Day 27: Restify–Build Correct REST Web Services in Node.js

NOVEMBER 24, 2013

BY [SHEKHAR GULATI](https://blog.openshift.com/author/shekhargulati/)

[Tweet](http://twitter.com/share)

Share[**4**](https://blog.openshift.com/day-27-restify-build-correct-rest-web-services-in-nodejs/)

Today for [my 30 day challenge](https://blog.openshift.com/learning-30-technologies-in-30-days-a-developer-challenge) I’m learning a Node.js module called [restify](http://mcavage.me/node-restify/). This module makes it easy to write correct RESTful APIs in Node.js and provides out-of-the-box support for features like versioning, error handling, CORS, and content negotiation. Restify intentionally borrows from [Express](http://expressjs.com/) as it’s the de facto API for writing web applications on top of node.js. We will develop a RESTful API for storing jobs and store the data in [MongoDB](http://www.mongodb.org/).

Restify Prerequisites

Restify requires NodeJS and the NPM package manager which comes by default with node.js installations. Download the latest version of NodeJS from [official website](http://nodejs.org/). Once you have node.js and NPM installed, we will use the NPM system to install Harp.

This application uses MongoDB as data storage choice. [Download the latest MongoDB release](http://www.mongodb.org/downloads) for your operation system.

Install Restify

Create a new directory at any convenient directory on your file system.

$ mkdir myapp

$ cd myapp

To install restify module issue this command:

$ npm install restify

We will use MongoJS as the MongoDB driver. Install the mongojs module:

$ npm install mongojs

Writing RESTful API

Now we have restify and mongojs installed, let’s write code. Create a new file called app.js file.

$ touch app.js

Copy and paste the following content to app.js.

var restify = require('restify');

var mongojs = require("mongojs");

The two lines show above load the restify and mongojs modules using the require function and assign them to variables.

Now create a new server using restify API:

var restify = require('restify');

var mongojs = require("mongojs");

var ip\_addr = '127.0.0.1';

var port = '8080';

var server = restify.createServer({

name : "myapp"

});

server.listen(port ,ip\_addr, function(){

console.log('%s listening at %s ', server.name , server.url);

});

The code shown above creates a new server. The createServer() function takes an options object. We passed ***myapp*** as the name of the server in options object. You can view the full list of options in the [documentation](http://mcavage.me/node-restify/#Creating-a-Server). After create the server instance, we call the listen function passing port, ip address, and a callback function.

Run the application by typing the following command.

$ node app.js

You will see following on the command line terminal.

myapp listening at http://127.0.0.1:8080

Configure Plugins

The restify module has a lot of built in plugins which we can use. Copy and paste the following in app.js. These should be added before the server.listen() function. Refer to [documentation](http://mcavage.me/node-restify/#Bundled-Plugins) to learn about all the supported plugins.

server.use(restify.queryParser());

server.use(restify.bodyParser());

server.use(restify.CORS());

The three lines shown above :

1. The restify.queryParser() plugin is used to parse the HTTP query string (i.e., /jobs?skills=java,mysql). The parsed content will always be available in req.query.
2. The restify.bodyParser() takes care of turning your request data into a JavaScript object on the server automatically.
3. The restify.CORS() configures [CORS](http://en.wikipedia.org/wiki/Cross-origin_resource_sharing) support in the application.

Configure MongoDB

Before adding the routes, let’s add code to connect to ***myapp*** to the MongoDB database.

var connection\_string = '127.0.0.1:27017/myapp';

var db = mongojs(connection\_string, ['myapp']);

var jobs = db.collection("jobs");

In the code shown above, we connect to local MongoDB instance. Next, we get the jobs collection using database object.

