Web Development: Module 2, Lesson 5  
REST Endpoints with Mocked Data Hands-On Lab

## Overview

In this lab, we'll implement in-memory (mocked data) RESTful API backend for a blog.

## Objectives

In this hands-on lab you will learn how to:

* Create a js file and implement Node RESTful API server with CRUD using Express
* Use mocked data from the in-memory store, i.e., array hard-coded in the ([code/lesson5/lab/posts.json](https://github.com/MSFTImagine/computerscience/tree/master/Complimentary%20Course%20Content/Module2/code/lesson5/lab2))

## Prerequisites

The following are required to complete this hands-on lab:

* A text editor
* Windows PowerShell, Mac Terminal, or some other shell with node.js and npm installed
* You should have completed [Module 2 Lessons 1, 2, 3, and 4](https://github.com/MSFTImagine/computerscience/tree/master/Complimentary%20Course%20Content/Module2/Lessons) as well as the [Module 2 Lesson 2 Lab](https://github.com/MSFTImagine/computerscience/tree/master/Complimentary%20Course%20Content/Module2/Labs).

## Exercises

This hands-on lab includes the following exercises:

* Exercise 1: Node RESTful API server

## Exercise 1: Node RESTful API server

In this exercise, you will write a Node RESTful API server with CRUD using Express. Be sure to refer to [Module 2 Lesson 5](https://github.com/MSFTImagine/computerscience/tree/master/Complimentary%20Course%20Content/Module2/Lessons) throughout this exercise.

NOTE: The following resources should prove helpful.

* <http://expressjs.com/en/api.html>
* <http://webapplog.com/express-js-4-node-js-and-mongodb-rest-api-tutorial/>
* <http://chaijs.com/api/bdd/>

To accomplish this exercise, we will be following these steps:

1. Create app.js file and implement GET, POST, PUT and DELETE for posts resource
2. Install dependencies with npm i
3. Test the server

Execute the following commands to create a new folder and add the default package.json using npm init.

$ mkdir rest-api

$ cd rest-api

$ npm init -y

Import the dependencies. You can either copy the content of package.json from code/lesson5 and run npm i, or execute the following install commands:

$ npm i body-parser@1.9.2 express@4.10.1 morgan@1.5.0 -S

$ npm i expect.js@0.3.1 mocha@2.5.3 superagent@0.20.0 -D

Create the data file posts.json. You can use your favorite code editor and insert the code below, or copy the file from [code/lesson5/lab2/posts.json](https://github.com/MSFTImagine/computerscience/tree/master/Complimentary%20Course%20Content/Module2/code/lesson4/lab2).

[{

"\_id": "0",

"title": "Azat",

"text": "Node is not JavaScript, but both JavaScript and Node are ECMAScript.",

"createdAt": "Sun May 29 2016 19:33:28 GMT-0500 (CDT)"

}]

**Implementing Tests**

To adhere to the principles of Test Driven Development, we are going to implement tests first. You can skip and download the test file but if you want to learn how to do testing in Node.js, follow along. Create a file called app.test.js.It will test the API by making requests to it and validating the responses. Here is a skeleton view of the file you will create, without the code for each individual function.

var superagent = require('superagent')

var expect = require('expect.js')

var app = require('./app.js')

let baseUrl = 'http://localhost:3007/api'

before(function(){

// Prepare for the tests (optional)

})

describe('express rest api server', function(){

var id // ID shared between multiple it statements

it('posts an object', function(done){

// Create a new post

})

it('retrieves an object', function(done){

// Fetch the post

})

it('retrieves a collection', function(done){

// Fetch the list of posts

})

it('updates an object', function(done){

// Update the post

})

it('checks an updated object', function(done){

// Check that the updated post has new values

})

it('removes an object', function(done){

// Remove the post

})

it('checks an removed object', function(done){

// Check that the post is no longer present

})

})

after(function(){

// Clean up (Optional)

})

To implement the tests, add code to import dependencies.

* **superagent** is used to make HTTP requests
* **expect** is used for BDD-style assertions
* **app.js** is the server app code we will create.

var superagent = require('superagent')

var expect = require('expect.js')

var app = require('./app.js')

let baseUrl = 'http://localhost:3007/api'

Since we have the Express web app code in app, we can invoke listen to boot up the server. Let's use port 3007 for testing to avoid conflicts with 3000 port which we will be using for the actual app.

Add the following code to your app.test.js file

before(function(){

app.listen(3007)

})

Next is the actual test suit. The convention is to use a noun as the name so this one will say express rest api server.

