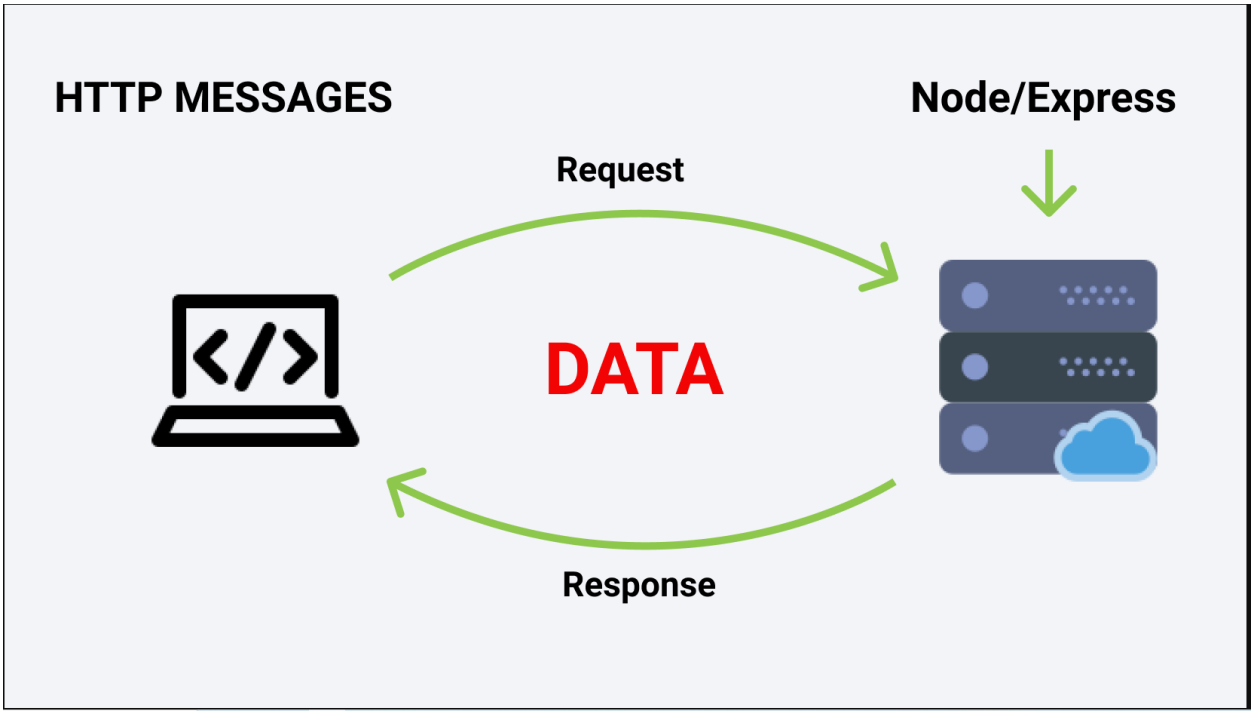
Express

HTTP Request/Response Cycle

Every time we open the browser and type the URL, we are performing a request to server that is responsible for serving a response. Now this is done using HTTP protocol and these are called HTTP messages.



Suer sends a HTTP request message and then server sends an HTTP response message and that's how we exchange data on web.

We mostly use Node but in order make our work easier we use a framework named Express JS. A server job is always to make a resource available. A server doesn't have an GUI (Graphical User Interface). Cloud is nothing but a bunch of servers and computers connected.

HTTP Messages

Let's see how HTTP messages are structured.

General structure for both messages (request and response) is similar.

They both have a start line, they both have optional headers, a blank line that indicates that all meta info has been sent and effectively headers are that meta info as well as optional body.

Request Messages – messages sent by a user.

Response Messages – messages sent by a server.

In General, when we talk about **request message** in start line there's going to be a method, then URL and then HTTP version as well.

Methods is the place where we communicate what we want to do.

> Ex: If we want to get the resource then we set it up as GET request.
>
> If we want to add the resource, then we set it up as POST request.
>
> ***GET request is the default request that the browser performs (since we open the browser and get some request from web, hence GET is the default request).***

URL is just the address. (Ex: freecodecamp.org)

Headers is essentially optional; it is meta information about our request. Headers have a key-value pair. We don't need to add headers manually but in few in cases we need to add headers. (Basically, it an information about our message).

Body, if we just need the data from resource then there is no body but if we want to add a resource to the server then we are expected to provide a body and that is called request payload.

When we talk about response message, the Node JS developers will be creating the response. Start line has the HTTP version, then we have a status code and status text.

HTTP version – it is mostly going to be 1.1

Status Code, it just signals what is the result of the request.

Ex: Status Code: 200 – Request was successful.

Status Code: 400 – There was an error in the request.

Status Code: 404 – Resource was not found.

Headers, we provide info about our message. (It is a setup of key value pairs).

Content-Type: text/html; we are sending back the html.

Content-Type: application/json: we are sending back the data.

When we communicate with API, mostly we are getting back the JSON data because over the web effectively we just send over the string.

In our headers we indicate that we are sending the data in application/json and then that application (web application) which is requesting knows that they are receiving application/json from the server.

Starter Project Install

Clone projects from https://github.com/john-smilga/node-express-course

Starter Overview

Express is built on top of Node and specifically built on HTTP module.

Follow the steps below to setup a express-tutorial project.

1. Create a new folder 02-express-tutorial
2. Run command npm init -y // to create a package.json file
3. Run command npm install // to install node modules
4. Run command npm install express // to install express package into our project
5. Run command npm install - - save-dev nodemon // to install nodemon as a dev dependency package
6. Create a .gitignore file and add node modules // this will make sure we don't push node modules to github.
7. Create app.js file and write some console.log statement
8. Edit the scripts object in package.json file and write the key start with value of nodemon app.js

HTTP Basics

In HTTP Protocol, we transfer data over the web using HTTP Request message and HTTP response message. We create a server using HTTP module and start the server. When we start the server, we try to listen to the requests by using a port number.

Port is a communication endpoint.

There are lot of port numbers,

Port Number 20: Used for File Transfer Protocol (FTP) Data Transfer.

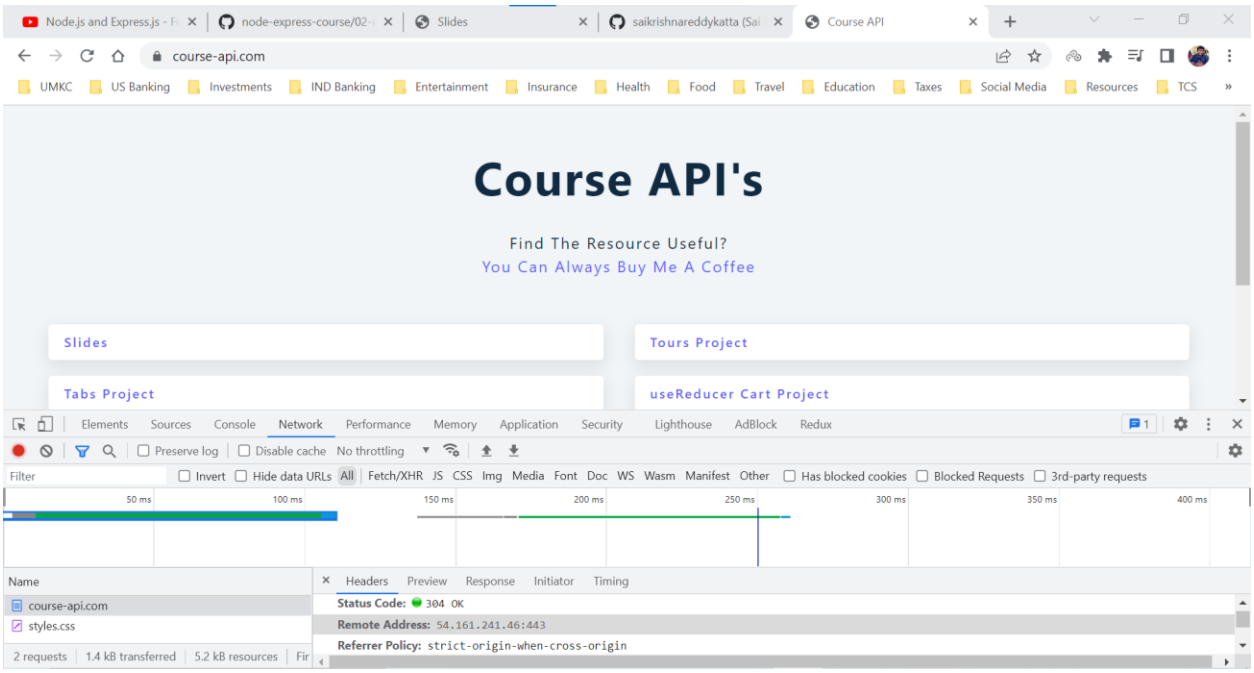
Port Number 21: Used for File Transfer Protocol (FTP) Command Control.

Port Number 80: Used for Hyper Text Transfer Protocol (HTTP) used in the world wide web.

Port Number 443: HTTP Secure (HTTPS) HTTP over TLS/SSL.

As of now in development phase we are using port number 5000 but once in production, we may use 80 or 443.

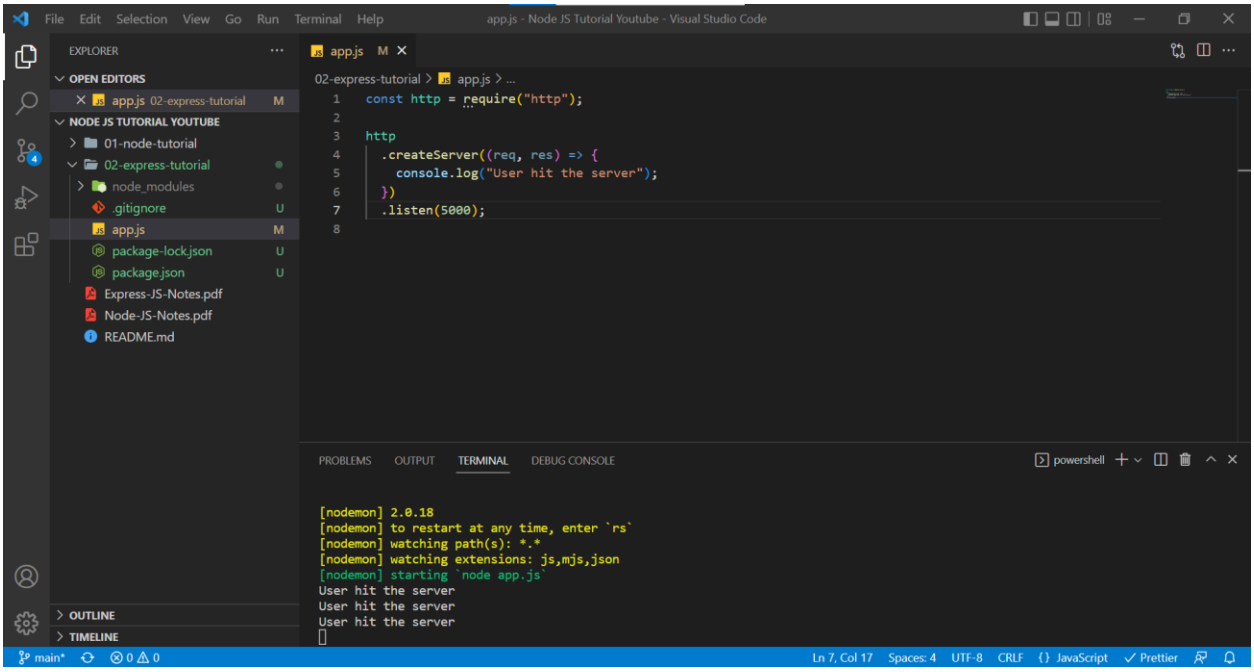
For course-api, we can see the port number (443) in Remote Address field which also contains the IP address.



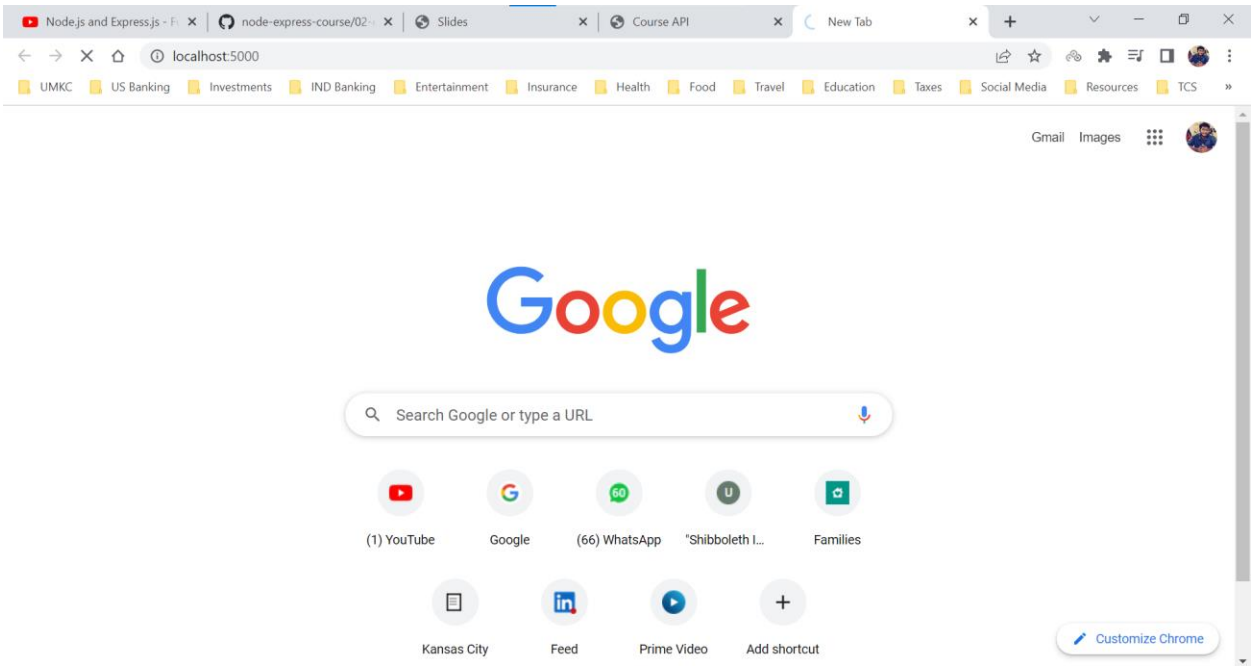
While in development we can use any port number, but 0 – 1024 port numbers are already taken.

Ex: React uses Port Number 3000, Gatsby uses Port Number 8000, Netlify CLI uses Port Number 8080.

When we don’t send any response to the server. We are just console logging the information.



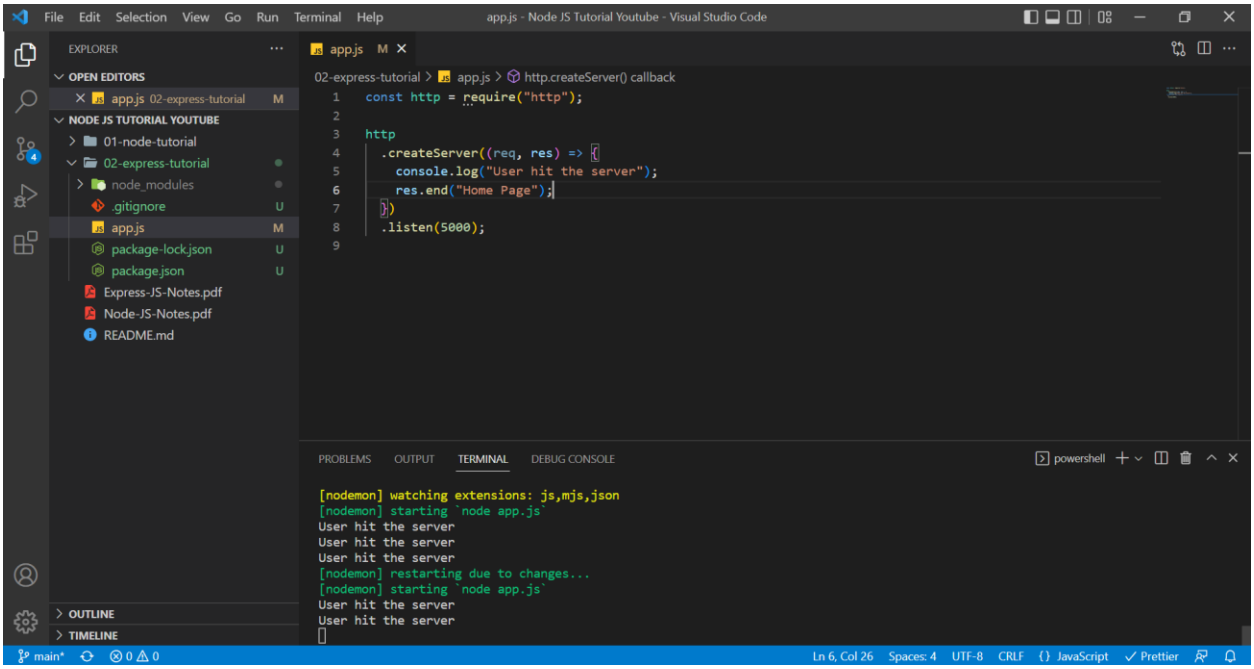
Server is waiting for the response, and it is still loading.



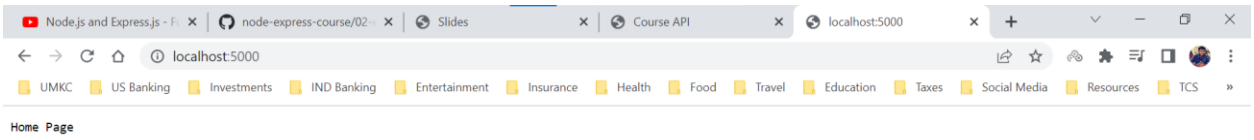
response.end() – this method signals server that all the response headers and body have been sent, that server should consider this message complete. This method response.end() must be called on each response.

createServer() method contains a callback which is invoked every time user hits the server and as parameters to the callback function, we have request and response objects.

app.js



and on localhost:5000



HTTP Headers

We have two major issues with our current setup.

