Table of Contents

[**part 01 – Decomposing Routes** 2](#_Toc28857012)

[**part 02 – Decomposing Controllers** 4](#_Toc28857013)

[**part 03 – MongoDB Setup** 6](#_Toc28857014)

[**part 04 – Setting up Mongoose** 7](#_Toc28857015)

[**part 05 – Using the Employee Schema** 9](#_Toc28857016)

[**part 06 –Add a new Document to the Database** 10](#_Toc28857017)

[**part 07 –Add a new Weight** 12](#_Toc28857018)

[**part 08 – MongoDB Nesting** 14](#_Toc28857019)

[**part 09 (Optional) – Server Side Validation with hapi/Joi** 17](#_Toc28857020)

Day02 Building the APIs and Adding a Database

Today we continue building our NodeJS application and add database interaction with MongoDB and Mongoose. We will start by separating out concerns in our NodeJS application. After that we create a MongoDB database and interact with it through Mongoose and it’s schema system. Once we have a schema we can use it to add documents and edit those documents to indicate transactions.

## part 01 – Decomposing Routes

You can copy the Day01 folder and rename it to Day01 or just keep using the same folder from Day01.

1. Create a new root 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(app){};** |

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

1. Next we will CUT the two **get()** functions from our index.js file into this one. Delete the **post()** function, unless you want to try that out as well.

|  |
| --- |
| **module.exports = function(app){**  **app.get('/', function(request, response){**  **response.sendFile('HTML\_STATIC');**  **})**  **//**  **app.get('/getweights', function(request, response){**  **response.send(data);**  **})**  **//**  **};** |

1. Back in the index.js file (the root 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 data = require('../Data/data.json');**  **const path = require('path');**  **const routes = require('../routes/routes');**  **const port = 8000;** |

1. Use the newly created **routes** object to register the Express **app** via it’s constructor. **Do this AFTER the app object is created**.

|  |
| --- |
| **const HTML\_STATIC = path.join(DIR\_DIST, 'index.html');**  **const app = express();**  **routes(app);**  **app.use(bodyParser.urlencoded({extended:false}))** |

1. Since routes.js will now be doing most of the routing, lets require **body-parser** in that file and remove it from the index.js file**.** Do the same for DIR\_DIST and HTML\_STATIC. Remove the corresponding lines from index.js

|  |
| --- |
| **const path = require('path');**  **const bodyParser = require('body-parser');**  **const DIR\_DIST = path.join(\_\_dirname, '../dist');**  **const HTML\_STATIC = path.join(DIR\_DIST, 'index.html');**  **//**  **module.exports = function(app){** |

Note once we do this, the relative paths must also change.

1. CUT the three **app.use()** method lines from index.js and put them inside the function in routes.js

|  |
| --- |
| **module.exports = function(app){**  **app.use(bodyParser.urlencoded({extended:false}));**  **app.use(bodyParser.json())**  **app.use(express.static(DIR\_DIST));**  **app.get('/', function(request, response){**  **response.sendFile(HTML\_STATIC);**  **})** |

8. Routes will need to know about the data being sent to the browser, so cut the data line from index and paste it into routes.js

|  |
| --- |
| **const path = require('path');**  **const data = require('../Data/data.json');**  **const bodyParser = require('body-parser');**  **const DIR\_DIST = path.join(\_\_dirname, '../dist');**  **const HTML\_STATIC = path.join(DIR\_DIST, 'index.html');** |

9. Routes will need its own copy of the Express package

|  |
| --- |
| **const bodyParser = require('body-parser');**  **const express = require('express');**  **const DIR\_DIST = path.join(\_\_dirname, '../dist');**  **const HTML\_STATIC = path.join(DIR\_DIST, 'index.html');** |

10. we no longer need path so remove this line from index.js

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

routes.js

|  |
| --- |
| **const path = require('path');**  **const data = require('../Data/data.json');**  **const bodyParser = require('body-parser');**  **const express = require('express');**  **const DIR\_DIST = path.join(\_\_dirname, '../dist');**  **const HTML\_STATIC = path.join(DIR\_DIST, 'index.html');**  **module.exports = function(app){**  **app.use(bodyParser.urlencoded({extended:false}));**  **app.use(bodyParser.json())**  **app.use(express.static(DIR\_DIST));**  **app.get('/', function(request, response){**  **response.sendFile(HTML\_STATIC);**  **})**  **app.get('/getweights', function(request, response){**  **response.send(data);**  **})**  **};** |

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 02 – Decomposing Controllers

At the moment our routes file is handling logic, we should port this responsibility to a separate file. This way we keep our concerns separate.

