Table of Contents

[**part 1 – Verify nodejs** 2](#_Toc19083745)

[**part 2 – Building a Simple nodejs App** 3](#_Toc19083746)

[**part 3 – Including and Working with node Packages** 5](#_Toc19083747)

[**part 4 – Routing Basics** 8](#_Toc19083748)

[**part 5a – Decomposing Routes** 10](#_Toc19083749)

[**part 5B – Decomposing Controllers** 12](#_Toc19083750)

[**part 6 – Including and Working with MongoDB** 14](#_Toc19083751)

[**part 7 – CRUD Operations with MongoDB** 15](#_Toc19083752)

[**part 8 – Setting up Mongoose** 17](#_Toc19083753)

[**part 9 – Expanding the Controller Functions to work with Database** 19](#_Toc19083754)

[**part 10 – Expanding the Controller to Delete From Database** 21](#_Toc19083755)

[**part 11 – Expanding the Controller to Add a new Document to the Database** 22](#_Toc19083756)

[**part 12 – Expanding the Controller to update a Document in the Database** 24](#_Toc19083757)

Day03 Building the APIs – in 12 Steps

## part 1 – Verify nodejs

1. Create a folder called **Part01**.
2. Open a terminal inside of **Part01** 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 **Part01** 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 2 – Building a Simple nodejs App

1. Create a folder called **Part02**.
2. Open a terminal inside of **Part02** 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. This time instead of accepting index.js, change it to http\_server.js.
4. Open a terminal window pointing to the **Part02** folder and using touch, create a .js file called http\_server.js.
5. Open http\_server.js inside of a text editor and type the following lines:

|  |
| --- |
| **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 myServer = 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 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 myServer = 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

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

Here is the entire http\_server.js file

|  |
| --- |
| **const http = require('http');**  **const hostname = "localhost";**  **const port = 8000;**  **const myServer = http.createServer(function(request, response){**  **response.writeHead(200, {'Content-Type':'text/plain'});**  **response.write("Hello from Skillsoft");**  **response.end();**  **});**  **myServer.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 3 – Including and Working with node Packages

1. Copy the folder called **Part02** paste it within the same folder and then rename it to **part03**.
2. Open a terminal inside of **Part03** and run the command **npm install** which will install everything that part02 had, it will use the json file from that directory.
3. While still in part03, install body parser by running this command from a terminal window that is pointing to Part03 directory: **npm install body-parser --save**
4. Install express by running this command from a terminal window that is pointing to Part03 directory: **npm install express --save**
5. Open the http\_server.js and replace the first line with this one

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

1. Next we will require body parser, and express does not need the hostname, so you could remove it or leave but do not use it

|  |
| --- |
| **const express= require('express'); const bodyParser = require('body-parser');**  **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 port = 8000; const app = express();** |

1. This example will use the POST method of the browser to pass data to our server, so we need body-parser to help with the identification of form values:

|  |
| --- |
| **const express = require('express');**  **const bodyParser = require('body-parser');**  **const port = 8000;**  **const app = express(); app.use(bodyParser.urlencoded({extended:false}));** |

9. At this point we can use the the **app** object again to call the **post()** method. 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 bodyParser= require(‘body-parser’);**  **const port = 8000;**  **const app = express(); app.use(bodyParser.urlencoded({extended:false})); app.post('/addnewdoc', function(request, response){});** |

With this code in place, we can use it to now get values from a form. For example on the form 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('/addnewdoc', function(request, response){**  **let empName = request.body.empName;**    **});** |

1. We can extend this to the **weight** value as well. Also for now lets just use the log to show that we did receive those values on the server end

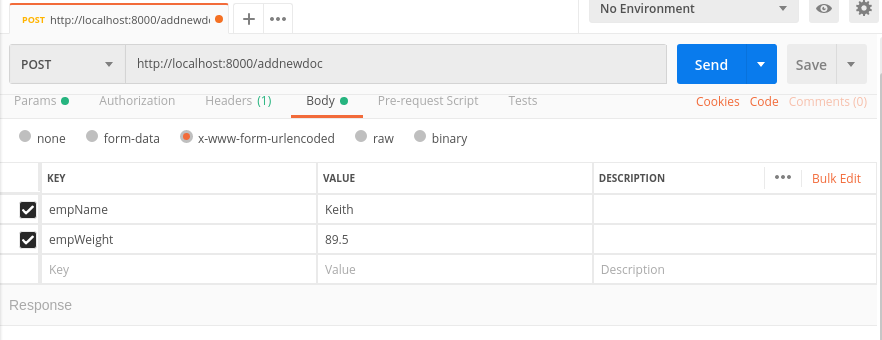
|  |
| --- |
| **app.post('/addnewdoc', function(request, response){**  **let empName = request.body.empName;**  **let empWeight = request.body.empWeight;**  **console.log(`POST success, you sent ${empName} and ${empWeight}, thanks!`);**  **response.end(`POST success, you sent ${empName} and ${empWeight}, thanks!`);**  **});** |