Writing CRUD API

Now, we have the server and database part ready. We still need routes to define the behaviour of the API. Copy and paste the following code to app.js.

var PATH = '/jobs'

server.get({path : PATH , version : '0.0.1'} , findAllJobs);

server.get({path : PATH +'/:jobId' , version : '0.0.1'} , findJob);

server.post({path : PATH , version: '0.0.1'} ,postNewJob);

server.del({path : PATH +'/:jobId' , version: '0.0.1'} ,deleteJob);

The code shown above does the following:

1. When a user makes a GET request to ‘/jobs’, then findAllJobs callback will be invoked. The another interesting part is the use of versioned routes. A client can specify the version using Accept-Versionheader.
2. When a user makes a GET request to ‘/jobs/123’, then findJob callback will be invoked.
3. When a user makes POST request to ‘/jobs’, then postNewJob callback will be invoked.
4. When a user makes DELETE request to ‘/jobs/123’, then postNewJob callback will be invoked.

Now we will write the callbacks. Copy and paste the following to app.js.

function findAllJobs(req, res , next){

res.setHeader('Access-Control-Allow-Origin','\*');

jobs.find().limit(20).sort({postedOn : -1} , function(err , success){

console.log('Response success '+success);

console.log('Response error '+err);

if(success){

res.send(200 , success);

return next();

}else{

return next(err);

}

});

}

function findJob(req, res , next){

res.setHeader('Access-Control-Allow-Origin','\*');

jobs.findOne({\_id:mongojs.ObjectId(req.params.jobId)} , function(err , success){

console.log('Response success '+success);

console.log('Response error '+err);

if(success){

res.send(200 , success);

return next();

}

return next(err);

})

}

function postNewJob(req , res , next){

var job = {};

job.title = req.params.title;

job.description = req.params.description;

job.location = req.params.location;

job.postedOn = new Date();

res.setHeader('Access-Control-Allow-Origin','\*');

jobs.save(job , function(err , success){

console.log('Response success '+success);

console.log('Response error '+err);

if(success){

res.send(201 , job);

return next();

}else{

return next(err);

}

});

}

function deleteJob(req , res , next){

res.setHeader('Access-Control-Allow-Origin','\*');

jobs.remove({\_id:mongojs.ObjectId(req.params.jobId)} , function(err , success){

console.log('Response success '+success);

console.log('Response error '+err);

if(success){

res.send(204);

return next();

} else{

return next(err);

}

})

}

The code shown above is self explanatory. We are using mongojs API to perform CRUD operations.

We can test the web services using curl. To create a new job, type the command shown below.

$ curl -i -X POST -H "Content-Type: application/json" -d '{"title":"NodeJS Developer Required" , "description":"NodeJS Developer Required" , "location":"Sector 30, Gurgaon, India"}' http://127.0.0.1:8080/jobs

To find all the jobs

$ curl -is http://127.0.0.1:8080/jobs

HTTP/1.1 200 OK

Access-Control-Allow-Origin: \*

Content-Type: application/json

Content-Length: 187

Date: Sun, 24 Nov 2013 16:17:27 GMT

Connection: keep-alive

[{"title":"NodeJS Developer Required","description":"NodeJS Developer Required","location":"Sector 30, Gurgaon, India","postedOn":"2013-11-24T16:16:16.688Z","\_id":"52922650aab6107320000001"}]

Deploy to Cloud

Before we deploy the application to OpenShift, we’ll have to do few setup tasks :

1. Sign up for an [OpenShift Account](https://www.openshift.com/app/account/new). It is completely free, and Red Hat gives every user three free Gears on which to run your applications. At the time of this writing, the combined resources allocated for each user is 1.5 GB of memory and 3 GB of disk space.
2. Install the [rhc client tool](https://openshift.redhat.com/community/get-started#cli) on the machine. The rhc is a ruby gem so you need to have ruby 1.8.7 or above on your machine. To install rhc type: sudo gem install rhc  
   If you already have one, make sure it is the latest one. To update the rhc, execute the command shown below.sudo gem update rhc  
   For additional assistance setting up the rhc command-line tool, see the following page: https://openshift.redhat.com/community/developers/rhc-client-tools-install
3. Setup the OpenShift account using rhc setup command. This command will help us create a namespace and upload your ssh keys to OpenShift server.

