

## Table of Contents

<b>PART 01 – VERIFY NODEJS</b>	2
<b>PART 02 – BUILDING A SIMPLE NODEJS APP</b>	4
<b>PART 03 – INTRODUCTION TO NODE PACKAGES AND EXPRESS</b>	6
<b>PART 04 – ROUTING BASICS</b>	10
<b>PART 05 – INSTALLING NODEMON</b>	12
<b>PART 06 – DECOMPOSING ROUTES</b>	13
<b>PART 07 – DECOMPOSING CONTROLLERS</b>	15
<b>PART 08 – INTRODUCTION TO MONGODB</b>	18
<b>PART 09 – SETTING UP MONGOOSE</b>	20
<b>PART 10 – EXPANDING THE CONTROLLER FUNCTIONS TO WORK WITH DATABASE</b>	21
<b>PART 10 – EXPANDING THE CONTROLLER TO DELETE FROM DATABASE (OPTIONAL)</b>	25
<b>PART 11 – EXPANDING THE CONTROLLER TO ADD A NEW DOCUMENT TO THE DATABASE</b>	27
<b>PART 12 – EXPANDING THE CONTROLLER TO UPDATE A DOCUMENT IN THE DATABASE</b>	29
<b>APPENDIX A – SIMPLE HTTP SERVER</b>	31
<b>APPENDIX B – LINUX COMMANDS</b>	31

# Day01 Building the APIs – in 12 Steps

## 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**

```
Press ^C at any time to quit.  
package name: (part01)  
version: (1.0.0)  
description:  
entry point: (index.js)  
test command:  
git repository:  
keywords:  
author:  
license: (ISC)
```

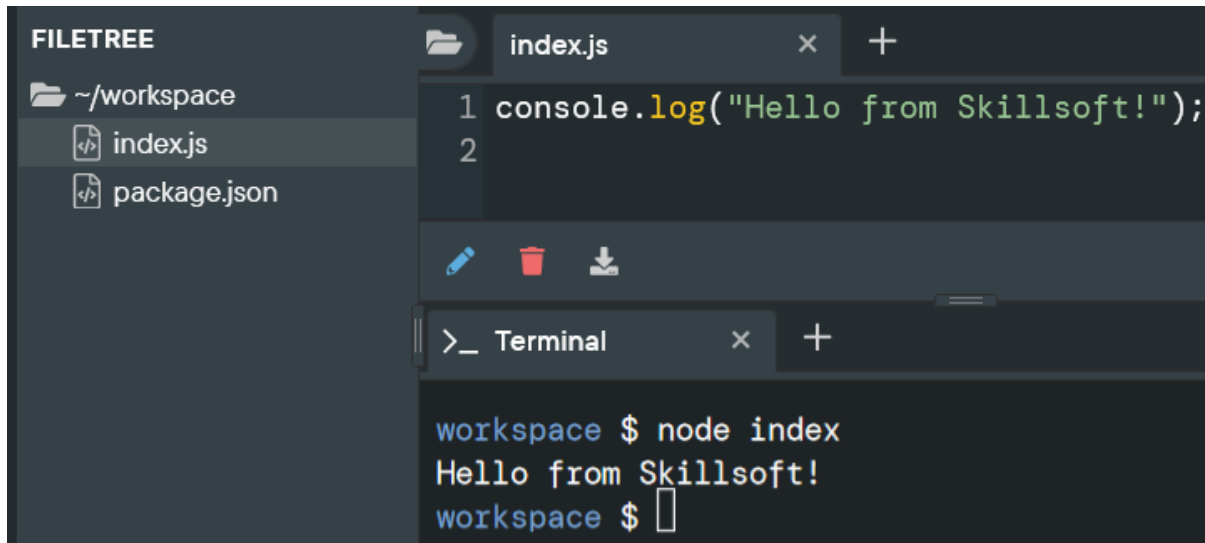
4. According to the .json file, node will look for index.js in order to execute the code inside, so use *touch* to create index.js inside of the **API** folder.
5. Add the following code to execute. This is just to make sure that node is working and it is executing properly.

```
console.log("Hello from Skillsoft!");
```