1. Finally for this file, remember when we ran the server file using node and the terminal window, we did not get any response. Lets change the **listen** method to use **app** and also to inform the developer that the service has started. This code goes to the bottom of the server file.

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

Here is the entire file, so far:

|  |
| --- |
| **const express = require('express');**  **const bodyParser = require('body-parser');**  **const port = 8000;**  **const app = express();**  **app.use(bodyParser.urlencoded({extended:false}));**  **app.post('/addnewdoc', function(request, response){**  **let empName = request.body.empName;**  **let empWeight = request.body.empWeight;**  **console.log(`POST success, you sent ${empName} and ${empWeight}, thanks!`);**  **response.end(`POST success, you sent ${empName} and ${empWeight}, thanks!`);**  **});**  app.listen(port, () => { console.log("Listening " + port); });  **app.listen(port, function(){**  **console.log("Listening " + port);**  **});** |

1. Now we have to test this out using a REST client in the browser, see below  
   Remember to turn on CORS and pass along a header  
   
2. Check the response in the terminal window and also on the browser’s console window.

|  |
| --- |
| **File with Arrow functions**  **const express = require('express');**  **const bodyParser = require('body-parser');**  **const port = 8000;**  **const app = express();**  **app.use(bodyParser.urlencoded({extended:false}));**  **app.post('/addnewdoc', function(request, response){**  **let empName = request.body.empName;**  **let empWeight = request.body.empWeight;**  **console.log(`POST success, you sent ${empName} and ${empWeight}, thanks!`);**  **response.end(`POST success, you sent ${empName} and ${empWeight}, thanks!`);**  **});**  **app.listen(port, () => {**  **console.log("Listening " + port);**  **});** |

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

## part 4 – Routing Basics

1. Copy the folder called **Part03** paste it inside of Day03 and then rename it to **part04**.
2. Run **npm install** using a terminal window.
3. Erase most of the code except for the first 5 lines and the listener at the bottom, so your http\_server.js file in part04 should look like the code below:

|  |
| --- |
| **const express = require('express');**  **const bodyParser= require('body-parser');**  **const port = 8000;**  **const app = express();**  **app.use(bodyParser.urlencoded({extended:false}));**  **app.listen(port, () => console.log("Listening " + port));** |

1. Next we will install *router*, so run this code in the terminal window that points to part04: **npm install router –save**
2. Create a new variable and point it to the **Router()** constructor from **express**:

|  |
| --- |
| **const app = express();**  **app.use(bodyParser.urlencoded({extended:false}));**  **//**  **const router = express.Router();** |

1. We now have router to construct routes and the first route is going to be the **root route**. Each route will use a method that represents an http REST verb. The first parameter will be the route path and the second will be a function that handles the request and response objects.

|  |
| --- |
| **const app = express();**  **app.use(bodyParser.urlencoded({extended:false}));**  **//**  **const router = express.Router();**  **router.get('/', function(req, res){**  **res.send("You are on the root route");**  **});** |

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

|  |
| --- |
| **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 the root address which remember is being served from port 8000, you should see “You are on the root route”. We can now proceed to build other routes.
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.**

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 bodyParser = require('body-parser');**  **const port = 8000;**  **const app = express();**  **app.use(bodyParser.urlencoded({extended:false}));**  **//**  **const router = express.Router();**  **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");**  **});**  **//**  **router.get('/employees/:employeeID', function(req, res){**  **res.send("You are viewing employee # " + req.params.employeeID);**  **});**  **//**  **app.use('/', router);**  **//**  **app.listen(port, () => console.log("Listening " + port));** |

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

## part 5a – Decomposing Routes

1. Copy the folder called **Part04** paste it inside and then rename it to **part05**.
2. Open a terminal inside of **Part05** and run the command **npm install** which will install everything that **part04** had, it will use the **JSON** file from that directory.
3. Create a new folder called **routes** and inside of that directory, create a new .js file called routes.js.
4. 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(app){};** |

Also notice that we have to pass the Express app into this function as a parameter, so that it becomes available to the entire function.