After setup, create a new OpenShift application by running the following command:

$ rhc create-app day27demo nodejs-0.10 mongodb-2 --from-code https://github.com/shekhargulati/day27-restify-openshift-demo.git

It will do all the stuff from creating an application, to setting up public DNS, to creating private git repository, and then finally deploying the application using code from my Github repository. The app is running here http://day27demo-{domain-name}.rhcloud.com//

That’s it for today. Keep giving feedback.

**Nodejs Restify MongoDB – Build your own REST API**

3100

In this post, we are going build a Node based REST API for a collection in our application. We will use Restify to handle our server endpoints. Then we use Mongojs to interact with the database. We will be using MongoLab, (*a free\* MongoDB as a Service*) to store our data. To make things a bit more interesting, we will use Runnable to deploy the app and share it with the world.

We will be building a simple product API, where we **C**reate a new product, **R**ead the product, **U**pdate the product and finally **D**elete the product using the traditional REST methods POST, GET, PUT & DELETE.

The final output of this post would be

To get your hands on the above demo, navigate to [myRunnable](http://runnable.com/U1OkqrgIj8pXRQ3p/restifymyapp-restify-and-mongodb-mongolab-mongojs-for-node-js), scroll to the very end of the page & wait for the terminal to show up. Then run

node client.js

to see the above output.

**Contents**

* [What is a RESTful API?](http://thejackalofjavascript.com/nodejs-restify-mongolab-build-rest-api/#rest)
* [What is Restify?](http://thejackalofjavascript.com/nodejs-restify-mongolab-build-rest-api/#restify)
* [What is Mongojs?](http://thejackalofjavascript.com/nodejs-restify-mongolab-build-rest-api/#mongojs)
* [Setup MongoLab account](http://thejackalofjavascript.com/nodejs-restify-mongolab-build-rest-api/#mongoLab)
* [Continue Development](http://thejackalofjavascript.com/nodejs-restify-mongolab-build-rest-api/#dev)
  + [GET – All Products](http://thejackalofjavascript.com/nodejs-restify-mongolab-build-rest-api/#get)
  + [POST – Add New Product](http://thejackalofjavascript.com/nodejs-restify-mongolab-build-rest-api/#post)
  + [PUT – Update an Existing Product](http://thejackalofjavascript.com/nodejs-restify-mongolab-build-rest-api/#put)
  + [DELETE – Delete a Product](http://thejackalofjavascript.com/nodejs-restify-mongolab-build-rest-api/#delete)
  + [GET – Fetch One Product](http://thejackalofjavascript.com/nodejs-restify-mongolab-build-rest-api/#getOne)
* [Super Asynchronous Client](http://thejackalofjavascript.com/nodejs-restify-mongolab-build-rest-api/#async)
* [Deploy to Runnable](http://thejackalofjavascript.com/nodejs-restify-mongolab-build-rest-api/#runnable)

You can find the complete code [here](https://github.com/arvindr21/restify-mongojs-app). Now, Let’s see how we can build the REST API.

**What is a RESTful API?**

*REST* is the underlying architectural principle of the web. The amazing thing about the web is the fact that clients (browsers) and servers can interact in complex ways without the client knowing anything beforehand about the server and the resources it hosts. The key constraint is that the server and client must both agree on the *media* used, which in the case of the web is *HTML*.

cont. on [SO](http://stackoverflow.com/a/671132/1015046)..

You can find another good tutorial on REST [here](http://www.restapitutorial.com/).

We are going to use Node as our server to host our application locally. You can read more above node [here](http://thejackalofjavascript.com/nodejs/) & you can set up node from [here](http://thejackalofjavascript.com/hello-node/).