Add the following code to your app.test.js file.

describe('express rest api server', function(){

var id

We define the first test case with a behavior, i.e., server posts an object. We use superagent to make a POST request to the http://localhost:3007/posts. The URL is constructed using ES6+/ES2015+ string template. The data being sent has fields author and text. When the response comes back, we check the body of the response contains an ID and that the ID is a number. We store the ID for the subsequent test cases.

Add the following code to your app.test.js file.

it('posts an object', function(done){

superagent.post(`${baseUrl}/posts`)

.send({ author: 'John',

text: `There's a better alternative to the ubiquitous JSON as the communication protocol of the web. It's Protocol Buffers (protobuf). In a nutshell, protobuf offers a more dense format (faster processing) and provides data schemas (enforcement of structure and better compatibility with old code). `

})

.end(function(e, res){

// console.log(res.body)

expect(e).to.eql(null)

expect(res.body).to.be.number

expect(res.body.id).to.be.number

id = res.body.id

done()

})

})

Next, we can test the ability of the server to fetch a single post. We already have the ID of our newly created post, so we don't have to rely on any data in the database.

In this test, we fetch the posts and compare the ID with our ID from the variable id. We also compare the field author to be John. The URL has the post ID in it.

Add the following code to your app.test.js file.

it('retrieves an object', function(done){

superagent.get(`${baseUrl}/posts/`+id)

.end(function(e, res){

// console.log(res.body)

expect(e).to.eql(null)

expect(typeof res.body).to.eql('object')

expect(res.body.id).to.be.number

expect(res.body.id).to.eql(id)

expect(res.body.author).to.eql('John')

done()

})

})

The next API to test is the fetching of a list of posts. The URL is the same for this test as it is for the creation of a new post, i.e., http://localhost:3007/posts. In the callback of the response (end()), we check that the array of the returned posts contains our newly created post. We do so with the map and to.contain(id) methods.

Add the following code to your app.test.js file.

it('retrieves a collection', function(done){

superagent.get(`${baseUrl}/posts`)

.end(function(e, res){

// console.log(res.body)

expect(e).to.eql(null)

expect(res.body.length).to.be.above(0)

expect(res.body.map(function (item){return item.id})).to.contain(id)

done()

})

})

We haven't tested the update yet. We can do so by making a PUT request to a URL /posts/:id. The new author name is Peter. Let's check for a property msg to be success in the response.

Add the following code to your app.test.js file.

it('updates an object', function(done){

superagent.put(`${baseUrl}/posts/`+id)

.send({author: 'Peter', id: id})

.end(function(e, res){

// console.log(res.body)

expect(e).to.eql(null)

expect(typeof res.body).to.eql('object')

expect(res.body.msg).to.eql('success')

done()

})

})

As soon as we updated the record, we should test that the change is there. We shouldn't rely only on the response from the PUT, but make a new request GET /posts/:id.

Add the following code to your app.test.js file

it('checks an updated object', function(done){

superagent.get(`${baseUrl}/posts/`+id)

.end(function(e, res){

// console.log(res.body)

expect(e).to.eql(null)

expect(typeof res.body).to.eql('object')

expect(res.body.id).to.eql(id)

expect(res.body.author).to.eql('Peter')

done()

})

})

Assuming the data has changed, we have just one last test: Delete. Use the DELETE HTTP method and the URL with ID. In the response, we check for msg to be success.

Add the following code to your app.test.js file

it('removes an object', function(done){

superagent.del(`${baseUrl}/posts/`+id)

.end(function(e, res){

// console.log(res.body)

expect(e).to.eql(null)

expect(typeof res.body).to.eql('object')

expect(res.body.msg).to.eql('success')

done()

})

})

Just checking for success in the DELETE response is not robust. It's better to test the database directly or try to retrieve the deleted post. We should NOT get this deleted post ID in the results.

Add the following code to your app.test.js file

it('checks an removed object', function(done){

superagent.get(`${baseUrl}/posts/`)

.end(function(e, res){

// console.log(res.body)

expect(e).to.eql(null)

expect(res.body.map(function (item){return item.id})).to.not.be(id)

done()

})

})

})

Finally, we exit the process and the testing in after. The after function is where we do any database or other test clean up as needed.

Add the following code to your app.test.js file

after(function(){

process.exit()

})

Congratulations, you implemented the tests. Remember, there is a copy of the completed file app.test.js if you did not complete all the steps. You will need it to test your API which we implement next. We run this test using the mocha test.js command.