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

|  |
| --- |
| **module.exports = function(app){**  **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”);**  **});**  **//**  **router.get(‘/employees/:employeeName’’, function(req, res){**  **res.send(“You are viewing employee# “ + req.params.employeeName’’);**  **});**  **//**  **};** |

1. However this file does not have access to **router**, it has access to **app**, which has access to **router**, so change the router object to the **app** object.

|  |
| --- |
| **module.exports = function(app){**  **app.get(‘/’, function(req, res){**  **res.send(“You are on the root route”);**  **});**  **//**  **app.get(‘/aboutus’, function(req, res){**  **res.send(“You are on the about us route”);**  **});**  **//**  **app.get(‘/employees/:employeeName’’, function(req, res){**  **res.send(“You are viewing employee “ + req.params.employeeName’’);**  **});**  **//**  **};** |

1. Back in the http\_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 Express **app** via it’s constructor

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

The rest of the http\_server.js file remain unchanged.

Here is the entire http\_server.js file, the routes.js file follows:

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

routes.js

|  |
| --- |
| **module.exports = function(app){**  **app.get('/', function(req, res){**  **res.send("You are on the root route");**  **});**  **//**  **app.get('/aboutus', function(req, res){**  **res.send("You are on the about us route");**  **});**  **//**  **app.get('/employees/:employeeName', function(req, res){**  **res.send("You are viewing employee# " + req.params.employeeName);**  **});**  **//**  **};** |

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
2. App is still using its own get() method, we can now change that to use route instead, but we have to re-structure the code a bit

|  |
| --- |
| **module.exports = function(app){**  **app.route('/').get(function(req, res){**  **res.send("You are on the root route");**  **});**  **app.route('/aboutus').get(function(req, res){**  **res.send("You are on the about us route");**  **});**  **app.route('/employees/:employeeName').get(function(req, res){**  **res.send("You are viewing employee# " + req.params.employeeName);**  **});**  **};** |

## part 5B – 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.employees=function(req, res){**  **res.send('You are viewing employee# ' + req.params.employeeName);**  **};**  **//**  **exports.getallrecords=function(req, res){**  **res.send('You are on the getallrecords route.');**  **};** |

I have just added a new function **getallrecords** 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.

|  |
| --- |
| **module.exports = function(app){**  **let controller = require('../controllers/controller');**  **app.route('/').get(function(req, res){**  **res.send("You are on the root route");**  **});**  **app.route('/aboutus').get(function(req, res){**  **res.send("You are on the about us route");** |

1. We can now replace the get function with the appropriate one from controller.js

|  |
| --- |
| **module.exports = function(app){**  **let controller = require('../controllers/controller');**  **app.route('/').get(controller.getdefault);**  **app.route('/aboutus').get(controller.aboutus);**  **app.route('/employees/:employeeName').get(controller.employees);**  **app.route('/getallrecords').get(controller.getallrecords);**  **//**  **};** |

Notice that each line represents a route. That route now points to a function in the controller file.

1. Test the application, it should work just like in **part04**. Note, there is nothing to do in the http\_server.js file.

**Do appendix on Nodemon.**

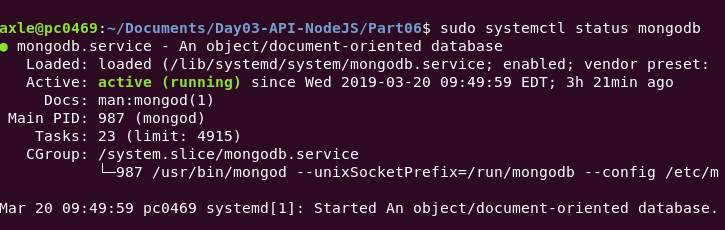
**------end of part 05-------**

## part 6 – Including and Working with MongoDB (Optional)

1. Create a new folder called **part06**.
2. Open a terminal inside of **Part06** and run the command **npm init** which will install a new JSON file.
3. We need to install the MongoDB package for our app, but the MongoDB software should already be installed on the OS, so run this command from the Ubuntu prompt, to make sure that the service is installed and running:

|  |
| --- |
| **sudo systemctl status mongodb** |

It should show something like the image below:



1. While still in **part06**, install MongoDB by running this command from a terminal window that is pointing to Part06 directory: **npm install mongodb --save**
2. If you are on linux, run the touch command to create the http\_server.js file
3. Open the http\_server.js and replace the first line with this one, **remove everything else**.

