
        });
    })
```

```
}
```

_dirname 은 현재 모듈의 위치를 나타냅니다.
router 모듈은 router 폴더에 들어있으니, data 폴더에 접근하려면
/../ 를 앞부분에 붙여서 먼저 상위폴더로
접근해야합니다.

자 서버를 실행해서 http://localhost/list 에
접속해보세요.



두번째 API: GET /getUser/:username

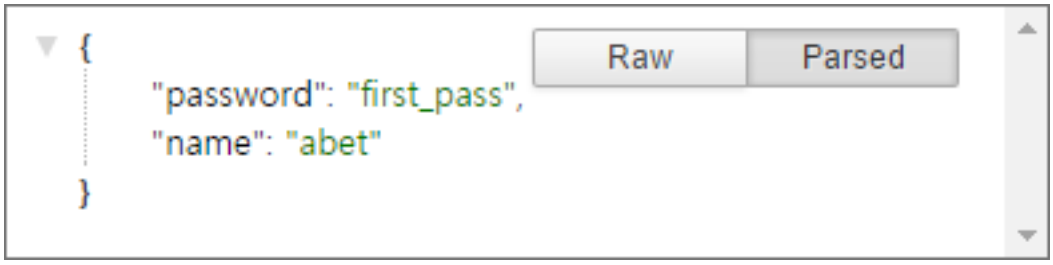
이번엔 특정 유저 username 의 디테일한 정보를 가져오는 GET API 를 작성해보도록 하겠습니다.

다음 코드를 router/main.js 의 list API 아래에 작성해주세요.

```
    app.get('/getUser/:username', function(req, res){

        fs.readFile( __dirname + "/../data/user.json", 'utf8', function (err, data) {

            var users = JSON.parse(data);

            res.json(users[req.params.username]);

        });

    });
```

파일을 읽은후, 유저 아이디를 찾아서 출력해줍니다.
유저를 찾으면 유저 데이터를 출력하고 유저가 없다면 {} 을 출력하게 됩니다.
fs.readFile()로 파일을 읽었을 시엔 텍스트 형태로읽어지기 때문에, JSON.parse() 를 해야합니다.

서버를 다시 실행 후 http://localhost:3000/getUser/first_user 에 접속해보세요.

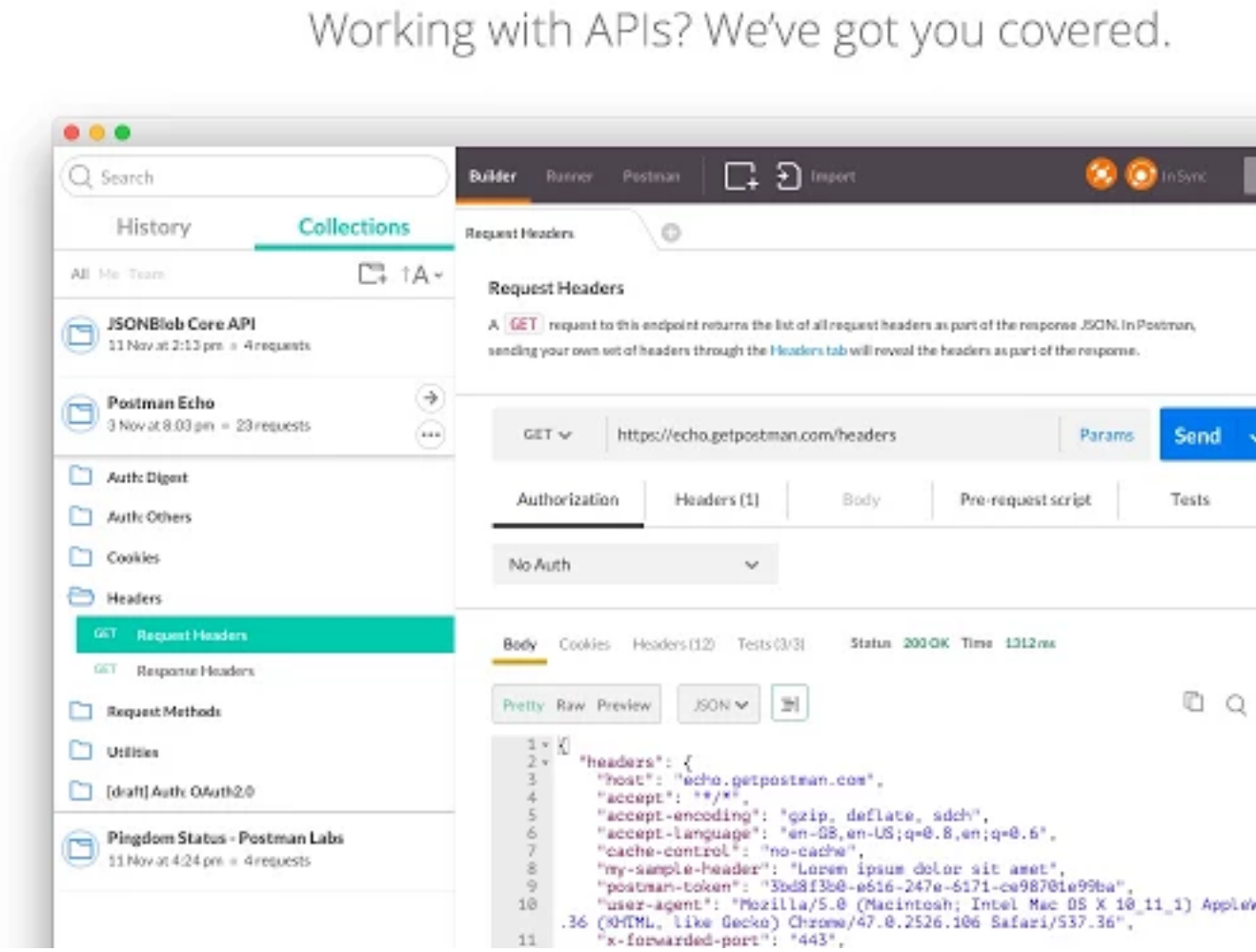


세번째 API: POST addUser/:username

body: { “password”: “____”, “name”: “____” }

이번 API 는 첫째 둘째와 다르게 POST 메소드를 사용합니다.

편한 테스팅을 위하여 구글 크롬 익스텐션 Postman 을 사용하겠습니다.



본격 Postman 리뷰글

HTTP 패킷을 요청하고 분석 할 수 있는 툴 입니다. 정말 괜찮은 앱이니 받아두세요. 물론 비슷한 프로그램이 이미 설치되있는사람들은 생략하셔도 됩니다.

다음 코드를 router/main.js 의 getUser API 하단에 작성해주세요:

```
    app.post('/addUser/:username',
function(req, res){



        var result = {  };

        var username = req.params.username;



        // CHECK REQ VALIDITY

        if(!req.body["password"]
|| !req.body["name"]){

            result["success"] = 0;

            result["error"] = "invalid
request";

            res.json(result);

            return;

        }
```

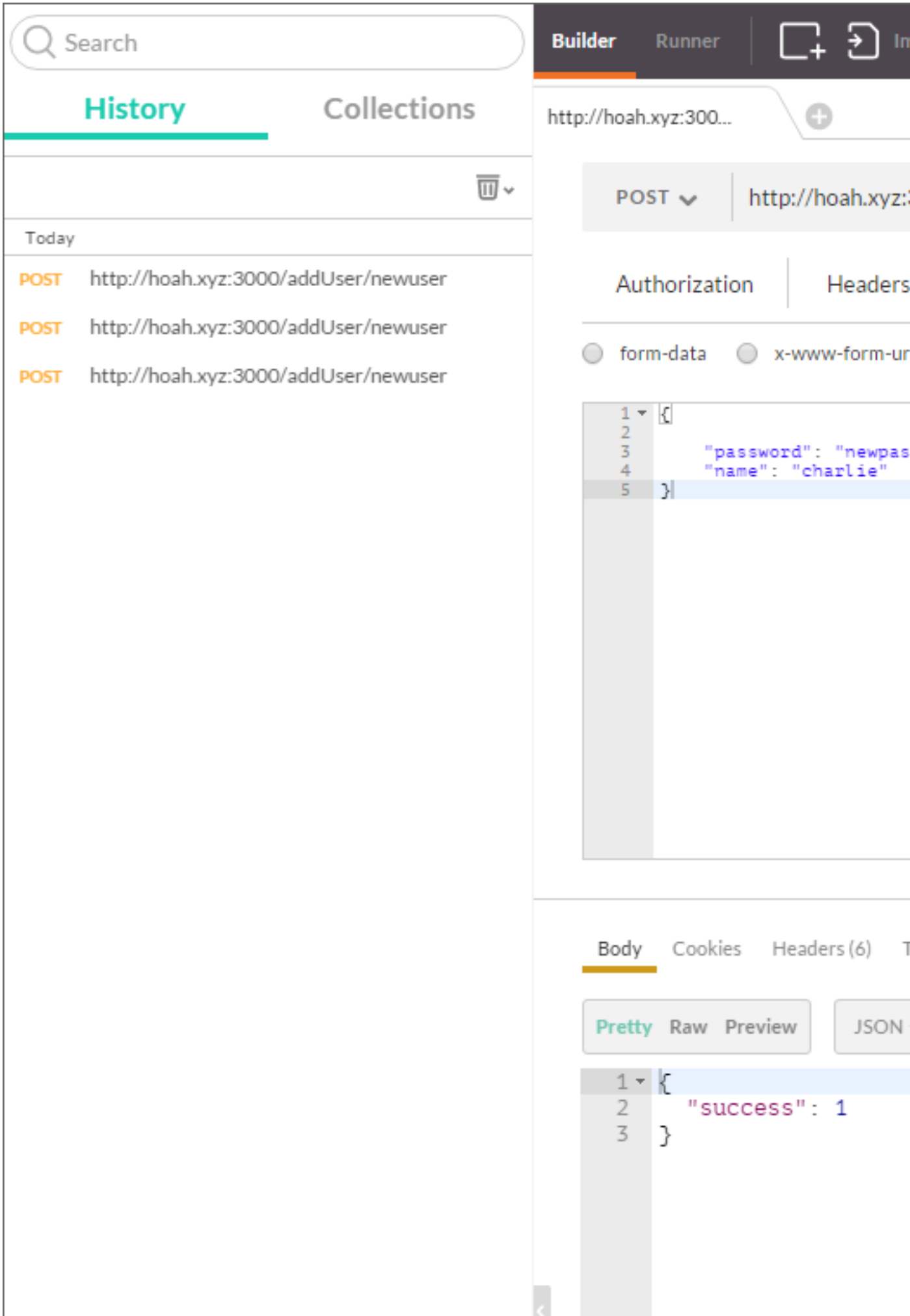
```
        // LOAD DATA & CHECK DUPLICATION

        fs.readFile( __dirname +
"/../data/user.json", 'utf8',
function(err, data){

            var users = JSON.parse(data);

            if(users[username]){

                // DUPLICATION FOUND

                result["success"] = 0;

                result["error"] =
"duplicate";

                res.json(result);

                return;