1. We don't send any information about the data that we are sending back. (We are sending any metadata about the response body we are sending back). As of now we are just sending back a string.
2. We are not sending data based on the request by user. Whatever may be the request we are sending only one response which is a string.

We use ***response.writeHead()*** method to write headers/ provide meta data to browser about the response we are sending back to the browser. ***response.writeHead()*** contains a status code and a headers object (which contains properties like content-type etc.) Browser renders the content of page based on property content-type. We also can add the status text (which is optional).

We use ***response.write()*** to send the response body to browser.

We use ***response.send()*** to tell the browser that the message is complete, and we have sent all the required data.

There are many status codes, and it is important to send the correct status code back to the browser.

- 100 – 199 -> Informational Responses
- 200 – 299 -> Successful Responses
- 300 – 399 -> Redirection Messages
- 400 – 499 -> Client Error Responses
- 500 – 599 -> Server Error Responses

MIME types – A media type also known as ***Multipurpose Internet Mail Extensions*** indicates the nature and format of a document, file, or assortment of bytes.

A MIME type most-commonly consists of just two parts: a type and a subtype, separated by a slash (/) — with no whitespace between.

Ex: type/subtype.

An optional parameter can be added to provide additional details.

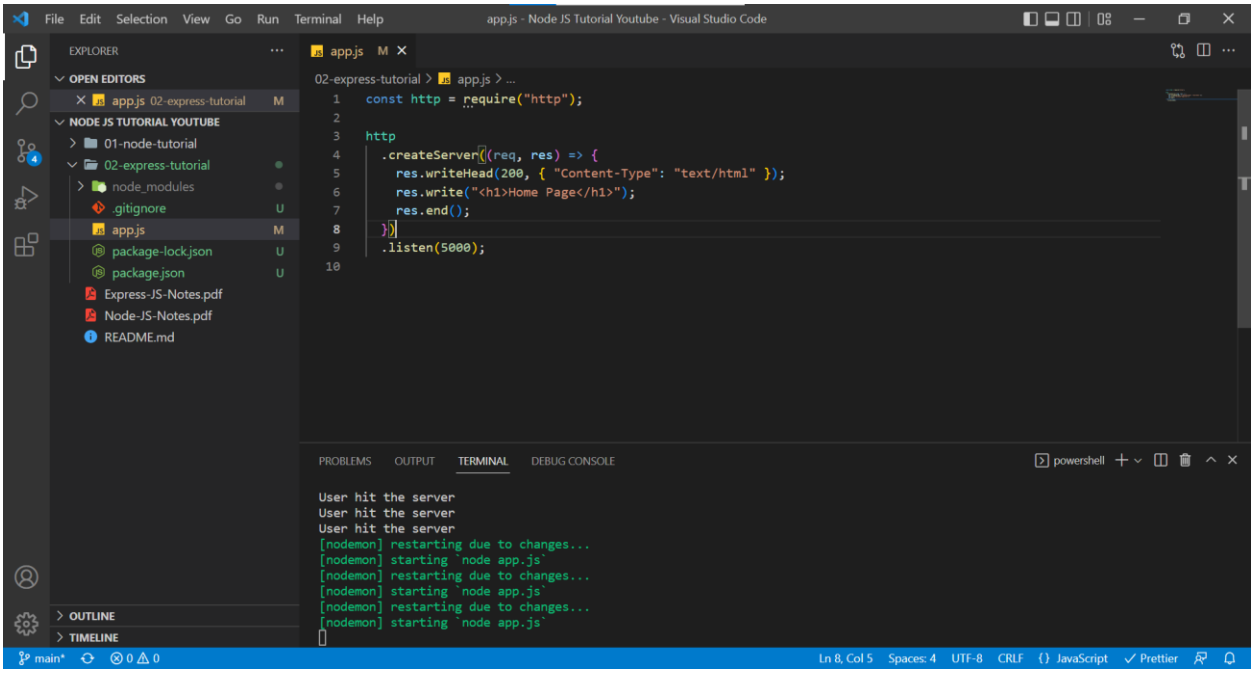
Ex: type/subtype; parameter = value

We use the MIME types to declare the type of data we are sending back to the browser. Types of MIME are

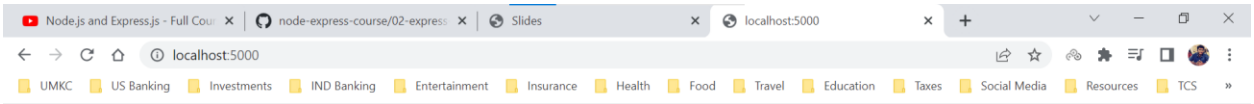
- application/octet-stream – This is default for binary files.
- text/plain - This is the default for textual files. Even if it really means "unknown textual file," browsers assume they can display it.
- text/css - CSS files used to style a Web page must be sent with text/css.
- text/html - All HTML content should be served with this type.
- text/javascript - Per the current relevant standards, JavaScript content should always be served using the MIME type text/javascript.

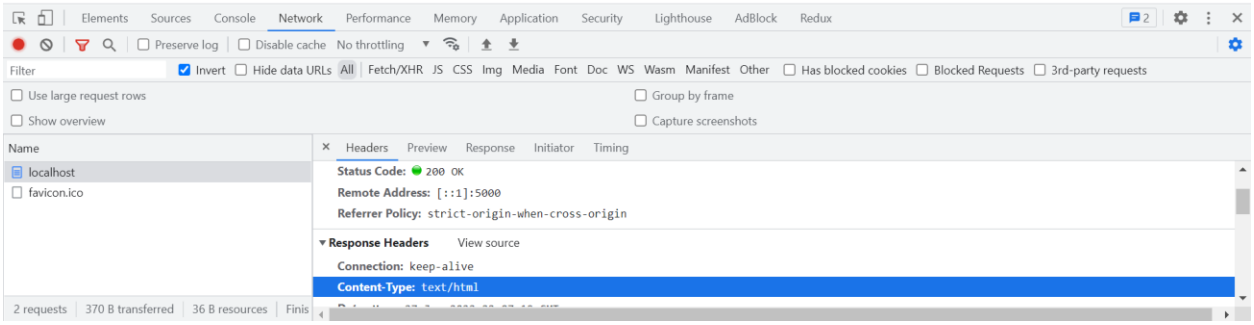
Express will take care of Headers, but we are learning here for Node JS.

app.js

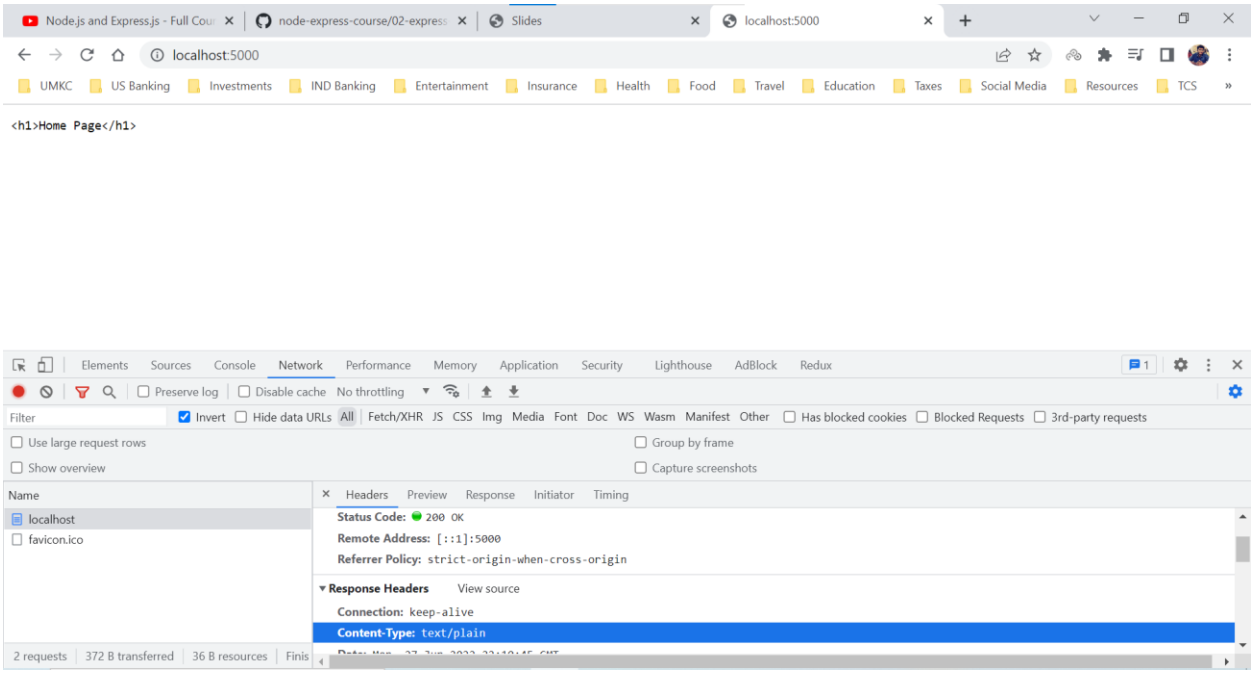


When Content-Type is text/html and Status code is 200





When Content-Type is text/plain and Status code is 200



Browser interprets the type of format and then renders the screen.

HTTP Request Object

Now, let us deal with the request object.

In the request object, we receive HTTP method, URL, HTTP Version, Headers, and Body (which is optional). Since we receive request body from the browser, we need to extract the properties and then send response to browser based on the request by the user.

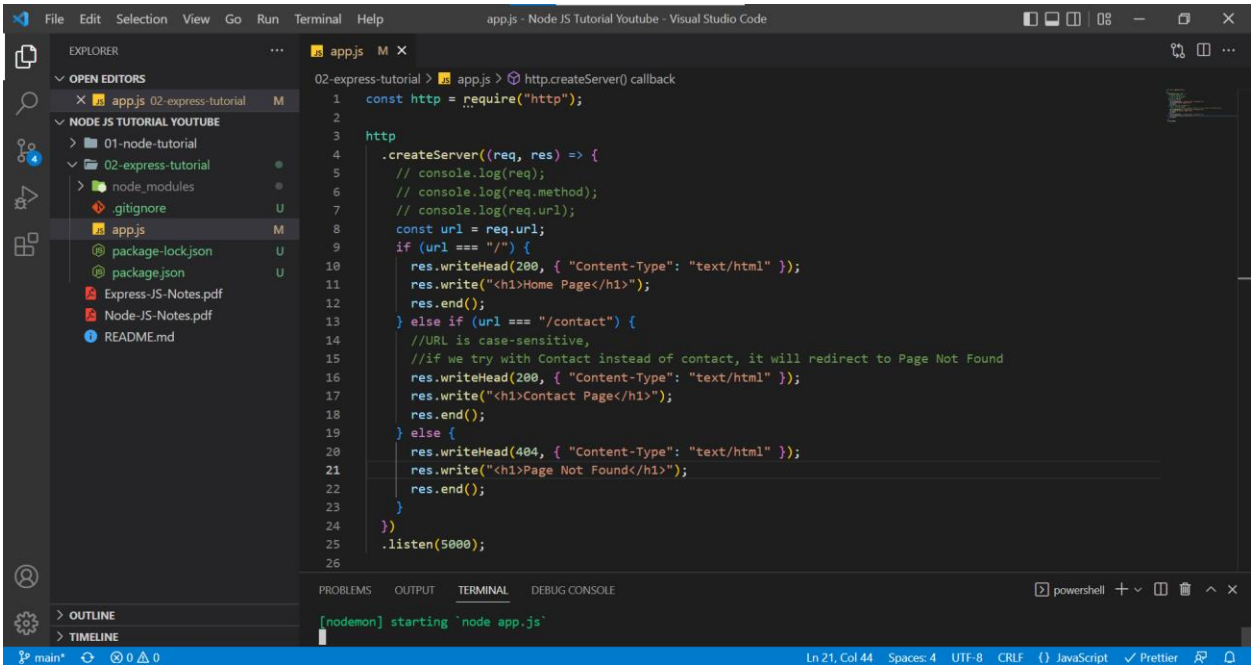
request.method – it is one of the properties which provides information about the HTTP method.

request.url – it is one of the properties which provide information about the URL.

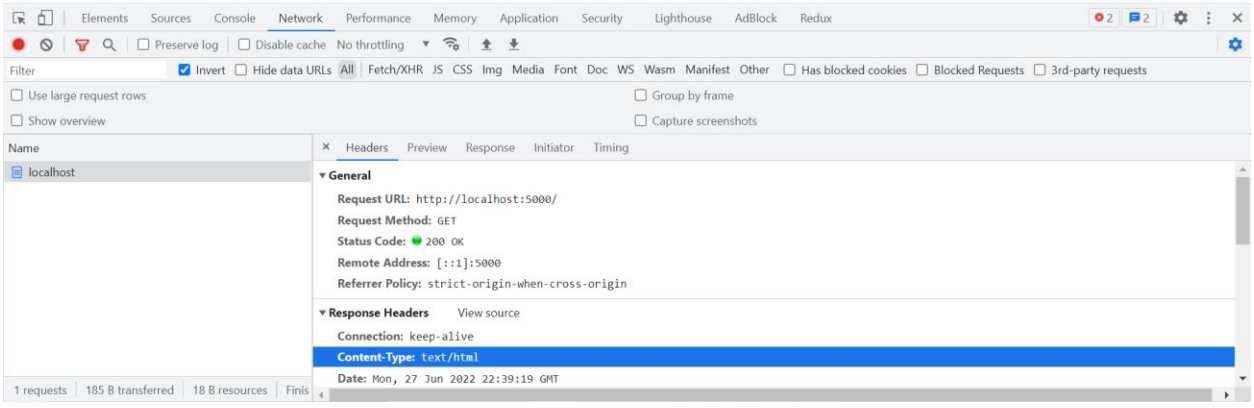
Forward Slash would be the Home Page (“/”).

If we want to Contact page resource, then URL can be (“/contact”).

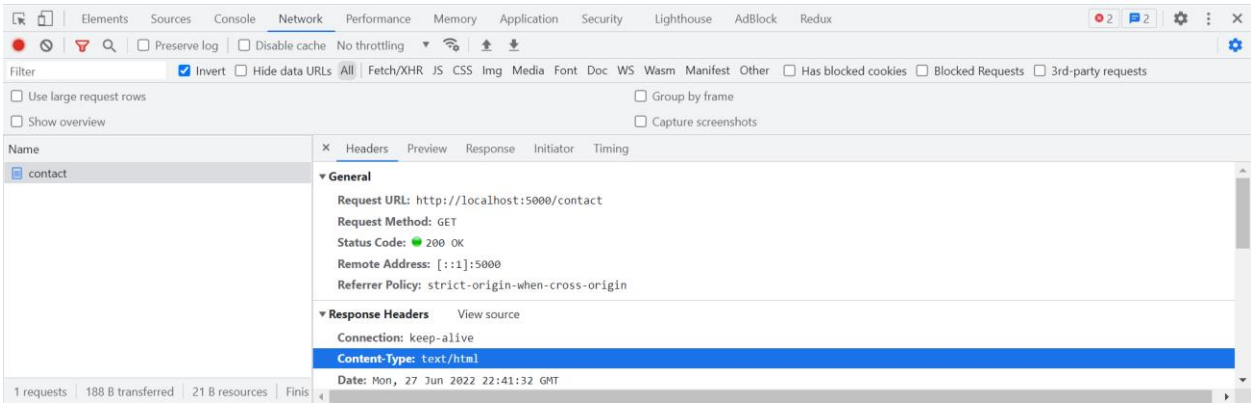
app.js



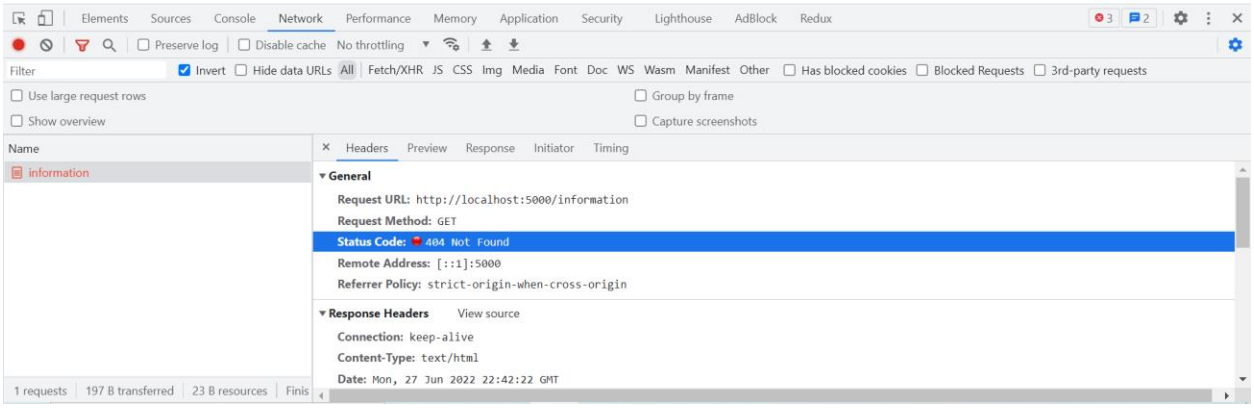
Home Page



Contact Page



Information Page, we are not holding the resource for Information Page.



