Table of Contents

**[part 01 – Verify nodejs](#_Toc58221180)** [2](#_Toc58221180)

[**part 02 – Building a Simple nodejs App** 3](#_Toc58221181)

[**part 03 – Introduction to node Packages and Express** 5](#_Toc58221182)

[**part 04 – Routing Basics** 8](#_Toc58221183)

[**part 05 – Installing Nodemon** 11](#_Toc58221184)

[**part 06 – Decomposing Routes** 12](#_Toc58221185)

[**part 07 – Decomposing Controllers** 13](#_Toc58221186)

[**part 08 – Introduction to MongoDB** 16](#_Toc58221187)

[**part 09 – Setting up Mongoose** 18](#_Toc58221188)

[**part 10 – Expanding the Controller Functions to work with Database** 19](#_Toc58221189)

[**part 10 – Expanding the Controller to Delete From Database (optional)** 22](#_Toc58221190)

[**part 11 – Expanding the Controller to Add a new Document to the Database** 23](#_Toc58221191)

[**part 12 – Expanding the Controller to update a Document in the Database** 25](#_Toc58221192)

Day01 Building the APIs – in 12 Steps

## part 01 – Verify nodejs

1. Create a folder called **API**.
2. Open a terminal inside of **API** and run the command **npm init**
3. Follow the prompts and just hit **enter** for each question, this is just to create a package.json file

|  |
| --- |
|  |

1. According to the .json file, node will look for index.js in order to execute the code inside, so use touch to create index.js inside of the **API** folder.
2. Add the following code to execute. This is just to make sure that node is working and it is executing properly.

|  |
| --- |
| **console.log("Hello from Skillsoft!");** |

1. Execute index.js by typing in the command **node index** from the command prompt. It should show “Hello from Skillsoft”. This step confirms that we can move on to other parts.

|  |
| --- |
|  |

**------end of part 01-------**

## part 02 – Building a Simple nodejs App

1. Open index.js inside of a text editor and type the following lines (delete the previous line)s:

|  |
| --- |
| **const http = require('http');**  **const hostname = "localhost";**  **const port = 8000;** |

This code means that we are using the http module of **nodejs**, and we will define the other two parameters that the http service requires.

1. Next we will define a variable to point to the **createServer()** method which will hold a reference to the server

|  |
| --- |
| **const SkillServer = http.createServer();** |

|  |
| --- |
| **A special note on the http.createServer() method**.  The createServer() method returns a web server object, which will listen for requests and then handle those requests by returning responses to the client, which could be a browser.  createServer() takes a function that is called each time a request is made.  Once a request is made and that request gets to the server, it is considered a request object and it is based on an HTTP method or verb. The headers object also exist on that request, but it is a separate object.  There are some requests that need special handling, such as POST and PUT. These need special handlers that can work with the ReadableStream interface. When the incoming data happens to be string, then it is possible to handle this string data as an array.  The response object on the other hand is an instance of the ServerResponse class. It is a WritableStream. To send back a response to the client means dealing with the stream methods such as write() and end(). |

1. The createServer() method takes a function that handles both the request and response objects. Extend the method to include that function as an anonymous function.

|  |
| --- |
| **const SkillServer = http.createServer(function(request, response){**    **});** |

1. This now gives us access to these two objects, so we can interrogate the **request** object for things like parameter values or form values and we can use the **response** object to send data or HTML back to the client. In this case we will only use the response object to send an ok as well as some text to the client

|  |
| --- |
| **const SkillServer = http.createServer(function(request, response){**  **response.writeHead(200, {'Content-Type':'text/plain'});**  **response.write("Hello from Skillsoft");**  **response.end();**  **});** |

1. Finally we can call the listen method and pass it the port and hostname

|  |
| --- |
| **SkillServer.listen(port, hostname);** |

Here is the entire index.js file

|  |
| --- |
| **const http = require('http');**  **const hostname = "localhost";**  **const port = 8000;**  **const SkillServer = http.createServer(function(request, response){**  **response.writeHead(200, {'Content-Type':'text/plain'});**  **response.write("Hello from Skillsoft");**  **response.end();**  **});**  **SkillServer.listen(port, hostname);** |

1. In a browser navigate to <http://localhost:8000> and you should see the message from the **response.write()** method call.

**------end of part 02-------**

## part 03 – Introduction to node Packages and Express

1. Stop the application by typing in CTRL-C in the terminal window. This will allow us to install packages. We would need to do this each time we have a new package to install.
2. Install express by running this command from a terminal window that is pointing to the Day01 directory: **npm install express --save**
3. Open the index.js and replace the first line with this one. You can remove the hostname variable, Express already knows its localhost.

|  |
| --- |
| **const express= require('express');**  **const port = 8000;** |

1. Create a new variable and point it to the constructor of express

|  |
| --- |
| **const express= require('express'); const bodyParser = require('body-parser');**  **const app = express();** |

1. Now we can use the app object to handle get and post requests, so simple APIs:

|  |
| --- |
| **const express = require('express');**  **const port = 8000;**  **const app = express();**  **app.get('/', (request, response)=> response.send('hello from skillsoft'));** |

1. The last 3 lines in this file will be a listener, replace the one we had before, this one use the app object

|  |
| --- |
| **app.listen(port, function(){**  **console.log("Listening " + port);**  **});** |

1. At this point we can use the the **app** object again to call various REST method like get() and **post()**. That **post()** method takes a route to send the request to and a function that handles the request and response objects.

|  |
| --- |
| **const express = require('express');**  **const port = 8000;**  **const app = express();**  **app.get('/', (request, response)=> response.send('hello from skillsoft'));**  **app.post('/addweight', function(request, response){**  **});**  **//**  **app.listen(port, function(){**  **console.log("Listening " + port);**  **});** |

1. With this code in place, we can use it to now get values from a form. For example on a form if there is a field called **empName**. We can get the value that the user put into that field by interrogating the **body** property of the **request** object.