Create a new folder named *myRESTAPI.*Open terminal/prompt here and lets create a new node project. Run

npm init

and fill it with *your* details

Next, we will create 2 new files inside the *myRESTAPI* folder. Name them *server.js* and *client.js*.

Now, we will install the project dependencies. The first dependency is Restify.

**What is Restify?**

Restify is a node.js module built specifically to enable you to build correct REST web services. It intentionally borrows heavily from express as that is more or less the de facto API for writing web applications on top of node.js.

Restify takes most of its functionality from Expressjs.

**Expressjs – Non-Restful stuff = Restify**

You can know more about Restify [here](http://mcavage.me/node-restify/).

Our second dependency is Mongojs.

**What is Mongojs?**

Mongojs is a node.js module for mongodb, that emulates the official mongodb API as much as possible. It wraps mongodb-native and is available through npm

If you are new to MongoDB, you can download the latest copy of MongoDB from [here](http://www.mongodb.org/downloads) to play around. But for this post, we are going to use MongoLab to manage our data.

If you are a beginner, MongoDB is a NoSQL database. Unlike MySQL or MSSQL or Oracle DBs, here database have collections instead of tables. We have documents in collections insteads of rows in a table. And best of all, all the documents are stored as JSON. You can know more about MongoDB [here](http://try.mongodb.org/).

If you have never worked with MongoDB before, you can remember the following commands to navigate around and perform basic operation

| **Command** | **Result** |
| --- | --- |
| mongod | will start the MongoDB service |
| mongo | will step you inside the MongoDB shell (*when run in a new tab, while Mongod is running*) |
| show dbs | will show the list of databases |
| use <<database name>> | will step you inside the database |
| show collections | will show the list of collections once you are inside the database |
| db.collectionName.find() | will show all the documents in that collection |
| db.collectionName.findOne() | will show the first document |
| db.collectionName.find().pretty() | will pretty print the JSON data in console |
| db.collectionName.insert({key :value}) | will insert a new record |
| db.collectionName.update({condition: value}, {$set : {key:value}},{upsert : true}) | will update a record with the given condition & sets the required value. If upsert is true a new document will be created if no documents with matching condition are found |
| db.collectionName.remove({}) | will remove all the documents in that collection |
| db.collectionName.remove({key :value}) | will remove the documents matching the condition |

You can learn more about MongoDB [here](http://docs.mongodb.org/manual/). And Mongojs [here](https://github.com/mafintosh/mongojs).

Now, we are well acquainted with our components. Lets add the dependencies to our project. Back to terminal/prompt run

npm i restify --save-dev

npm i mongojs --save-dev

This will install both our dependencies.

**Setup MongoLab account**

Navigate to [MongoLab](https://mongolab.com/signup/) and sign up for a new account if you don’t have one, else login. Next, create a new database then fill the details as

You can give a database name you like and click on create.

Then to access the database, we need to create a new user to access the database. From the home page, click on the database name and click on the user’s tab. Then create a new user. Remember the username and password.

To access the data remotely, we will build a connection string. On the database homepage, you will see a section on top of the page,

Fill in the placeholders in the above URL and you are done!! We will use this URL in just a moment.

**Continue Development**

Open server.js in your favorite editor. To get started with our project, we will first add the required dependencies.

server.js



|  |  |
| --- | --- |
| 1  2 | var restify = require('restify');  var mongojs = require('mongojs'); |

Next, let’s create a new server and start it. Add the below lines to server.js

server.js



|  |  |
| --- | --- |
| 1  2  3  4  5 | var server = restify.createServer();    server.use(restify.acceptParser(server.acceptable));  server.use(restify.queryParser());  server.use(restify.bodyParser()); |

Finally, ask it start listening to any free port

server.js



|  |  |
| --- | --- |
| 1  2  3 | server.listen(3000, function () {      console.log("Server started @ 3000");  }); |

Then back to prompt/terminal run

node server.js

And you should see the server started message. If you want you can open a browser and navigate to  http://localhost:3000 to see a message like

{"code":"ResourceNotFound","message":"/ does not exist"}

Because we have not yet created any end points.

