Table of Contents

[**part 01 – Verify nodejs** 2](#_Toc129953557)

[**part 02 – Building a Simple nodejs App** 4](#_Toc129953558)

[**part 03 – Introduction to node Packages and Express** 6](#_Toc129953559)

[**part 04 – Routing Basics** 10](#_Toc129953560)

[**part 05 – Installing Nodemon** 12](#_Toc129953561)

[**part 06 – Decomposing Routes** 13](#_Toc129953562)

[**part 07 – Decomposing Controllers** 15](#_Toc129953563)

[**part 08 – Introduction to MongoDB** 18](#_Toc129953564)

[**part 09 – Setting up Mongoose** 20](#_Toc129953565)

[**part 10 – Expanding the Controller Functions to work with Database** 21](#_Toc129953566)

[**part 11 – Expanding the Controller to Add a new Document to the Database** 26](#_Toc129953567)

[**part 12 – Expanding the Controller to update a Document in the Database** 28](#_Toc129953568)

[**Appendix A – Simple HTTP Server** 30](#_Toc129953569)

[**Appendix B – Linux Commands** 30](#_Toc129953570)

[**Appendix C –Delete From Database** 31](#_Toc129953571)

Day01 - Building the APIs

## part 01 – Verify nodejs

1. Create a project folder called **FSD** or something similar.
2. Open a terminal inside of your folder 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. Alternatively just do **npm init -y**

|  |
| --- |
|  |

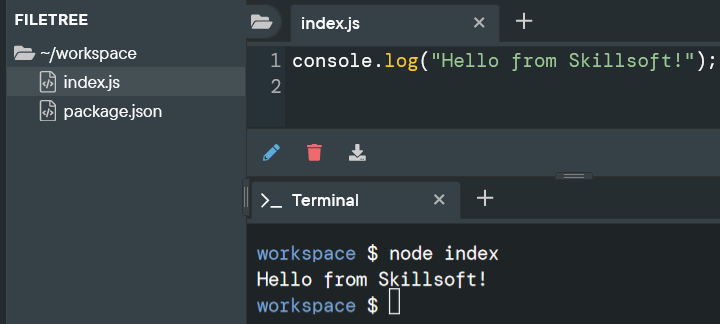
1. According to the .json file, node will look for index.js in order to execute the code inside, so use *touch* to create index.js inside of the **API** folder.
2. Add the following code to execute. This is just to make sure that node is working and it is executing properly.

|  |
| --- |
| **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.

|  |
| --- |
|  |

If you are using the third party service next.tech, you browser screen should look something like the image below:



**------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 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);**  **});** |

|  |
| --- |
| If you got this far, go to the terminal window and run node index again. If you see a message *Listening on 8000* then you can open a browser window to that location on your localhost, so <http://localhost:8000>  You should see your message there, *Hello from Skillsoft* |

1. At this point we can use the the **app** object again to call various REST method like **get()** and **post()**. The **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('/addemployee', 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('/addemployee', function(request, response){**  **let empName = request.body.empName;**    **});** |

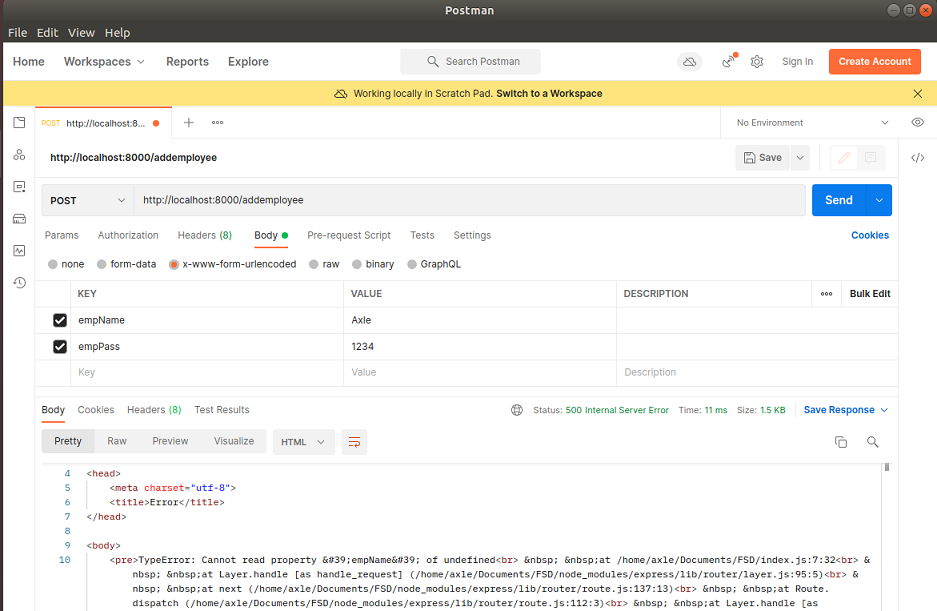
1. Lets add one more field, empPass. After that we can log the results or send them back to the browser using response.end(). I also changed the inner function to an arrow:

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

If your app is shutdown, just start it up again using **node index** from the command line.