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(request, response){**  **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(request, response){**  **res.send('You are on the root route.');**  **};**  **//**  **exports.getweights = function(request, response){**  **res.send('You are on the get weights route.');**  **};** |

This is the entire controller.js file so far, notice the naming conventions, this is a personal choice, but I try to keep the HTTP verbs in the name of the function.

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

|  |
| --- |
| **const path = require('path');**  **const data = require('../Data/data.json');**  **const controller = require('../controllers/controller');**  **const bodyParser = require('body-parser');**  **const express = require('express');** |

1. Before we continue we need to implement **Express** **Router**, this is a special class that works with **Express** to handle routing. Over in index.js file, create a variable that points to the **Router** class of **Express**. Then pass that variable to the **app.use()** function, basically saying that everything on the root route should be handle by **Express** **Router**.

|  |
| --- |
| **routes(app);**  **const router = express.Router();**  **app.use('/', router)**  **//**  **app.listen(port, function(){** |

1. Back in routes.js we can now replace the *get* function with *route*. This allows us to then chain the appropriate **get()** method of the controller file.

|  |
| --- |
| **module.exports = function(app){**  **app.use(bodyParser.urlencoded({extended:false}));**  **app.use(bodyParser.json())**  **app.use(express.static(DIR\_DIST));**  **app.route('/').get(controller.getdefault);**  **app.route('/getweights').get(controller.getweights);**  **};** |

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

1. For now, comment out the line where we tell our app to use **express.static()**

|  |
| --- |
| **module.exports = function(app){**  **app.use(bodyParser.urlencoded({extended:false}));**  **app.use(bodyParser.json())**  **//app.use(express.static(DIR\_DIST));**  **app.route('/').get(controller.getdefault);**  **app.route('/getweights').get(controller.getweights);**  **};** |

8. Run the command npm run dev and test the application in the browser

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

## part 03 – MongoDB Setup

Before proceeding, delete any Weights database that is currently in the system. Use a terminal window. Do a show dbs and all databases will be shown. To delete a db, just use it then issue the command **db.dropDatabase()**.

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

1. Change the database to **Weights**

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

1. Add a collection

|  |
| --- |
| **db.createCollection(“employeeWeights”)** |

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

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

1. Enter a record

|  |
| --- |
| **db.employeeWeights.insertOne( {empName : “Joe”,  empWeight : 95.6 })** |

1. Verify the record.

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

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

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

1. Verify the new record

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

1. Lets change (update) Joe’s record:

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

1. Verify the change

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

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

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

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

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

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

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

1. At this point you may remove the **data** line from routes.js, we will not be using it anymore. It may be line 2.

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

## part 04 – Setting up Mongoose

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

1. We need to install **Mongoose**, so run this command:  
   **>npm install mongoose.**
2. Create a new directory called **models** and touch a new .js file inside of models called employee.js and add the following lines. This directory called **models**, would be in the root folder of the app.

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

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

1. Next we will define the schema.

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

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

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

1. Here is the entire file

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

At this point, run **npm run dev**, just to make sure there are no errors.

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

## part 05 – Using the Employee Schema

We will now incorporate our employee.js file into the **Node** application. Just to make sure everything is working we will introduce functionality to interact with our database. **If you are using the community verion of MongoDB, remember to start the service**.

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(request, response){**  **res.send('You are on the root route.');**  **};** |

Note that **MongoDB** is considered **Middleware**, as such it has access to the **request** and **response** objects of the **Express** app.