6. 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.

```
admin2@pc0456:~/Documents/day05/part01$ touch index.js  
admin2@pc0456:~/Documents/day05/part01$ node index  
Hello from Skillsoft  
admin2@pc0456:~/Documents/day05/part01$
```

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



The image shows a code editor interface with a dark theme. On the left is a 'FILETREE' sidebar showing a directory structure: '~ /workspace' containing 'index.js' and 'package.json'. The main editor area has two tabs: 'index.js' (active) and an empty tab. The 'index.js' file contains two lines of code: `1 console.log("Hello from Skillsoft!");` and `2`. Below the code editor is a toolbar with icons for editing, deleting, and downloading. At the bottom is a 'Terminal' window with a prompt `>_`. The terminal shows the command `workspace $ node index` being executed, followed by the output `Hello from Skillsoft!`, and then a new prompt `workspace $` with a cursor.

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

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.

2. 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()`.

3. 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){  
});
```

4. 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();  
});
```

5. 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);
```

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

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

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;
```

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

```
const express= require('express');  
const app = express();
```

5. 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'));
```

6. 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*

7. 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);
});
```

8. 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;
});
```

10. 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:

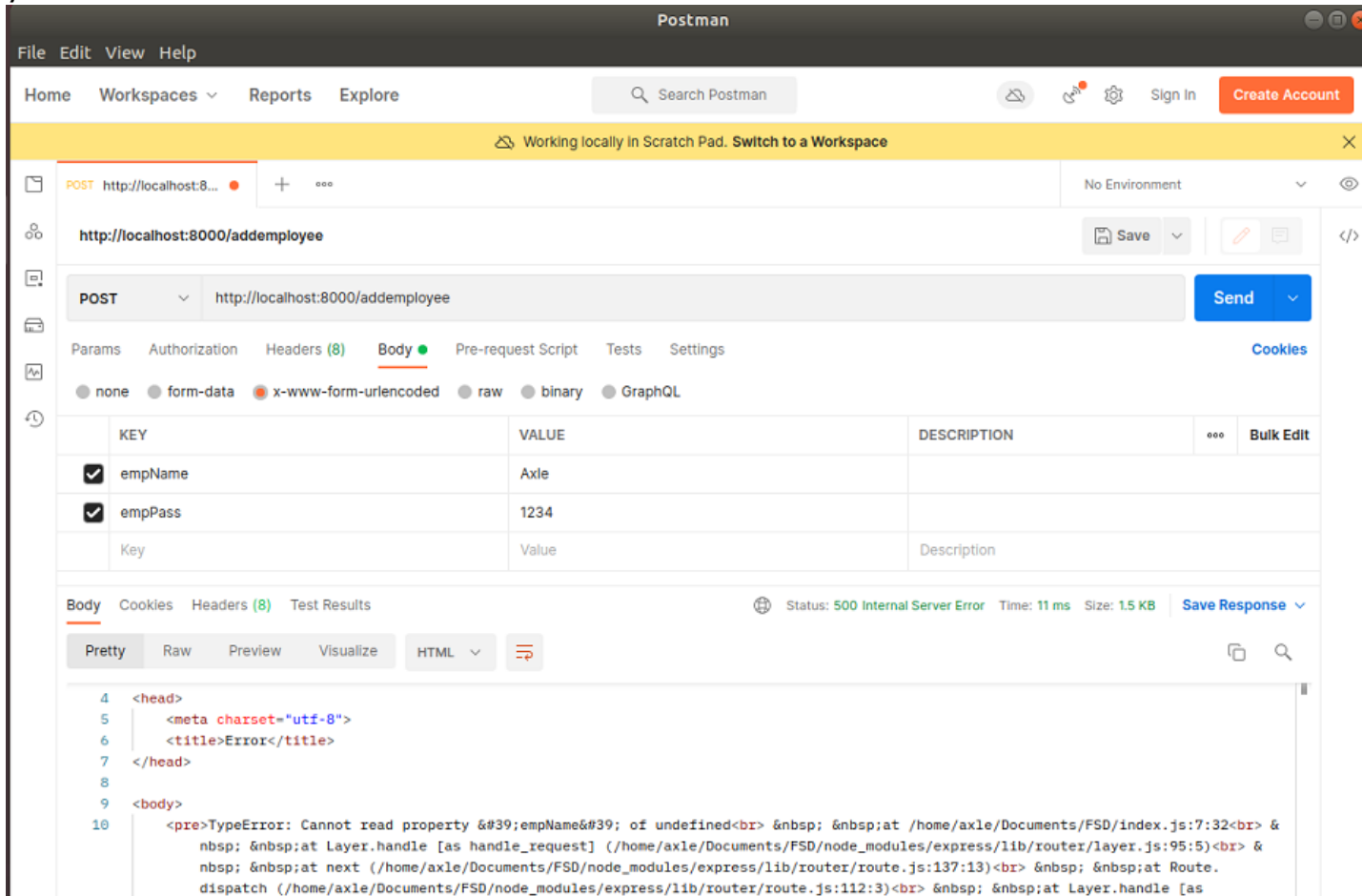
```
app.use(express.urlencoded({extended:false}))
app.get('/', (request, response)=> response.send('hello from skillsoft'));
app.post('/addemployee', (request, response)=>{
  let empName = request.body.empName;
  let empPass = request.body.empPass;
  response.end(` POST success, you sent ${empName} and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

