**Web Server Basics: Express CRUD Routing I**

In this blog post, we’ll be doing the same thing we did in **our previous blog**, but this time by configuring our own express server.

To keep things simple, we’ll be using an array of objects as our database. Because of this reason, you’ll be able to perform only: get and post operations. As patch/delete operations are heavily DB dependent, we’ll reserve them for later (when we will be using a real database).

So let’s get started.

**Prerequisite**

Npm package called nodemon. It helps with live server reloads (or restarts). Meaning you don’t have to execute node index.js every time you want to restart your server. This happens on it own when you save index.js file.

<video clip goes here…>

**Creating Our Express Server**

By now you should know this well. Anyway, here’s the video clip for reference. Setup the server to run at port 3000.

<video clip goes here…>

**Setting Up Our Simple Database**

A simple array of objects will replace JSON server as our database this time.

**Note:** Don’t worry we will eventually get to using a real database. But when it comes to programming, it’s best to focus on one thing at a time. So an array of objects within our index.js will do for now.

So add the following code to your index.js

<embedded code goes here>

As you can see, each object (or record since it’s basically a database) contains a name, designation and id property. And since we are trying to store employee related information, we name our little database as ‘employees’.

This data arrangement is an exact replica of what you will see if you were using mongodb. Only the naming conventions would change. [The array would be called Collection and the objects as documents. More on MongoDB, here: read [this]].

**Note:** To know more about JavaScript array of objects, read [this]

**Setting Up Our GET Route**

Our goal is to create a ‘/employees’ route or endpoint that will ‘get’ and display all the records available in our database. [In other words, all the information we just stored in our employees array].

How do we do this? By making use of app.get() method configured to serve ‘employees’ array at the ‘/employees’ route.

So setup a app.get() method as shown in the video clip below.

<video clip goes here…>

**How to Send the Data to the Browser?**

So far we made use of res.send() method for our responses. Now we’ll use res.json() method, as res.send() is usually reserved for only html based messages.

**Note:** Though res.send() works fine in our case, it’s ideal to use res.json() for sending across array based information [more on res.json() here]

So add the following line of code in your app.get() method.

<cut to video clip showing the same….>

Save index.js file and restart your server using node index.js command.

**Accessing Our GET Route**

<video clip goes here…>

Try hitting the endpoint http://localhost:3000/employees) in the browser and our database information will get displayed.

Great! You just configured an express GET route.

**Setting Up Our POST Route**

Our goal is to create another ‘/employees’ route or endpoint. This route is going to help us ‘post’ data to our server.

How are we going to do this? Yes, by making use of app.post( ) method.

And since we don’t have a frontend yet just like in our previous blog, we’ll be making use of Postman for posting the data.

So let’s setup our post route as shown above.

<video clip showing app.post scaffolding code…>

**Read Posted or Incoming Data Using express.json()**

Before we post the data to our ‘/express’ endpoint, we have to enable our server so that it can read the data being posted. So, just like **we did in our previous blog**, we have include the express.json( ) middleware in our index.js file.

<video clip indicating middleware inclusion goes here….>

Now that our server is ready to read the data being posted to its post route, let’s try sending over the same data using Postman.

**Posting Data To Our Server Using Postman**

Ensure you select ‘POST’ method from the dropdown and select raw option and select JSON(application/json) option as shown in the video clip. Also, ensure the data you send across follows the data pattern shown in our database (that is: name, designation, id).

<video clip showing the send operation>

Postman will try sending across the data to our server but fail (indicated by the loading… message). This is because our post route is empty at the moment. In other words, the logic to capture the incoming data is missing. So let’s create it now.

**Creating New Object for Capturing Posted Data**

Now since our database is an array of objects, we have to create a new object (to store the data being posted) every time the data gets posted to our route and then push it into our database array.

Just like in our previous blog, we have to make use of request body object (req.body) to capture the posted data.

**Note:** The new object we create should have the same keys (that is, name, designation and id) as the records in our database. After all, it’s going to be pushed into our database.

<image of code containing only the object keys…>

**Capturing Data Using req.body**

Just like in our previous blog, we have to make use of request body object (req.body) to capture the posted data (and assign it to the appropriate keys of our newly created object).

Once the incoming data is captured, next is to push the newly created object into our employees’ array.

<code goes here…>

**Checking Whether Our Database Got Updated**

**IMPORTANT:** Since our database is just an array of objects. We won’t be able to persist any data in it. [Once our server connection is closed, the data will be lost].

But temporarily, our data does get stored in the employees array and it can be accessed only by hitting the GET route (‘/employees’) endpoint and not in the index.js file (like we did with the json server).

So you just have to access our GET route (‘/employees’) to view our updated database.

<video showing the same…>

**Accessing Individual Record Using GET Route**

Just like with our previous blog, we’ll also try and get individual data using our GET route. But since we are not using a real database, we’ll have to make some tweaks to our code.

**Using req.params.id Object**

Now since we are dealing with the request object and need to pass a parameter (the id of the record being fetched) to get access to a specific record within the database, we need to make use of req.params method.

**What is a params in req.params.id?**

params stands for parameters. Simply put, parameters are variables you pass to the URL.

They are of 2 types: URL parameter and query parameter. The one we are dealing with is a URL parameter (since it’s being passed to the URL directly). More on parameters here.

**What Does req.param.id Do?**

This piece of code is basically responsible for capturing whatever that’s passed after a route (and deliver the information belonging to the value captured).

For example:

Value of the /:id in <http://localhost:3000/employees/:id>

**Note:** Make note of the syntax. That is, the : before the id.

In our case, req.params.id is needed to capture the id of the record (or object) we want to fetch, so that the relevant record information can be extracted (using the id obtained with req.params.id) and displayed.

**Note:** We are duplicating database behaviour here. With databases, you need ids to fetch relevant record.

Let’s create our individual record fetching app.get() route.

<video showing app.get( ) scaffolding code here>

Now try console logging (or res.sending) id variable by hitting the endpoint: localhost:3000/employees/:id. You’ll notice that whatever you type in the URL after the employee/ route gets console logged.

**Using Array Index as Our Record ID**

Now IDs within our temporary employees database array are dummy. They do not successfully identify the record and extract information for them. Because of this, we are going to make use of array indexes instead.

**Note:** More on array index here

Copy/paste the following code in your index.js.

<code goes here…>

You’ll notice we are simply sending the employee object information over. And the [id] value obtained using req.params.id is going to serve as the array index for fetching the relevant record.

More on accessing objects within array of objects here.

Now try passing an array index of our employees array and see what happens.

<video clip goes here…>

The data belonging to the passed array index gets displayed on the browser.

There you go! We just created a temporary/makeshift database and used app.get( ) and app.post( ) methods to access our data!