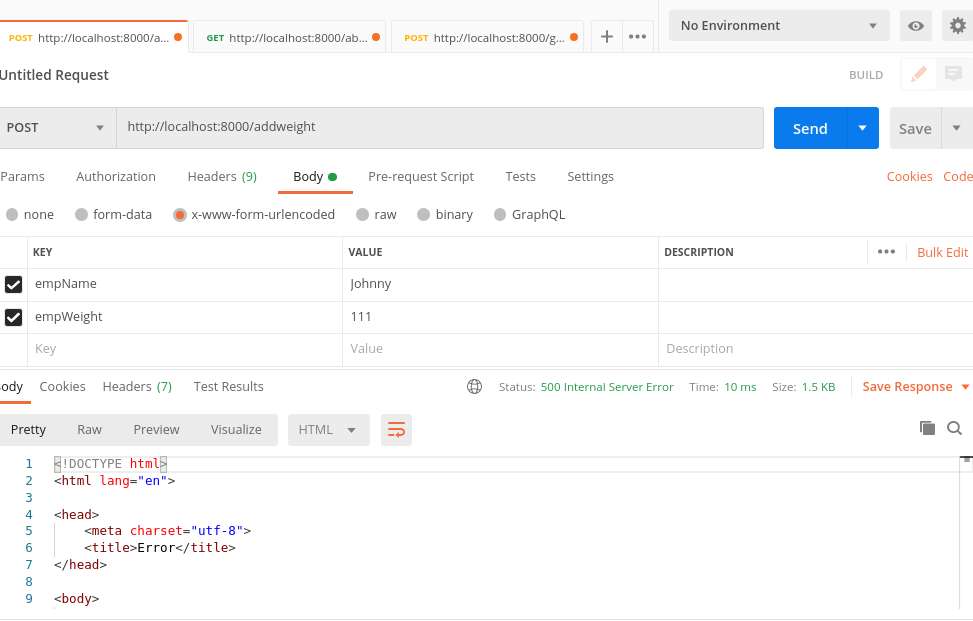
|  |
| --- |
| **app.post('/addweight', function(request, response){**  **let empName = request.body.empName;**    **});** |

1. Lets add one more field, empWeight. After that we can log the results or send them back to the browser using response.end(). I also changed the inner function to an arrow:

|  |
| --- |
| **app.use(express.urlencoded({extended:false}))**  **app.get('/', (request, response)=> response.send('hello from skillsoft'));**  **app.post('/addweight', (request, response)=>{**  **let empName = request.body.empName;**  **let empWeight = request.body.empWeight;**  **response.end(`POST success, you sent ${empName} and ${empWeight}, thanks!`);**  **});** |

Here is the entire file, so far:

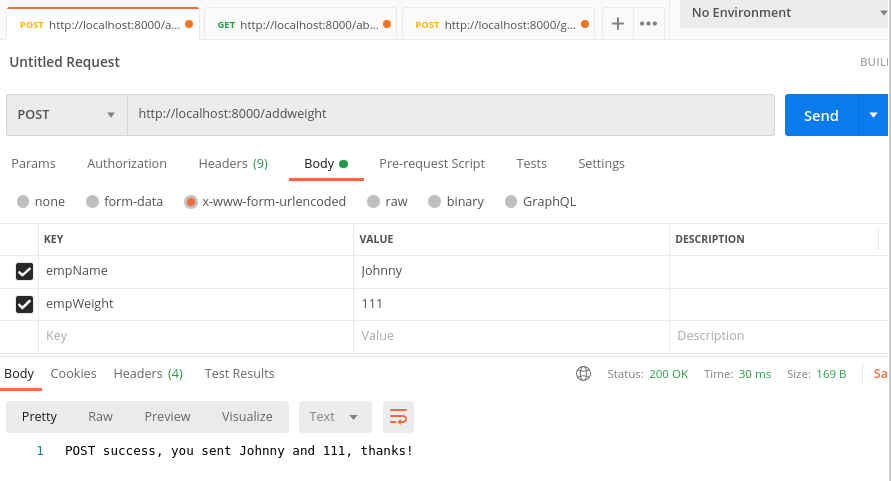
|  |
| --- |
| **const express = require('express');**  **const port = 8000;**  **const app = express();**  **app.get('/', (request, response)=> response.send('hello from skillsoft'));**  **app.post('/addweight', (request, response)=>{**  **let empName = request.body.empName;**  **let empWeight = request.body.empWeight;**  **response.end(`POST success, you sent ${empName} and ${empWeight}, thanks!`);**  **});**  **//**  **app.listen(port, function(){**  **console.log("Listening " + port);**  **});//** |