1. Next we will expand the **getweights()** function. That function will use the **Employee** variable created above and its attached **find()** method. Delete the **response.send()** function or comment it out.

|  |
| --- |
| **exports.getweights = function(request, response){**  **Employee.find({}, function(err, results){});**  **};** |

This is a pattern that will repeat itself. Whenever we interact with **MongoDB** we interact with its methods, and usually the methods will require an object in curly braces. The second parameter is usually a function that will have access to any **errors** or the **result** of what we are querying. In this case however, we are interacting with the **db** via a **Mongoose** model and we can access data via the **find()** method of the model directly.

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

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

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

1. In the routes.js file, make sure we have a route to match the function. It is this routing that gives the controller file access to the app object.

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

1. Test in a browser, go to: [**http://localhost:8000/getweights**](http://localhost:8000/getweights)
2. You will only get an empty array, but we will insert a document soon. To see the empty array, on Firefox, hit the *Raw Data* tab.

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

## part 06 –Add a new Document to the Database

We can now add a new function **addnewemployee()** in the controller.js file to handle the adding a new employee to the database. In this way when we finally implement getweights() there will be a document to look at.

1. Create the new **addnewemployee()** function just like the other functions. This time we are converting the model exported from employees.js into an object so that we can do things like **Emp.save()**

|  |
| --- |
| **response.json(results);**  **});**  **};**  **exports.addnewemployee = function(request, response){**  **const Emp = new Employee();**  **};** |

1. With our **Emp** object we can directly access it’s properties so add the two lines

|  |
| --- |
| **exports.addnewemployee = function(request, response){**  **const Emp = new Employee();**  **Emp.empName = request.body.empName;**  **Emp.empPass = request.body.empPass;**  **};;** |

1. Now that we have a complete object (date goes in automatically) we can fire the **save()** method on that object and prepare for any errors or results.

|  |
| --- |
| **const Emp = new Employee();**  **Emp.empName = request.body.empName;**  **Emp.empPass = request.body.empPass;**  **Emp.save({}, function(err){**  **});**  **//** |

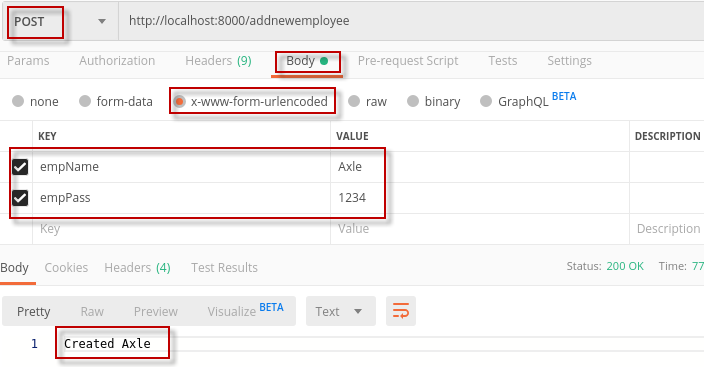
A Mongoose **save()** is converted into a MongoDB **insert()** for a new document in the background. When we do a **save()**, we need the entire object in terms of the database object, an **update()** is handled differently.

1. There are several ways to handle errors and results but for brevity, lets just respond with the error and of course display some message that the post was successfu. Later we will change this to actual JSON.

|  |
| --- |
| **Emp.empName = request.body.empName;**  **Emp.empPass = request.body.empPass;**  **Emp.save({}, function(err, results){**  **if (err)**  **response.end(err);**  **response.end(`Created ${Emp.empName}`);**  **});** |

1. Make sure we have a route to handle this POST request.

|  |
| --- |
| **app.route('/').get(controller.getdefault);**  **app.route('/getweights').get(controller.getweights);**  **app.route('/addnewemployee').post(controller.addnewemployee);**  **//**  **};** |

1. Test the function using Postman  
     
   

**Note:** I am leaving out validation and error handling for now, you can insert it later. Also, the results object, the second parameter of the inner function is full of details that will become important in the final rendition of this app.

1. In your browser go to /getweights, you should no longer see empty [] but the document we inserted

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

## part 07 –Add a new Weight

We can now create a new function **addnewweight()**in the controller.js file to handle the daily transactional activity of adding weights to the employee.