Install mocha globally or locally with the following commands. Run one of the following two commands (we recommend using the second command to install locally and avoid conflicts)

Install globally

$ npm i –g mocha

Install locally

$ npm i mocha

If you copied the package.json or run npm i mocha -D, then you have mocha installed locally already. All you need to do is to run the following command.

$ ./node\_modules/mocha/bin/mocha app.test.js

The same command can be saved as an npm script and run with npm test. In either case, when you run the tests, they should FAIL with either Error: Cannot find module './app.js' or some other error (if you create app.js already) because we have not implemented app.js.

**Implementing API**

Create the file app.js. This is a high-level overview to give you a sense of where are we going in terms of the implementation. All the routes are in one file. Eventually, you'll want to abstract (modularize) them into separate files, but for this rather small API we will keep them all in one file.

The middleware and the bootup could also be separated into their own files for better code organization. Here is a skeleton view of the file you will create, without the code for each individual function.

var express = require('express'),

bodyParser = require('body-parser')

logger = require('morgan')

let posts = require('./posts.json')

var app = express()

app.use(bodyParser.json())

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

app.use(logger('dev'))

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

// Serve an info message

})

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

// Fetch a list of posts and send it to the client

})

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

// Create a new post in the database and send its ID to the client

})

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

// Fetch an individual post by ID and send back the post data as JSON

})

app.put('/api/posts/:id', function(req, res, next) {

// Partially update the post by ID and return success if everything went fine

})

app.delete('/api/posts/:id', function(req, res, next) {

// Remove the post by ID from the database and return success if everything went fine

})

if (require.main === module) { // Is this run as a standalone program or a module

app.listen(3000, function(){ // Boot up the server if standalone program

console.log('Express server listening on port 3000')

})

} else { // Export the code if run as a module

module.exports = app

}

Now let's create the file.

First, import the required modules. **body-parser** and **morgan** are Express plugins (a.k.a. middleware) which we use for parsing of the incoming payloads and logging of the requests respectively.

Add the following code to the app.js file

var express = require('express'),

bodyParser = require('body-parser')

logger = require('morgan')

To populate our blog with some posts we will read data from a JSON file. You can use fs.readFile or fs.readFileSync, but a better and easier way is to use require which allows to load JSON files.

Add the following code to the app.js file

let posts = require(‘./posts.json’)

Next, we create an instance of the Express app and apply middleware to parse the incoming requests as well as log requests information in terminal. The middleware must be applied with app.use() before any routes to take advantage of the functionality provided in the middleware. Otherwise, routes won’t get payload if they precede the middleware.

Add the following code to the app.js file

var app = express()

app.use(bodyParser.json())

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

app.use(logger(‘dev’))

Implement the GET routes for the home page and for the /posts.

Add the following code to the app.js file

app.get(‘/’, function(req, res, next) {

res.send(‘please select a collection, e.g., /posts’)

})

app.get(‘/api/posts’, function(req, res, next) {

let results = posts

res.send(results)

})

Implement a POST /api/posts which will create a new post in our “database” (i.e., array posts). The ID will be the index of the element so we can use length and then push because length will give the last index plus 1 and push will create a new element with that index:

Add the following code to the app.js file

app.post(‘/api/posts’, function(req, res, next) {

let post = req.body

post.id = posts.length

posts.push(post)

res.send(post)

})

We want to fetch a single element and we can do so by using :id in the URL pattern. The value will be accessible from req.params.id:

Add the following code to the app.js file

app.get(‘/api/posts/:id’, function(req, res, next) {

let result = posts.find((post)=>{

return post.id == req.params.id

})

res.send(result)

})

We need to update a post (CRUD- create, read, update, delete). If we cannot find a post which index value equals the value from the request and URL (req.body.id and req.params.id), then we should continue with an error by invoking next(new Error()). Otherwise, we update the post using the index and send a success message.

Add the following code to the app.js file

app.put(‘/api/posts/:id’, function(req, res, next) {

let postIndex = posts.findIndex((post)=>{

return (post.id == req.params.id && post.id == req.body.id)

})

if (postIndex == -1) return next(new Error(‘Post not found’))

posts[postIndex] = req.body

res.send({msg:’success’})

})

Implement the route to remove the post from a collection (list of posts). To do this, we need to extract the index with postIndex() function and use splice method on the array. We send back success if we deleted 1 item from the array.

Add the following code to the app.js file

app.delete(‘/api/posts/:id’, function(req, res, next) {

let postIndex = posts.findIndex((post)=>{

return (post.id == req.params.id && post.id == req.body.id)

})

let result = posts.splice(postIndex, 1)

res.send((result.length == 1)?{msg: ‘success’} : {msg: ‘error’})

})

Add the bootup and export section. Require.main is a global property which will be equal to module if this is a standalone program.