HTTP – HTML File

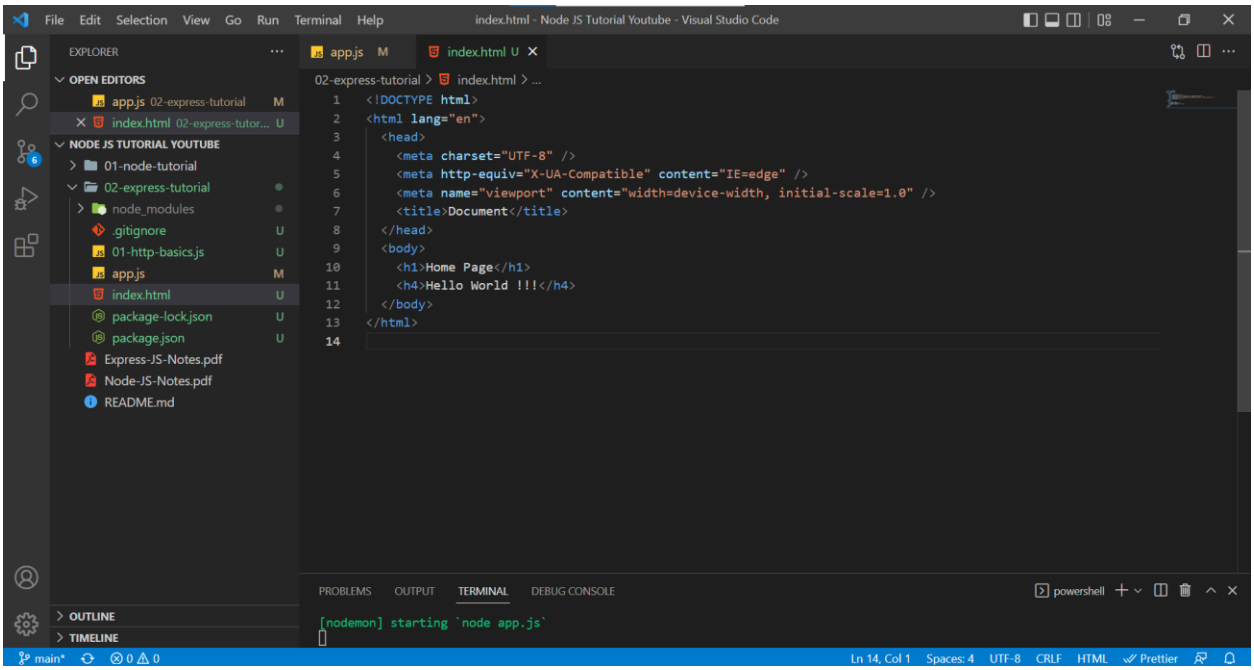
We are not limited to send the HTML directly into ***response.write()*** method or ***response.end()*** method. Instead, we can set up a file, request the file using ***File System*** and just passing it.

Remember that we are passing in the contents of the file not the entire file.

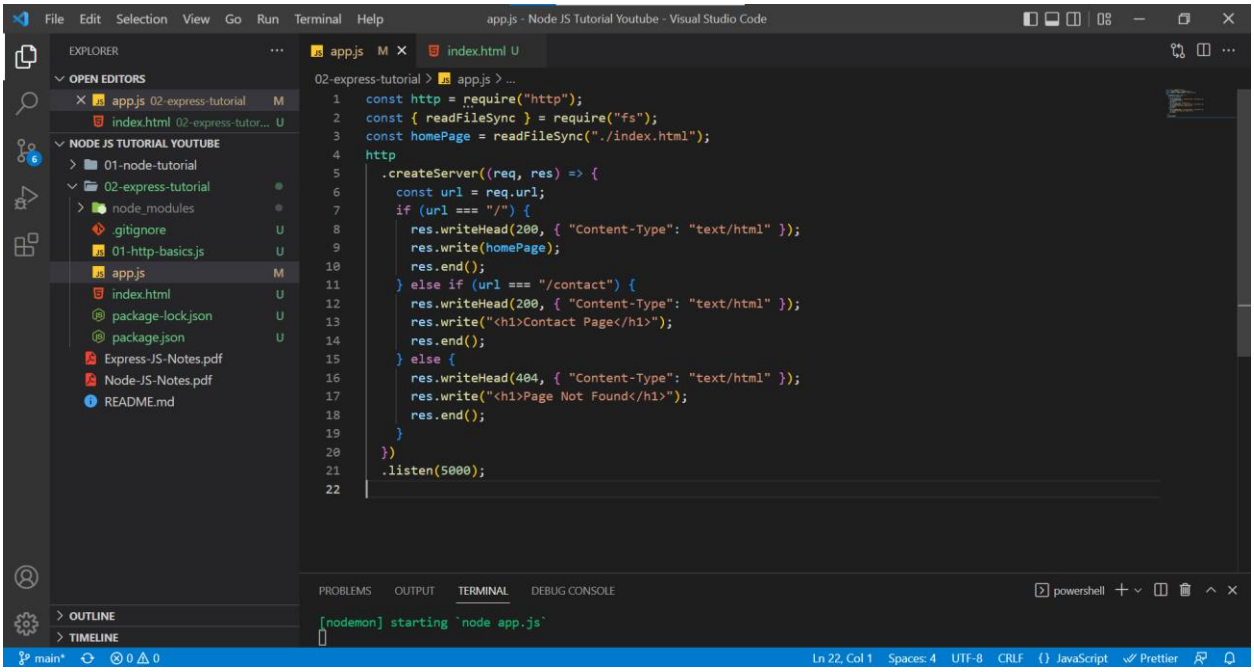
The reason we are using readFileSync is

1. We are not invoking the readFileSync every time when someone hits the server. We require that file when we instantiate the server (basically the initial time when the server starts running). We are just requesting it only once.
2. It is just an example as of now.

index.html



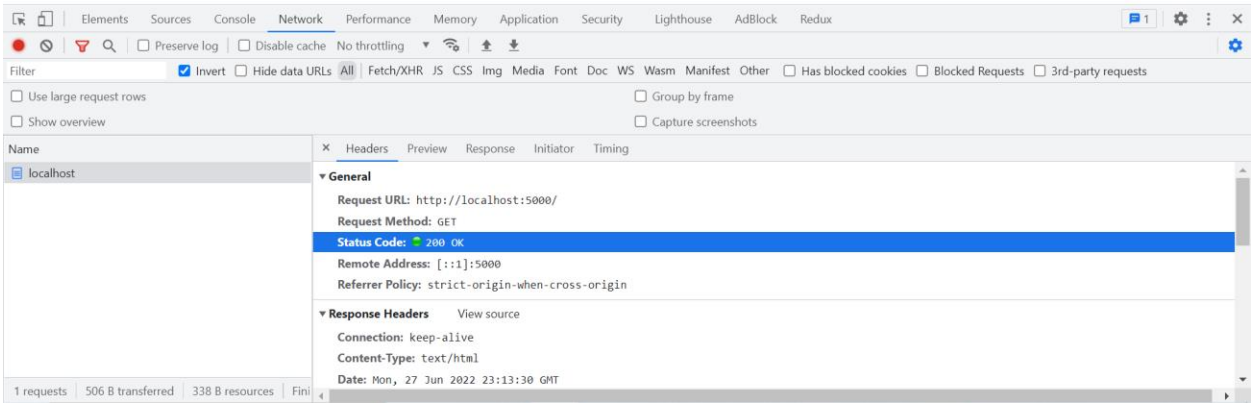
app.js



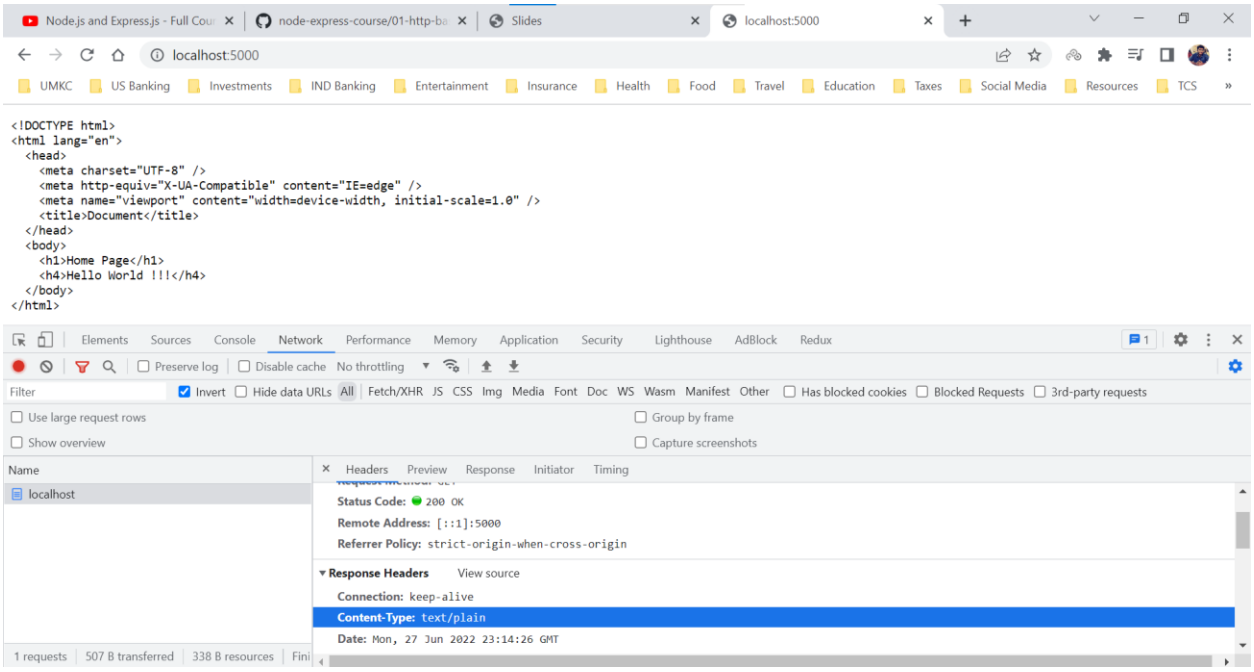
Home Page when Content Type is text/html.

Home Page

Hello World !!!



Home Page when Content Type is text/plain.



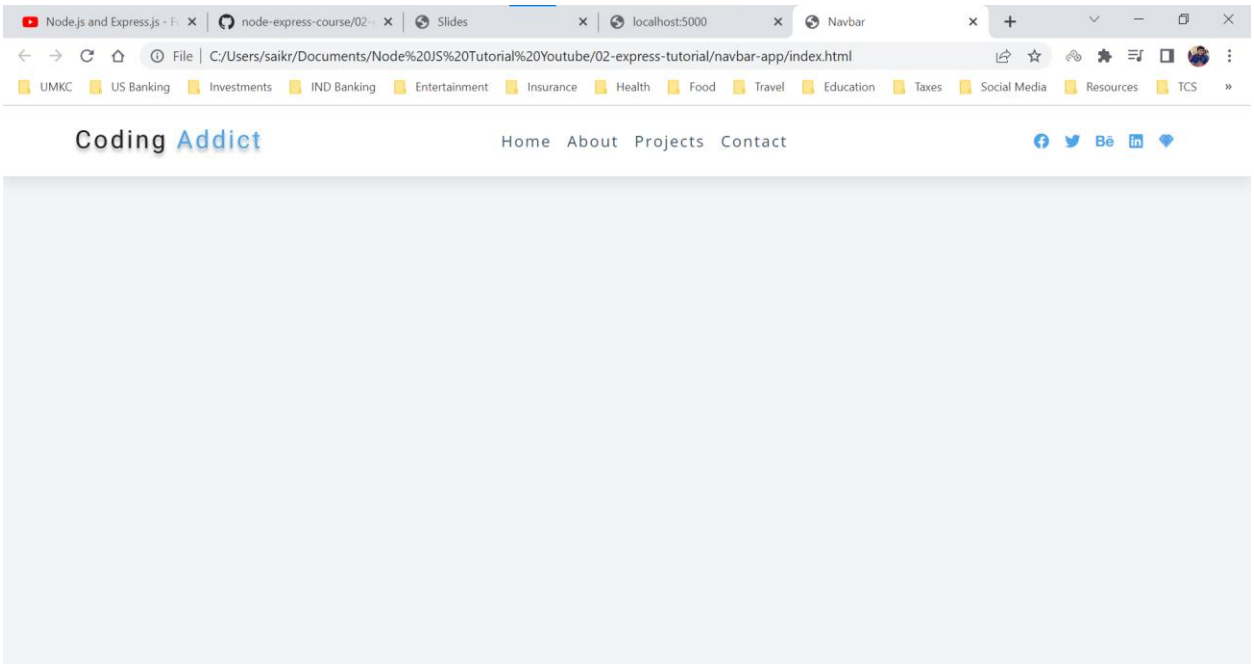
HTTP – App Example

In this section, we already have a web application named navbar-app which contains a JS file, CSS file, SVG file and an HTML file.

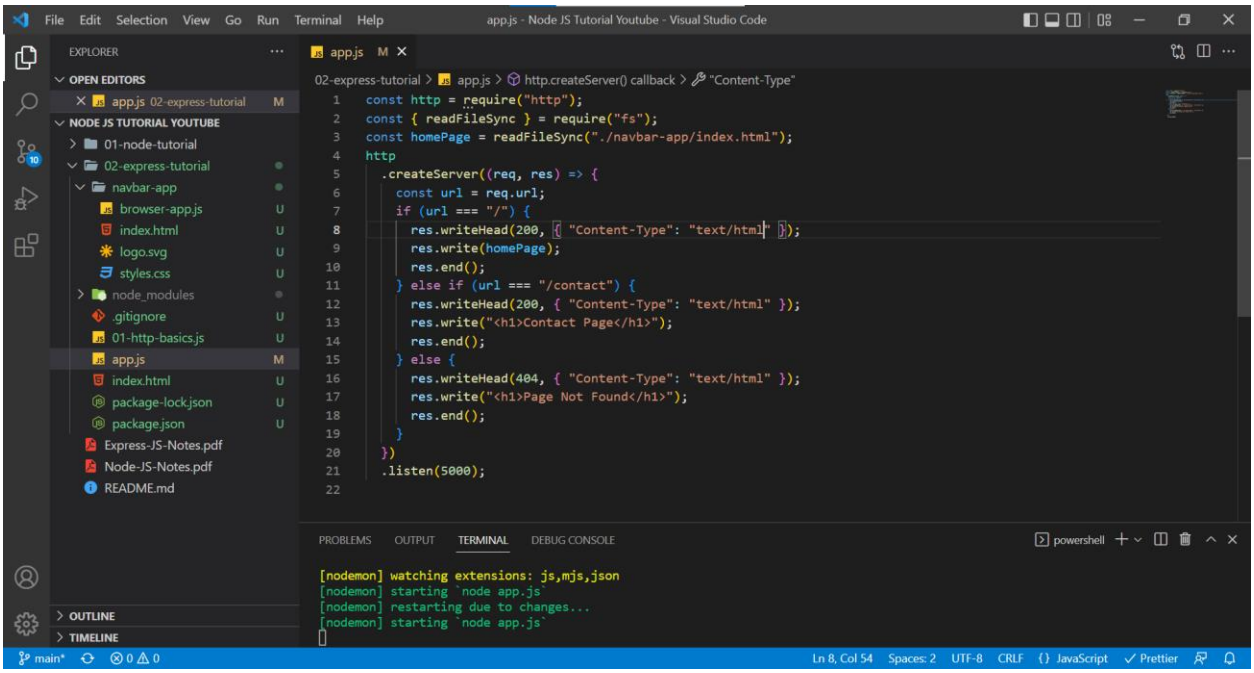
We try to run our server and connect to this application and provide responses based on the request.

We are changing the index.html file path since the web application already has an index.html file.

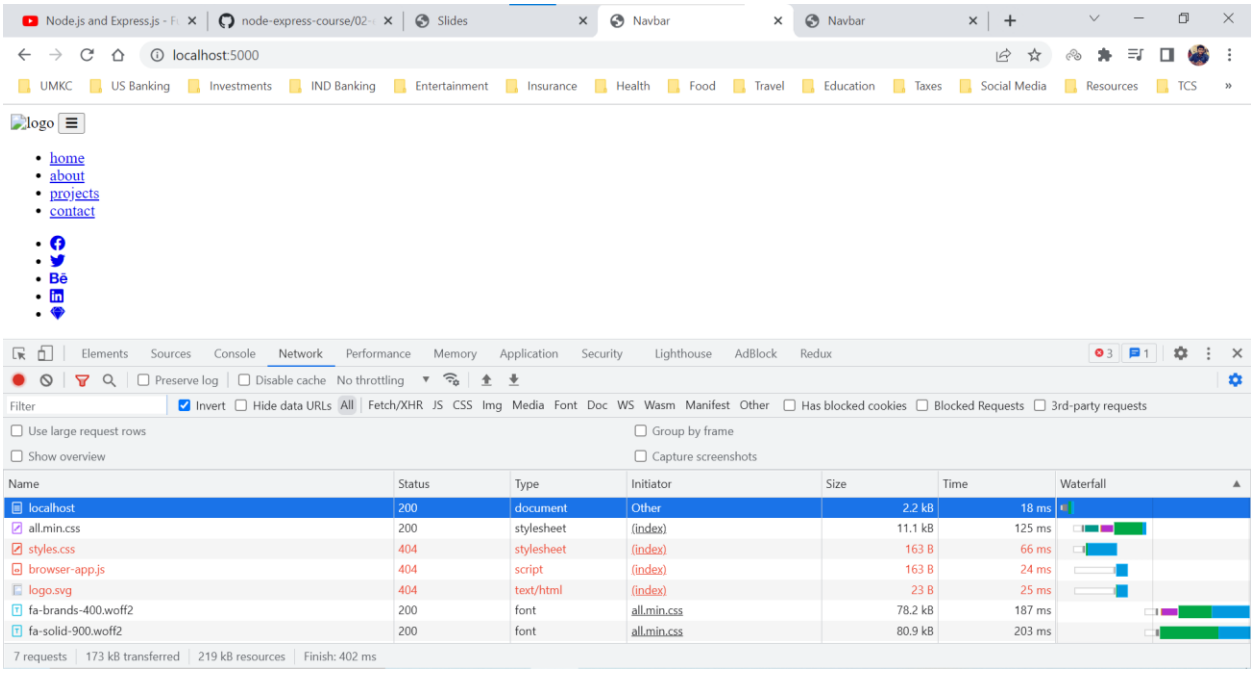
Web Application



app.js



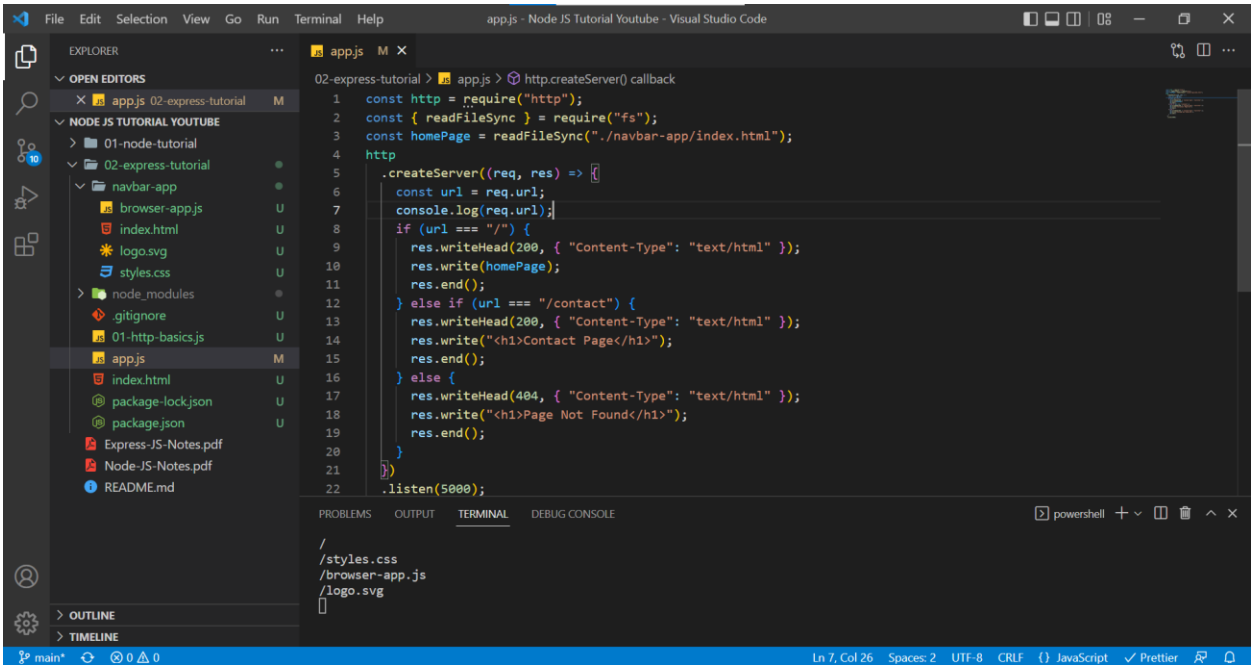
localhost:5000



The reason why the page doesn't have the same outlook after connecting to the server.