            }


            // ADD TO DATA

            users[username] = req.body;


            // SAVE DATA
```

```
            fs.writeFile(__dirname +
"/../data/user.json",

                         JSON.stringify(users,
null, '\t'), "utf8", function(err, data){

                result = {"success": 1};

                res.json(result);

            })

        })

    });
```

JSON 형태가 INVALID 하다면 오류를 반환하고, VALID 하다면 파일을 열어서 username 의 중복성을 확인 후 JSON 데이터에 추가하여 다시 저장합니다.

34 번줄에서 stringify(users, null, 2) 은 JSON 의 pretty-print 를 위함 입니다.

Postman 을 통하여 API 를 테스트해볼까요?


Search
History
Collections
Today
POST http://hoah.xyz:3000/addUser/newuser
POST http://hoah.xyz:3000/addUser/newuser
POST http://hoah.xyz:3000/addUser/newuser
Builder
Runner
http://hoah.xyz:300...
POST
http://hoah.xyz:
Authorization
Headers
form-data
x-www-form-ur
1 {
2
3 "password": "newpas
4 "name": "charlie"
5 }
Body
Cookies
Headers (6)
Pretty
Raw
Preview
JSON
1 {
2 "success": 1
3 }

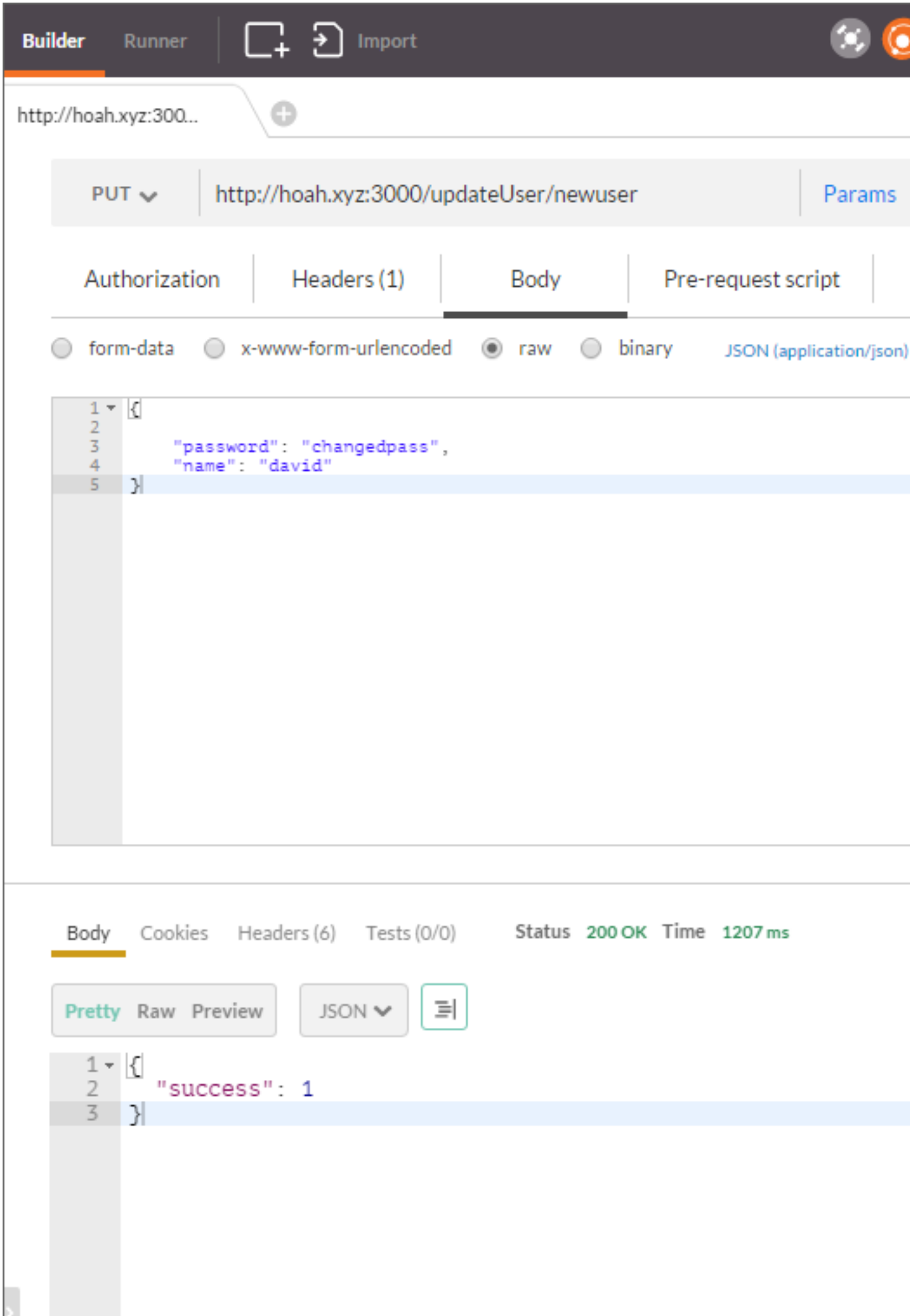
body 에서 Content-type 를 JSON 으로 하셔야 정상적으로 처리됩니다.

네번째 API: PUT updateUser/:username

body: { “password”: “_____”, “name”: “_____” }

이 API 는 위 API 와 비슷합니다. 사용자 정보를 업데이트 하는 API 이구요, PUT 메소드를 사용합니다.

PUT API 는 idempotent 해야 합니다, 쉽게말하자면 즉 요청을 몇번 수행하더라도, 같은 결과를 보장해야합니다.


Builder
Runner
Import
http://hoah.xyz:300...
PUT
http://hoah.xyz:3000/updateUser/newuser
Params
Authorization
Headers (1)
Body
Pre-request script
form-data
x-www-form-urlencoded
raw
binary
JSON (application/json)
{
"password": "changedpass",
"name": "david"
}
Body
Cookies
Headers (6)
Tests (0/0)
Status 200 OK
Time 1207 ms
Pretty
Raw
Preview
JSON
{
"success": 1
}

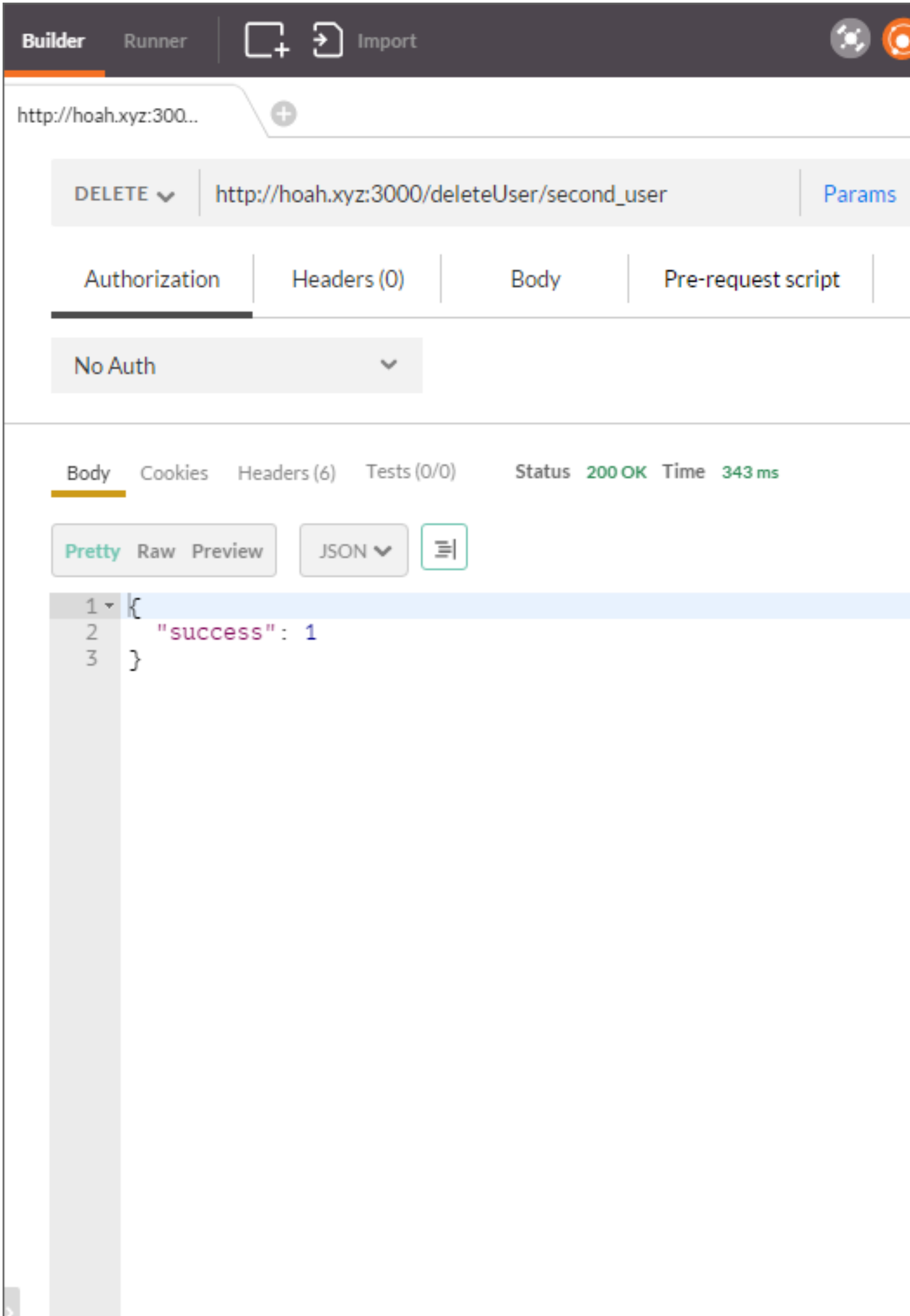
마지막 API: DELETE deleteUser/:username

유저를 데이터에서 삭제하는 API 입니다. DELETE 메소드를 사용합니다.

네번째 API 아래에 다음 코드를 작성해주세요:

```
    app.delete('/deleteUser/:username',
function(req, res){

        var result = { };

        //LOAD DATA

        fs.readFile(__dirname +
"/../data/user.json", "utf8",
function(err, data){

            var users = JSON.parse(data);



            // IF NOT FOUND


if(!users[req.params.username]){

                result["success"] = 0;

                result["error"] = "not
found";

                res.json(result);

                return;
```

```
            }


            delete
users[req.params.username];

            fs.writeFile(__dirname +
"/../data/user.json",

                         JSON.stringify(users,
null, '\t'), "utf8", function(err, data){

                result["success"] = 1;

                res.json(result);

                return;

            })

        })


    })
```


Builder
Runner
Import
http://hoah.xyz:300...
DELETE
http://hoah.xyz:3000/deleteUser/second_user
Params
Authorization
Headers (0)
Body
Pre-request script
No Auth
Body
Cookies
Headers (6)
Tests (0/0)
Status 200 OK
Time 343 ms
Pretty
Raw
Preview
JSON
1 {
2 "success": 1
3 }

router/main.js 전체 코드

```
module.exports = function(app, fs)

{


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

        res.render('index', {

            title: "MY HOMEPAGE",

            length: 5

        })

    });


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

      fs.readFile( __dirname +
"/../data/user.json", 'utf8', function
(err, data) {

          console.log( data );

          res.end( data );

      });
```

```
    });


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

      fs.readFile( __dirname +
"/../data/user.json", 'utf8', function
(err, data) {

          var users = JSON.parse(data);


res.json(users[req.params.username]);

      });

    });


    app.post('/addUser/:username',
function(req, res){


       var result = {  };

       var username = req.params.username;


       // CHECK REQ VALIDITY
```

```
        if(!req.body["password"]
|| !req.body["name"]){

            result["success"] = 0;

            result["error"] = "invalid
request";

            res.json(result);

            return;

        }



        // LOAD DATA & CHECK DUPLICATION

        fs.readFile( __dirname +
"/../data/user.json", 'utf8',
function(err, data){

            var users = JSON.parse(data);

            if(users[username]){

                // DUPLICATION FOUND

                result["success"] = 0;

                result["error"] =
"duplicate";

                res.json(result);
```

```
            return;

        }


        // ADD TO DATA

        users[username] = req.body;


        // SAVE DATA

        fs.writeFile(__dirname +
"/../data/user.json",

                     JSON.stringify(users,
null, '\t'), "utf8", function(err, data){

            result = {"success": 1};

            res.json(result);

        })

    })

});
```

```
    app.put('/updateUser/:username',
function(req, res){



        var result = {  };

        var username = req.params.username;



        // CHECK REQ VALIDITY

        if(!req.body["password"]
|| !req.body["name"]){

            result["success"] = 0;

            result["error"] = "invalid
request";

            res.json(result);

            return;

        }



        // LOAD DATA

        fs.readFile( __dirname +
"/../data/user.json", 'utf8',
function(err, data){
```

```
            var users = JSON.parse(data);

            // ADD/MODIFY DATA

            users[username] = req.body;


            // SAVE DATA

            fs.writeFile(__dirname +
"/../data/user.json",

                         JSON.stringify(users,
null, '\t'), "utf8", function(err, data){

                result = {"success": 1};

                res.json(result);

            })

        })

    });



    app.delete('/deleteUser/:username',
function(req, res){

        var result = { };
```

```
        //LOAD DATA

        fs.readFile(__dirname +
"/../data/user.json", "utf8",
function(err, data){

            var users = JSON.parse(data);



            // IF NOT FOUND


if(!users[req.params.username]){

                result["success"] = 0;

                result["error"] = "not
found";

                res.json(result);

                return;

            }



            // DELETE FROM DATA

            delete
users[req.params.username];
```

```
            // SAVE FILE

            fs.writeFile(__dirname +
"/../data/user.json",

                         JSON.stringify(users,
null, '\t'), "utf8", function(err, data){

                result["success"] = 1;

                res.json(result);

                return;

            })

        })



    })


}
```

이렇게 Express 응용 RESTful API 편을 마치도록 하겠습니다.
다음 편에서는 세션을 다루도록 하겠습니다.

[Node.JS] 강좌 10-3편: Express 프레임워크 응용하기 - express-session 편

2016년 2월 13일 velopertdev.log / node.js / tech.log7 Comments

이 강좌는 강좌 10-2 편과 이어지는 강좌입니다.

강좌를 작성하다가 글이 너무 길어져서 3 편으로 나누어 작성한 점 유의해주세요

10-1: EJS
10-2: RESTful API
10-3: express-session

---

express-session
express-session 은 Express 프레임워크에서 세션을 관리하기 위해 필요한 미들웨어 입니다.

이번 강좌에서는 express-session 을 통해 로그인 및 로그아웃을 구현해보도록 해보겠습니다.

사용 예제

Express에 적용

```
var session = require('express-session');


app.use(session({
 secret: '@#@$MYSIGN#@$#$',
 resave: false,
 saveUninitialized: true
}));
```

- **secret** – 쿠키를 임의로 변조하는것을 방지하기 위한 값 입니다. 이 값을 통하여 세션을 암호화 하여 저장합니다.
- **resave** – 세션을 언제나 저장할 지 (변경되지 않아도) 정하는 값입니다. express-session documentation 에서는 이 값을 false 로 하는것을 권장하고 필요에 따라 true 로 설정합니다.
- **saveUninitialized** – 세션이 저장되기 전에 uninitialized 상태로 미리 만들어서 저장합니다.

세션 초기 설정 (initialization)

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

    sess = req.session;

});
```

간단하게 이렇게 라우터 콜백함수 안에서 req.session 으로 세션에 접근 할 수 있습니다.

세션 변수 설정

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

    sess = req.session;

    sess.username = "velopert"

});
```

따로 키를 추가해줄 필요 없이 sess.[키 이름] = 값 으로 세션 변수를 설정 할 수 있습니다.

세션 변수 사용

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

    sess = req.session;

    console.log(sess.username);

});
```

예상 하셨겠지만 세션 변수를 사용하는 것 역시 이렇게 간단합니다.

세션 제거

```
req.session.destroy(function(err){

   // cannot access session here

});
```

세션을 제거할땐 (로그아웃) 위와 같이 합니다.
destroy() 메소드의 콜백함수에서는 세션에 접근 할 수 없다는점 명심하세요.

적용

자, 이제 express-session 을 저희가 강좌 10-1 과 10-2 에서 만들던 프로젝트에 적용해보겠습니다.

LOGIN API

강좌 10-2 REST API 편에서 배웠던 지식을 토대로 로그인 API 를 작성해봅시다.
(router/main.js 에 추가)

```
    app.get('/login/:username/:password',
function(req, res){

        var sess;

        sess = req.session;
```

```
        fs.readFile(__dirname +
"/../data/user.json", "utf8",
function(err, data){

            var users = JSON.parse(data);

            var username =
req.params.username;

            var password =
req.params.password;

            var result = {};

            if(!users[username]){

                // USERNAME NOT FOUND

                result["success"] = 0;

                result["error"] = "not
found";

                res.json(result);

                return;

            }
```

```
            if(users[username]["password"]
== password){

                result["success"] = 1;

                sess.username = username;

                sess.name =
users[username]["name"];

                res.json(result);



            }else{

                result["success"] = 0;

                result["error"] =
"incorrect";

                res.json(result);

            }

        })

    });
```

로그인에 성공했다면 세션에 username 과 name 을 저장하도록 했습니다.

LOGOUT API

로그인이 있었으면 로그아웃도 만들어야겠죠?

다음 코드를 작성해주세요 (router/main.js 에 추가)

```
    app.get('/logout', function(req, res){
        sess = req.session;
        if(sess.username){

req.session.destroy(function(err){
                if(err){
                    console.log(err);
                }else{
                    res.redirect('/');
                }
            })
        }else{
            res.redirect('/');
        }
    })
```

로그아웃을 하고 메인페이지로 redirect 됩니다.

메인페이지 수정

메인페이지에서 세션을 조회 할 수 있도록 수정해줍니다.
우선 라우터 상단의 app.get('/'...) 부분을 이렇게 업데이트 하세요.

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

    var sess = req.session;



    res.render('index', {

        title: "MY HOMEPAGE",

        length: 5,

        name: sess.name,

        username: sess.username

    })