**GET – All Products**

Lets create a new GET endpoint. Add the below code to our server.js

GET - server.js



|  |  |
| --- | --- |
| 1  2  3  4 | server.get("/products", function (req, res, next) {      res.send("You will see all the products in the colection with this end point");      return next();  }); |

server.get()  method takes 2 arguments. One the endpoint to which it needs to listen to and the second argument is the callback function, which will be called when that endpoint is hit.

Restify server methods, like any Express middleware, provides 3 arguments to its callback method.

**req** – The request object, which contains all the data that comes from the client

**res** – The Response object, which contains all the data that the server sends to the client

**next** – The next method, which is used to invoke the next middleware method in queue.

If you want to write back anything to the client, you need to use  res.send();. Since we are dispatching a REST response, its our duty to dispatch the HTTP response code as well as the content header.

Back to terminal, kill the running node app by pressing **ctrl + c** and the restart it by running  node server.js . You may need to do this again and again when you make changes to the server.js. If you are *lazy* like me, you can install *nodemon* a node package for restarting the node app automatically when changes are made.

| **Windows** | **Mac/\*nix** |
| --- | --- |
| npm i -g nodemon | sudo npm i -g nodemon |

Now, run  nodemon server.js and then navigate to  http://localhost:3000/products, to see the newly updated message.

Now, lets add the db config to our server.js. Update the below URL with your db config.

var db = mongojs('mongodb://admin:admin123@ds053718.mongolab.com:53718/restifymyapp', ['products']);

If you are planning to user your local db instance, your connection string would be

var db = mongojs('productsdb', ['products']);

Where, *productdb* is the name of the database & *products* is the name of the collection.

And the updated server.js would be

server.js



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | var restify = require('restify');  var mongojs = require('mongojs');    var db = mongojs('mongodb://admin:admin123@ds053718.mongolab.com:53718/restifymyapp', ['products']);    // Server  var server = restify.createServer();    server.use(restify.acceptParser(server.acceptable));  server.use(restify.queryParser());  server.use(restify.bodyParser());    server.listen(3000, function () {      console.log("Server started @ 3000");  });    server.get("/products", function (req, res, next) {  res.writeHead(200, {          'Content-Type': 'application/json; charset=utf-8'      });      res.send("You will see all the products in the colection with this end point");      return next();  }); |

Now, instead of spitting out a static message, lets hit the DB and dispatch a list of products. Our updated get method would be

get all products



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | server.get("/products", function (req, res, next) {      db.products.find(function (err, products) {          res.writeHead(200, {              'Content-Type': 'application/json; charset=utf-8'          });          res.end(JSON.stringify(products));      });      return next();  }); |

You can navigate back to the browser and refresh the */products* endpoint. And you should see an empty array, since we do not have any products yet.

**POST – Add New Product**

Now, lets add a POST method so that we can save a product. You can add the below POST method to server.js

serve.js Save a product



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | server.post('/product', function (req, res, next) {      var product = req.params;      db.products.save(product,          function (err, data) {              res.writeHead(200, {                  'Content-Type': 'application/json; charset=utf-8'              });              res.end(JSON.stringify(data));          });      return next();  }); |

As you can see, we are getting the *product* object from the *req.params*. This is sent by our client. You can use chrome extensions like [Postman](http://www.getpostman.com/) or [Advanced REST client](https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddffdnphfgcellkdfbfbjeloo) to perform the REST operations.