1. Now we have to test this out using a REST client or a plugin for the browser, see below. I will be using Postman for this part. Remember to turn on CORS in your browser:  
   
2. Now we did get an error and its because Express on its own, does not have the capability to handle form fields, we have to either add a specific package that does this part or use the one that came with Express. We will choose the latter:

|  |
| --- |
| **const express = require('express');**  **const port = 8000;**  **const app = express();**  **app.use(express.json());**  **app.get('/', (request, response)=> response.send('hello from skillsoft'));**  **app.post('/addweight', (request, response)=>{** |

1. Hit the SEND button on Postman again and take a look at the result. It succeeded but the values did not propagate properly, we need another line of code to interpret the form field values:

|  |
| --- |
| **const app = express();**  **app.use(express.json());**  **app.use(express.urlencoded({extended:false}));**  **app.get('/', (request, response)=> response.send('hello from skillsoft'));**  **app.post('/addweight', (request, response)=>{** |

****

**------end of part 03-------**

## part 04 – Routing Basics

1. So far we have been using Express itself (via the app object) to perform simple routing. Next we will use *router*, to handle all of our routing needs. First create a variable to point to the Router constructor:

|  |
| --- |
| **const express = require('express');**  **const port = 8000;**  **const app = express();**  **const router = express.Router();**  **app.use(express.json());**  **app.use(express.urlencoded({extended:false}));** |

1. Create a new variable and point it to the **Router()** constructor from **express**:

|  |
| --- |
| **const app = express();**  **const router = express.Router();**  **app.use(express.json());** |

1. We now have router to construct routes and the first route is going to be the **root route**. So where ever we have a route with app, just change it to router:

|  |
| --- |
| **const app = express();**  **const router = express.Router();**  **app.use(express.json());**  **app.use(express.urlencoded({extended:false}));**  **router.get('/', (request, response)=> response.send('hello from skillsoft'));**  **router.post('/addweight', (request, response)=>{**  **let empName = request.body.empName;** |

1. Before we can run this code, we need to tell our express app, to use **router** for executing routes. The **app.use()** method is saying to use router once you get to the root of this server path. If you attempt to spin the app, it will start but it will crash every time we go to a route, unless we register router with app as shown below:

|  |
| --- |
| **router.get('/', function(req, res){**  **res.send("You are on the root route");**  **});**  **//**  **app.use('/', router);**  **//**  **app.listen(port, function(){** |

1. Spin the application and go to a browser and everything should work like it did before, only now we are using Router.
2. Create an “About Us” route by copying the **get()** route and replacing the first parameter with something like “/aboutus”.

|  |
| --- |
| **router.get('/', function(req, res){**  **res.send("You are on the root route");**  **});**  **//**  **router.get('/aboutus', function(req, res){**  **res.send("You are on the about us route");**  **});**  **//**  **app.use('/', router);** |

**NOTE: whenever we make a change on the server code, we must stop and start the application, unless we use nodemon, coming soon.**

1. Continue to include other routes as necessary, here is the entire file. Note that the last route gets a specific document, based on some parameter passed in via the **URL**.

|  |
| --- |
| **const express = require('express');**  **const port = 8000;**  **const app = express();**  **const router = express.Router();**  **//**  **app.use(express.json());**  **app.use(express.urlencoded({extended:false}));**  **//**  **router.get('/', (request, response)=> response.send('hello from skillsoft'));**  **//**  **router.post('/addweight', (request, response)=>{**  **let empName = request.body.empName;**  **let empWeight = request.body.empWeight;**  **response.end(`POST success, you sent ${empName} and ${empWeight}, thanks!`);**  **});**  **//**  **router.get('/aboutus', function(req, res){**  **res.send("You are on the about us route");**  **});**  **//**  **app.use('/', router);**  **//**  **app.listen(port, function(){**  **console.log("Listening " + port);**  **});** |

**------end of part 04-------**

## part 05 – Installing Nodemon

1. Nodemon will restart the application whenever there is a change to any of the files in the application.
2. First install Nodemon generally with this command:

**sudo npm install -g nodemon**

1. Install Nodemon again in the folder you are using to build your application

**npm install nodemon --save-dev**

Note that Nodemon is a development dependency, it does not have to be installed in the final application, hence --save-dev will ensure that this does not happen.

1. With Nodemon installed, once you are in the directory just issue the command Nodemon and the app will spin, you don’t even have to point it to the file index.js.
2. However, it is customary to add a script that will solidify what Nodemon should do when we start the application. So open your package.json file and change the following lines:

|  |
| --- |
| **"main": "index.js",**  **"scripts": {**  **"start": "nodemon index.js"**  **},**  **"author": "",**  **"license": "ISC",**  **"dependencies": {** |

1. So now we can start the app using just **npm start** or **nodemon.** Also when we make changes in the future and save the file, index.js, the server will restart and accept our changes. Of course if there is a syntax error in our code, it will report it as well.