});
```

세션 변수를 EJS 템플릿 엔진에 전달 하게 했습니다.
이제 EJS 파일을 수정해야겠죠?

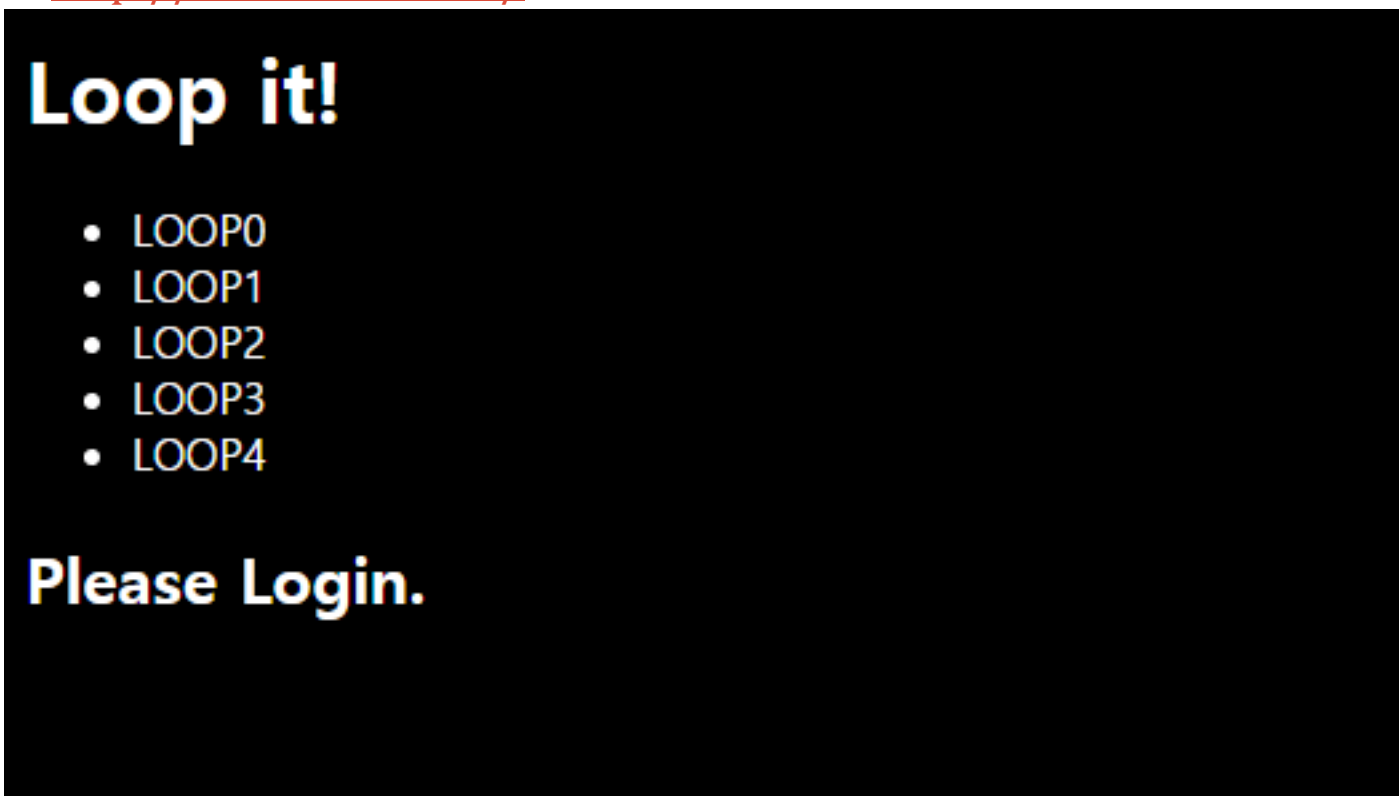
views/body.ejs 를 다음과같이 수정 하세요.

```
<h1>Loop it!</h1>

<ul>

    <% for(var i=0; i<length; i++){ %>

        <li>

            <%= "LOOP" + i %>

        </li>

    <% } %>

</ul>



<% if(username){ %>

    <h2>Welcome! <%= username %> (name:
<%= name %>)</h2>

<% }else{ %>

    <h2> Please Login. </h2>

<% } %>
```

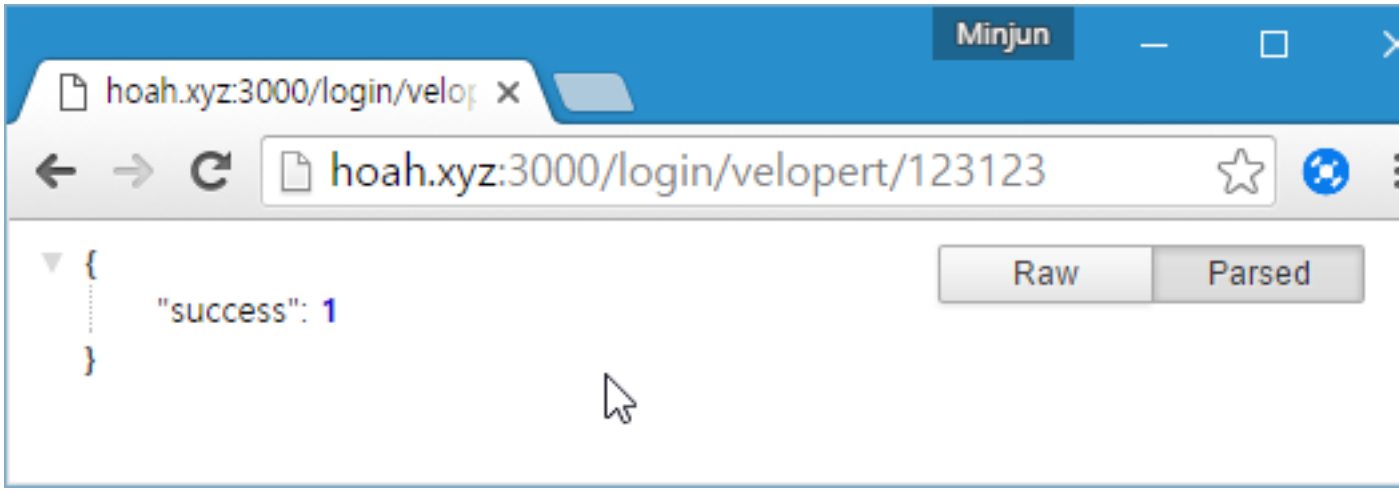
로그인 되었는지 안되어있는지 확인하고
로그인상태라면 환영메시지를,
그렇지 않다면 로그인 하라는 메시지를 띄웁니다.

테스트

1. http://localhost:3000/ 접속

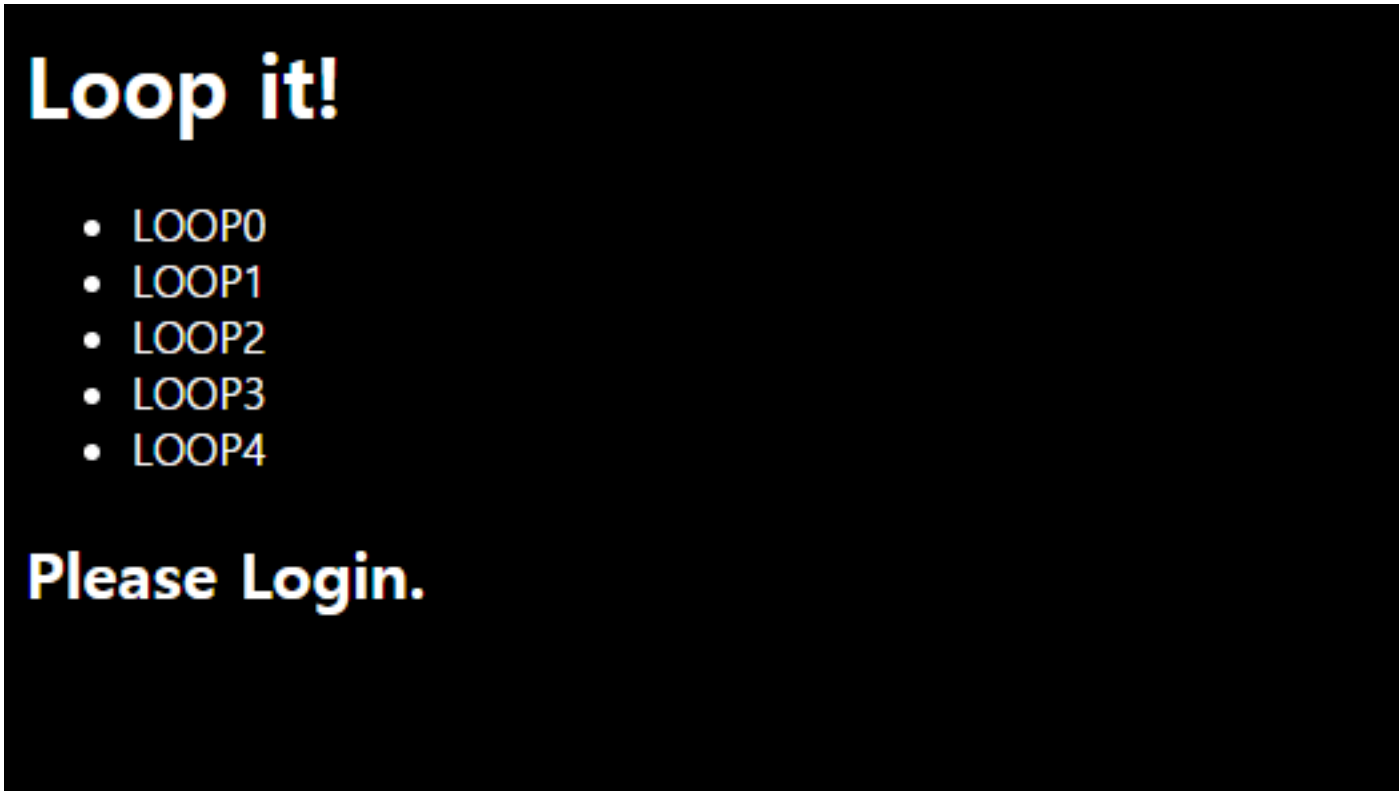


2. http://localhost:3000/user/pass 접속 (유저 데이터는 직접 입력하세요)



3. http://localhost:3000/ 다시 접속

4. http://localhost:3000/logout 접속



마무리

수고하셨습니다. Express.js 를 사용해보면서 Node.js 와 Express.js 웹 프레임워크에 대한 기초를 다졌습니다. 강좌에서 사용된 코드는 https://github.com/velopert/express-tutorial 에서 모두 확인 할 수 있습니다.

[Node.JS] 강좌 11편: Express와 Mongoose를 통해 MongoDB와 연동하여 RESTful API 만들기

2016년 3월 2일 velopertdev.log / node.js / tech.log22 Comments

이번 강좌에서는 Mongoose 를 통하여 Node.js 에서 MongoDB 와 연동하는것을 배워보겠습니다.

1. 소개

Mongoose 는 MongoDB 기반 ODM(Object Data Mapping) Node.JS 전용 라이브러리입니다. ODM 은 데이터베이스와 객체지향 프로그래밍 언어 사이 호환되지 않는 데이터를 변환하는 프로그래밍 기법입니다. 즉 MongoDB 에 있는 데이터를 여러분의 Application 에서 JavaScript 객체로 사용 할 수 있도록 해줍니다.

Note: 이 강좌는 Node.js 와 MongoDB 가 설치되있다는 전제하에 진행됩니다.
또한, MongoDB 에 전반적인 지식이 없다면 mongoose 사용이 다소 어려울 수 있습니다.
MongoDB 기초 강좌는 여기서 볼 수 있습니다.

2. 프로젝트 생성 및 패키지 설치

Mongoose 를 배워가면서, 간단한 Express 기반의 RESTful 프로젝트를 만들어보도록 하겠습니다.

2.1 프로젝트 생성

우선 npm init 을 통하여 package.json 을 생성하세요.
엔터를 계속 눌러 설정값은 기본값으로 하시면 됩니다.