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('/addemployee', (request, response)=>{**  **let empName = request.body.empName;**  **let empPass = request.body.empPass;**  **response.end(`POST success, you sent ${empName} and ${empPass}, thanks!`);**  **});**  **//**  **app.listen(port, function(){**  **console.log("Listening " + port);**  **});//** |

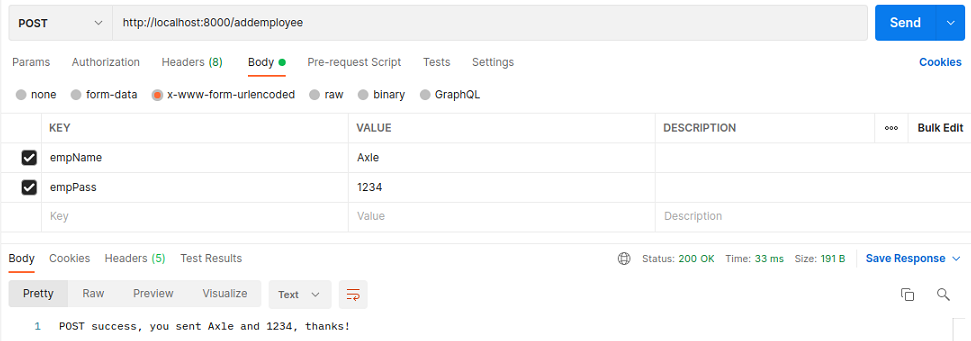
1. Now we have to test this out using a REST client or a plugin for the browser, see below. I will be using Postman for this part. Remember to turn on CORS in your browser:  
   
2. Now we did get an error and its because Express on its own, does not have the capability to handle form fields in JSON format, 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('/addemployee', (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('/addemployee', (request, response)=>{** |

express.urlencoded() is a middleware Express function designed to recognize a posted request object as strings or arrays.

****

The entire index.js file so far

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

**Using arrow functions:**

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

**------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. We now have router to construct routes and the first route is going to be the **root route**. So wherever 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('/addemployee', (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.

|  |
| --- |
| **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('/addemployee', (request, response)=>{**  **let empName = request.body.empName;**  **let empPass = request.body.empPass;**  **response.end(`POST success, you sent ${empName} and ${empPass}, 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('/addemployee', (request, response)=>{**  **let empName = request.body.empName;**  **let empPass = request.body.empPass;**  **response.end(`POST success, you sent ${empName} and ${empPass}, 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('/addemployee', (request, response)=>{**  **let empName = request.body.empName;**  **let empPass = request.body.empPass;**  **response.end(`POST success, you sent ${empName} and ${empPass}, 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('/addemployee', (request, response)=>{**  **let empName = request.body.empName;**  **let empPass = request.body.empPass;**  **response.end(`POST success, you sent ${empName} and ${empPass}, 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.addemployee=function(req, res){**  **res.send('You are on the addemployee route');**  **};**  **//**  **exports.getemployees=function(req, res){**  **res.send('You are on the getdocs route.');**  **};** |

I have just added a new function **getemployees** to do some interacting with the Employees 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 in routes.js.

|  |
| --- |
| **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('/addemployee', (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('/addemployee', controller.addemployee);**  **//**  **router.get('/aboutus', controller.*aboutus);***  **};** |