## part 06 – Decomposing Routes

1. Create a new folder called **routes** and inside of that directory, create a new .js file called routes.js.
2. The first line will be a variable pointing to a function, we have to do this in order for other files in our application to know that the routes file exists.

|  |
| --- |
| **module.exports = function(){};** |

1. Next we will CUT the three get() functions from our index.js file into this one

|  |
| --- |
| **module.exports = function(){**  **//**  **router.get('/', (request, response)=> response.send('hello from skillsoft'));**  **//**  **router.post('/addweight', (request, response)=>{**  **let empName = request.body.empName;**  **let empWeight = request.body.empWeight;**  **response.end(`POST success, you sent ${empName} and ${empWeight}, thanks!`);**  **});**  **//**  **router.get('/aboutus', function(req, res){**  **res.send("You are on the about us route");**  **});**  **};** |

1. However this file does not have access to **router**, but we can pass router to this file when we call the exported function.

|  |
| --- |
| **module.exports = function(router){**  **//**  **router.get('/', (request, response)=> response.send('hello from skillsoft'));**  **//**  **router.post('/addweight', (request, response)=>{**  **let empName = request.body.empName;**  **let empWeight = request.body.empWeight;**  **response.end(`POST success, you sent ${empName} and ${empWeight}, thanks!`);**  **});**  **//**  **router.get('/aboutus', function(req, res){**  **res.send("You are on the about us route");**  **});**  **};** |

1. Back in the server file, we have to let it know where to find routes.js, so create a variable and point it to the new routes.js file inside of the routes directory.

|  |
| --- |
| **Const app = express();**  **const router = express.Router();**  **const routes = require(‘./routes/routes’);** |

Remember we had cut the three route functions, so this file should be very short.

1. Use the newly created **routes** object to register the routing functionality via it’s constructor

|  |
| --- |
| **const app = express();**  **const router = express.Router();**  **routes(router);**  **//**  **app.use(express.json());** |

The rest of the index.js file remain unchanged.

Here is the entire index.js file, the routes.js file follows:

|  |
| --- |
| **const express = require('express');**  **const routes = require('./routes/routes');**  **const port = 8000;**  **const app = express();**  **const router = express.Router();**  **routes(router);**  **//**  **app.use(express.json());**  **app.use(express.urlencoded({extended:false}));**  **app.use('/', router);**  **//**  **app.listen(port, function(){**  **console.log("Listening " + port);**  **});** |

routes.js

|  |
| --- |
| **module.exports = function(router){**  **//**  **router.get('/', (request, response)=> response.send('hello from skillsoft'));**  **//**  **router.post('/addweight', (request, response)=>{**  **let empName = request.body.empName;**  **let empWeight = request.body.empWeight;**  **response.end(`POST success, you sent ${empName} and ${empWeight}, thanks!`);**  **});**  **//**  **router.get('/aboutus', function(req, res){**  **res.send("You are on the about us route");**  **});**  **};** |

1. Test the application, it should work just like before, no changes. But we have now ported our routes into a separate file, making future changes easier

## part 07 – Decomposing Controllers

1. Create a new directory called **controllers** and create a new **.js** file called controller.js
2. Open the controller.js file in an editor and start entering the first controller function. Remember controllers will take responsibility for making several decisions. The first controller should handle what happens when the user navigates to the root route:

|  |
| --- |
| **exports.getdefault = function(req, res){**  **res.send('You are on the root route.');**  **};** |

In this case we are not exporting the entire file, but each function is exported individually

1. Continue to develop this file by completing all the route functions, in other words, write functions that match the routes we had before. For now these functions are very simple, but soon, they will become a bit more complicated.

|  |
| --- |
| **exports.getdefault = function(req, res){**  **res.send('You are on the root route.');**  **};**  **//**  **exports.aboutus=function(req, res){**  **res.send('You are on the about us route.');**  **};**  **//**  **exports.addweight=function(req, res){**  **res.send('You are on the addweight route');**  **};**  **//**  **exports.getdocs=function(req, res){**  **res.send('You are on the getdocs route.');**  **};** |

I have just added a new function **getdocs** to do some interacting with the Weights database soon. This is the entire controller.js file so far.

1. Back in the routes.js file, we need to let this file know that there is a controller handling each route, so basically routes.js is now acting like a pointer to a controller function, which does the final piece in deciding what to serve to the client. Add this line at the top of the function.

|  |
| --- |
| **const controller = require('../controllers/controller');**  **module.exports = function(router){**  **//**  **router.get('/', (request, response)=> response.send('hello from skillsoft'));** |

1. We can now replace the router function in routes with the appropriate one from controller.js

|  |
| --- |
| **const controller = require('../controllers/controller');**  **module.exports = function(router){**  **//**  **router.get('/', controller.getdefault);**  **//**  **router.post('/addweight', (request, response)=>{** |

1. Test the root route, it should work just like in **part04**. Note, there is nothing to do in the index.js file. Now complete the rest of the routes with their respective controller functions:

|  |
| --- |
| **const controller = require('../controllers/controller');**  **module.exports = function(router){**  **//**  **router.get('/', controller.getdefault);**  **//**  **router.get('/addweight', controller.addweight);**  **//**  **router.get('/aboutus', controller.aboutus);**  **};** |

1. Remember that /addweight is a post method, so let’s change that on both sides, on the routes side and on the controller side, first the controller. First notice that each controller function has access to the request and response objects:

|  |
| --- |
| **exports.addweight=function(req, res){**  **let empName = req.body.empName;**  **let empWeight = req.body.empWeight;**  **res.end(`POST success, you sent ${empName} and ${empWeight}, thanks!`);**  **};** |