```
$ npm init
```

2.2 패키지 설치

프로젝트에서 사용 할 패키지를 설치하겠습니다.

1. express: 웹프레임워크
2. body-parser: 데이터 처리 미들웨어
3. mongoose: MongoDB 연동 라이브러리

```
$ npm install --save express mongoose
body-parser
```

(설치할 때 mongoose 라고 폴더이름을 주면 안된다.
이름이 충돌난다. 18/4/2 일 체크)

명령어를 입력하시면 자동으로 패키지를 설치하고,
package.json 파일에 패키지 리스트를 추가합니다.

3. 서버 설정하기

3.1 디렉토리 구조

먼저 저희가 만들 프로젝트의 디렉토리 구조를
살펴봅시다.

```
- models/
----- book.js
- node_modules/
- routes
----- index.js
- app.js
- package.json
```

이 파일들은 강좌를 진행하면서 만들도록 하겠습니다.

3.2 Express 로 이용한 웹서버 생성

mongoose 를 사용하기 위해서 우선 book 데이터를 조회·수정·삭제 하는간단한 RESTful 웹서버를 작성해보겠습니다.
이 서버에 만들 API 목록은 다음과 같습니다.

ROUTE	METHOD	DESCRIPTION
/api/books	GET	모든 book 데이터 조회
/api/books/:book_id	GET	_id 값으로 데이터 조회

ROUTE	METHOD	DESCRIPTION
/api/books/author/:author	GET	author 값으로 데이터 조회
/api/books	POST	book 데이터 생성
/api/books/:book_id	PUT	book 데이터 수정
/api/books/:book_id	DELETE	book 데이터 제거

우선 서버의 메인 파일인 **app.js** 를 작성하세요.

```
// app.js


// [LOAD PACKAGES]
var express    = require('express');
var app        = express();
var bodyParser  = require('body-parser');
var mongoose    = require('mongoose');
```

```
// [CONFIGURE APP TO USE bodyParser]

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

app.use(bodyParser.json());



// [CONFIGURE SERVER PORT]

var port = process.env.PORT || 8080;



// [CONFIGURE ROUTER]

var router = require('./routes')(app)



// [RUN SERVER]

var server = app.listen(port, function(){

 console.log("Express server has started
on port " + port)

});
```

line 17 에서 router 모듈을 불러오게 했죠? 이제 router 를 작성해봅시다.

routes/index.js 에 다음 코드를 입력하세요.

```
// routes/index.js


module.exports = function(app)

{

    // GET ALL BOOKS

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

        res.end();

    });



    // GET SINGLE BOOK

    app.get('/api/books/:book_id',
function(req, res){

        res.end();

    });



    // GET BOOK BY AUTHOR
```

```
    app.get('/api/books/author/:author',
function(req, res){

        res.end();

    });



    // CREATE BOOK

    app.post('/api/books', function(req,
res){

        res.end();

    });



    // UPDATE THE BOOK

    app.put('/api/books/:book_id',
function(req, res){

        res.end();

    });



    // DELETE BOOK
```

```
    app.delete('/api/books/:book_id',
function(req, res){

        res.end();

    });


}
```

4. MongoDB 서버 연결

MongoDB 서버에 연결 하는 방법은 다음과 같습니다.

```
// app.js


// ......


var mongoose    = require('mongoose');


// ......


// [ CONFIGURE mongoose ]
```

```
// CONNECT TO MONGODB SERVER

var db = mongoose.connection;

db.on('error', console.error);

db.once('open', function(){

    // CONNECTED TO MONGODB SERVER

    console.log("Connected to mongod
server");

});



mongoose.connect('mongodb://localhost/mon
godb_tutorial');



// ......
```

mongoose.connect() 메소드로 서버에 접속을 할 수 있으며, 따로 설정 할 파라미터가 있다면 다음과 같이 uri 를 설정하면 됩니다.

```
mongoose.connect('mongodb://username:pass
word@host:port/database?options...');
```

이 강좌에서는 mongodb_tutorial db 를 사용하도록 하겠습니다.

5. Schema & Model

5.1 schema

schema 는 document 의 구조가 어떻게 생겼는지 알려주는 역할을 합니다.

schema 를 만드는 방법은 다음과 같습니다. 이 코드를 **models/book.js** 에 입력하세요.

```
var mongoose = require('mongoose');

var Schema = mongoose.Schema;



var bookSchema = new Schema({

    title: String,

    author: String,

    published_date: { type: Date, default: Date.now  }

});



module.exports = mongoose.model('book', bookSchema);
```

schema 에서 사용되는 SchemaType 은 총 8 종류가 있습니다.

1. String
2. Number
3. Date
4. Buffer
5. Boolean
6. Mixed
7. Objectid
8. Array

이를 사용하는 예제는 매뉴얼을 참고하세요.

5.2 model

model 은 데이터베이스에서 데이터를 읽고, 생성하고, 수정하는프로그래밍 인터페이스를 정의합니다.

```
// DEFINE MODEL

var Book = mongoose.model('book', bookSchema);
```

첫번째 인자 book' 은 해당 document 가 사용 할 collection 의 단수적 표현입니다. 즉, 이 모델에서는 'books' collection 을 사용하게 되겠죠. 이렇게 자동으로 단수적 표현을 복수적(plural) 형태로 변환하여 그걸 collection 이름으로 사용합니다. collection 이름을 plural 형태로 사용하는건 mongodb 의 네이밍컨벤션 중 하나입니다.

만약에, collection 이름을 임의로 정하고 싶다면, schema 를 만들 때 따로 설정해야 합니다.

```
var dataSchema = new Schema({..},
{ collection: 'COLLECTION_NAME' });
```

model 을 만들고 나면, model 을 사용하여 다음과 같이 데이터를 데이터베이스에 저장하거나 조회 할 수 있습니다.

```
var book = new Book({

    name: "NodeJS Tutorial",

    author: "velopert"

});

book.save(function(err, book){

    if(err) return console.error(err);

    console.dir(book);

});
```

이와 같이 model 을 사용하여 데이터베이스와 연동하는 자세한 방법에 대해서는 다음 섹션에서 다루도록 하겠습니다.

저희는 model 을 **models/bear.js** 를 모듈화해서 만들게 할 것이므로, 다음과 같이 해당파일의 마지막줄에서 model 을 모듈화하세요.

```
// models/book.js
```

```
var Schema = mongoose.Schema;


var bookSchema = new Schema({

    title: String,

    author: String,

    published_date: { type: Date, default: Date.now  }

});


module.exports = mongoose.model('book', bookSchema);
```

app.js 에서 이 모듈을 불러와보겠습니다.

```
// app.js


// ...


// CONNECT TO MONGODB SERVER
```

```
// ...


// DEFINE MODEL

var Book = require('./models/book');


// ...
```

6. CRUD (Create, Retrieve, Update, Delete)

6.0 시작하기 전에..

3 번 섹션에서 만들었던 API 를 직접 구현해가면서 데이터를 생성/조회/수정/제거 하는 방법을 알아보겠습니다.|
라우터에서 Book 모델을 사용해야 하므로, 라우터에 Book 을 전달해줘야겠죠?
따라서 **/routes/index.js** 와 **app.js** 를 우선 수정해야합니다.

```
// routes/index.js

modules.exports = function(app, Book)

{

    // ....
```

```
}

// app.js


// ...


var router = require('./routes')(app,
Book);


// ...
```

6.1 Create (POST /api/books)

book 데이터를 데이터베이스에 저장하는 API 입니다.

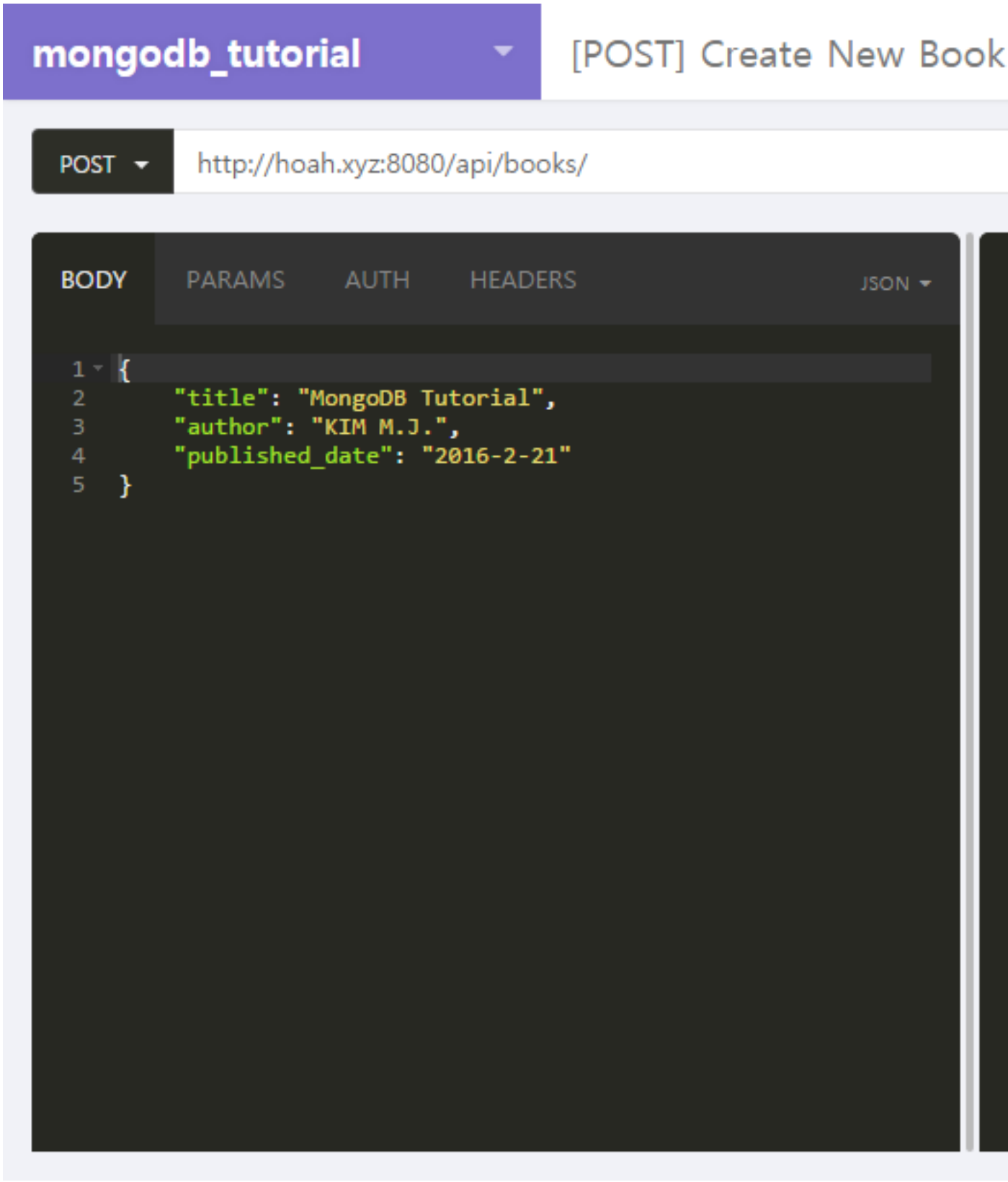
```
app.post('/api/books', function(req,
res){

    var book = new Book();

    book.title = req.body.name;

    book.author = req.body.author;

    book.published_date = new
Date(req.body.published_date);
```

```
    book.save(function(err){
        if(err){
            console.error(err);
            res.json({result: 0});
            return;
        }

        res.json({result: 1});