Add the following code to the app.js file

if (require.main === module) {

app.listen(3000, function(){

console.log(‘Express server listening on port 3000’)

})

} else {

module.exports = app

}

You can try running this server with node app.js, but I recommend running tests first. To test your app run npm test. You know the server is working and all the endpoints are working when tests print this output:

express rest api server

POST /api/posts 200 8.967 ms - 326

✓ posts an object (70ms)

GET /api/posts/1 200 5.737 ms - 326

✓ retrieves an object

GET /api/posts 200 1.848 ms - 487

✓ retrieves a collection

PUT /api/posts/1 200 0.778 ms - 17

✓ updates an object

GET /api/posts/1 200 0.346 ms - 25

✓ checks an updated object

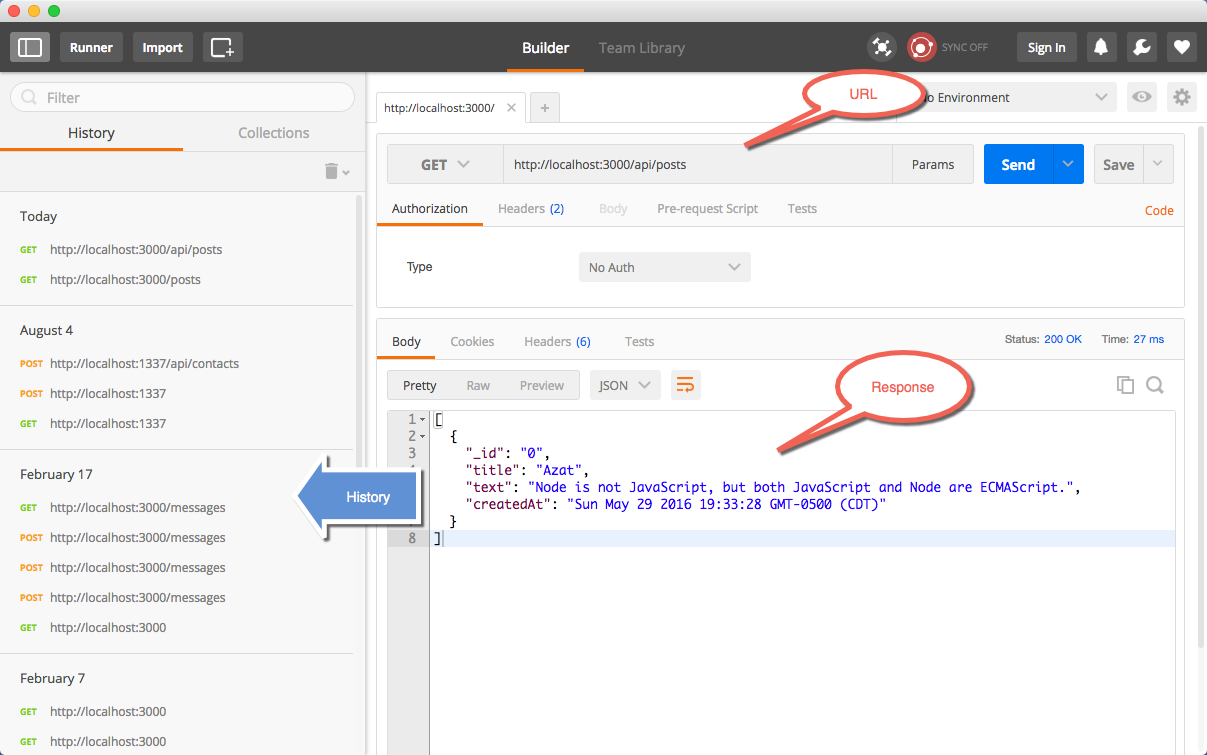
DELETE /api/posts/1 200 0.626 ms - 17

✓ removes an object

GET /api/posts/ 200 0.926 ms - 160

✓ checks an removed object

Another way to test the rest API is to do so manually by using Postman or CURL. Postman is a Chrome app and has a nice user-friendly GUI as shown in the capture below. It is available for [Windows](https://www.getpostman.com/app/download/win64) and [Mac](https://www.getpostman.com/app/download/osx64). Just remember to start your server first with node app, or you'll get the error: "*Could not get any response*"



…

## Summary

In this hands-on lab, you learned how to:

* Implement basic RESTful API server
* Use Express