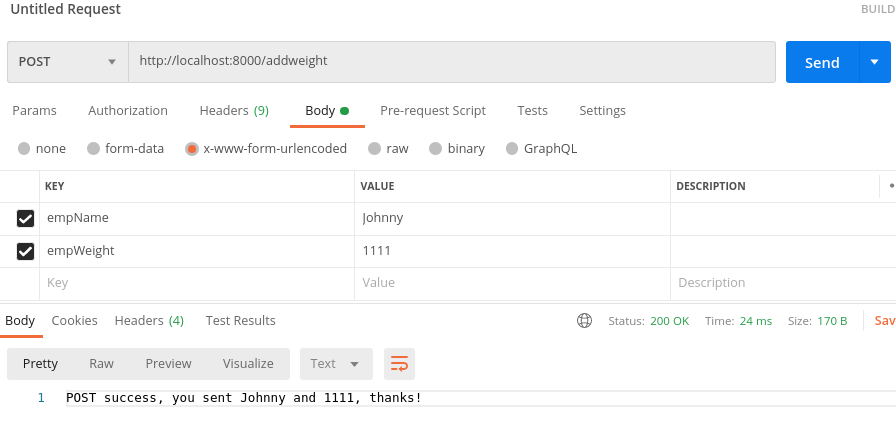
**Note:** I used request and response previously, now those two keywords have been shortened to req and res.

1. On the routes side, change the type of request from GET to POST

|  |
| --- |
| **module.exports = function(router){**  **//**  **router.get('/', controller.getdefault);**  **//**  **router.post('/addweight', controller.addweight);**  **//**  **router.get('/aboutus', controller.aboutus);**  **};** |

1. Test the post route using Postman

|  |
| --- |
| **module.exports = function(router){**  **//**  **router.get('/', controller.getdefault);**  **//**  **router.post('/addweight', controller.addweight);**  **//**  **router.get('/aboutus', controller.aboutus);**  **};** |



**------end of part 07-------**

## part 08 – Introduction to MongoDB

Before proceeding, either open a new terminal window or tab. For working with MongoDB via the command line, you do not have to be in any particular directory within the terminal window.

In order to get into the MongoDB shell, use the command **sudo mongo**

1. Change the database to Weights and create a new table using the following code:

|  |
| --- |
| **use Weights** |

1. Add a collection

|  |
| --- |
| **db.createCollectoin("EmployeeWeights")** |

1. Perform a find(), it should not return anytihng but at least we know we now have a database and a collection

|  |
| --- |
| **db.EmployeeWeights.find()** |

1. Enter a record

|  |
| --- |
| **db.EmployeeWeights.insertOne( {empName : "Joe", empWeight : 95.6 })** |

1. Verify the record.

|  |
| --- |
| **db.EmployeeWeights.find()** |

1. Add another record by using the up arrow key and just changing the name and weight

|  |
| --- |
| **db.EmployeeWeights.insertOne( {empName : “mary”,  empWeight : 65.9 })** |

1. Verify the new record

|  |
| --- |
| **db.EmployeeWeights.find()** |

1. Lets change (update) Joe’s record:

|  |
| --- |
| **db.EmployeeWeights.update(**  **{empName : "Joe"},**  **{$set: {empWeight : 96.5 } }**  **)** |

1. Verify the change

|  |
| --- |
| **db.EmployeeWeights.find()** |

1. Enter a new document but this one will have a date in addition to the name and weight

|  |
| --- |
| **db.EmployeeWeights.insertOne(**  **{**  **empName : "Sally",**  **empWeight : 65.9,**  **Date : new Date()**  **}**  **)** |

1. Verify the change but this tiime chain the pretty() method

|  |
| --- |
| **db.EmployeeWeights.find().pretty()** |

1. Finally update Joes’s record to include a date and then do a find pretty

|  |
| --- |
| **db.EmployeeWeights.update (   {empName : "Joe"},    {$set: {Date : new Date() } },  false, false )** |

**------end of part 08-------**

## part 09 – Setting up Mongoose

1. Return to the existing Node application and using a terminal pointing to your project, run the following install: **npm install mongoose**

Mongoose is an ORM which interacts with the **Weights** database and abstracts away much of the annoyances of working directly with the database natively.

1. Create a new directory called **models** and touch a new .js file inside of models called employees.js and add the following lines. Do this using your editor which should have the application opened:

|  |
| --- |
| **const mongoose = require('mongoose');**  **mongoose.connect('mongodb://localhost:27017/Weights', { useNewUrlParser: true });** |

The first line is simply requiring the mongoose package and the second is using the **connect()** method which takes 2 parameters, the location of the **mongod** service and a json object which is required and standard according to the documentation.