|  |
| --- |
| **const MongoClient = require('mongodb').MongoClient;** |

1. Next we will create another variable and make it equal to the address of where mongo lives

|  |
| --- |
| **const MongoClient = require('mongodb').MongoClient;**  **const url = 'mongodb://localhost:27017';** |

1. Create a new variable and have it represent the database we created in mongo on day4

|  |
| --- |
| **const MongoClient = require('mongodb').MongoClient;**  **const url = 'mongodb://localhost:27017';**  **const dbName = 'Weights';** |

1. Create a new variable and have it point to the **MongoClient** constructor. The constructor takes 2 parameters, a url of where mongo lives and a json object that is directly out of the documentation.

|  |
| --- |
| **const MongoClient = require('mongodb').MongoClient;**  **const url = 'mongodb://localhost:27017';**  **const dbName = 'Weights';**  **const mClient = new MongoClient(url, {**  **useNewUrlParser: true**  **});** |

9. At this point we can verify that we can connect to the database using the connect() method of mClient. Pass it a function which takes the error object.

|  |
| --- |
| **mClient.connect(function(err) {**  **if(err) console.log("Error");**  **console.log("Success!!!");**  **mClient.close();**  **});** |

1. Run the application using **nodemon http\_server** if you installed nodemon. If you see a “success!!!” Message in the terminal window, then we are able to connect to **mongodb**

**------end of part 06-------**

## part 7 – CRUD Operations with MongoDB

1. Copy the folder called **Part06** paste it and then rename it to **part07**.
2. Open a terminal inside of **Part07** and run the command **npm install** which will install everything that **Part06** had, it will use the json file from that directory.
3. This file will insert a new document into the **Weights** database. Open the http\_server.js file, which is basically the same file from **Part06**. Then add, below the existing code, the following **variable** which points to **function**. The function takes two **parameters**, the first represents our **Weights** database and the second is a **function** that will run, after the **putDocuments()** function is ran.

|  |
| --- |
| **const putDocuments = function(db, callback) {**  **};** |

1. Next we will add a variable to represent the collection inside of our database. So the database is represented by **db** and the **EmployeeWeights** collection is represented by **collection**.

|  |
| --- |
| **const putDocuments = function(db, callback) {**  **const collection = db.collection('EmployeeWeights');**  **};** |

1. Use the **insertMany()** function of the **collection** object to insert 2 documents

|  |
| --- |
| **const putDocuments = function(db, callback) {**  **const collection = db.collection('EmployeeWeights');**  **collection.insertMany([**  **{"empName":"Axle", "empWeight" : "85.8"},**  **{"empName":"John", "empWeight" :"102"}**  **]);**  **};** |

1. Although this code will work, we should also supply a second parameter to the **insertMany()** method which is a function and it will handle any errors as well as feedback from the server, after the insert.

|  |
| --- |
| **const putDocuments = function(db, callback) {**  **const collection = db.collection('EmployeeWeights');**  **collection.insertMany([**  **{"empName":"Axle", "empWeight" : "85.8"},**  **{"empName":"John", "empWeight" :"102"}**  **], function(err, result) {**  **console.log("Inserted 2 documents/records");**  **callback(result);**  **});**  **};** |

1. Also the second parameter includes a callback. That will send execution back to the caller function which we will change next.
2. With the **putDocuments()** function complete, go back to the **connect()** function and call **putDocuments**(). This function should be in the http\_server.js file already from part 06. But first we need to create a variable that represents our database object. **mClient** in our case has a **db** method to which we pass the database name. We are returned with an object representing the **Weights** database.

|  |
| --- |
| **mClient.connect(function(err) {**  **if(err) console.log("Error");**  **const db = mClient.db(dbName);**  **putDocuments(db, callback);**  **});** |

1. We now need to replace the word **callback** with an actual anonymous function, which will execute the **mClient.close()** function, thereby closing the database connection. So make sure the **mClient.close()** function is wrapped inside of the anonymous function.

|  |
| --- |
| **const db = mClient.db(dbName);**  **putDocuments(db, function(){**  **mClient.close();**  **});** |

1. Before executing the code, show the current database with the current set of documents. Then run this new code to show that the 2 documents were inserted.