    });
});
```

.save 메소드는 데이터를 데이터베이스에 저장합니다. err 을 통하여 오류처리가 가능합니다.

이 API 에서는 데이터 저장에 성공하면 result: 1 을, 실패하면 result: 0 을 반환합니다.

(기존 포스트맨 플러그인이 디프리케이트되어서 맥용 앱을 설치함. 19/4/2 일 체크)


mongodb_tutorial
[POST] Create New Book
POST
http://hoah.xyz:8080/api/books/
BODY
PARAMS
AUTH
HEADERS
JSON
{
"title": "MongoDB Tutorial",
"author": "KIM M.J.",
"published_date": "2016-2-21"
}

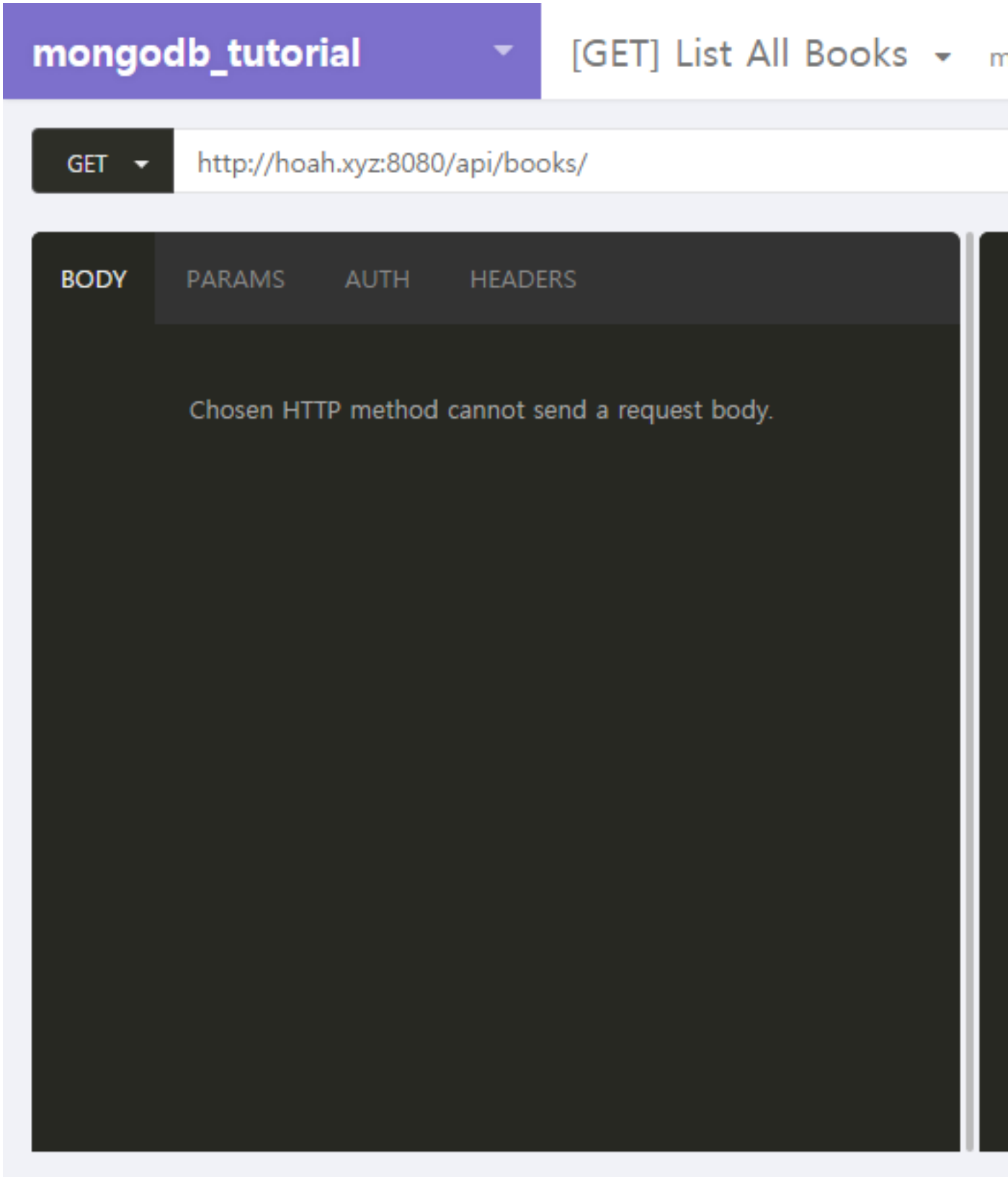
(REST API 테스팅에 사용된 어플리케이션은 크롬 확장 프로그램 Insomnia 입니다)

6.2.1 RETRIEVE (GET /api/books)

모든 book 데이터를 조회하는 API 입니다.

```
// GET ALL BOOKS

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

    Book.find(function(err, books){

        if(err) return
res.status(500).send({error: 'database
failure'});

        res.json(books);

    })

});
```

데이터를 조회 할 때는 find() 메소드가 사용됩니다.
query 를 파라미터 값으로 전달 할 수 있으며,
파라미터가 없을 시, 모든 데이터를 조회합니다.
데이터베이스에 오류가 발생하면 HTTP Status 500 과 함께 에러를 출력합니다.



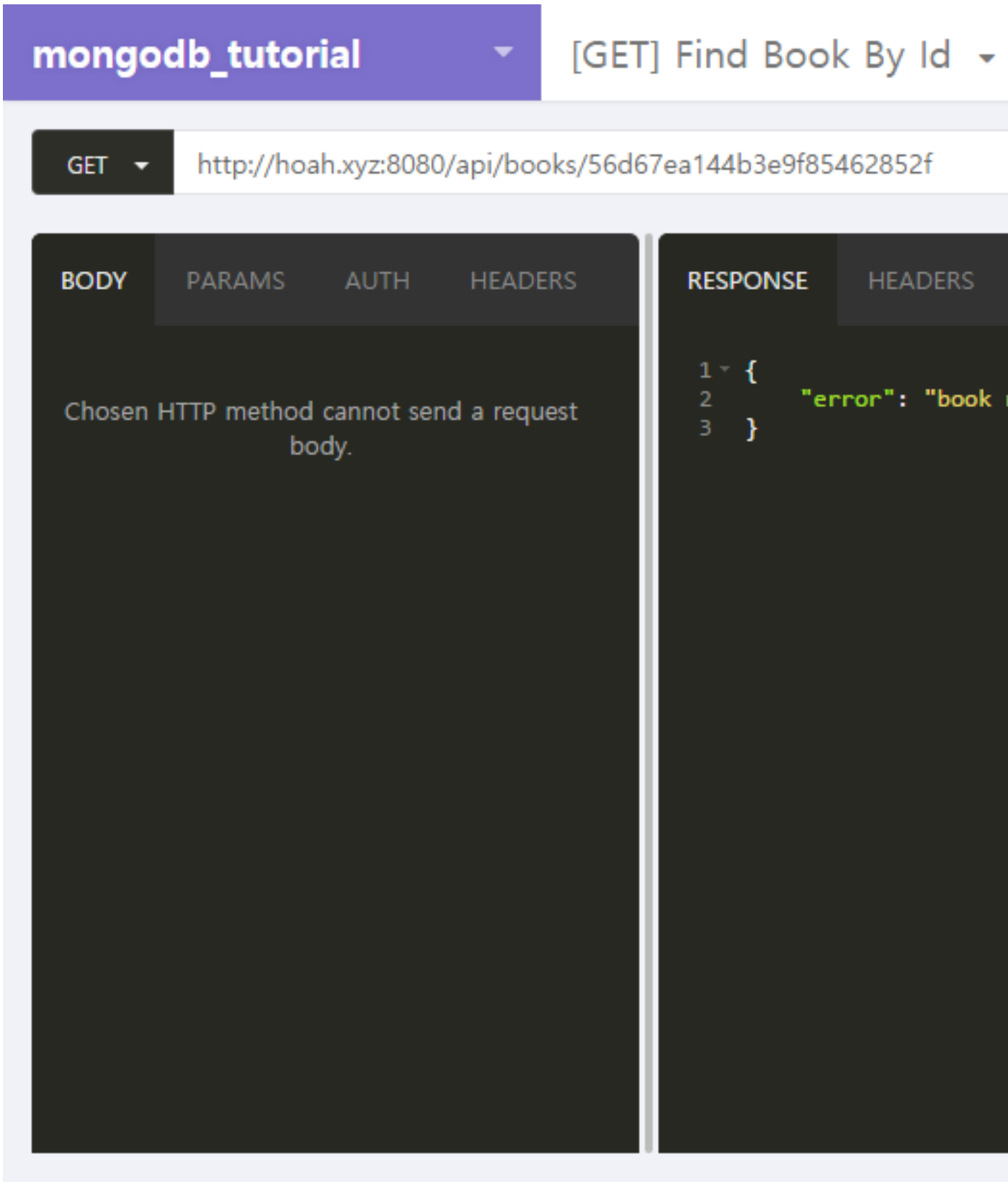
6.2.2 RETRIEVE (GET /api/books/:book_id)

데이터베이스에서 Id 값을 찾아서 book document 를 출력합니다.