1. Next we will define the schema with the name wSchema:

|  |
| --- |
| **const mongoose = require('mongoose');**  **mongoose.connect('mongodb://localhost:27017/Weights', { useNewUrlParser: true });**  **const wSchema = new mongoose.Schema({**  **empName: String,**  **empWeight: Number,**  **created: {type: Date, default: Date.now }**  **});** |

1. We also need to let the client files know which collection we are working with, so expand the code to include the collection name:

|  |
| --- |
| **const mongoose = require('mongoose');**  **mongoose.connect('mongodb://localhost:27017/Weights', { useNewUrlParser: true });**  **const wSchema = new mongoose.Schema({**  **empName: String,**  **empWeight: String,**  **created: {type: Date, default: Date.now }**  **},{**  **collection:'EmployeeWeights'**  **});** |

1. Finally for the models.js file, we need to export our schema

|  |
| --- |
| **module.exports = mongoose.model('Weights', wSchema);** |

1. Here is the entire file

|  |
| --- |
| **const mongoose = require('mongoose');**  **mongoose.connect('mongodb://localhost:27017/Weights',**  **{**  **useNewUrlParser: true,**  **useUnifiedTopology: true**  **}**  **);**  **//**  **const wSchema = new mongoose.Schema({**  **empName: String,**  **empWeight: Number,**  **created: {type: Date, default: Date.now }**  **},{**  **collection:'Employee'**  **});**  **module.exports = mongoose.model('Weights', wSchema);** |

At this point, test the application to make sure there are no errors.

**------end of part 09-------**

## part 10 – Expanding the Controller Functions to work with Database

1. Open controller.js in an editor and the first line will be a variable pointing to the **models** directory and its contents.

|  |
| --- |
| **const Weight = require('../models/employees);**  **exports.getdefault=function(req, res){**  **res.send('You are on the root route.');**  **};**  **//** |

1. Next we will expand the **getdocs** function. That function will use the **Weight** variable created above and its attached **find()** method. Delete the **res.send()** function or comment it out. The find() method, like almost ALL Mongoose methods, takes an object as the first parameter and a function as the second.

|  |
| --- |
| **exports.getdocs=function(req, res){**  **Weight.find({}, function(err, results){});**  **//res.send('You are on the getdocs route.');**  **};** |

1. The **find()** method will handle any errors and any returns from the query, so lets expand on it.

|  |
| --- |
| **exports.getdocs=function(req, res){**  **Weight.find({}, function(err, results){**  **if (err)**  **res.end(err);**  **res.json(results);**  **});**  **//res.send('You are on the getdocs route.');**  **};** |

Now with this new code, we end the connection to the server if any errors occur and respond to the client with any data we got from executing the **find()** method.

1. In the routes.js file, make sure we have a route to match the function

|  |
| --- |
| **router.get('/getdocs', controller.getdocs);** |

1. Test the code by opening a browser and navigating to **http://localhost:8000/getdocs**



1. We can now try to get a single record by passing in the *name* to get in the url. So in **controller** create a controller method called get**employee**. Expand the get**employee()** controller method to find an employee by her name:

|  |
| --- |
| **exports.getemployee = function(req, res) {**  **};** |

1. The getemployee() method will interrogate the request object like before, but this time we are looking into the parameter property (called params). In the code below we are asking the params property for the value in employeeName, that the user was supposed to pass to this route:

|  |
| --- |
| **exports.getemployee = function(req, res) {**  **let empToFind = req.params.employeeName;**  **};** |

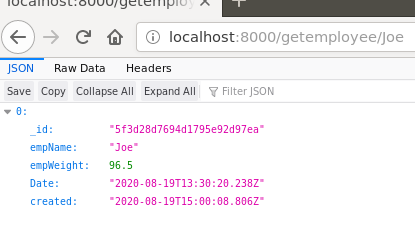
1. We can now pass this value to the find() method of our Weight object and handle any errors, as well as the result of our search:

|  |
| --- |
| **exports.getemployee = function(req, res) {**  **let empToFind = req.params.employeeName;**  **Weight.find({empName:empToFind}, function(err, results){**  **if (err)**  **res.end(err);**  **res.json(results);**  **});**  **};** |

1. Add a route to the routes.js file

|  |
| --- |
| **router.get('/getemployee/:employeeName', controller.getemployee);** |

1. Test the code by opening a browser and navigating to [**http://localhost:8000/employees/**](http://localhost:8000/employees/)**Joe**



Of course you can test in Postman also

1. (optional) We can cater for no records found by adding a simple if statement. Here is the entire function

|  |
| --- |
| **exports.getemployee=function(req, res){**  **let empToFind = req.params.employeeName;**  **Weight.find({empName:empToFind}, function(err, results){**  **if (err)**  **res.end(err);**  **if(!results.length)**  **res.status(404).send('We could not find that name');**  **else {**  **res.json(results);**  **}**  **});**  **};** |

1. (Optional) we could also work with the database in a more asynchronous way, all of Mongoose functions are asynchronous and they do return a promise.

|  |
| --- |
| **exports.getemployee = function(req, res) {**  **let empToFind = req.params.employeeName;**  **Weight.find({empName:empToFind})**  **.then(results=>{**  **if(!results.length)**  **res.status(404).send('We could not find that name');**  **else**  **res.json(results);**  **})**  **.catch(err=>console.log(err.message))**  **};** |

**------end of part 09-------**

## part 10 – Expanding the Controller to Delete From Database (optional)

1. In the routes.js file, copy any of the previous route lines and change the route to be **deletebyname**.

|  |
| --- |
| **router. delete('/deletebyname/:employeeName', controller.deletebyname);** |