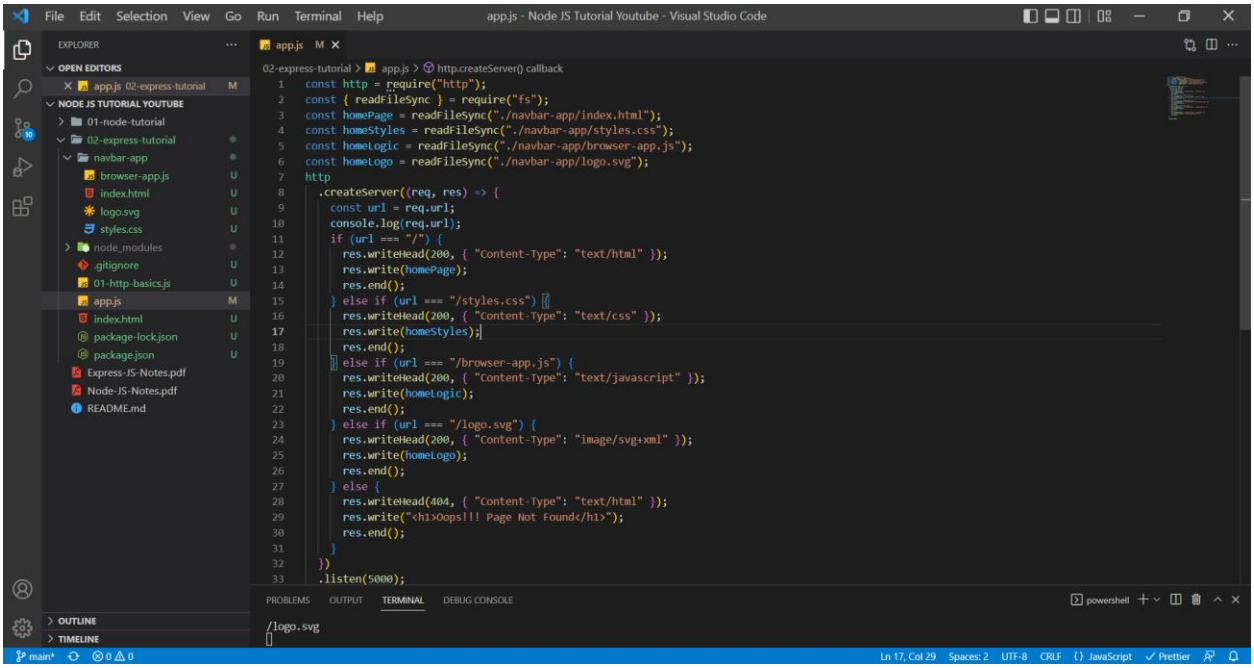
> Web Application is requesting resources like (index.html, styles.css, logo.svg, browser-app.js) but as of now we are serving the index.html request, hence other requests are receiving 404 error response. We ae not handling those requests (styles.css, logo.svg, browser-app.js) in our server.

We can console log the requests by the web application

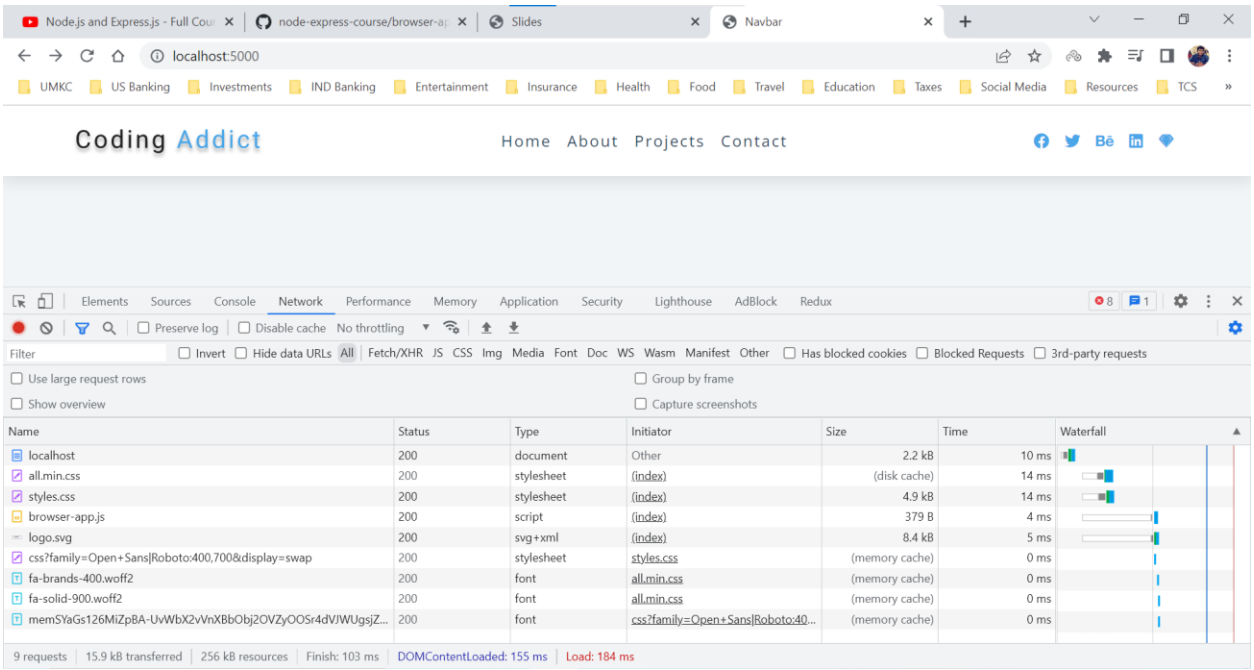


We need to handle those requests and provide responses.

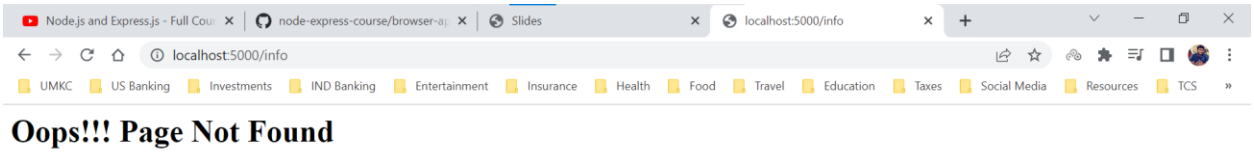
app.js

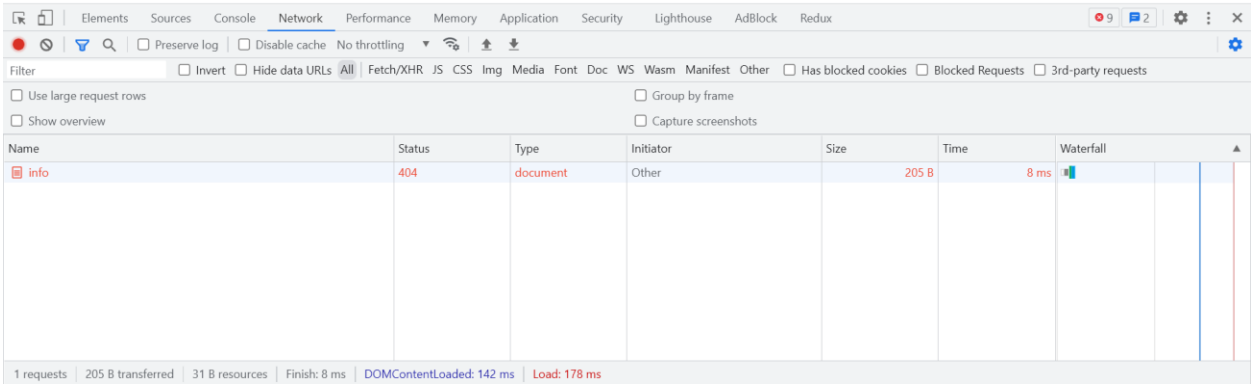


We are serving all the requests and let us see the response in browser.



If we try to request a resource which isn't available, then below would be the response.





Express Info

We can setup our server just with HTTP module but imagine a scenario where we have a website with tons of resources and then we need to setup for every single resource.

Express JS is a minimal and flexible Node JS Web App Framework designed to develop websites, web apps and API's much faster and easier.

Express is not one of the built-in modules of Node. Express is a standard when creating web applications with Node JS.

Command to install Express JS

npm install express - -save

Express JS team suggests using the - -save flag and effectively the reason is because in the earlier Node versions if you didn't add this flag then package wasn't saved to the package.json file meaning whenever we push the code without - -save flag then when another person is using the project, they didn't have reference to the project. Currently that issue is fixed but it still a precaution to use the save flag.

Command to install Express JS with a specific version

npm install express@4.17.1 - -save

Express Basics

HTTP methods are also known as HTTP verbs, and it is the important part of the HTTP request message that we look for.

HTTP Methods represent what the user is trying to do where to read the data, insert the data, update the data, or delete the data.

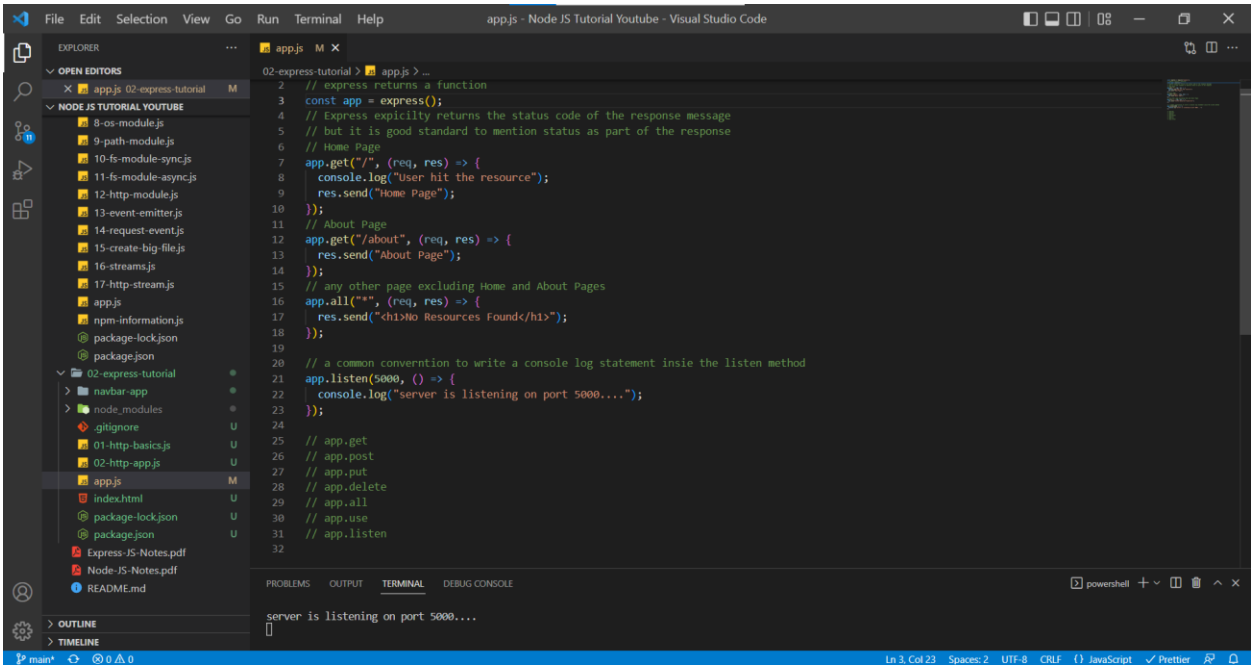
By default, all the browsers perform the GET request.

HTTP METHODS

GET	Read Data
POST	Insert Data
PUT	Update Data
DELETE	Delete Data

GET	***www.store.com/api/orders***	get all orders
POST	***www.store.com/api/orders***	place an order (send data)
GET	***www.store.com/api/orders/:id***	get single order (path params)
PUT	***www.store.com/api/orders/:id***	update specific order (params + send data)
DELETE	***www.store.com/api/orders/:id***	delete order (path params)

app.js



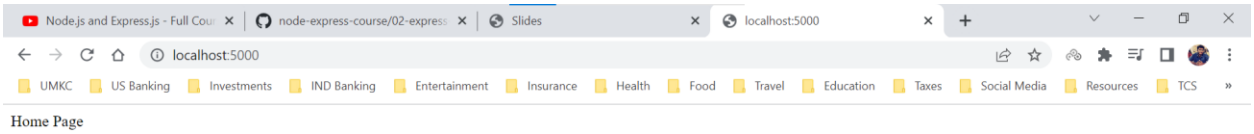
app.all – this method works with all of them. It is used handle to requests which doesn't have any resources.

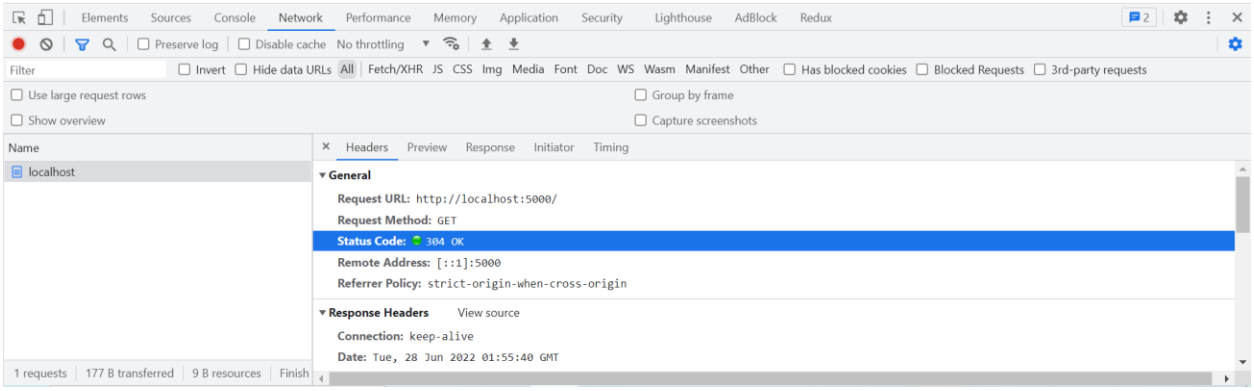
app.use – this method is responsible for middleware, and it is a crucial part of Express.

app.get – In this method, we need to specially add two things. A path (what resource user is trying to request) and a callback function and this callback function will be invoked every time a user is performing a get request on our route or on our domain.

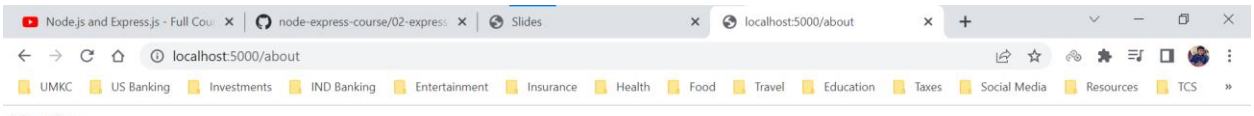
app.listen – In this method our server gets started and server listens to requests sent by the browser.