```
// GET SINGLE BOOK

app.get('/api/books/:book_id',
function(req, res){

    Book.findOne({_id: req.params.book_id},
function(err, book){

        if(err) return
res.status(500).json({error: err});

        if(!book) return
res.status(404).json({error: 'book not
found'});

        res.json(book);

    })

});
```

하나의 데이터만 찾을 것이기 때문에, findOne 메소드가 사용되었습니다.

오류가 발생하면 500, 데이터가 없으면 404 HTTP Status 와 함께 오류를 출력합니다.

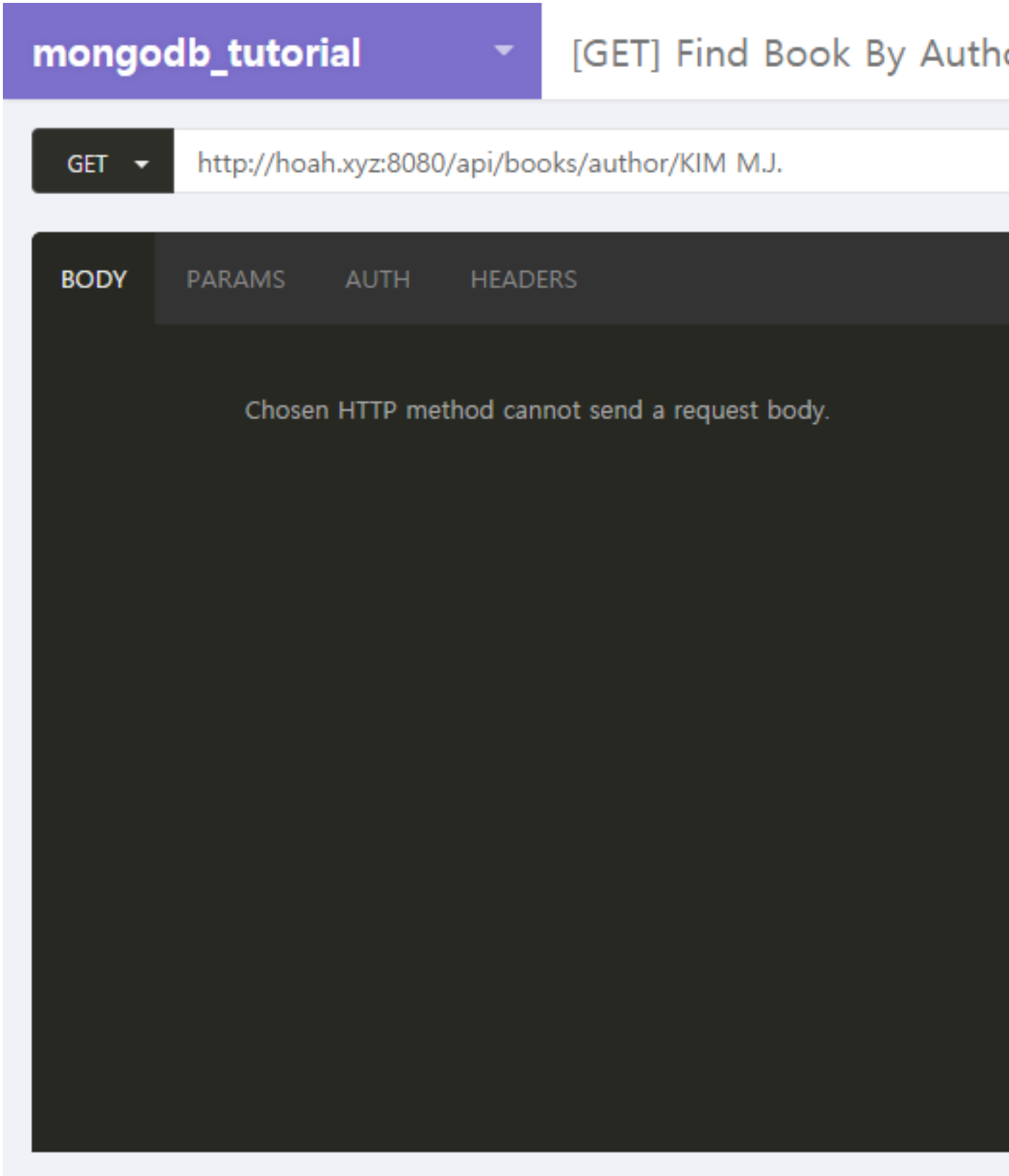


6.2.3 RETRIEVE (GET /api/books/author/:author)

author 값이 매칭되는 데이터를 찾아 출력합니다.

```
// GET BOOKS BY AUTHOR

app.get('/api/books/author/:author',
function(req, res){

    Book.find({author: req.params.author},
{_id: 0, title: 1, published_date: 1},
function(err, books){

        if(err) return
res.status(500).json({error: err});

        if(books.length === 0) return
res.status(404).json({error: 'book not
found'});

        res.json(books);

    })

});
```

find() 메소드에서 첫번째 인자에는 query 를, 두번째는 projection 을 전달해주었습니다.
이를 통하여 author 값으로 찾아서 title 과 published_date 만 출력합니다.
(만약에 projection 이 생략되었다면 모든 field 를 출력합니다.)


mongodb_tutorial
[GET] Find Book By Autho
GET
http://hoah.xyz:8080/api/books/author/KIM M.J.
BODY
PARAMS
AUTH
HEADERS
Chosen HTTP method cannot send a request body.

6.3 UPDATE (PUT /api/books/:book_id)

book_id 를 찾아서 document 를 수정합니다.

```
// UPDATE THE BOOK

app.put('/api/books/:book_id',
function(req, res){

    Book.findById(req.params.book_id,
function(err, book){

        if(err) return
res.status(500).json({ error: 'database
failure' });

        if(!book) return
res.status(404).json({ error: 'book not
found' });



        if(req.body.title) book.title =
req.body.title;

        if(req.body.author) book.author =
req.body.author;

        if(req.body.published_date)
book.published_date =
req.body.published_date;



        book.save(function(err){
```

```
            if(err)
res.status(500).json({error: 'failed to
update'});

            res.json({message: 'book
updated'});

        });



    });



});
```

데이터를 수정 할 땐, 데이터를 먼저 찾은 후, save() 메소드를 통하여 수정하면 됩니다.
update 하는 방법은 이 외에도 다른 방법이 있는데요, 만약 어플리케이션에서 기존 document 를 굳이 조회 할 필요가없다면
update() 메소드를 통하여 바로 document 를 업데이트 할 수 있습니다.
아래 코드는 코드와 같은 동작을 하지만 업데이트하는 과정에서 document 를 조회 하지 않습니다.

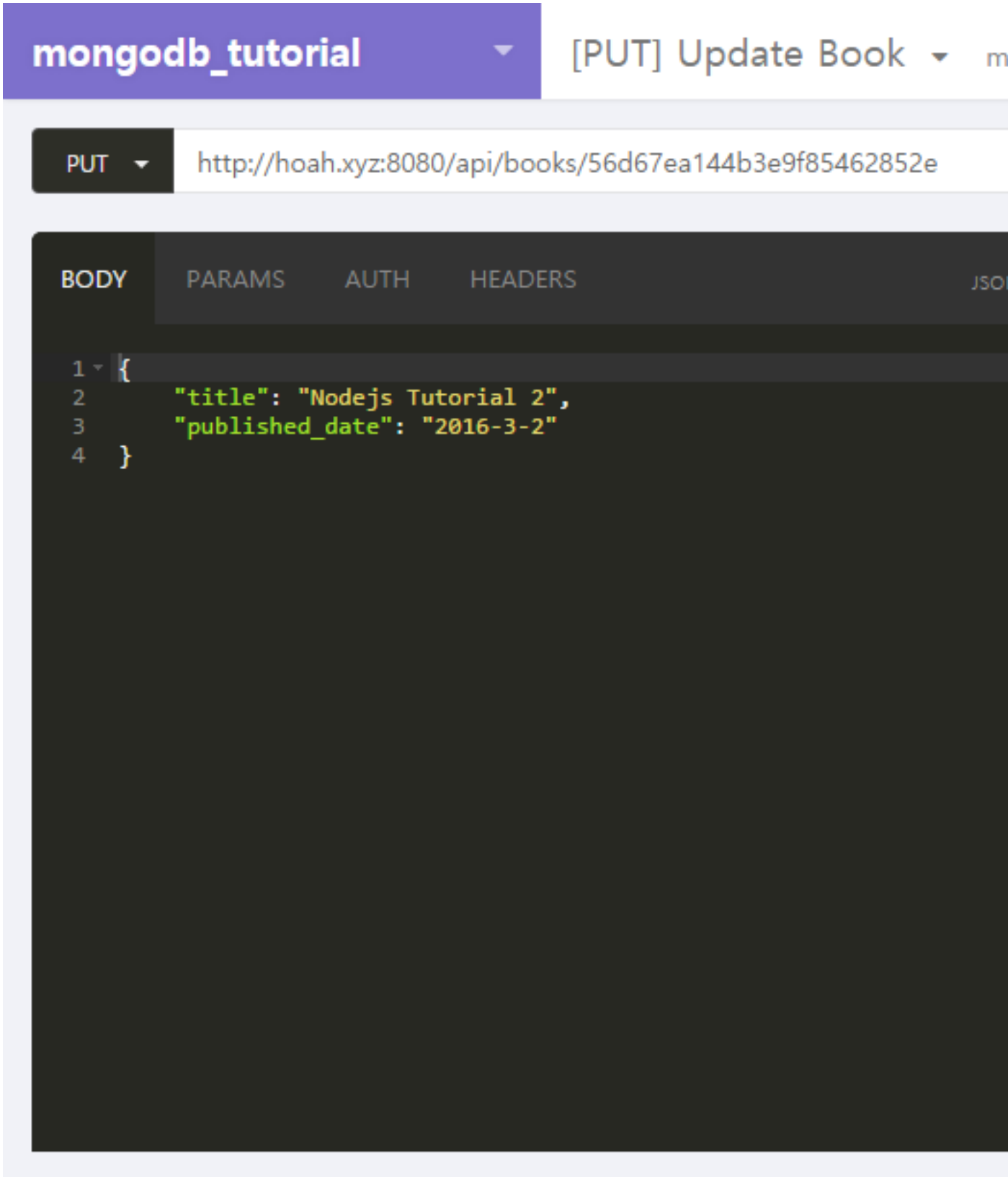
```
// UPDATE THE BOOK (ALTERNATIVE)

app.put('/api/books/:book_id',
function(req, res){
```

```
    Book.update({ _id:
req.params.book_id }, { $set: req.body },
function(err, output){

        if(err)
res.status(500).json({ error: 'database
failure' });

        console.log(output);

        if(!output.n) return
res.status(404).json({ error: 'book not
found' });

        res.json( { message: 'book
updated' } );

    })

});
```

여기서 output 은 mongod 에서 출력하는 결과물입니다.