1. For this method you can just copy the first few lines of **addnewemployee()** and change the code accordingly (**controller.js**). Remove the const Emp.

|  |
| --- |
| **exports.addnewweight = function(request, response){**  **let empName = request.body.empName;**  **let empWeight = request.body. empWeight;**  **};** |

1. We can add the two fields we require for the update, **empName** to find the document and **empWeight** to add a new weight

|  |
| --- |
| **exports.addnewweight = function(request, response){**  **let empName=request.body.empName;**  **let empWeight=parseInt(request.body.empWeight);** |

Since the **Mongoose** schema requires a numeric value for **empWeight**, lets ensure we send an integer value to that field, hence **parseInt().**

1. Now we just need to figure out which MongoDB function to use, I will go with **updateOne**() and the $**addToSet** Operator.

|  |
| --- |
| **exports.addnewweight = function(request, response){**  **let empName=request.body.empName;**  **let empWeight=parseInt(request.body.empWeight);**  **Employee.updateOne();**  **//** |

Note that this is an update of an existing document so we do not need a database object here as we did for the **save()** method from Part07.

1. The parameter within the **updateOne()** structure (so everything between the parenthesis) will represent our query and update, it takes several name value pairs but the basic ones are as follows:

|  |
| --- |
| **exports.addnewweight = function(request, response){**  **let empName = request.body.empName;**  **let empWeight=parseInt(request.body.empWeight);**  **Employee.updateOne(**  **{empName: },**  **{$addToSet: },**  **{upsert : },**  **function(err, doc) { }**  **);** |

We just need to determine the values of each of these names/keys, but the $**addToSet** operator takes an object itself.

1. Fill out what you can up to this point:

|  |
| --- |
| **Employee.updateOne(**  **{empName: empName},**  **{$addToSet: { } },**  **{upsert : true},**  **function(err, doc) {**  **if(err) {**  **return console.log(err);**  **} else {**  **return response.send("done");**  **}**  **}**  **);** |

**empName** is the employee whose weight is being added to the db, **upsert** is set to true to prevent duplicates. The last parameter is a function that deals with any errors or the result of transaction we are performing.

1. Finally we come to $**addToSet**. We would need to first create an array in the collection to hold multiple name value pairs and each time we have a new ‘record’ to insert into the document, we run this **addnewweight**() method. So assuming that the structure of the Mongoose collection has a name value pair where the value is an array, then we could do this:

|  |
| --- |
| **Employee.updateOne(**  **{empName: empName},**  **{$addToSet:**  **{employeeWeights :**  **{**  **empWeight:empWeight**  **}**  **}**  **},**  **{upsert : true},**  **function(err, doc) {**  **if(err) {** |

With $**addToSet** we can target the **employeeWeights** collection by its name and the value will be another inner structure consisting of the field of that inner schema and the value we got from our client (HTML form or postman).

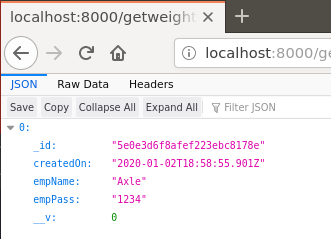
1. Following the pattern so far, if we have a controller function usually there should be a route into this function (not always but usually). Add a new route to your routes.js file but this time it’s a **put()** method.

|  |
| --- |
| **app.route('/').get(controller.getdefault);**  **app.route('/addnewemployee').post(controller.addnewemployee);**  **app.route('/getalldocs').get(controller.getalldocs);**  **app.route('/addnewweight').put(controller.addnewweight);** |

**Note:** I am leaving out validation and error handling for now, we will insert it later. Trying to keep the code simple to understand. **Do not test this as yet, just make sure there are no general errors**.

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

## part 08 – MongoDB Nesting

Before we can test this **addnewweight()** function, we need to re-structure the database in order to handle this new subset of data. The browser result from part06 shows the structure of our collection so far, **\_id, createdOn, empName** and **empPass**.

What we need now is a strategy to accommodate multiple transactions against the collection.