Here is the entire code so far:

|  |
| --- |
| **const MongoClient = require('mongodb').MongoClient;**  **const url = 'mongodb://localhost:27017';**  **const dbName = 'Weights';**  **const mClient = new MongoClient(url, {**  **useNewUrlParser: true**  **});**  **mClient.connect(function(err) {**  **if(err) console.log("Error");**  **const db = mClient.db(dbName);**  **putDocuments(db, function(){**  **mClient.close();**  **});**  **});**  **const putDocuments = function(db, callback) {**  **const collection = db.collection('EmployeeWeights');**  **collection.insertMany([**  **{"empName":"Axle", "empWeight" : "85.8"},**  **{"empName":"John", "empWeight" :"102"}**  **],**  **function(err, result) {**  **if(err) throw err;**  **console.log("Inserted 2 documents/records");**  **callback(result);**  **}**  **);**  **};** |

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

## part 8 – Setting up Mongoose

1. Copy the folder called **Part05** paste it and then rename it to **part08**.
2. Open a terminal inside of **Part08** and run the command **npm install** which will install everything that **part07** had, it will use the json file from that directory.
3. Install mongodb like you did for part06
4. We also need to install **Mongoose**, so once inside of **Part08** run this command:  
   **npm install mongoose –save.** Mongoose is an ORM which interacts with the **Weights** database and abstracts away much of the annoyances of working directly with the database natively.
5. Create a new directory called **models** and touch a new .js file inside of models called employees.js and add the following lines.

|  |
| --- |
| **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.

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

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 });**  **const wSchema = new mongoose.Schema({**  **empName: String,**  **empWeight: Number,**  **created: {type: Date, default: Date.now }**  **},{**  **collection:'EmployeeWeights'**  **});**  **module.exports = mongoose.model('Weights', wSchema);** |

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

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

## part 9 – Expanding the Controller Functions to work with Database

1. Copy the folder called **Part08** paste it and then rename it to **part09**.
2. Open a terminal inside of **Part09** and run the command **npm install** which will install everything that **part08** had.
3. 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 **getallrecords** function. That function will use the **Weight** variable created above and its attached **find()** method. Delete the **res.send()** function or comment it out.

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

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

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

|  |
| --- |
| **app.route('/getallrecords').get(controller.getallrecords);** |

1. Test the code by opening a browser and navigating to **http://localhost:8000/getallrecords**
2. We can now try to get a single record by passing in the *name* to get in the url. Remember we had a route called **employees** and a controller called **employees**. Expand the **employees()** controller method to find an employee by her name:

|  |
| --- |
| **exports.employees = 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

|  |
| --- |
| **app.route('/employees/: employeeName').get(controller.employees);** |

1. Test the code by opening a browser and navigating to [**http://localhost:8000/employees/Axle**](http://localhost:8000/employees/Axle)
2. (optional) We can cater for no records found by adding a simple if statement. Here is the entire function

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

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

## part 10 – Expanding the Controller to Delete From Database

1. Copy the folder called **Part09** paste it and then rename it to **part10**.
2. Open a terminal inside of **Part10** and run the command **npm install** which will install everything that **part09** had, it will use the JSON file from that directory.
3. In the routes.js file, copy any of the previous route lines and change the route to be **deletebyname**.

|  |
| --- |
| **app.route('/deletebyname/:employeeName').delete(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 **employees()** function. Jut change **find()** to **deleteOne()** and create a new variable to hold the name to be deleted.

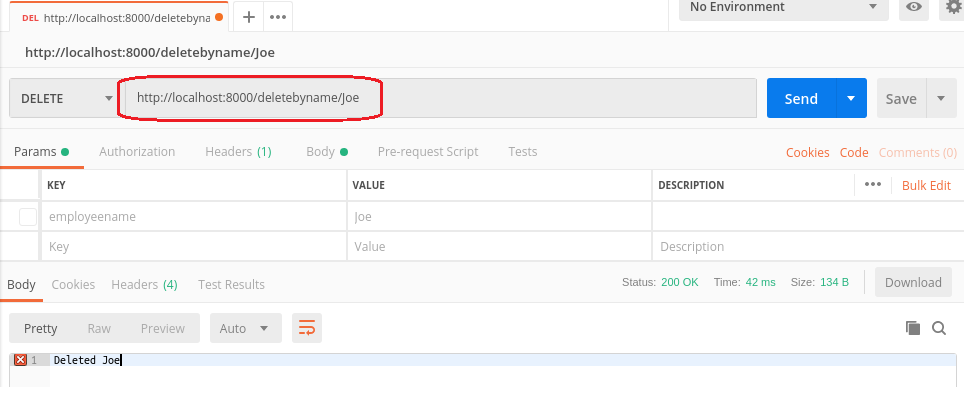
|  |
| --- |
| **exports.deletebyname=function(req, res){**  **let empToDelete = req.params.employeeName;**  **Weight.deleteOne({empName:empToDelete}, function(err, result) {**  **if (err)**  **res.send(err);**  **res.end(`Deleted ${empToDelete}`);**  **});**  **};** |