```
{

    ok: 1,

    nModified: 0,

    n: 1

}
```

여기서 nModified 는 변경한 document 갯수, n 은 select 된 document 갯수입니다.
update() 를 실행하였을 때, 기존 내용이 업데이트 할 내용과 같으면 nModified 는 0 으로 되기 때문에, n 값을 비교하여 성공여부를 판단합니다.


mongodb_tutorial
[PUT] Update Book
PUT
http://hoah.xyz:8080/api/books/56d67ea144b3e9f85462852e
BODY
PARAMS
AUTH
HEADERS
{
"title": "Nodejs Tutorial 2",
"published_date": "2016-3-2"
}

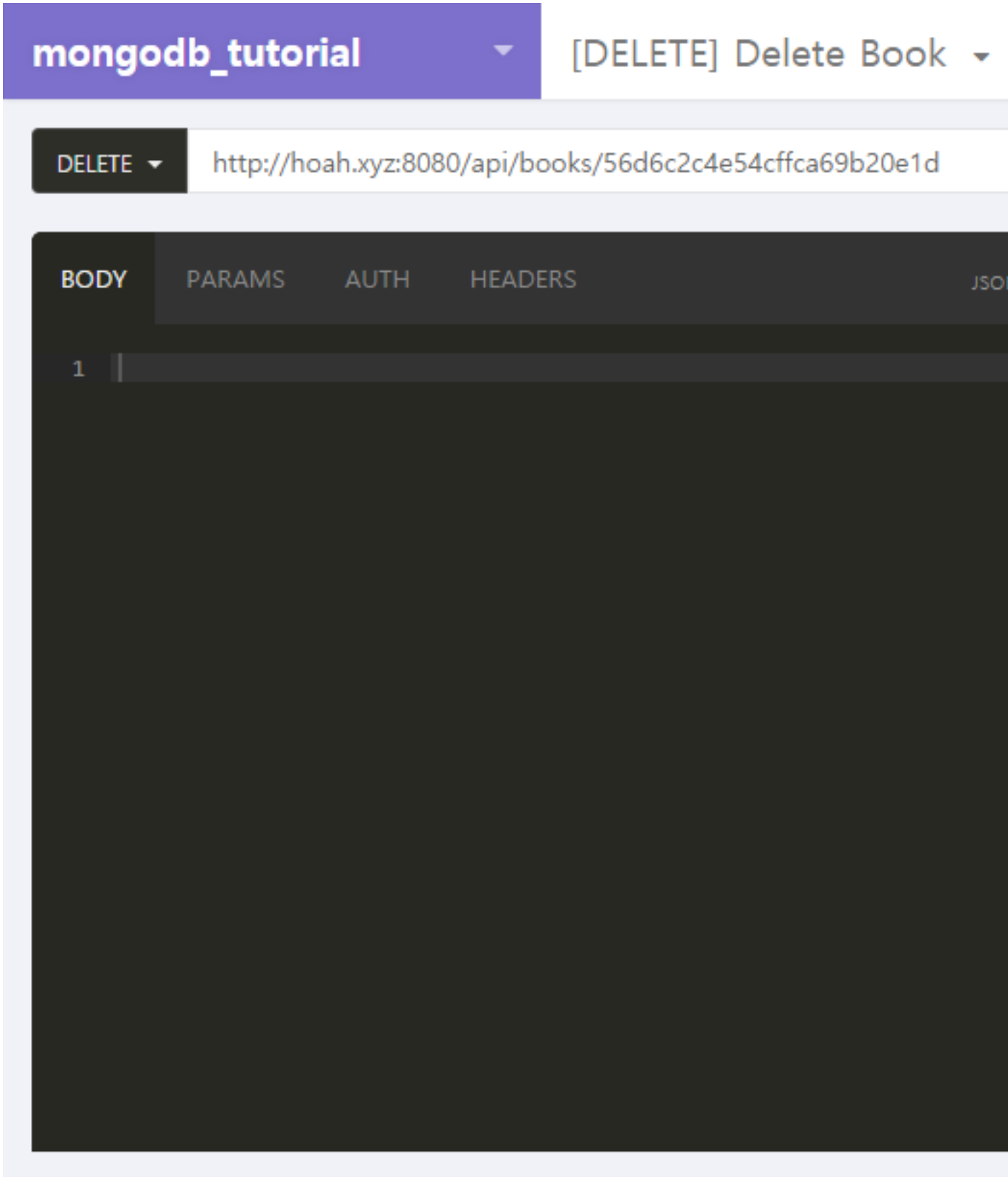
6.4 DELETE (/api/books/:book_id)

book_id 를 찾아서 document 를 제거합니다.

```
// DELETE BOOK

app.delete('/api/books/:book_id',
function(req, res){

    Book.remove({ _id:
req.params.book_id }, function(err,
output){

        if(err) return
res.status(500).json({ error: "database
failure" });



        /* ( SINCE DELETE OPERATION IS
IDEMPOTENT, NO NEED TO SPECIFY )

        if(!output.result.n) return
res.status(404).json({ error: "book not
found" });

        res.json({ message: "book
deleted" });

        */



        res.status(204).end();
```

```
    })

});
```

document 를 제거 할 땐 remove() 메소드가 사용됩니다. DELETE 는 idempotent*(어떤 과정을 몇번이고 반복 수행 하여도 결과가 동일함; 즉 삭제한 데이터를 삭제하더라도, 존재하지 않는 다큐먼트를 제거 시도를하더라도 달라질게 없음)* 하므로, 성공하였을떄나 실패하였을때나 결과값이 같습니다. 여기서 204 HTTP status 는 **No Content** 로서, 요청한 작업을 수행하였고 데이터를 반환 할 필요가 없다는것을 의미합니다.6~9 번줄은 실제로 존재하는 데이터를 삭제하였는지 확인해주는 코드이나, 그럴 필요가 없으므로 주석처리되었습니다.


mongodb_tutorial
[DELETE] Delete Book
DELETE
http://hoah.xyz:8080/api/books/56d6c2c4e54cffca69b20e1d
BODY
PARAMS
AUTH
HEADERS
1

마치면서..

이 강좌에서 사용된 프로젝트는 깃헙 (/velopert/mongoose_tutorial) 에 업로드되어있습니다. 테스트 하고싶으신 분은 참조하세요.

다음 강좌에서는 gulp 를 다뤄보도록하겠습니다.

api / express / mongodb / mongoose / Node.js / restful / tutorial / 강좌

Post navigation

Previous Post

Previous post:

Next Post

Next Post:

디바이스 기반으로 약간 수정한 내용

Server.js

```
var express = require('express');
var app = express();
```

```
var bodyParser = require('body-parser');
var session = require('express-session');
var fs = require("fs")

app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.engine('html', require('ejs').renderFile);


var server = app.listen(3000, function(){
  console.log("Express server has started on port 3000")
});

app.use(express.static('public'));

app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(session({
  secret: '@#@$MYSIGN#@$#$',
  resave: false,
  saveUninitialized: true
}));


var router = require('./router/main')(app, fs);
```

main.js

```
module.exports = function(app, fs)
{

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

```
            res.render('index', {
                  title: "MY HOMEPAGE",
                  length: 5
            })
      });

       app.get('/currentTime', function(req,res) {
            res.send(new Date().toISOString());
      });

     app.get('/list', function (req, res) {
          fs.readFile( __dirname + "/../data/" + "device.json", 'utf8', function
(err, data) {
               console.log( data );
               res.end( data );
          });
     })

     app.get('/getDevice/:devicename', function(req, res){
          fs.readFile( __dirname + "/../data/device.json", 'utf8', function (err,
data) {
                var devices = JSON.parse(data);
                res.json(devices[req.params.devicename]);
          });
     });

     app.post('/addDevice/:devicename', function(req, res){

          var result = {    };
          var devicename = req.params.devicename;

          // CHECK REQ VALIDITY
          if(!req.body["numbers"] || !req.body["name"]){
```

```
                result["success"] = 0;
                result["error"] = "invalid request";
                res.json(result);
                return;
            }

            // LOAD DATA & CHECK DUPLICATION
            fs.readFile( __dirname + "/../data/device.json", 'utf8',    function(err,
data){
                var devices = JSON.parse(data);
                if(devices[devicename]){
                    // DUPLICATION FOUND
                    result["success"] = 0;
                    result["error"] = "duplicate";
                    res.json(result);
                    return;
                }

                // ADD TO DATA
                devices[devicename] = req.body;

                // SAVE DATA
                fs.writeFile(__dirname + "/../data/device.json",
                            JSON.stringify(devices, null, '\t'), "utf8",
function(err, data){
                    result = {"success": 1};
                    res.json(result);
                })
            })
        });

        app.put('/updateDevice/:devicename', function(req, res){
```

```
        var result = {    };
        var devicename = req.params.devicename;

        // CHECK REQ VALIDITY
        if(!req.body["numbers"] || !req.body["name"]){
            result["success"] = 0;
            result["error"] = "invalid request";
            res.json(result);
            return;
        }

        // LOAD DATA
        fs.readFile( __dirname + "/../data/device.json", 'utf8',    function(err,
data){
            var devices = JSON.parse(data);
            // ADD/MODIFY DATA
            devices[devicename] = req.body;

            // SAVE DATA
            fs.writeFile(__dirname + "/../data/device.json",
                        JSON.stringify(devices, null, '\t'), "utf8",
function(err, data){
                result = {"success": 1};
                res.json(result);
            })
        })
    });


    app.delete('/deleteDevice/:devicename', function(req, res){
        var result = { };
        //LOAD DATA
        fs.readFile(__dirname + "/../data/device.json", "utf8", function(err,
data){
```

```
                var devices = JSON.parse(data);

                // IF NOT FOUND
                if(!devices[req.params.devicename]){
                        result["success"] = 0;
                        result["error"] = "not found";
                        res.json(result);
                        return;
                }

                // DELETE FROM DATA
                delete devices[req.params.devicename];

                // SAVE FILE
                fs.writeFile(__dirname + "/../data/device.json",
                                JSON.stringify(devices, null, '\t'), "utf8",
function(err, data){
                        result["success"] = 1;
                        res.json(result);
                        return;
                })
        })

    })

}
```