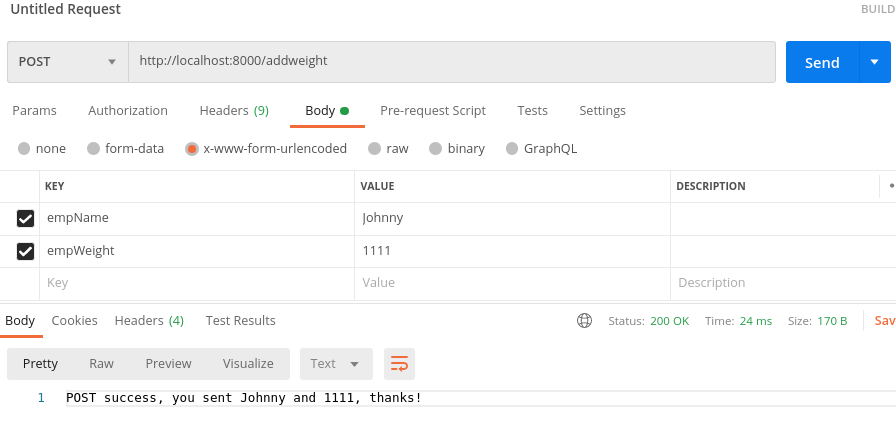
1. Remember that /addemployee is a post method, so let’s change that on both sides, on the **routes** side and on the **controller** side, first the controller. First notice that each controller function has access to the request and response objects:

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

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

1. On the routes side, make sure that the /addemployee route is a POST

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



Here are the controller and routes files:

|  |
| --- |
| routes.js  const controller = require('./../controllers/controller');  module.exports = function(router){  router.get('/', controller.getdefault);  //  router.post('/addemployee', controller.addemployee);  //  router.get('/aboutus', controller.aboutus);  //  router.get('/getemployees', controller.getemployees);  } |

|  |
| --- |
| controller.js  exports.getdefault = function(req, res){  res.send('You are on the root route.');  };  //  exports.addemployee = function(req, res){  let empName = req.body.empName;  let empPass = req.body.empPass;  res.end(`POST success, you sent ${empName} and ${empPass}, thanks!`);  };  //  exports.aboutus = function(req, res){  res.send("You are on the about us route!!!");  };  //  exports.getemployees = function(req, res){  res.send('You are on the getemployees route.');  }; |

**------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.

Note: if you do not have MongoDB installed, install it now using:  
**sudo apt install mongodb**

Assuming that you are on Ubuntu Linux 20+

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

Note: for older versions use **sudo mongo** or **mongod**

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

|  |
| --- |
| **use Employees** |

1. Add a collection

|  |
| --- |
| **db.createCollection("FTEmployees")** |

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

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

1. Enter a record

|  |
| --- |
| **db.FTEmployees.insertOne( {empName : "Joe", empPass : "1234" })** |

1. Verify the record.

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

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

|  |
| --- |
| **db.FTEmployees.insertOne( {empName : "mary",  ", empPass : "1234"})** |

1. Verify the new record

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

1. Lets change (update) Joe’s record:

|  |
| --- |
| **db.FTEmployees.update(**  **{empName : "Joe"},**  **{$set: {empPass : "Joe"} }**  **)** |

1. Verify the change

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

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

|  |
| --- |
| **db.FTEmployees.insertOne(**  **{**  **empName : "Sally",**  **empPass : "1234",**  **Date : new Date()**  **}**  **)** |

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

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

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

|  |
| --- |
| **db.FTEmployees.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 **Employees** 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 employee.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/Employees', { 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 empSchema:

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

Note, if you are using version 6+ of MongoDB you no longer need the object passed as the second parameter to the connect() method. It is greyed out here and in #6 below.

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

|  |
| --- |
| **module.exports = mongoose.model('Employees', empSchema);** |

1. Here is the entire file

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

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

1. Next we will change the **getemployees** function. That function will use the **Employee** variable created above and its attached **find()** method

|  |
| --- |
| **exports.getemployees=function(req, res){**  **Employee.find();**  **};** |

1. The find() method, like almost ALL Mongoose methods, takes an object as the first parameter and then, depending on the version, a function as the second. For a find all, the first parameter object must be blank.

|  |
| --- |
| **exports.getemployees=function(req, res){**  **Employee.find({})**  **.then(**  **employeeData => res.send(employeeData)**  **)**  **};** |

Note: as of version 6+ we are now being forced to interact with the database asynchronously. This means that we either use async/await or chain a .then() method to the find() method.

1. In order to handle any errors we need to chain a .catch() method in addition to the then() method:

|  |
| --- |
| **exports.getemployees=function(req, res){**  **Employee.find({})**  **.then(**  **employeeData => res.send(employeeData)**  **)**  **.catch((err)=>{**  **res.send(err);**  **})**  **//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('/getemployees', controller.getemployees);** |

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



Here is the entire function

|  |
| --- |
| **exports.getemployees=function(req, res){**  **Employee.find({})**  **.then(**  **employeeData => res.send(employeeData)**  **)**  **.catch((err)=>{**  **res.send(err);**  **})**  **};** |

1. We can try to get just one employee. First in **controller** create a controller method called **getemployee**.

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

1. Notice, this method is implying singularity. We can now try to get a single record by passing in the *name* to get in the url .Add a route to the routes.js file

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

Notice the colon and the path name after the path. This will accept any parameters passed by the user into the getemployee() function.