Notice that the method call is a **delete()** NOT **get().**

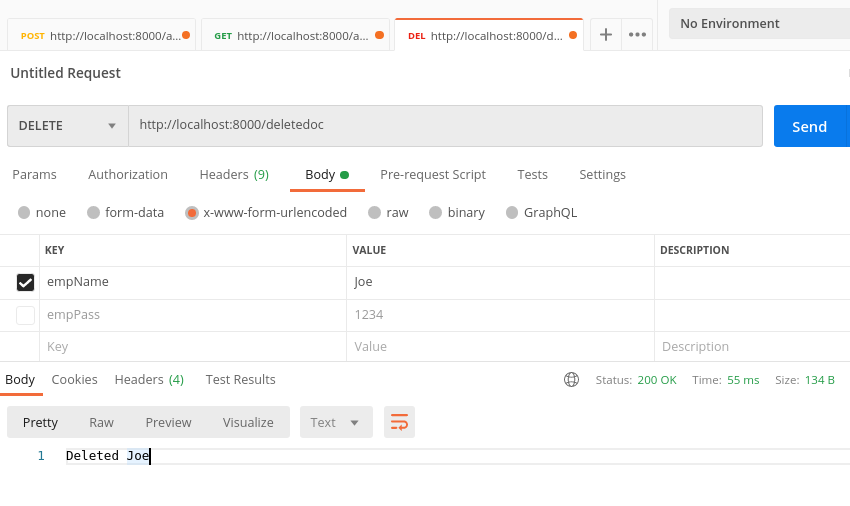
1. Create a matching function in the controller.js file, in fact we can just copy, paste and edit the get**employees()** function. Jut change **find()** to **deleteOne()** and create a new variable to hold the name to be deleted. In fact we can do both a restful delete or a delete by form, I am showing the latter:

|  |
| --- |
| **exports.deletedoc = function(req, res) {**  **let empToDelete = req.body.empName;**  **Weight.deleteOne({empName:empToDelete}, function(err, result) {**  **if (err)**  **res.send(err);**  **res.end(`Deleted ${empToDelete}`);**  **});**  **};** |

In this function, we get the name to delete from the form, store it in a variable, then pass the variable as a value to the **deleteOne()** method. If no errors we send a text message to the client.

1. We can now try to delete a single doc using the **REST** client. Remember to change the method to **DELETE**. Also **CORS** must be turned on.

**Note the function is looking for empName, so if the document was not stored with that name/value type of structure, the delete will fail.**



**Also notice that the empPass box is NOT ticked.**

The entire deletedoc() method

|  |
| --- |
| **exports.deletedoc = function(req, res) {**  **let empToDelete = req.body.empName;**  **Weight.deleteOne({empName:empToDelete}, function(err, result) {**  **if (err)**  **res.send(err);**  **res.end(`Deleted ${empToDelete}`);**  **});**  **};** |

**------end of part 10-------**

## part 11 – Expanding the Controller to Add a new Document to the Database

1. In the routes.js file, copy any of the previous route lines and change the route to be add a new document.

|  |
| --- |
| **router.route('/addnewdoc').post(controller.addnewdoc);** |

Notice that the method call is a **post()** NOT **get().**

1. Create a matching function in the controller.js file, in fact we can just copy, paste and edit the **deletebyname()** function.

|  |
| --- |
| **exports.addnewdoc=function(req, res){**  **let empName = req.body.empName;**  **let empPass = req.body.empPass; };** |

In this function, we get the name and new employee from an HTML form, NOT the **URL**. Notice that we now introduce a new field, empPass for password.

1. Create a variable and point it to the **Weight** object, which represents our database

|  |
| --- |
| **exports.addnewdoc = function(req,res){**  **let empName = req.body.empName;**  **let empWeight = req.body.empPass;**  **const Weights = new Weight();** |

1. Use the new variable and its properties to pass values from the form to the database properties

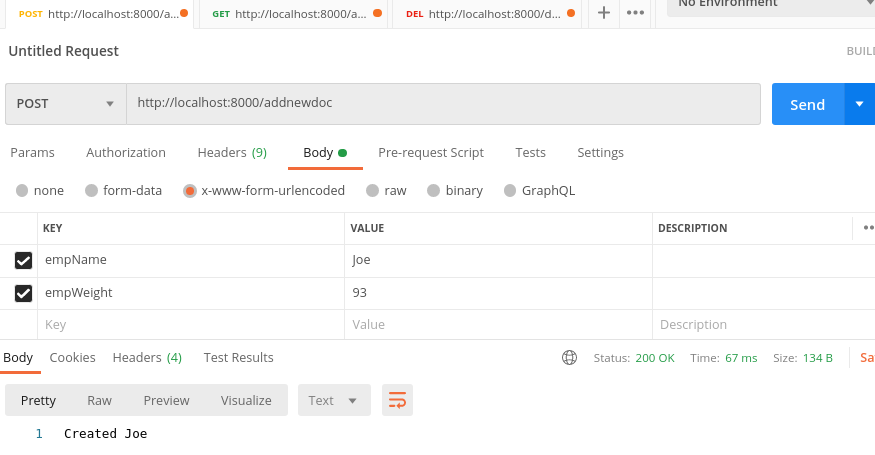
|  |
| --- |
| **const Weights = new Weight();**  **Weights.empName = empName;**  **Weights.empPass = empPass;** |