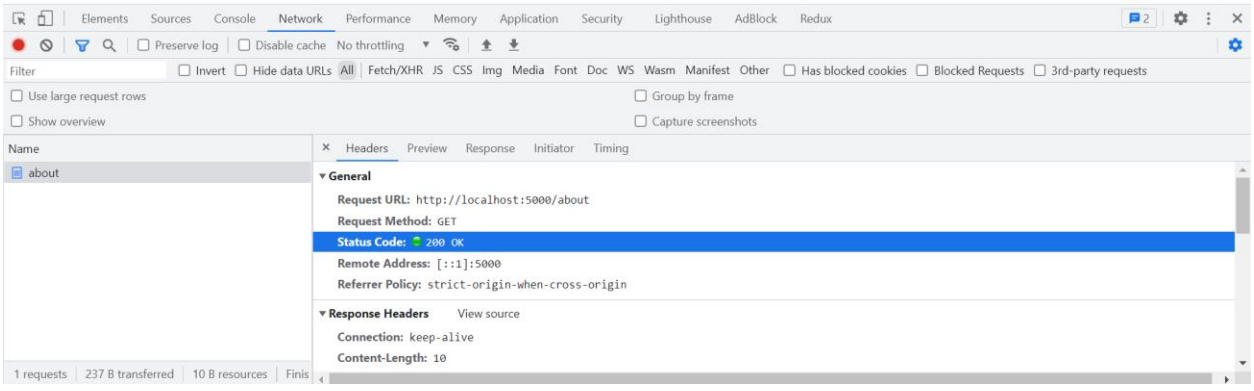
Home Page



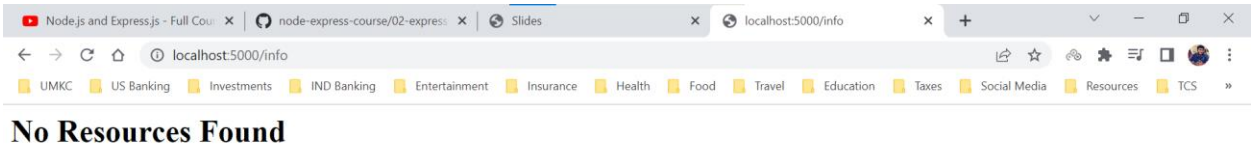


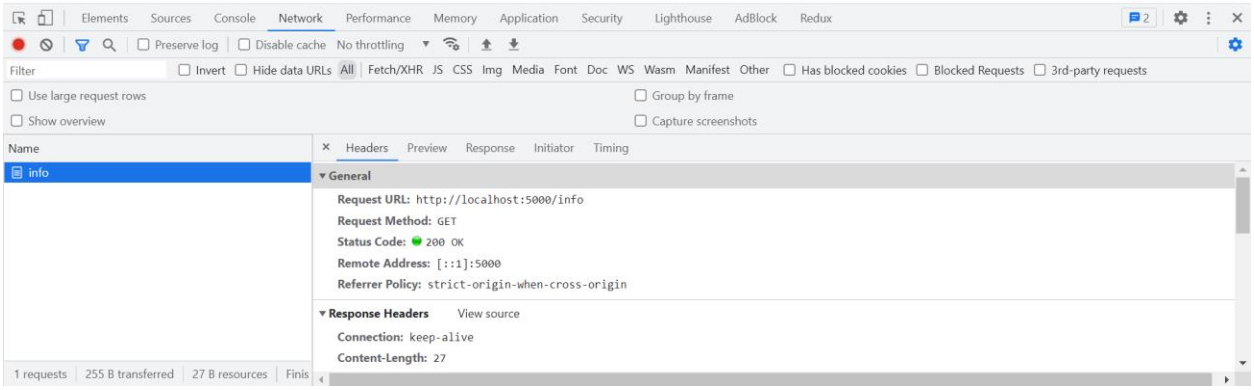
About Page





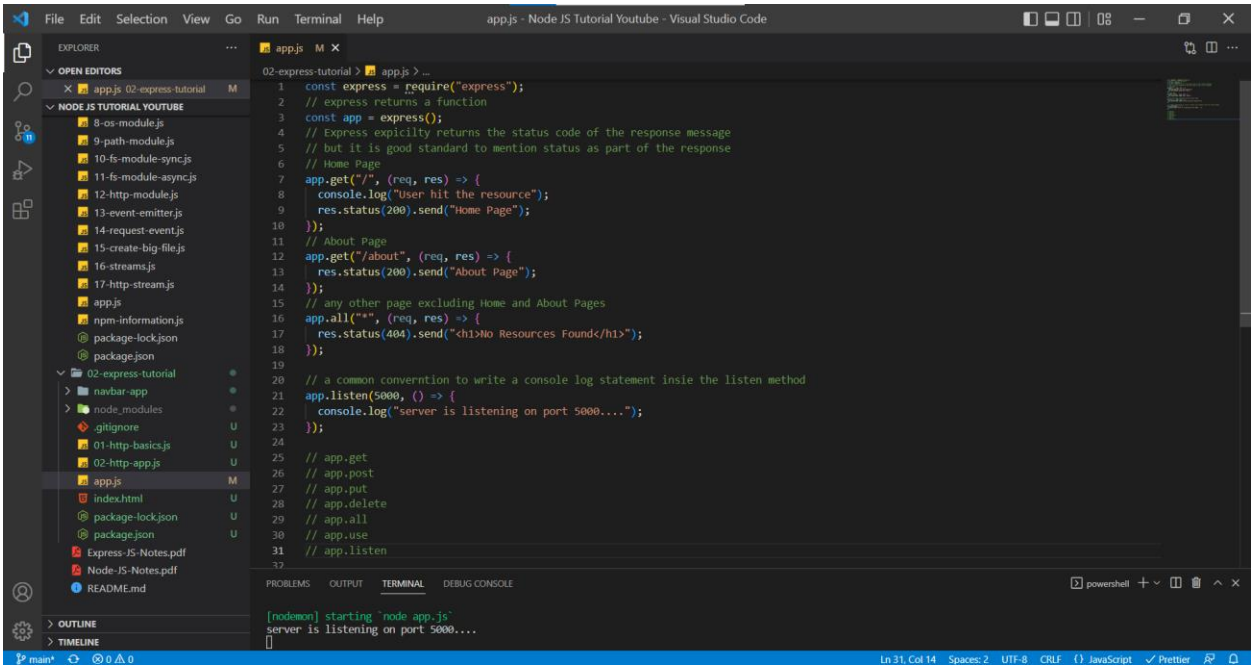
Resources which are not handled



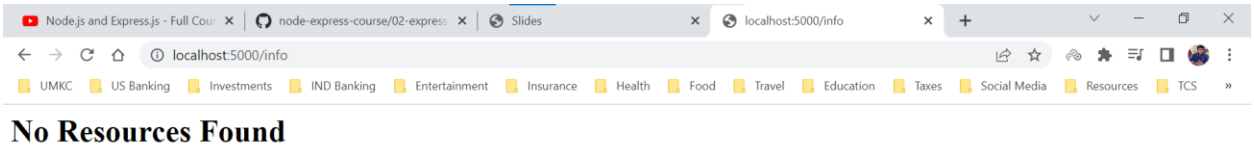


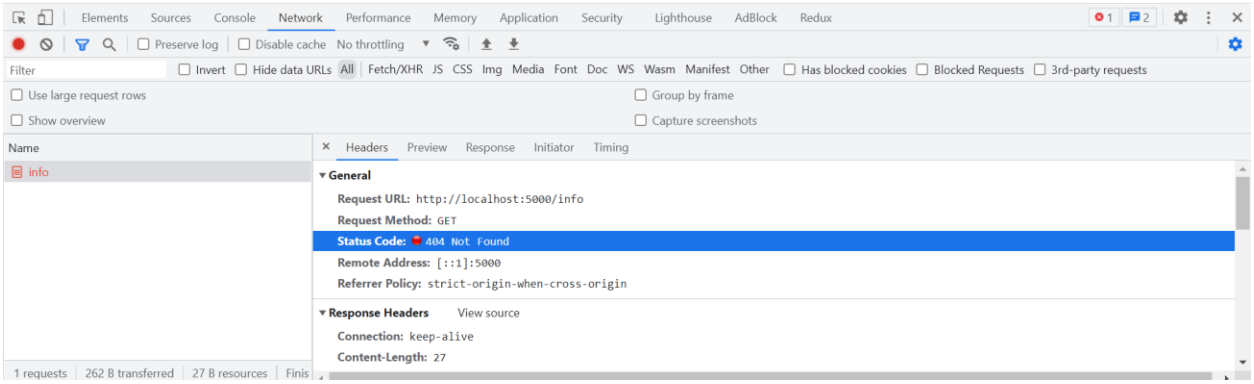
One thing which doesn't add up here is the Status Code. Hence, we need to provide a status code whenever we are sending a response to the browser.

app.js



Resources which are not handled



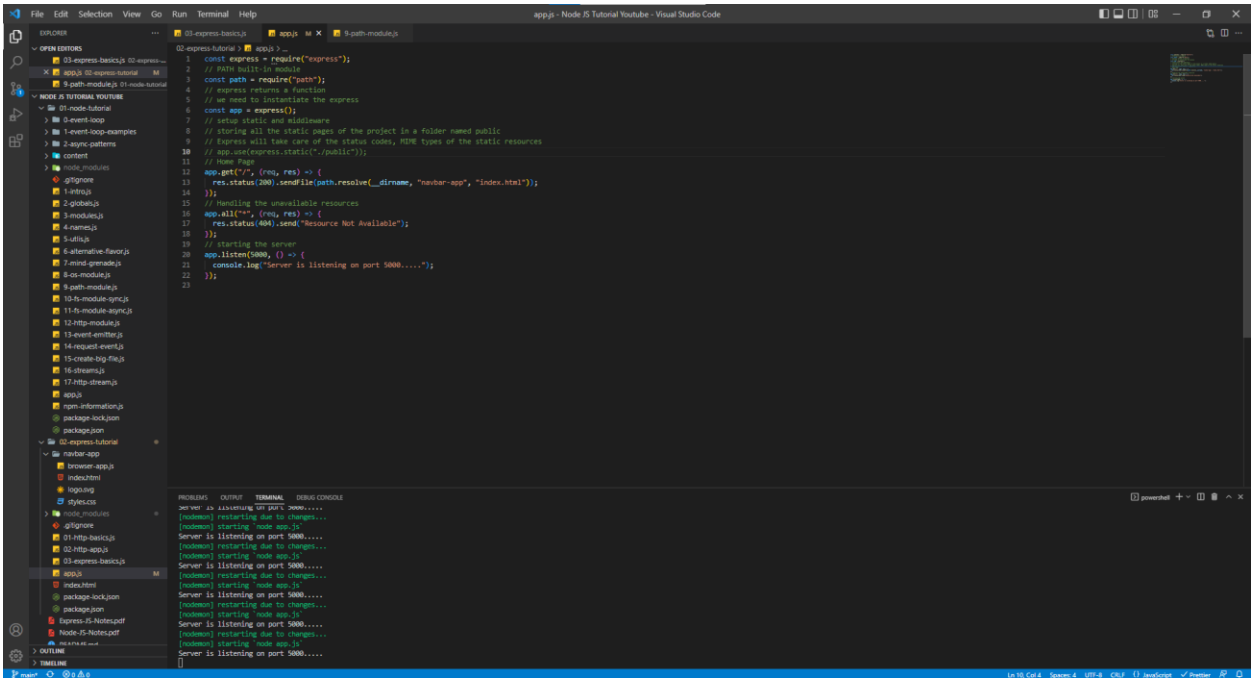


As we can it is a way less code when compared to creating a server using built-in HTTP module.

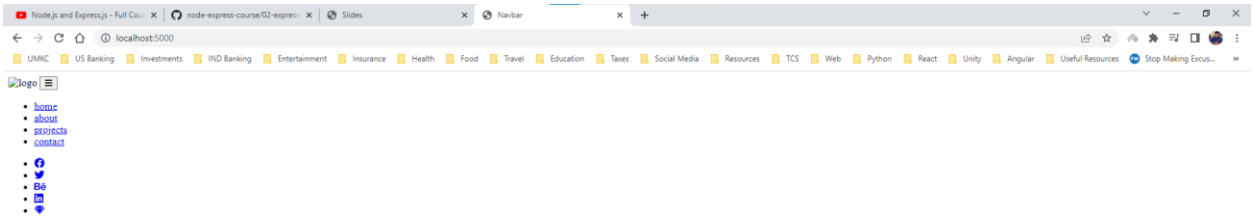
Express – App Example

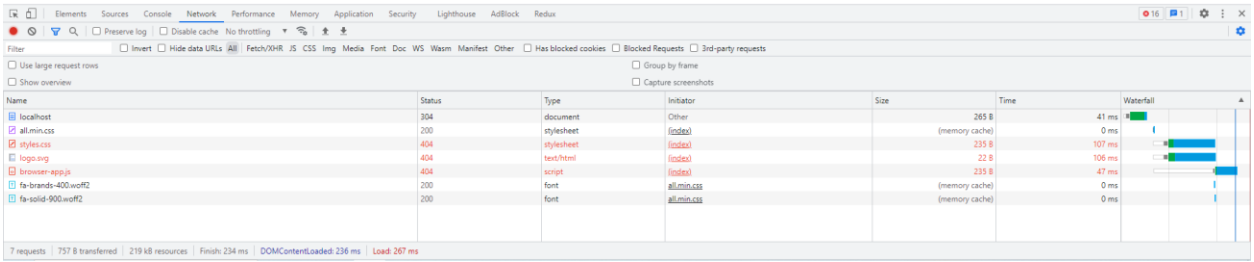
In this section, we are using the same navbar-app example with express framework. Express reduces a lot of code and provides the easier way to develop a server rather than HTTP Module server.

app.js



localhost:5000





We are experiencing the same thing, we have sent index.html as part of the response but we haven't sent the styles.css, browser-app.js and logo.svg. These needs to be taken care. In HTTP server we used to write a piece of code for each resource (styles.css, browser-app.js and logo.svg) but in express we don't write 3 different pieces of code instead we only use a method name app.use() which will take care of the middleware and static files of the project.

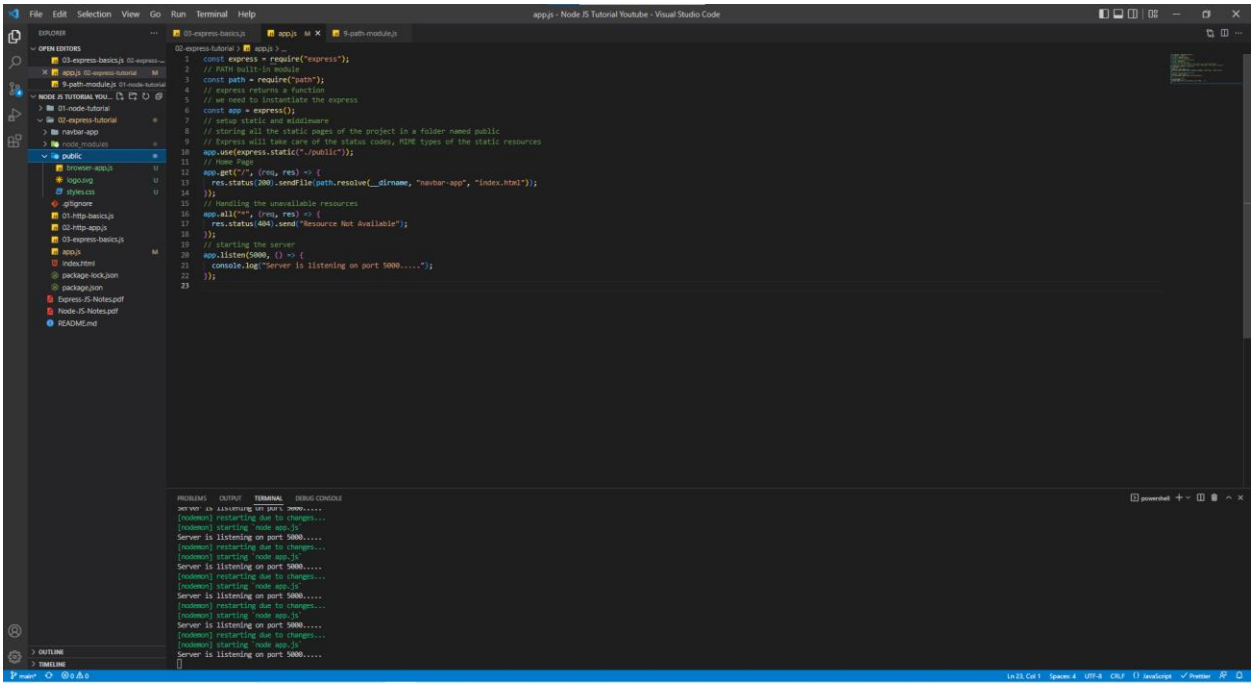
We place all the static files of the application (browser-app.js, styles.css, logo.svg) in folder named public (common convention name).

With the help of app.use() method we won't need to write paths, status codes, MIME types and content type of the static resources. Express takes care of it all.

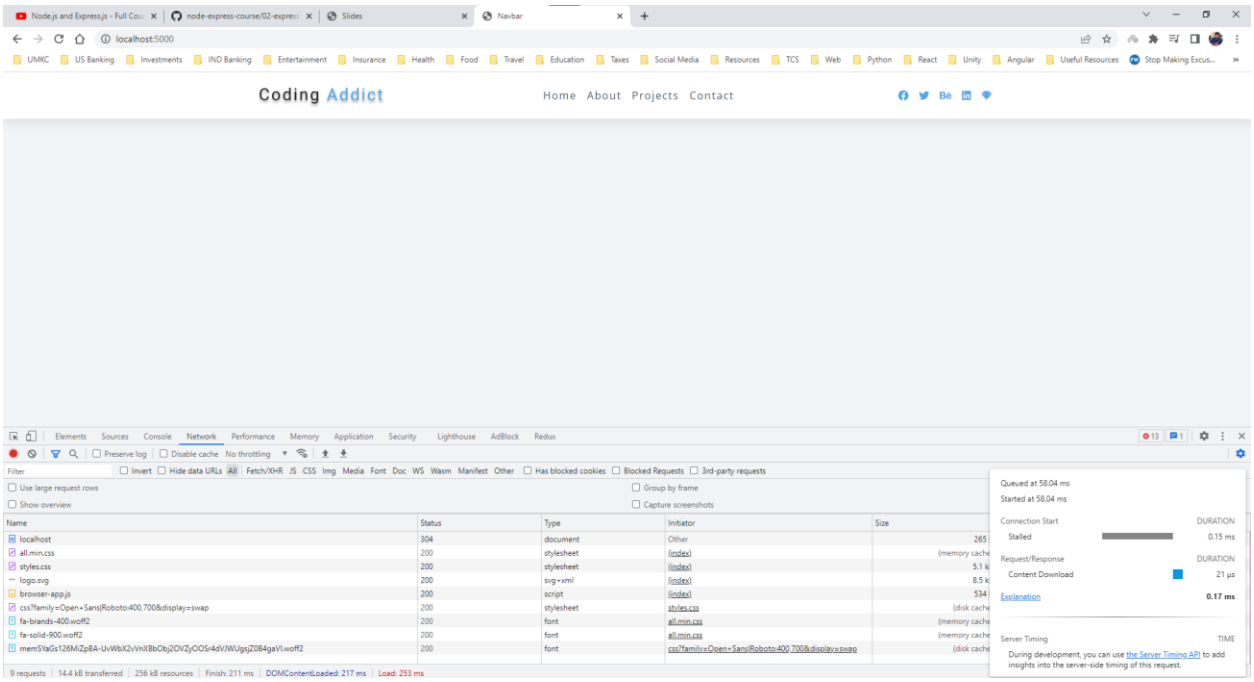
Static Asset – it means that it is a file that server doesn't need to change it.