But, to get a complete understanding of Restify, we will leverage Restify’s JSONClient. Open client.js and add the below code.

client.js



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | var restify = require('restify');  var server = require('./server');    var client = restify.createJsonClient({      url: 'http://localhost:3000'  });    // a static product to CREATE READ UPDATE DELETE    var testProduct = {      id: "1",      name: "Apple iPad AIR",      os: "iOS 7, upgradable to iOS 7.1",      chipset: "Apple A7",      cpu: "Dual-core 1.3 GHz Cyclone (ARM v8-based)",      gpu: "PowerVR G6430 (quad-core graphics)",      sensors: "Accelerometer, gyro, compass",      colors: "Space Gray, Silver"  };    client.post('/product', testProduct, function (err, req, res, product) {      if (err) {          console.log("An error ocurred >>>>>>");          console.log(err);      } else {          console.log('Product saved >>>>>>>');          console.log(product);      }  }); |

Do notice Line no 2. Here we are referring to our server.js file. The reason behind this is, if we need to run the server and client, we need to run in 2 terminals, one for server and one for client. Instead, I am invoking the server.js from client.js so that the server starts and then using client.js, we will access the server.

Lines 21 to 29, the response is node style. i.e. they are all *async*. IMO, this is good for performance but bad for development. The callback will be fired when the server responds with either an error or the actual response.

Next, add the below line of code to server.js at the very end

module.exports = server;

This is a node way of exporting stuff to be used in other files. You can learn more [here](http://thejackalofjavascript.com/node-js-design-patterns/).

Now, back to terminal/prompt, run

nodemon client.js

And you should see something like

Our product has been inserted. *yay*!! Now, lets add a client method to fetch all the products

client.js fetch all products



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | client.get('/products', function (err, req, res, products) {  if (err) {      console.log("An error ocurred >>>>>>");      console.log(err);  } else {      console.log("Total products " + products.length);      console.log('All products >>>>>>>');      console.log(products);  }  }); |

*PS : If you save the file, remember that the POST will run again a new record will get saved and now we will see 2 products.*If you dont want this to happen, you can comment out the post part on the client.

**PUT – Update an Existing Product**

The PUT method is for updating an existing product, whereas POST is for saving a new product. The only key difference is that in PUT we know the object we are dealing with, whereas in POST we don’t. So, if you are ever stuck in making a decision between PUT & POST, all you need to do is answer the above question.

Update a Product



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | server.put('/product/:id', function (req, res, next) {      // get the existing product      db.products.findOne({          id: req.params.id      }, function (err, data) {          // merge req.params/product with the server/product            var updProd = {}; // updated products          // logic similar to jQuery.extend(); to merge 2 objects.          for (var n in data) {              updProd[n] = data[n];          }          for (var n in req.params) {              updProd[n] = req.params[n];          }          db.products.update({              id: req.params.id          }, updProd, {              multi: false          }, function (err, data) {              res.writeHead(200, {                  'Content-Type': 'application/json; charset=utf-8'              });              res.end(JSON.stringify(data));          });      });      return next();  }); |

In PUT, we will be updating only the changed values, But since we are not aware of the changes on the server side, I am merging the existing object with the one I get from client. And I override the product on the server with the one from client.

I know, its very convoluted. If you know a better method, please do comment.

And our client to test the above code would be

PUT client.js



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | testProduct.price = "1000 USD",  client.put('/product/' + testProduct.id, testProduct, function (err, req, res, status) {      if (err) {          console.log("An error ocurred >>>>>>");          console.log(err);      } else {          console.log('Product saved >>>>>>>');          console.log(status);      }    }); |

Do notice on Line no 1, we are adding a new property named price before we send the data to the server. This is our test for checking if the update has actually worked.

If you save the files, nodemon will restart the server & client and you should see

**DELETE – Delete a Product**

The Final HTTP method, where we send a Product ID to the server and the server will remove it from the DB using Mongojs API.