1. Now all we have to do is call the **save**() method of our **weight** object and deal with errors, here is the entire function

|  |
| --- |
| **exports.addnewdoc = function(req, res) {**  **let empName = req.body.empName;**  **let empPass = req.body.empPass;**  **const Weights = new Weight();**  **Weights.empName = empName;**  **Weights.empPass = empPass;**  **Weights.save({}, function(err) {**  **if (err)**  **res.end(err);**  **res.end(`Created ${empName}`);**  **});**  **};** |

1. Before we test the new function using the **REST** client we should change the database schema to accommodate this new field:

|  |
| --- |
| **const wSchema = new mongoose.Schema({**  **empName: String,**  **empWeight: Number,**  **empPass: String,**  **created: {type: Date, default: Date.now }**  **},{** |



You can also verify that Joe is in the database by going to localhost:8000/getdocs

**------end of part 11-------**

## part 12 – Expanding the Controller to update a Document in the Database

1. In the routes.js file, copy any of the previous route lines and change the route to be **updatedoc**.

|  |
| --- |
| **app.route('/updatedoc’).put(controller. updatedoc);** |

Notice that the method call is a **put()** NOT **get().**

1. Create a matching function in the controller.js file, in fact we can just copy, paste and edit the **addnewdoc()** function.

|  |
| --- |
| **exports.updatedoc= function(req,res){};** |

In this function, we get the name and/or weight from an HTML form, NOT the url.

1. Since we did most of what is needed in the **putdoc()** function, just copy paste and change accordingly

|  |
| --- |
| **exports.updatedoc=function(req, res){**  **let empName = req.body.empName;**  **let newWeight = req.body.newWeight;**    **};** |

1. Next we add a query object that we can pass to the updateOne() method of Mongoose in order to find the document we want to update

|  |
| --- |
| **exports.updatedoc=function(req, res){**  **let empName = req.body.empName;**  **let newWeight = req.body.newWeight;**  **let query = { empName : empName };**  **};** |

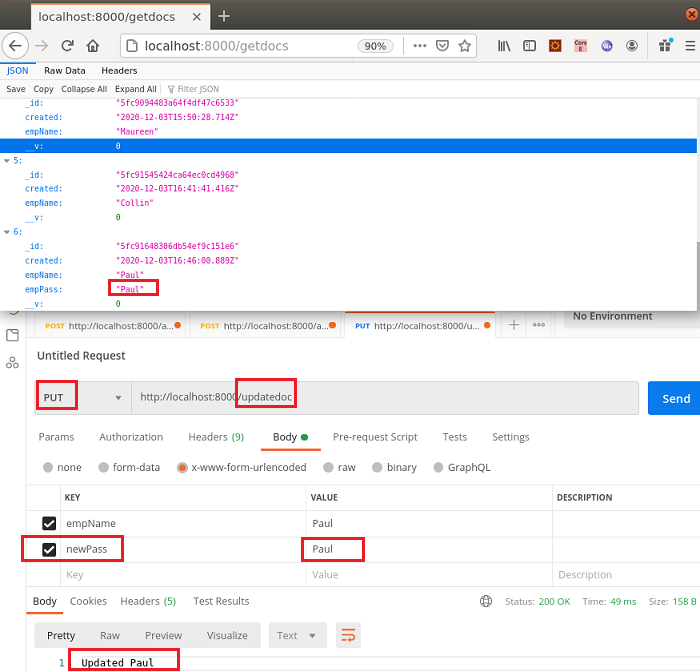
1. After that, we add an object that would pass the new value to be changed along with the command to do so, which is $set:

|  |
| --- |
| **exports.updatedoc=function(req, res){**  **let empName = req.body.empName;**  **let newWeight = req.body.newWeight;**  **let query = { empName : empName };**  **let data = { $set : {empWeight : newWeight } };** |

1. Finally we deal fire the updateOne() method and deal with any errors or results

|  |
| --- |
| **let data = { $set : {empWeight : newWeight } };**  **Weight.updateOne(query, data, function(err, result) {**  **if (err)**  **res.send(err);**  **res.end(`Updated ${empName}`);**  **});** |

Test the new function using the REST client. Update any of the documents you have, in this case we are updating Joe from 93 to 97 Kgs



If you wanted to see what is in result, use this line instead

**res.end(`Updated ${fixName}, ${result.nModified}`);**

1. Here is the entire updatedoc() function

|  |
| --- |
| **//**  **exports.updatedoc = function(req, res) {**  **let empName = req.body.empName;**  **let newPass = req.body.newPass;**  **let query = { empName : empName };**  **let data = { $set : {empPass : newPass } };**  **Weight.updateOne(query, data, function(err, result) {**  **if (err)**  **res.send(err);**  **res.end(`Updated ${empName}`);**  **});**  **};** |

**------end of part 12-------**

|  |
| --- |
| Linux commands:   1. To copy the current directory to a new one: axle@pc0469:~/Documents/FSD/Day03/Part04$ **cp -r ./ ../Part05**   This code will copy Part04 into Part05   1. To create a new directory: **mkdir routes** |