Imagine a scenario where we have 20,000 images and if we must use the HTTP server then we need to have 20,000 pieces of code for each resource. It isn't viable and is not a good approach to have that many lines of code.

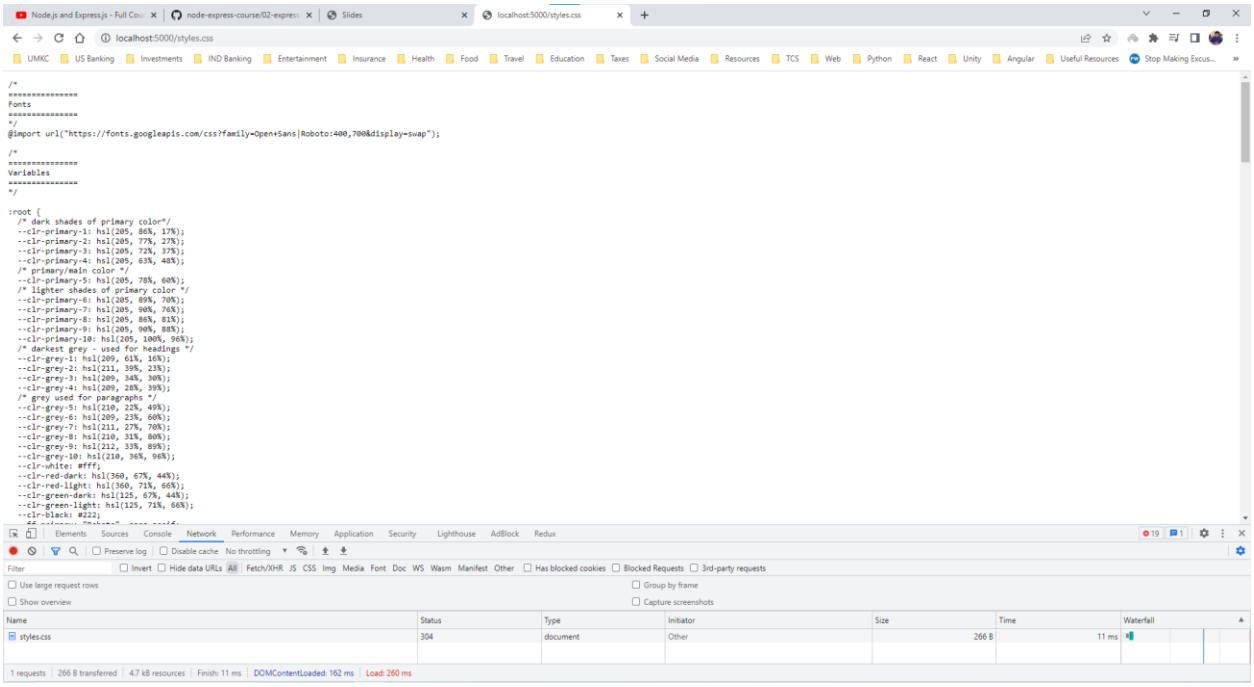
app.js



localhost:5000



localhost:5000/styles.css



Express – All Static

While working on the above express application, we must get a doubt that even index.html file is static and why are not adding it to the folder.

We can do that in two approaches.

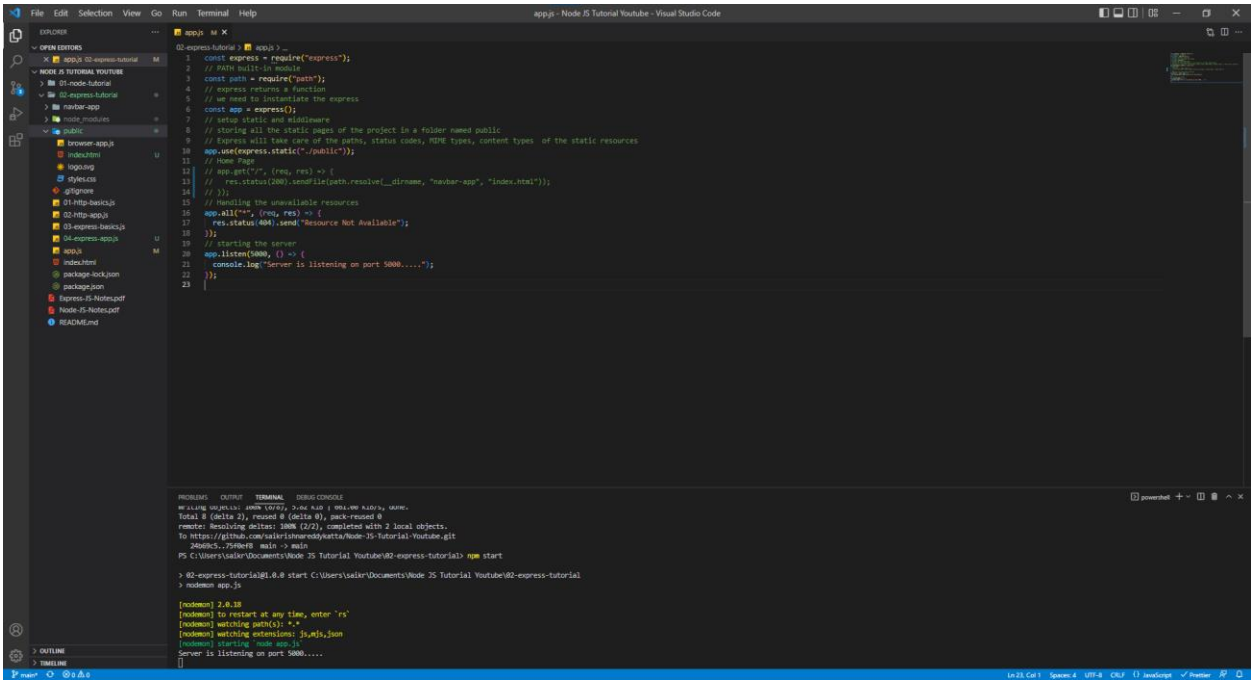
1. Yes, we can add index.html file in the static folder. So, in this section we are going to add the index.html file in static folder.

   Index.html file is always a root file, so when the user hits the server by default the server will serve the index.html file present in the static folder. Since our index.html file has all the paths to browser-app.js, logo.svg and styles.css we are going to be in a good shape, and we don't even need to setup the send File option.

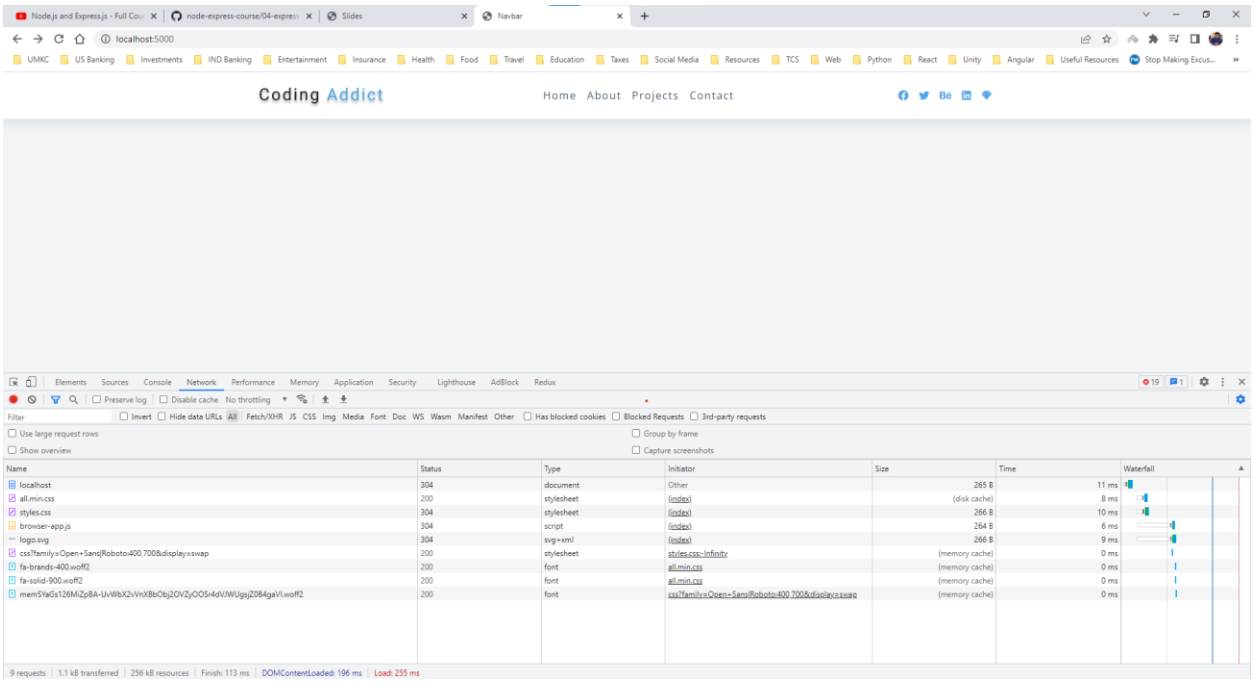
2. SSR (Server-Side Rendering)

In this section, let us discuss only about first approach.

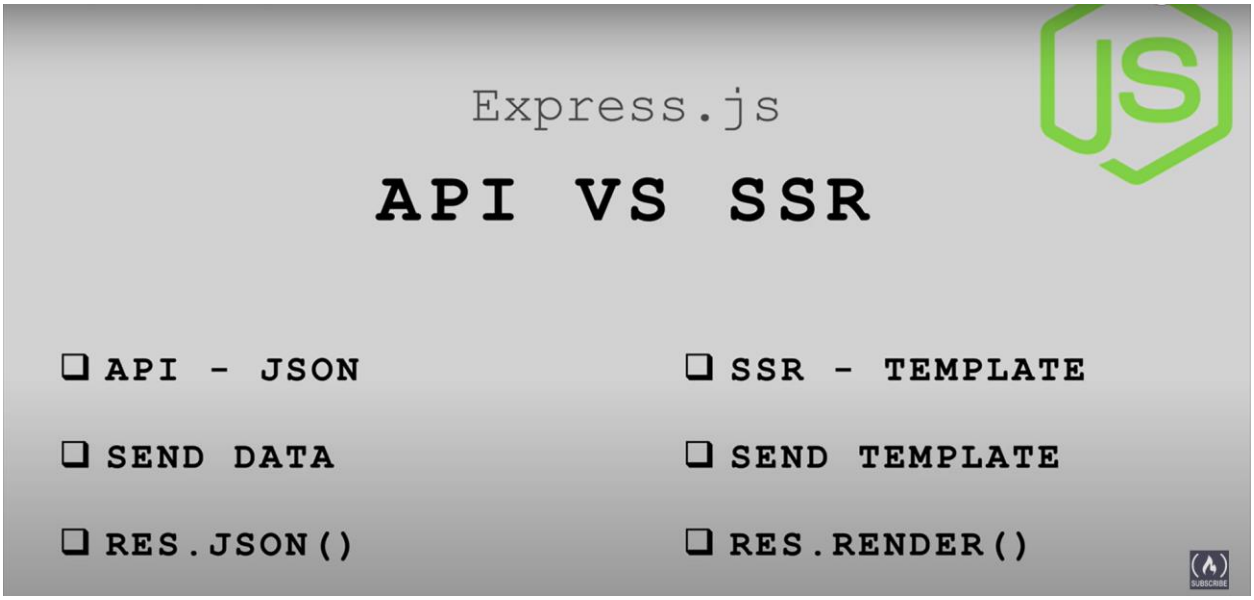
app.js



localhost:5000



API vs SSR (Application Programming Interface vs Server-Side Rendering)



Express can be used in two ways

1. Using Express to create an API
2. Using Express to create templates with Server-Side Rendering

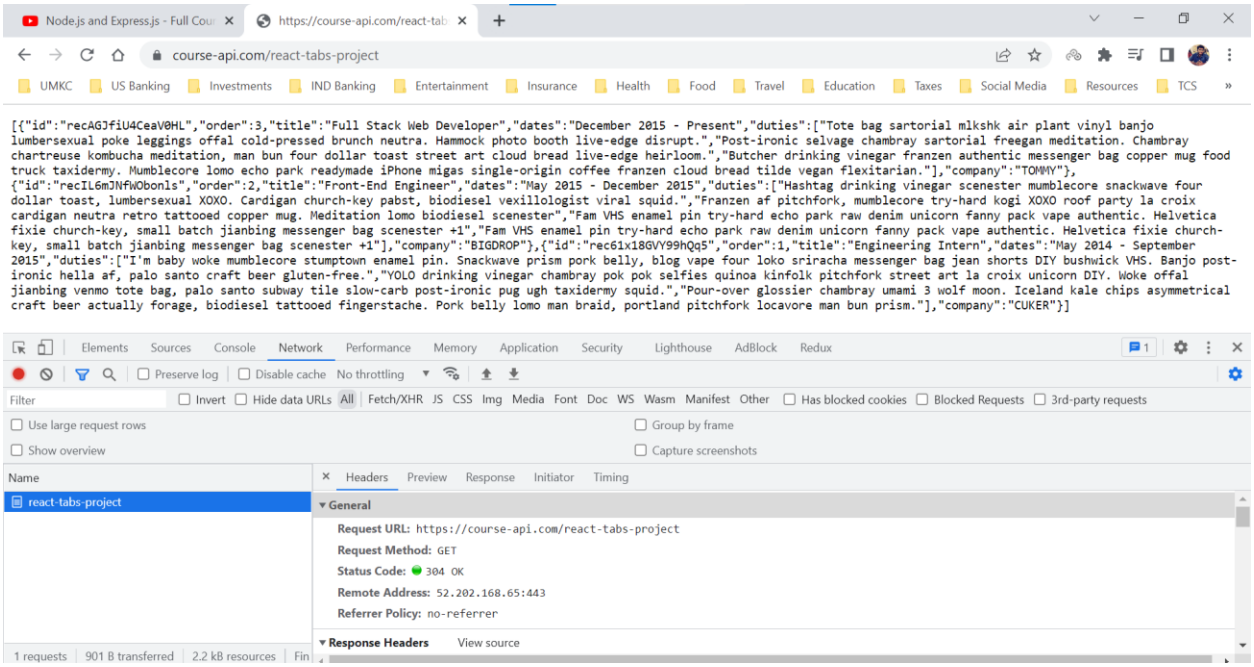
In general, API is used to set up an HTTP interface to interact with data. Data is sent through JSON (JavaScript Object Notation) and to send back our response we use **res.json()** method. This method will do all the heavy lifting like setting up the proper content-type and stringify our data.

In Server-Side Rendering, we will set up templates and send back the entire HTML, CSS and JavaScript to frontend using **res.render()** method.

We will mostly try to learn about API rather than the SSR because API is main part of the Express and once API knowledge is done, we can easily handle SSR part as well.

Main Idea around API is that our server provides the data to any frontend application that's wanting to access and use the data.

React Tabs Project (API Call)



JSON Basics

JSON stands for JavaScript Object Notation

Whatever may be the frontend, API provides the data to the browser.

res.json() sends a JSON response. This method sends a response (with the correct content-type) that is the parameter converted to a JSON string using JSON.stringify()

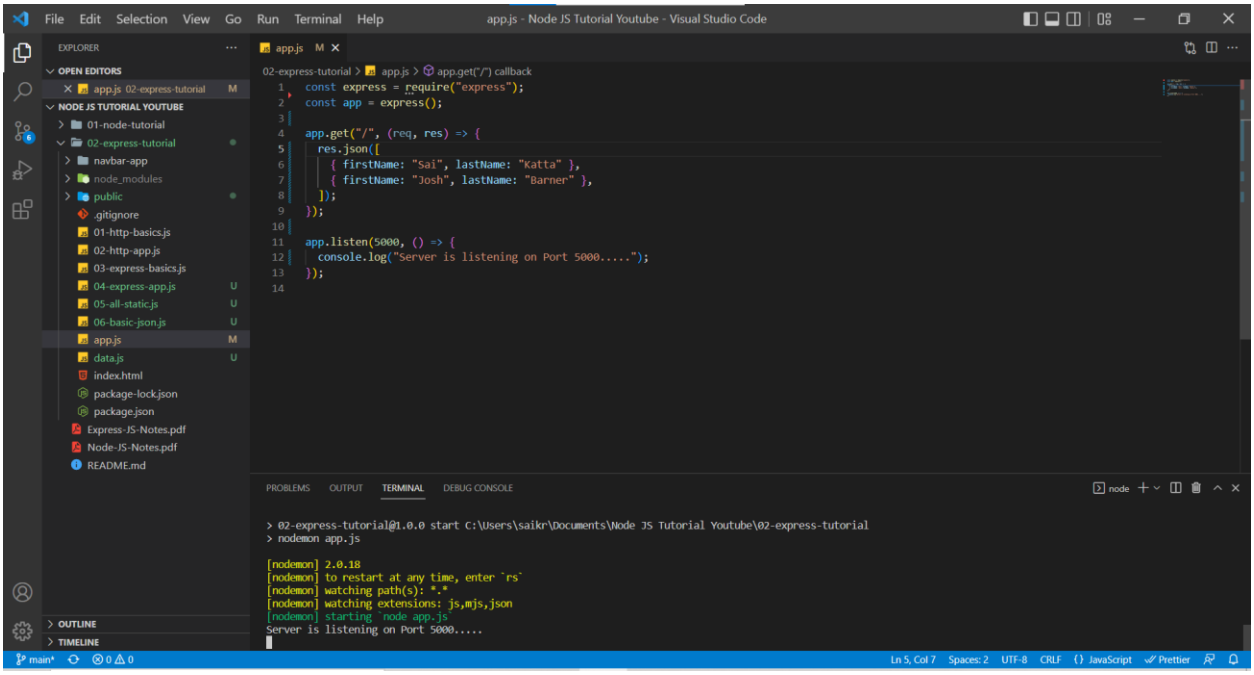
The parameter can be any JSON type, including object, array, string, Boolean, Number, or null and you can also use it to covert other values to JSON.