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;**  **};** |

I could have used empName instead.

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

|  |
| --- |
| **let empToFind = req.params.employeeName;**  **Employee.find({empName:empToFind})**  **.then(**  **employeeData => res.send(employeeData)**  **)**  **.catch((err)=>{**  **res.send(err);**  **})** |

Note this is almost exactly the code for the getemployees() function, the only difference is that we passed an object to be searched.

1. Test the code by opening a browser and navigating to [**http://localhost:8000/employees/**](http://localhost:8000/employees/)**Joe**Graphical user interface, text, application, email

   Description automatically generated

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

|  |
| --- |
| **Employee.find({empName:empToFind})**  **.then(**  **employeeData => {**  **if(employeeData.length === 0)**  **res.send("No data!");**  **else**  **res.send(employeeData);**  **}**  **)**  **.catch((err)=>{**  **res.send(err);**  **})** |

1. (Optional) The above will return an empty array if not configured. However we know it is already configured to work with JSON, so lets return JSON if no records found:

|  |
| --- |
| **employeeData => {**  **if(employeeData.length === 0)**  **res.send({"message":"No Data!"});**  **else**  **res.send(employeeData);**  **}** |

The entire getemployee function

|  |
| --- |
| exports.getemployee= function(req, res){  let empToFind = req.params.employeeName;  Employee.find({empName:empToFind})  .then(  employeeData => {  if(employeeData.length === 0)  res.send({"message":"No Data!"});  else  res.send(employeeData);  }  )  .catch((err)=>{  res.send(err);  })  }; |

The entire routes.js file so far:

|  |
| --- |
| const controller = require('./../controllers/controller');  module.exports = function(router){  router.get('/', controller.getdefault);  router.get('/aboutus', controller.aboutus);  router.post('/addemployee', controller.addemployee);  router.get('/getemployees', controller.getemployees);  router.get('/getemployee/:employeeName', controller.getemployee);  } |

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

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

1. In the routes.js file, you should already have a function called addemployee. If not copy any of the previous route lines and change the route to be add a new document.

|  |
| --- |
| **router.post('/addemployee', controller.addemployee);** |

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

1. If you do not have a corresponding function, create a matching function in the controller.js file, in fact we can just copy, paste and edit the **deletebyname()** function.

|  |
| --- |
| **exports.addemployee = 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**.

1. Create a variable called Emp and point it to the **Employee** object, which represents our database. Remember in line 1 of the controller.js file we required the employee.js file:

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

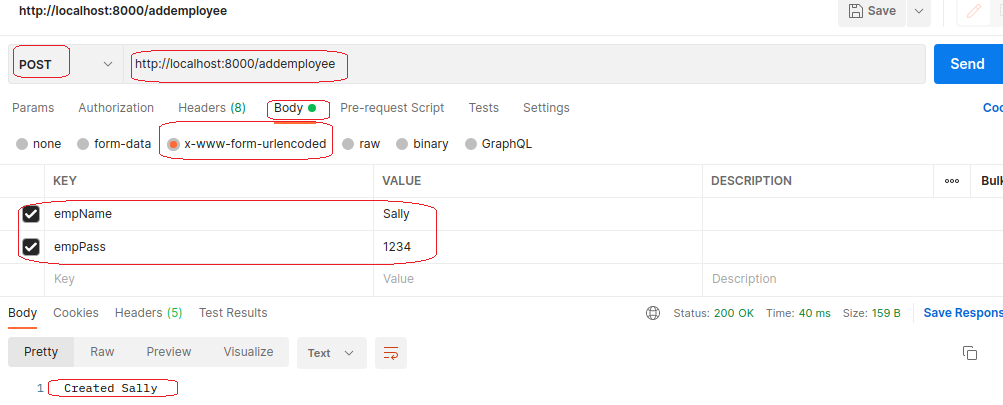
1. Use the new variable, Emp, and its properties to pass values from the form to the database properties

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

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

|  |
| --- |
| **Emp.empName = empName;**  **Emp.empPass = empPass;**  **Emp.save()**  **.then(msg => {**  **res.send({"message":"Created " + Emp.empName});**  **})** |

As usual in an asynchronous operation, you have to add the then() method, pass a parameter to accept any return from the save() method and then call the send() method to pass that value back to the user