In this function, we get the name to delete from the **URL**, 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 record by passing in the *name* to get in the **URL**. We will need to use 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 use a REST client for this request**



**Note, if using Postman, do not use Params tab, the parameter is actually being passed in the URL itself.**

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

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

1. Copy the folder called **Part10** paste it inside and then rename it to **part11**.
2. Open a terminal inside of **Part11** and run the command **npm install** which will install everything that **part10** had, it will use the json file from that directory.
3. In the routes.js file, copy any of the previous route lines and change the route to be **putdoc**.

|  |
| --- |
| **app.route('/putnewdoc').post(controller.putnewdoc);** |

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.putnewdoc = function(req, res){};** |

In this function, we get the name and weight from an HTML form, NOT the **URL**.

1. We can now expand the **putnewdoc()** function to interrogate the REST client’s body values for name and weight. **NOTE: body parser must be setup properly for this to work:**

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

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

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

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

|  |
| --- |
| **const weight = new Weight();**  **weight.empName = empName;**  **weight.empWeight = empWeight;** |

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

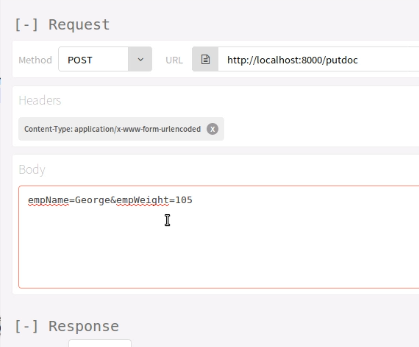
|  |
| --- |
| **exports.putnewdoc = function(req,res){**  **let empName = req.body.empName;**  **let empWeight = req.body.empWeight;**  **const weight = new Weight();**  **weight.empName = empName;**  **weight.empWeight = empWeight;**  **weight.save({}, function(err) {**  **if (err)**  **res.end(err);**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

Test the new function using the **REST** client.

**NOTE (if using Firefox Rest Client):**

1. **body parser must be installed properly for this to work**
2. **CORS must be enable in the browser**
3. **body-parser must be installed and *app.use(bodyparser)* must be directly underneath *const app = express();***

**For Postman, use x-www-form-urlencoded with empName and empWeight**



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

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

1. Copy the folder called **Part11** paste it and then rename it to **part12**.
2. Open a terminal inside of **Part12** and run the command **npm install** which will install everything that **part11** had, it will use the json file from that directory.
3. 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 **putnewdoc()** function.

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

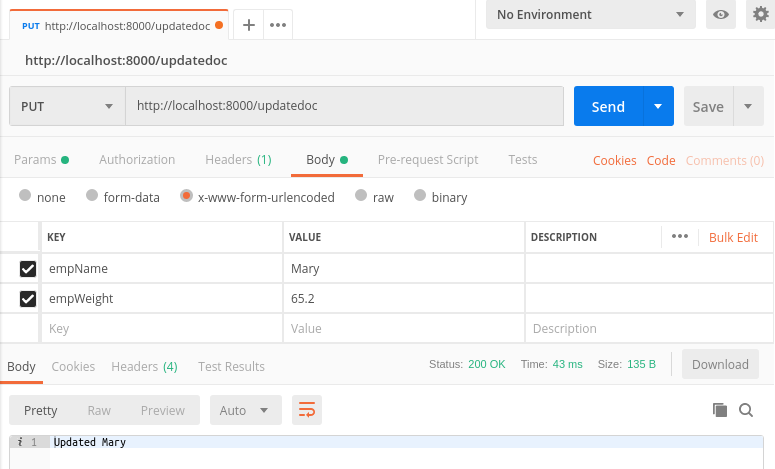
In this function, we get the name and weight from an HTML form, NOT the url.

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

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

In this function, we use the **updateOne()** method of the weight object. Also we need to first find the record we need to update and then update it by using the **$set** keyword.

Test the new function using the REST client. NOTE: body parser must be installed properly for this to work.



**------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** |