11. 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:



12. 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) => {
```

13. 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.



POST http://localhost:8000/addemployee Send

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/>	empName	Axle			
<input checked="" type="checkbox"/>	empPass	1234			
	Key	Value	Description		

Body Cookies Headers (5) Test Results Status: 200 OK Time: 33 ms Size: 191 B Save Response

Pretty Raw Preview Visualize Text

```
1 POST success, you sent Axle and 1234, thanks!
```

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-----

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}));
```

2. 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;
```

3. 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(){
```

4. Spin the application and go to a browser and everything should work like it did before, only now we are using Router.
5. 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.**

6. 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-----

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`
3. 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.

4. 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`.
5. 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": {
```

6. 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.

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(){};
```

3. 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");
  });
};
```

4. 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");
  });
};
```

5. 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.

6. 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");
  });
};
```

7. 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

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

3. 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.

4. 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'));
```

5. 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)=>{
```

6. 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);
};
```

7. 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:

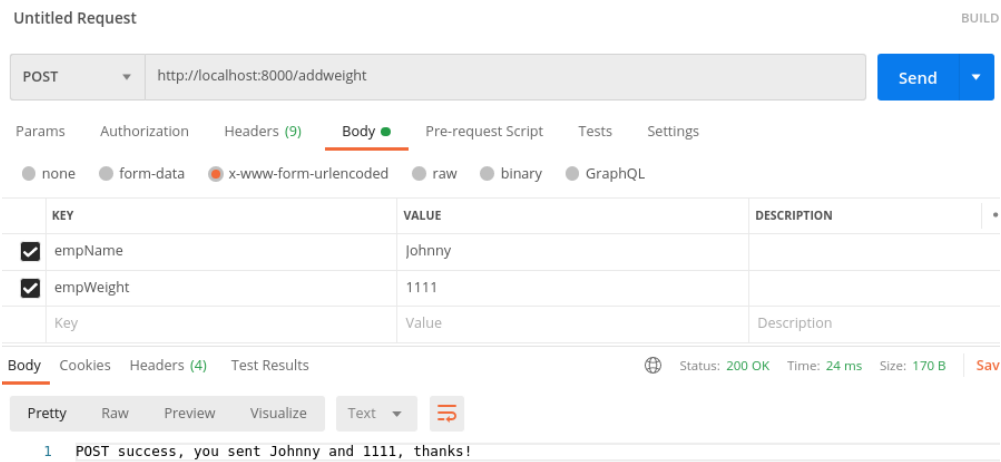
```
exports.addemployee=function(req, res){
  let empName = req.body.empName;
  let empPass = req.body.empPass;
  res.end(` POST success, you sent ${empName} and ${empPass}, thanks!`);
};
```