res.json(null)

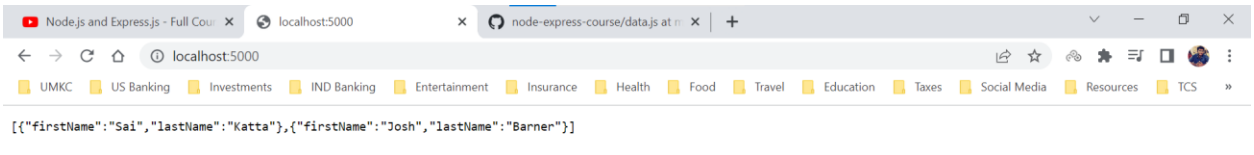
res.json({user: “Sai”})

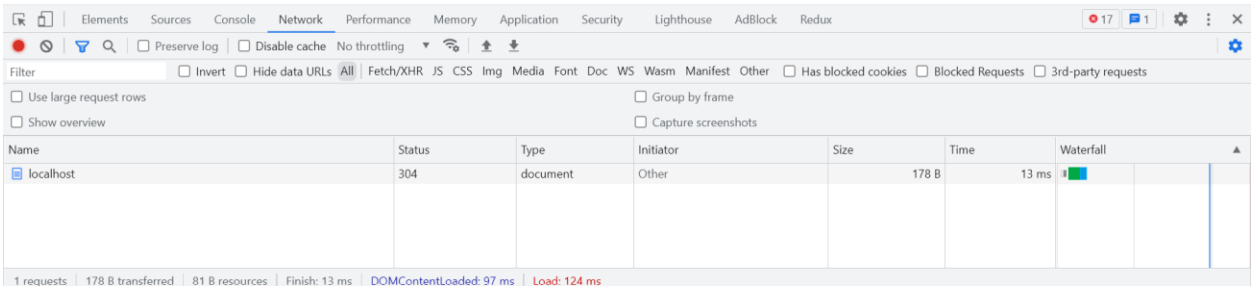
res.status(500).json({error: “message”})

app.js



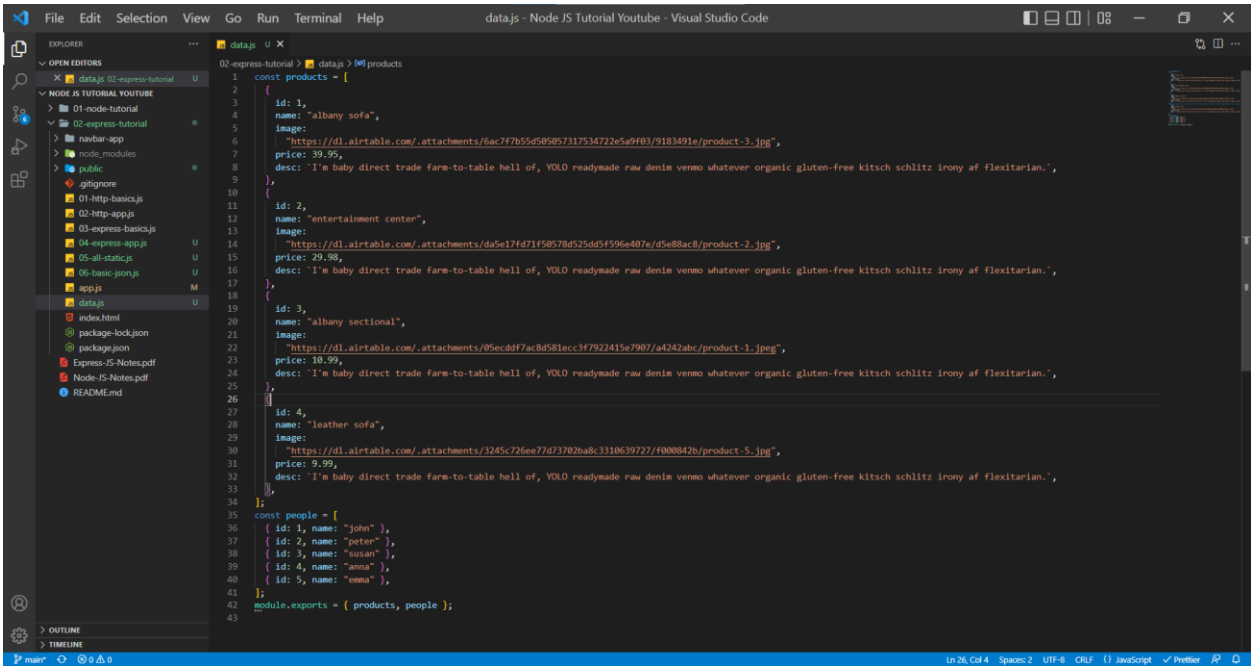
localhost:5000





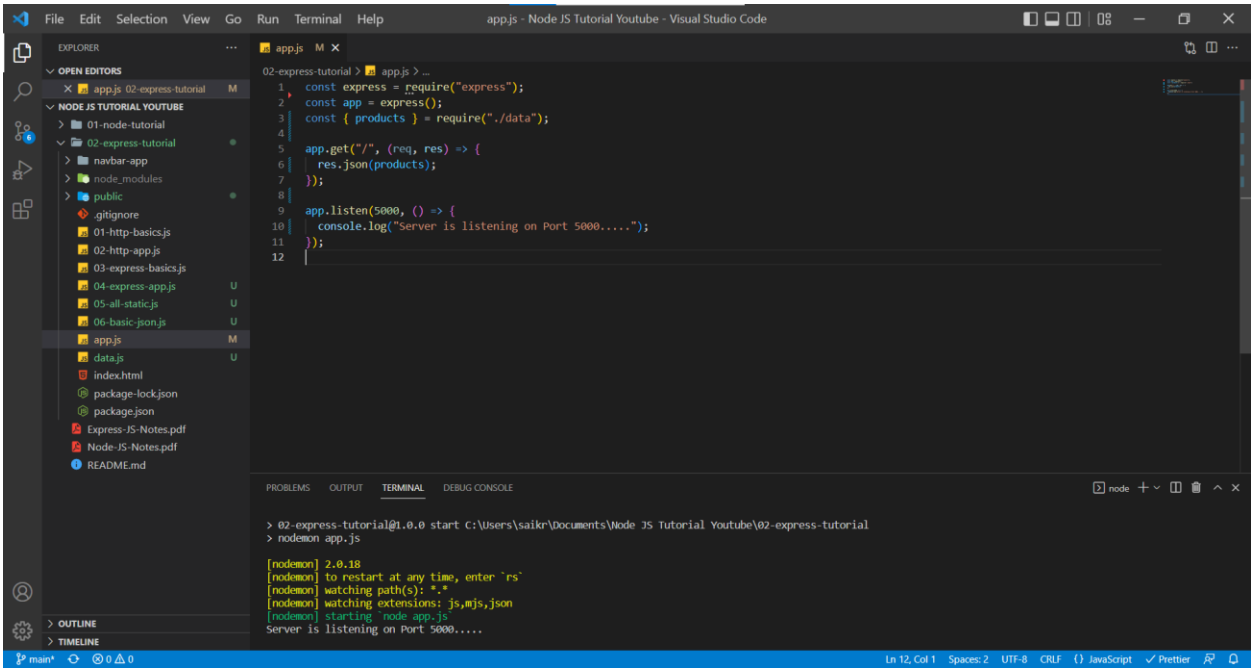
Now we will a file name data.js which contains data consisting of arrays.

data.js

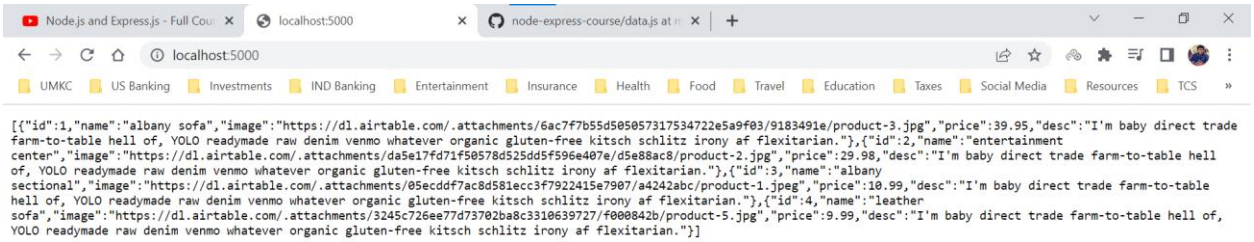


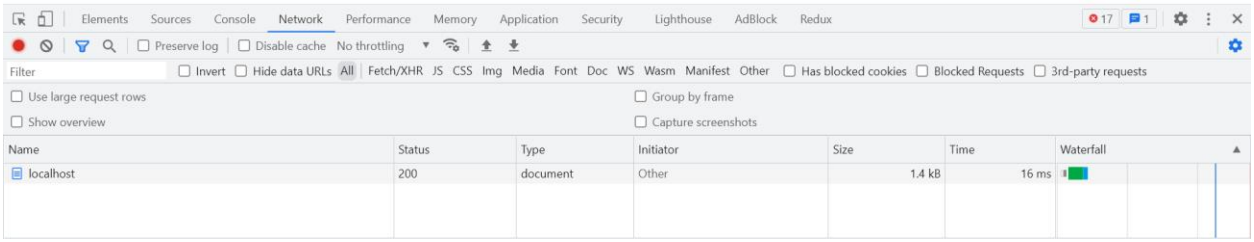
Instead of Hardcoding the data in app.js, we will try to import data from data.js file and send it to browser.

app.js



localhost:5000





Params, Query String – Setup

Route Params

In this section, we are going to learn about requesting data based on the parameters. Assume a scenario where we have 2000 resources on the page and user wants to request each resource based upon the click.

We define each resource with an ID which will be unique for a resource. When a user tries to request (either through a click or through entering address on address bar), we need to provide data based on the request.

In general, we need to capture the parameter of the request and provide the data according to that. In a request we have a property name ***params*** which provides a javascript object containing parameters of that request. We need to extract the parameter and find the respective data.

Note: We need to remember that values in params object would be of data type String.

app.all() method is not working when we try to request a response from a product which is not available.

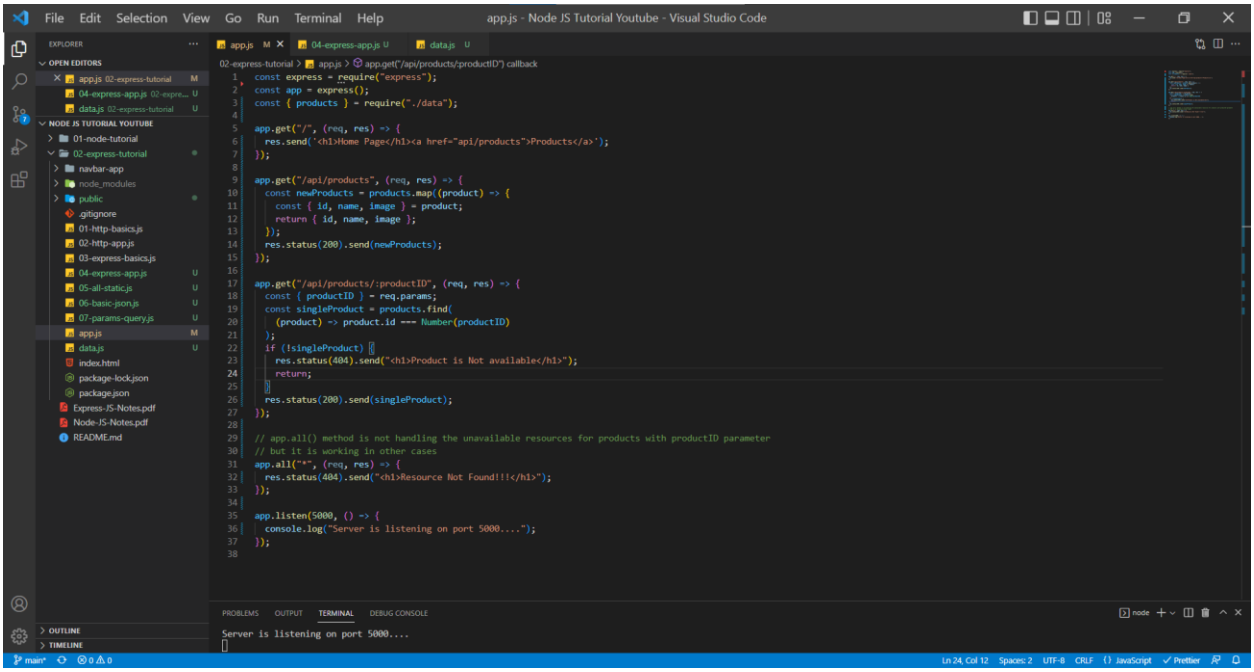
Ex: localhost:5000/api/products/hgjh

But it is working when we are trying to request a below resource

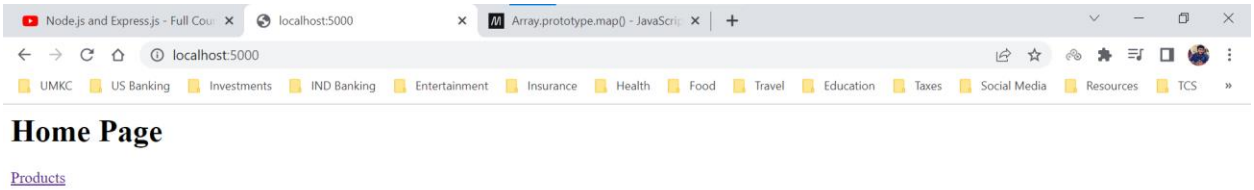
Ex: localhost:5000/api/hgdsjgh – It is providing the Resource Not Available response as expected.

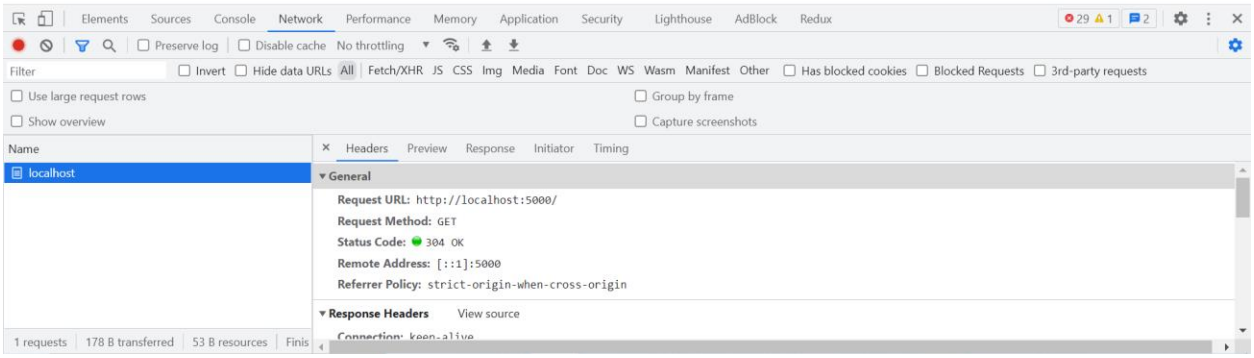
Maye be it is not checking inside the Query Parameters.