1. If you do not have the MongoDB terminal open, then open a terminal and run the command **sudo mongo** which will get you into the MongoDB shell. Once there switch to the **Weights** database and finally the **employees** collection. Make sure that the **mongod** service is running, if you are on the community version.
2. Back in Atom, in the employee.js file, copy the **empSchema** that is already there, paste it and rename the schema to **weightSchema**.

|  |
| --- |
| **//**  **const weightSchema = new mongoose.Schema({**  **empName: String,**  **empPass: String,**  **created: {type: Date, default: Date.now }**  **});**  **//**  **const empSchema = new mongoose.Schema({**  **empName: String,**  **empPass: String,**  **created: {type: Date, default: Date.now }**  **});**  **//** |

1. Our weight transaction should include the date and the weight numeric value itself, so change the field names accordingly.

|  |
| --- |
| **//**  **const weightSchema = new mongoose.Schema({**  **weighedDate: {type: Date, default: Date.now },**  **empWeight: Number**  **});**  **//**  **const empSchema = new mongoose.Schema({**  **empName: String,** |

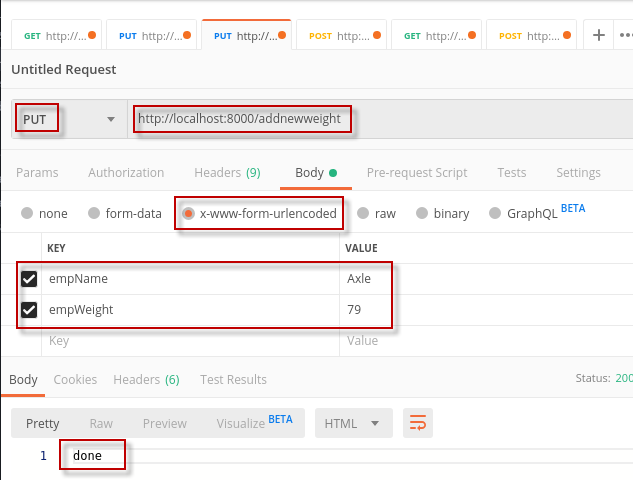
1. Now we need to connect the two schemas, add a reference to the **weightSchema** from **empSchema**

|  |
| --- |
| **//**  **empWeight: Number**  **});**  **//**  **const empSchema = new mongoose.Schema({**  **empName: String,**  **empPass: String,**  **employeeWeights:[],**  **createdOn: {type: Date, default: Date.now }**  **});** |

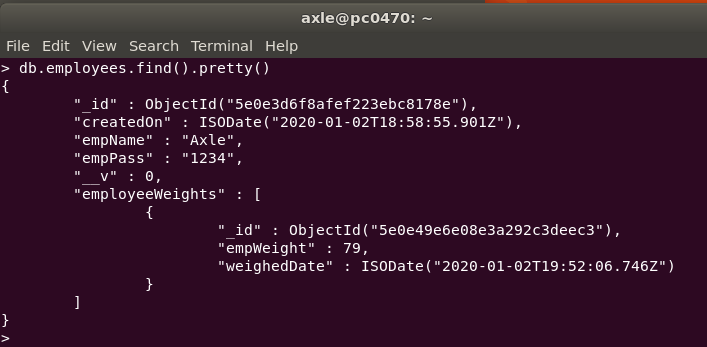
1. Now just pass the **weightSchema** into the array of **employeeWeights**.

|  |
| --- |
| **const empSchema = new mongoose.Schema({**  **empName: String,**  **empPass: String,**  **employeeWeights:[weightSchema],**  **createdOn: {type: Date, default: Date.now }**  **});** |

1. Now we are in a position to add some weights to the document we already have. Fire up Postman and remember that update equates to a PUT method in REST, so your Postman interface should look like this:



1. Using your mongodb terminal run the **find()** method against the **employees** collection



The employeeWeights array will keep expanding as Axle weighs himself each day.

1. Now you can experiment with different employee names and weights. Again I am leaving out several software development details in order for you to understand the core concepts. I also did not do one other CRUD operation and that is delete. By now the keen observer should be able to follow the pattern established and create your own delete method 😊

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

## part 09 (Optional) – Server Side Validation with hapi/Joi

The happi/joi package helps developers to parse any incoming values and validates this data for type and format. This process helps ensure the integrity of the data in our database.