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

8. 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-----

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 mongo**

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

```
use Employees
```

2. Add a collection

```
db.createCollection("FTEmployees")
```

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

```
db.FTEmployees.find()
```

4. Enter a record

```
db.FTEmployees.insertOne( {empName : "Joe", empPass : "1234" })
```

5. Verify the record.

```
db.FTEmployees.find()
```

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

```
db.FTEmployees.insertOne( {empName : "mary", ", empPass : "1234"})
```

7. Verify the new record

```
db.EmployeeWeights.find()
```

8. Lets change (update) Joe's record:

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

9. Verify the change

```
db.FTEmployees.find()
```

10. 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()  
  }  
)
```

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

```
db.FTEmployees.find().pretty()
```

12. 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-----

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.

2. 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.

3. 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 }
});
```

4. 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'
});
```

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

```
module.exports = mongoose.model('Employees', empSchema);
```

6. 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.');
```

```
};
//
```

2. 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();
};
```

3. . The `find()` method, like almost ALL Mongoose methods, takes an object as the first parameter and a function as the second. For a find all, the first parameter object must be blank. The second parameter, the function, has to handle any result of making this call

```
exports.getemployees =function(req, res){
  Employee.find({}, function(err, results){});
};
```

4. The `find()` method will handle any errors and any returns from the query, so let's expand on it.

```
exports.getemployees = function(req, res){
  Employee.find({}, function(err, results){
    if (err)
      res.end(err);
    res.json(results);
  });
  //res.send('You are on the getemployees 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.

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

```
router.get('/getemployees', controller.getemployees);
```

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

```
▼ 0:
  _id:      "5f3d28d7694d1795e92d97ea"
  empName:  "Joe"
  empWeight: 96.5
  Date:      "2020-08-19T13:30:20.238Z"
  created:   "2020-08-19T14:44:18.734Z"
▼ 1:
  _id:      "5f3d2945694d1795e92d97eb"
  empName:  "Mary"
  empWeight: 65.7
  created:   "2020-08-19T14:44:18.735Z"
▼ 2:
  _id:      "5f3d295a694d1795e92d97ec"
  empName:  "Sally"
  empWeight: 65.9
  Date:      "2020-08-19T13:30:02.506Z"
  created:   "2020-08-19T14:44:18.735Z"
```

Here is the entire function

```
exports.getemployees = function(req, res){
  Employee.find({}, function(err, results){
    if(err) res.send(err);
    res.json(results);
  })
};
```

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

```
exports.getemployee = function(req, res) {  
  };
```

8. 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);
```

9. 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:

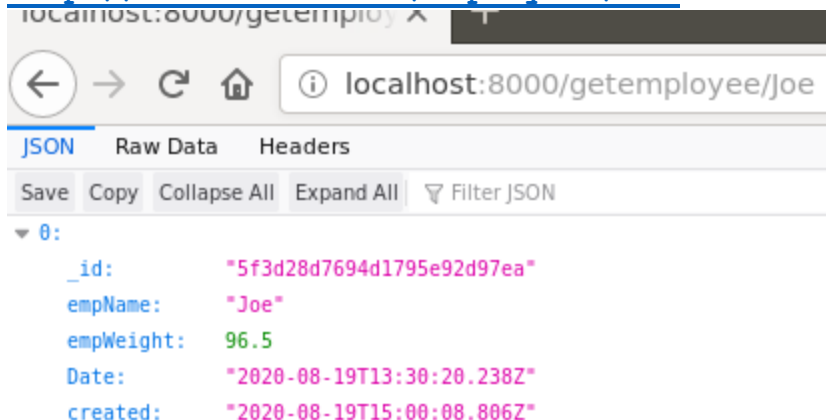
```
exports.getemployee = function(req, res) {  
  let empToFind = req.params.employeeName;  
};
```

I could have used `empName` instead.

10. 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;  
  Employee.find({empName:empToFind}, function(err, results){  
    if (err)  
      res.end(err);  
    res.json(results);  
  });  
};
```

11. Test the code by opening a browser and navigating to <http://localhost:8000/employees/Joe>



Of course you can test in Postman also

12. (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;
  Employee.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);
    }
  });
};
```

13. (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;
  Employee.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))
};
```

The entire getemployee function

```
exports.getemployees = function(req, res){
  Employee.find({}, function(err, results){
    if(err) res.send(err);
    res.json(results);
  })
};
```

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 09-----



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

```
router.delete('/deletebyname', controller.deletebyname);
```

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

2. Create a matching function in the `controller.js` file, in fact we can just copy, paste and edit the `getemployees()` function. Just 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.deletebyname = function(req, res) {  
  let empToDelete = req.body.empName;  
  Employee.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.