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

1. As we did before, you should also chain the catch() method to handle any errors:

|  |
| --- |
| **Emp.empName = empName;**  **Emp.empPass = empPass;**  **Emp.save()**  **.then(msg => {**  **res.send({"message":"Created " + Emp.empName});**  **})**  **.catch(**  **err => res.send({"message":err.message})**  **);** |

In this scenario there is a *message* property attached to the err object

1. The entire addemployee() function

|  |
| --- |
| **exports.addemployee=function(req, res){**  **let empName = req.body.empName;**  **let empPass = req.body.empPass;**  **const Emp = new Employee();**  **Emp.empName = empName;**  **Emp.empPass = empPass;**  **Emp.save()**  **.then(msg => {**  **res.send({"message":"Created " + Emp.empName});**  **})**  **.catch(**  **err => res.send({"message":err.message})**  **);**  **};** |

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

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

1. Create an **updateemployee()** function in the controller.js file:

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

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

|  |
| --- |
| **router.put('/updateemployee', controller.updateemployee);** |

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

1. Get the two values from the HTML form (Postman). Notice that I changed the password field to newPass. This is not necessary but good programming:

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

1. Next we need one of the update methods, here I use findOneAndUpdate(). Check the documentation for each method. For this method, we need to pass two objects, the document we need to find and the field that we want to change:

|  |
| --- |
| **let empName = req.body.empName;**  **let newPass = req.body.newPass;**  **let empToFind = empName;**  **Employee.findOneAndUpdate(**  **{empName:empToFind},**  **)** |

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:

|  |
| --- |
| **let empName = req.body.empName;**  **let newPass = req.body.newPass;**  **let empToFind = empName;**  **Employee.findOneAndUpdate(**  **{empName:empToFind},**  **{$set : { empPass : newPass}},**  **{new : true}**  **)** |

The last object is optional but it will return the new document, with the changes

1. At this point the document will be updated but good programming will need us to verify the results and handle the update in an asynchronous way:

|  |
| --- |
| **l Employee.findOneAndUpdate(**  **{empName:empToFind},**  **{$set : { empPass : newPass}},**  **{new : true}**  **)**  **.then(**  **result => {**  **if(!result)**  **res.send({message : empName + " was NOT updated!"});**  **else**  **res.send({message : "Updated " + empName})**  **}**  **)** |

We must return something to the client. If there is nothing in the result object, then the document was not updated, otherwise we return a message in JSON format.

1. Finally add a catch and return an appropriate message:

|  |
| --- |
| **{new : true}**  **)**  **.then(**  **result => {**  **if(!result)**  **res.send({message : empName + " was NOT updated!"});**  **else**  **res.send({message : "Updated " + empName})**  **}**  **)**  **.catch((err) => {**  **res.send({message : "Update Failed " + err.message})**  **})** |

**------End of part 12-------**

## Appendix A – Simple HTTP Server

* 1. Choose a directory and run: npm install http-server -g
  2. To start the server, find a directory and type http-server .

(note the period signifies that you are starting the server on the current directory that you are in at the moment)

* 1. If the above does not work, you can try installing the server as a dev server
  2. If as a dev server, then your command will be:   
     sudo npm install –save-dev http-server
  3. Then in your package.json file, use the script section to point to that package, so:  
      "scripts": {

"start": "http-server ."

},

* 1. Now from any directory just type npm start

## Appendix B – Linux Commands

|  |
| --- |
| 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** 2. sudo service mongod status 3. sudo service mongod start 4. To kill any process on any port: **fuser -k 8000/tcp** 5. From npm we will be using the latest packages or at least: a. "express": "^4.18.2",   b. "mongoose": "^7.0.1" c. "nodemon": "^2.0.21" |

## Appendix C –Delete From Database

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

|  |
| --- |
| **exports.deleteemployee=function(req, res){**  **let empName = req.body.empName;**  **Employee.deleteOne(**  **{empName:empName}**  **)**  **.then(**  **result => {**  **if(!result)**  **res.send({message : empName + " was NOT deleted!"});**  **else**  **res.send({message : "Deleted " + empName})**  **}**  **)**  **.catch((err) => {**  **res.send({message : "Delete Failed " + err.message})**  **})**  **};** |

1. Add a route for the delete function:

|  |
| --- |
| **erouter.delete('/deleteemployee', controller.deleteemployee);** |

Changes made to the updateemployee() function to make it a delete function:

1. We do not need the password of the document, just the \_id or name
2. The deleteOne() method only needs one object as a parameter, others are optional
3. The messages we send to the user has to be changed as it is now a delete and not an update

**------ End of Appendix C -------**