

Table of Contents

- PART 01 – VERIFY NODEJS 2
- PART 02 – BUILDING A SIMPLE NODEJS APP..... 3
- PART 03 – INTRODUCTION TO NODE PACKAGES AND EXPRESS 5
- PART 04 – ROUTING BASICS 9
- PART 05 – INSTALLING NODEMON 10
- PART 06 – DECOMPOSING ROUTES 11
- PART 07 – DECOMPOSING CONTROLLERS..... 13
- APPENDIX A – SIMPLE HTTP SERVER 17
- APPENDIX B – LINUX COMMANDS..... 17
- APPENDIX C –DELETE FROM DATABASE..... 18

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**

```
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 inside of the index.js file 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$
```

-----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-----

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`
(Note, you do not have to pass the `--save` flag anymore, but the `--save dev` flag remains)
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 `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:

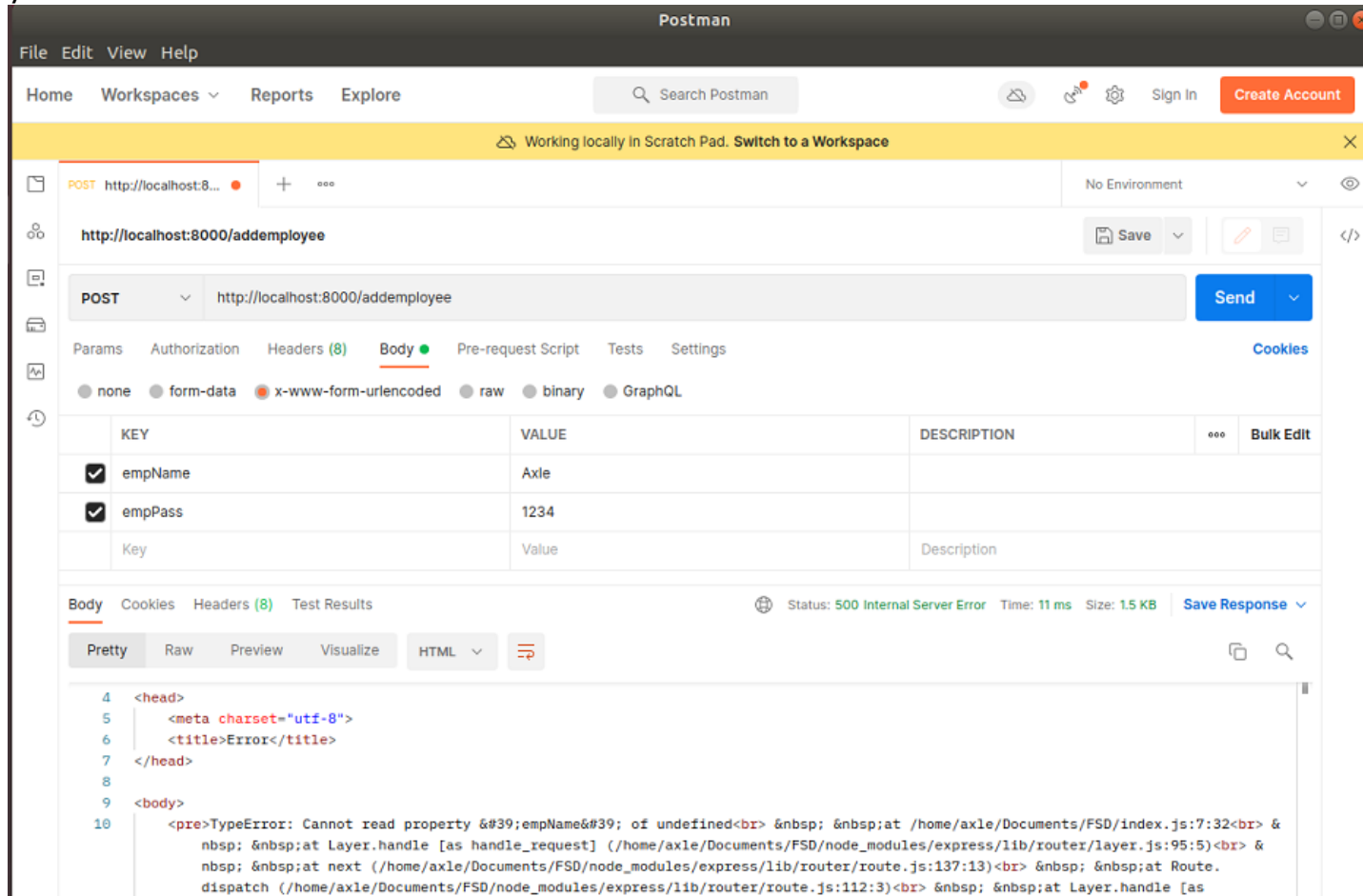
```
app.use(express.urlencoded({extended:false}))
app.get('/', (request, response)=> response.send('hello from skillsoft'));
app.post('/addemployee', (request, response)=>{
  let empName = request.body.empName;
  let empPass = request.body.empPass;
  response.end(` POST success, you sent ${empName} and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



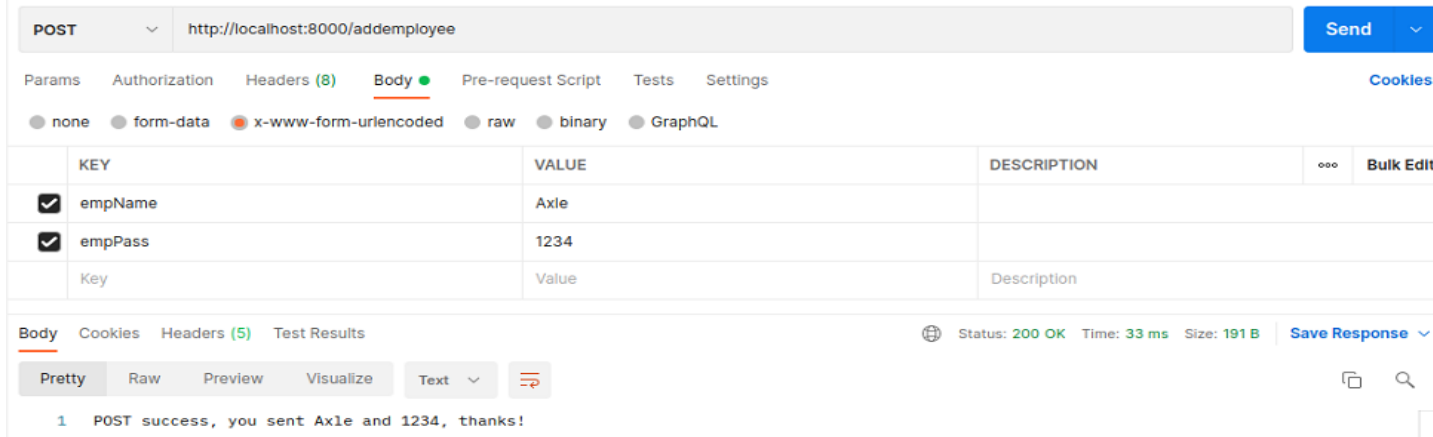
12. Now we did get an error and its because Express on its own, does not have the capability to handle form fields in JSON format, we have to either add a specific package that does this part or use the one that came with Express. We will choose the latter:

```
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.



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}));
```

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-----

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
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.

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(){};
```

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

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

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 getemployees 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);
  //
};
```

Untitled Request BUILD


POST ▼ http://localhost:8000/addweight Send ▼

Params Authorization Headers (9) **Body** ● Pre-request Script Tests Settings

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

	KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/>	empName	johnny	
<input checked="" type="checkbox"/>	empWeight	1111	
	Key	Value	Description

Body Cookies Headers (4) Test Results 🌐 Status: 200 OK Time: 24 ms Size: 170 B Sav

Pretty Raw Preview Visualize Text ▼ 

1 POST success, you sent Johnny and 1111, thanks!

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-----

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. 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})  
  })  
};
```

2. Add a route for the delete function:

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

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

- a. We do not need the password of the document, just the `_id` or name
- b. The `deleteOne()` method only needs one object as a parameter, others are optional
- c. 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 -----