1. Create a new file inside of the controllers folder called validate.js.
2. Install the package so, >**npm i @hapi/joi**
3. Now require the package in the validate.js file:

|  |
| --- |
| **const Joi = require('@hapi/joi');** |

1. Begin to develop a schema for our data by using the following code (comes directly from the website)

|  |
| --- |
| **const Joi = require('@hapi/joi');  module.exports = Joi.object().keys({**  **//schema goes here**  **});** |

We will define our schema between the curly braces, for now just use the two main fields.

1. Next, we will define the schema for two fields

|  |
| --- |
| **module.exports = Joi.object().keys({**  **empName: Joi.string()**  **.alphanum()**  **.min(2)**  **.max(30)**  **.required(),**  **empPass:Joi.string()**  **.alphanum()**  **.min(2)**  **.max(8)**  **.required()**  **});** |

This code should be self explanatory. We do not have to *export as default* as we are using **module.exports**.

1. In the controller.js file we need to implement this **@hapi/joi** package. First require the validate.js file at the top of controller.js.

|  |
| --- |
| **const schema = require('./validate');**  **//**  **exports.getdefault = function(request, response){**  **response.send('You are on the root route.');**  **};** |

1. From #6 above we now have schema that represents the **joi** package. That package has a **validate()** function that will accept the entire form as an object and return a false value if one of those conditions in the schema is violated.

|  |
| --- |
| **exports.addnewemployee = function(request, response){**  **schema.validate(request.body);**  **let empName = request.body.empName;**  **let empWeight = request.body.empWeight;** |

8. What we have to do is pass any error value to a variable that we can then check for errors.

|  |
| --- |
| **exports.addnewemployee = function(request, response){**  **const {error} = schema.validate(request.body);**  **let empName = request.body.empName;** |

1. If the error object is initialized, then we know that we have bad data, we could just do something like this: **if(error) response.end('POST Failed, bad data!');**

However it is better to handle bad data like this:

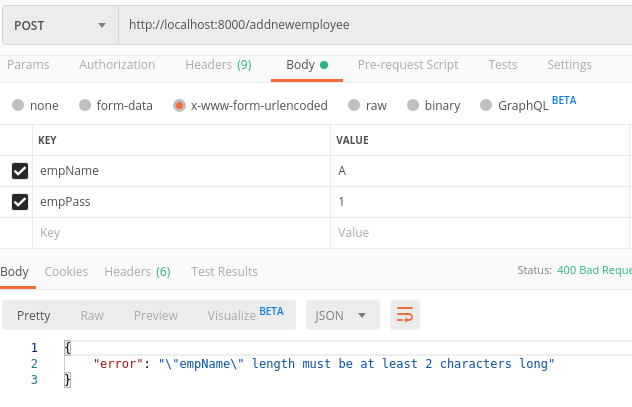
|  |
| --- |
| **exports.addnewemployee = function(request, response){**  **const {error} = schema.validate(request.body);**  **if(error)**  **return response.status(400).json({error:error.details[0].message});**  **let empName = request.body.empName;**  **let empPass = request.body.empPass;** |

In this way, we return the proper status as well as the exact field that has the bad data in it. Notice also that we *return from* the function, so **nothing** after the return statement gets executed if we have bad data.  
  
10. At this point we can send our data to the database, but first we have to setup the database and have it ready to receive data from the Node application.

|  |
| --- |
| **if(error)**  **return response.status(400).json({error:error.details[0].message});**  **let empName = request.body.empName;**  **let empPass = request.body.empPass;**  **//we can now send empPass and empName to the database** |

1. Here is the entire function

|  |
| --- |
| **exports.addnewemployee = function(request, response){**  **const {error} = schema.validate(request.body);**  **if(error)**  **return response.status(400).json(**  **{error:error.details[0].message}**  **);**  **const Emp = new Employee();**  **Emp.empName = request.body.empName;**  **Emp.empPass = request.body.empPass;**  **Emp.save({}, function(err){**  **if (err)**  **response.end(err);**  **response.end(`Created ${Emp.empName}`);**  **});**  **};** |



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

**At the end of today you would have a NodeJS server application and database connection that will become the environment you need to build a ReactJS client application. From this point on there will be two development tracks, one for Node and one for React.**