3. 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 (record) was not stored with that name/value type of structure, the delete will fail.**

The screenshot shows a REST client interface. At the top, there are tabs for GET and DELETE requests to `http://localhost:8000/deletebyname/Sally`. The DELETE request is selected, and its method is highlighted in a red box. The URL is also highlighted in a red box. Below the request bar, the 'Body' tab is active, showing a table with columns KEY, VALUE, and DESCRIPTION. The response section at the bottom shows a status of 200 OK, and the response body is 'Deleted Sally', which is also highlighted in a red box.

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body: Deleted Sally

**Also notice that no form field values were sent because we are using URL parameters.**

The entire `deletebyname()` method

```
exports.deletebyname = function(req, res){
  let empToDelete = req.body.empName;
  Employee.deleteOne({empName:empToDelete}, function(err, result) {
    if (err)
      res.send(err);
    res.end(` Deleted ${empToDelete} `);
  });
};
```

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

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()`.

2. 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.

3. 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();
```

4. 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;
```

5. 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.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({}, function(err) {
    if (err)
      res.end(err);
    res.end(`Created ${empName}`);
  });
};
```

http://localhost:8000/addemployee

POST http://localhost:8000/addemployee

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

none form-data **x-www-form-urlencoded** raw binary GraphQL

	KEY	VALUE	DESCRIPTION	...	Bulk
<input checked="" type="checkbox"/>	empName	Sally			
<input checked="" type="checkbox"/>	empPass	1234			
	Key	Value	Description		

Body Cookies Headers (5) Test Results

Status: 200 OK Time: 40 ms Size: 159 B Save Response

Pretty Raw Preview Visualize Text

1 Created Sally

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

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

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()`.

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

```
exports.updateemployee= function(req,res){};
```

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

3. Since we did most of what is needed in the `addemployee()` function, just copy paste and change accordingly

```
exports.updateemployee=function(req, res){
  let empName = req.body.empName;
  let newPass = req.body.newPass;

};
```

4. 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.updateemployee =function(req, res){
  let empName = req.body.empName;
  let newPass = req.body.newPass;
  let query = { empName : empName };

};
```

5. 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.updateemployee =function(req, res){
  let empName = req.body.empName;
  let newPass = req.body.newPass;
  let query = { empName : empName };
  let data = { $set : {empPass : newPass } };
};
```

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

```
let data = { $set : {empPass : newPass } };
Employee.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, for example you can change Joe's password to Joe or 1234.

PUT http://localhost:8000/updateemployee

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

none form-data **x-www-form-urlencoded** raw binary GraphQL

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> empName	Joe	
<input checked="" type="checkbox"/> newPass	1234	
Key	Value	Description

Body Cookies Headers (5) Test Results

Status: 200 OK Time: 13 ms Size: 157 B Save Response

Pretty Raw Preview Visualize Text

1 Updated Joe

If you wanted to see what is in result, use this line instead  
`res.end(`Updated ${fixName}, ${result.nModified}`);`

7. Here is the entire updateemployee () function

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

-----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)
3. If the above does not work, you can try installing the server as a dev server
4. If as a dev server, then your command will be:  
`sudo npm install -save-dev http-server`
5. Then in your package.json file, use the script section to point to that package, so:

```
"scripts": {  
  "start": "http-server ."  
},
```
6. 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
2. To create a new directory: **mkdir routes**
3. `sudo service mongod status`
4. `sudo service mongod start`
5. To kill any process on any port: **fuser -k 8000/tcp**
- 6.