The server side code for DELETE method would be

DEL - server.js



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | server.del('/product/:id', function (req, res, next) {      db.products.remove({          id: req.params.id      }, function (err, data) {          res.writeHead(200, {              'Content-Type': 'application/json; charset=utf-8'          });          res.end(JSON.stringify(true));      });      return next();  }); |

And the client code would be



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | client.del('/product/' + testProduct.id, function (err, req, res, status) {      if (err) {          console.log("An error ocurred >>>>>>");          console.log(err);      } else {          console.log('Product deleted >>>>>>>');          console.log(status);      }  }); |

And the output would be

**GET – Fetch One Product**

Finally, we will create another GET method to fetch only one product based on the Product ID. The server side code would be

GET - One Product - Server.js



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | server.get('/product/:id', function (req, res, next) {      db.products.findOne({          id: req.params.id      }, function (err, data) {          res.writeHead(200, {              'Content-Type': 'application/json; charset=utf-8'          });          res.end(JSON.stringify(data));      });      return next();  }); |

And the client would be

GET - One Product - client.js



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | client.get('/product/' + testProduct.id, function (err, req, res, product) {      if (err) {          console.log("An error ocurred >>>>>>");          console.log(err);      } else {          console.log('Product with id ' + product.id + '  >>>>>>>');          console.log(product);      }  }); |

On the client.js, If you have commented out the POST method and ran the delete method, you will not have any records in DB. Uncomment POST, save and you should see

Simple right?

**Super Asynchronous Client**

As you can see that our client is Async in nature. We can’t have 2 methods one after the other and expect to see the output in order of execution.

So, to test all the above server methods sequentially, add the below piece of code to client.js

Async nested client



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | // to see the output when you the run the client, A nested async callback system to do the above steps  console.log("\n>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> \n");  client.get('/products', function (err, req, res, products) {      if (err) console.log("Oops : ", err);      else console.log('Total products : ', products.length);      console.log("\n>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> \n");        client.post('/product', testProduct, function (err, req, res, prod) {          if (err) console.log("Oops : ", err);          else console.log('Inserted product : ', prod);          console.log("\n>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> \n");            client.get('/product/' + testProduct.id, function (err, req, res, prod) {              if (err) console.log("Oops : ", err);              else console.log('Product with ID :' + testProduct.id + ' :: ', prod);              console.log("\n>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> \n");                client.put('/product/' + testProduct.id, {                  price: "999 USD"              }, function (err, req, res, status) {                  if (err) console.log("Oops : ", err);                  else console.log('Product Updated status :', status);                  console.log("\n>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> \n");                    client.get('/product/' + testProduct.id, function (err, req, res, prod) {                      if (err) console.log("Oops : ", err);                      else console.log('Product with ID :' + testProduct.id + ' :: ', prod);                      console.log("\n>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> \n");                        client.del('/product/' + testProduct.id, function (err, req, res, status) {                          if (err) console.log("Oops : ", err);                          else console.log('Product deleted status :', status);                          console.log("\n>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> \n");                          client.get('/products', function (err, req, res, products) {                              if (err) console.log("Oops : ", err);                              else console.log('Total products : ', products.length);                              console.log("\n>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> \n");                          });                        });                  });              });          });      });  }); |

Yeah.. Super Async!! You can see the output in the console. All the methods running one after the other.

You can find the complete code [here](https://github.com/arvindr21/restify-mongojs-app).

**Deploy to Runnable**

I have recently come across [Runnable.com](http://runnable.com/) and they are *amazing*!!. You can create runnables that execute in a certain environment like node and the share the results with your team.

You can create a new account with them, which is free (*duh!*). Then you can create a new runnable or you can upload your code to GitHub and give that link. Runnable will do the rest to make it *runnable*.

I have deployed the above code to runnable [here](http://runnable.com/U1OkqrgIj8pXRQ3p/restifymyapp-restify-and-mongodb-mongolab-mongojs-for-node-js). You can check it out. Also do check out their other environments on Runnable.com too.

Hope you got an understanding of REST API and how you can create your own using Nodejs and Restify.