app.js

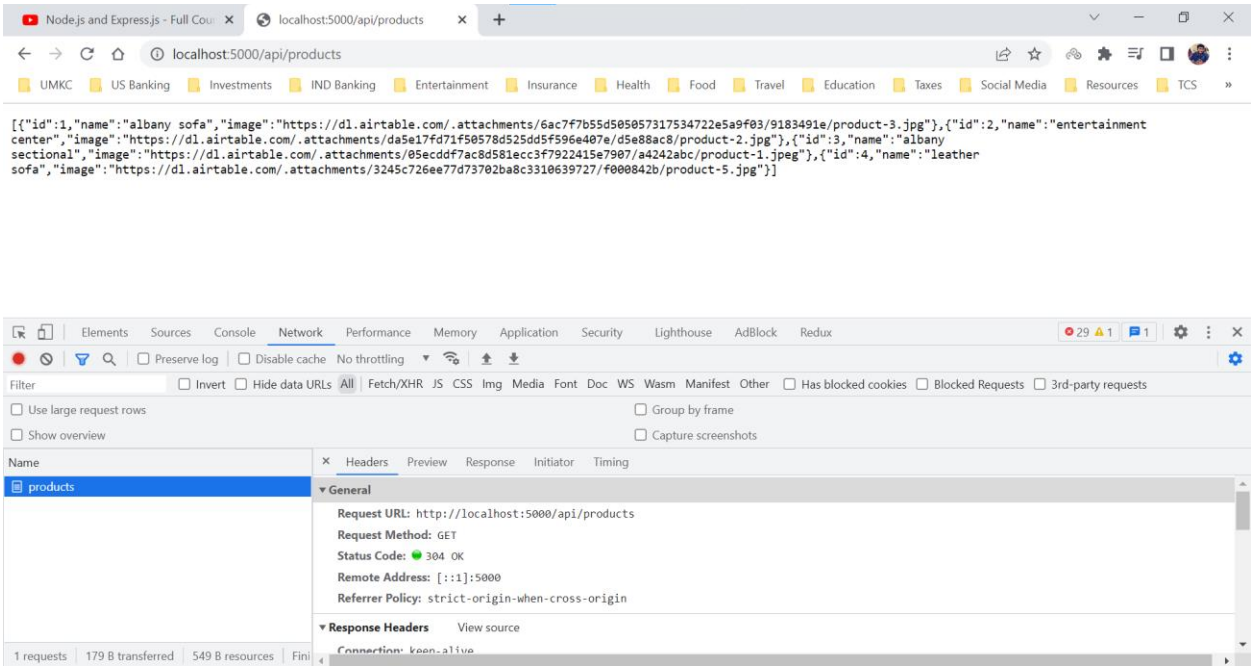


localhost:5000 – Home Page

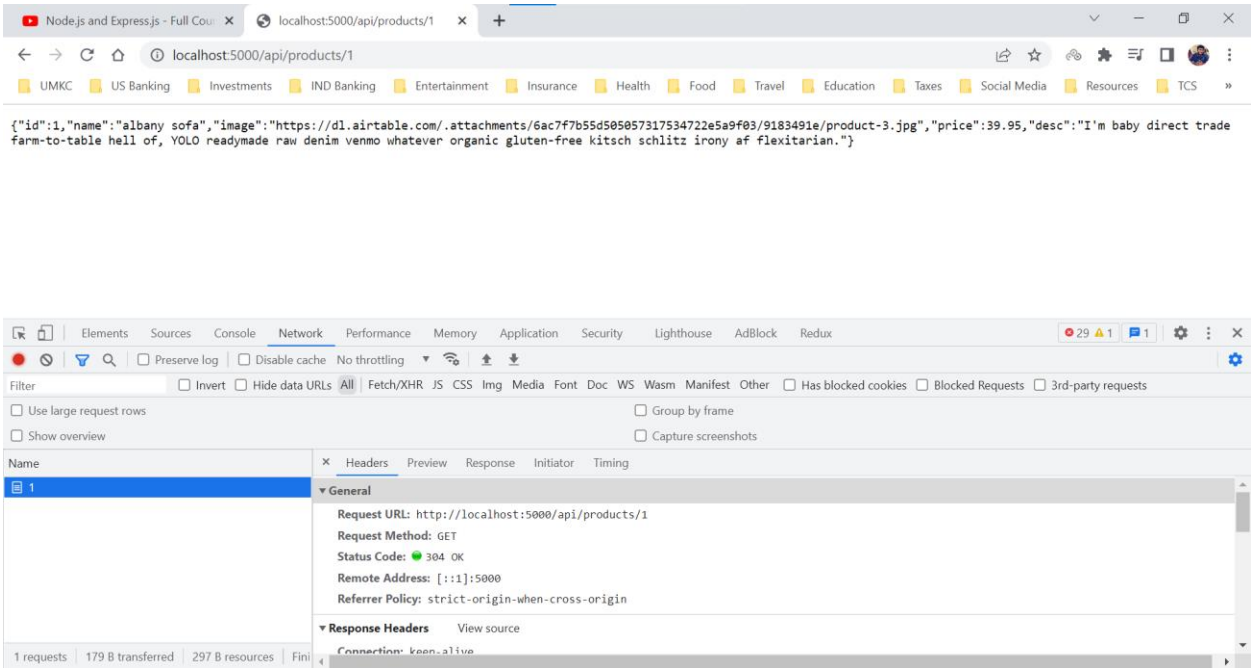




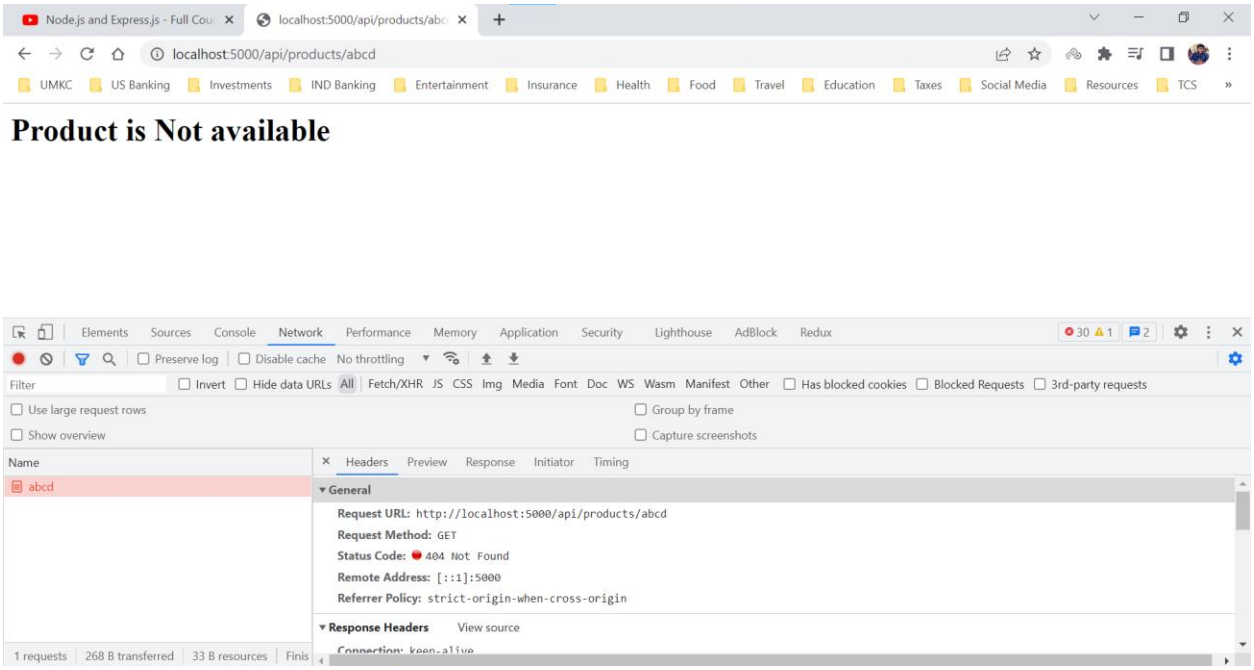
localhost:5000/api/products – Products Page



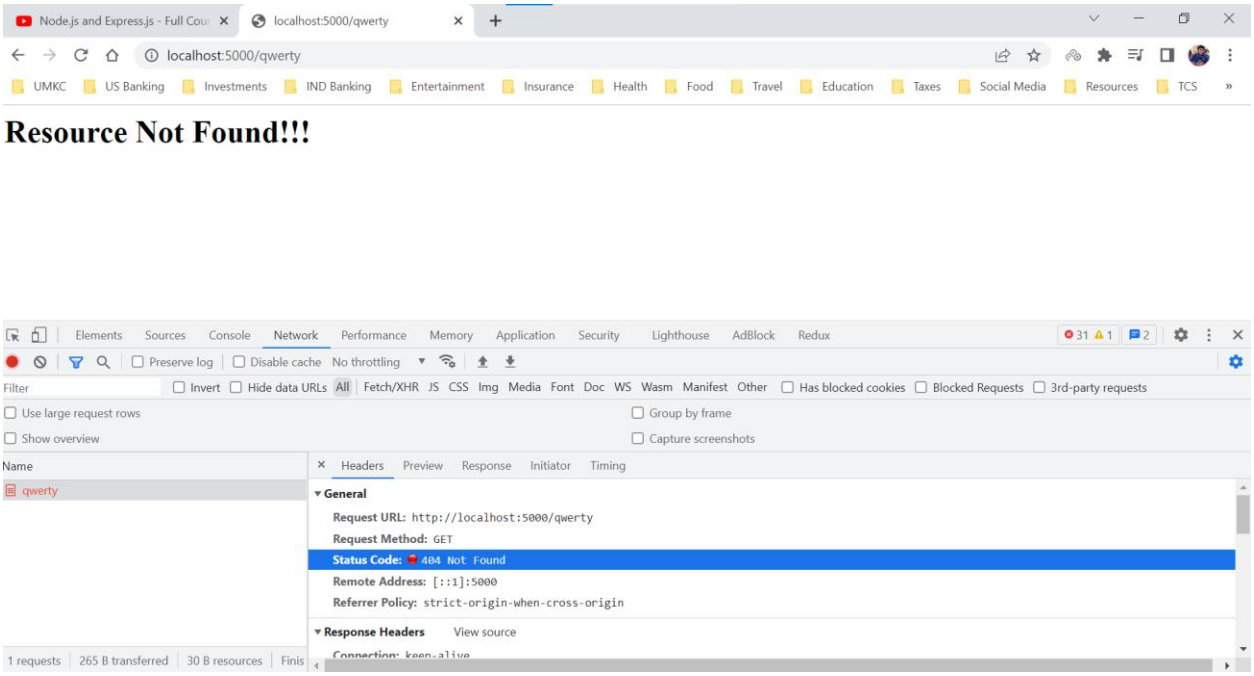
localhost:5000/api/products/1 – Product with ID #1



localhost:5000/api/products/abcd – Unavailable Product



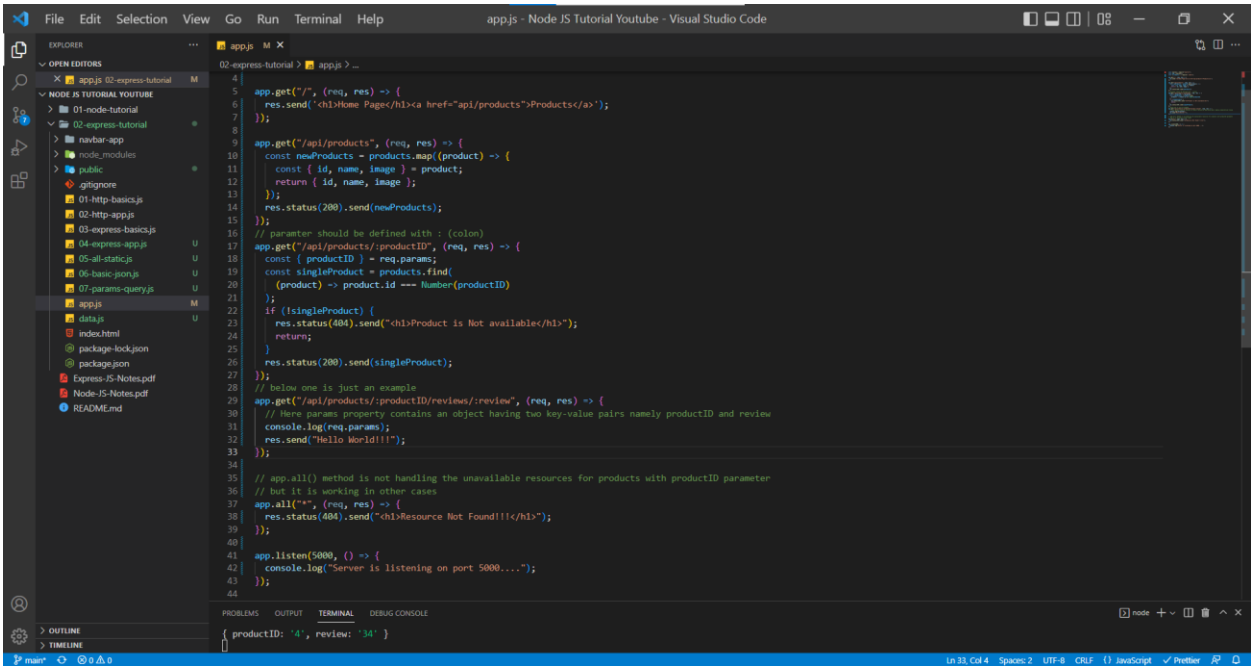
localhost:5000/qwerty – Unavailable Resource on the Server



Params – Extra Info

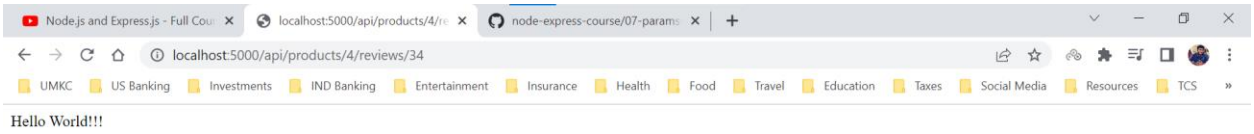
Route Parameters can get complex way more than the above example.

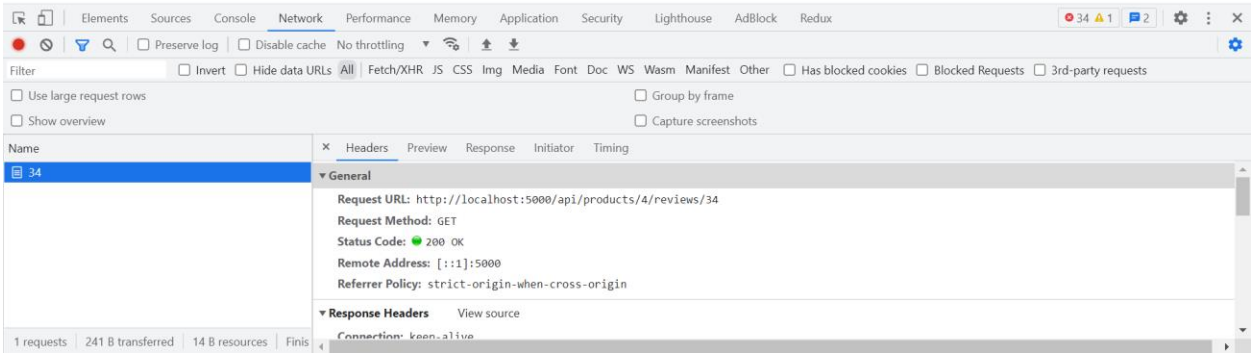
app.js



In console statement of app.js, we can find the req.params property data.

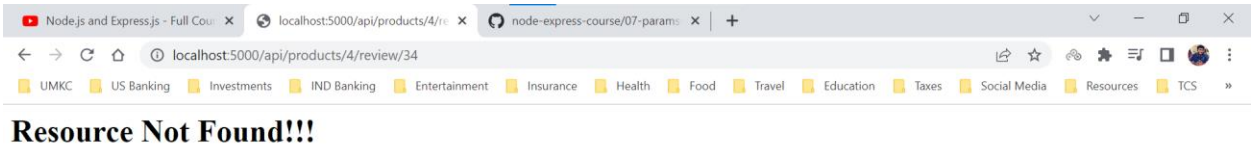
localhost:5000/api/products/4/reviews/34

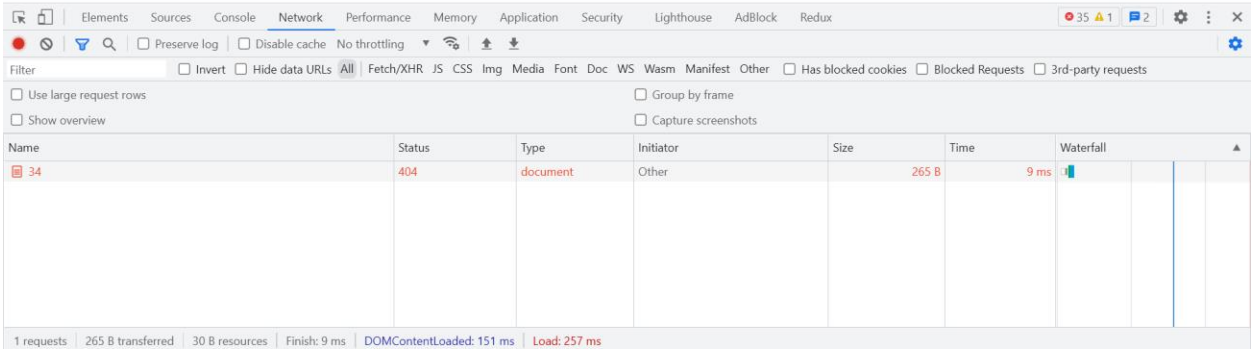




When we try to enter localhost:5000/api/products/4/review/34. Here I have changed the name from reviews to review and hence we are getting a message “Response Not Found” by app.all() method.

By this example, we can understand that app.get() provides an error message only when it couldn’t find the URL it doesn’t take responsibility for the placeholders/ route parameters like (productid and review) in the request URL.





Query String Parameters

Query String Parameters are also known as URL Parameters. Essentially this is the way for us to send amounts of information to the server using the URL.

This information is usually used as parameters for database query or filtering the results.

Here are few examples to have a look at the Query String Parameters. (HackerNews Story API)

Basically, they won’t be the part of the URL and are separated by a question mark (?) in the URL.

All stories matching foo

http://hn.algolia.com/api/v1/search?query=foo&tags=story

All comments matching bar

http://hn.algolia.com/api/v1/search?query=bar&tags=comment

All URLs matching bar

http://hn.algolia.com/api/v1/search?query=bar&restrictSearchableAttributes=url

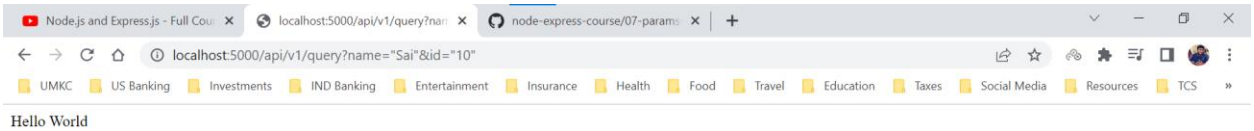
Whatever is after the question mark is not technically part of the URL meaning it is just a way for us to send that data to the server and then server decides what to do with this data.

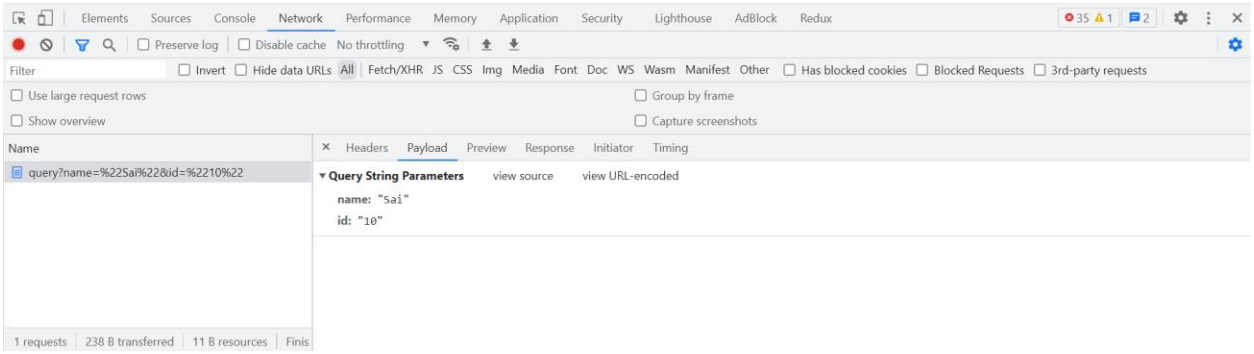
The phase after question mark is nothing but a specific information about the data a user is requesting to the server.

We can add how may query parameters, but they need to start after question mark (?) and all of them need to be combined with ampersand (&), based on the query parameters we can develop the functionality and provide the response.

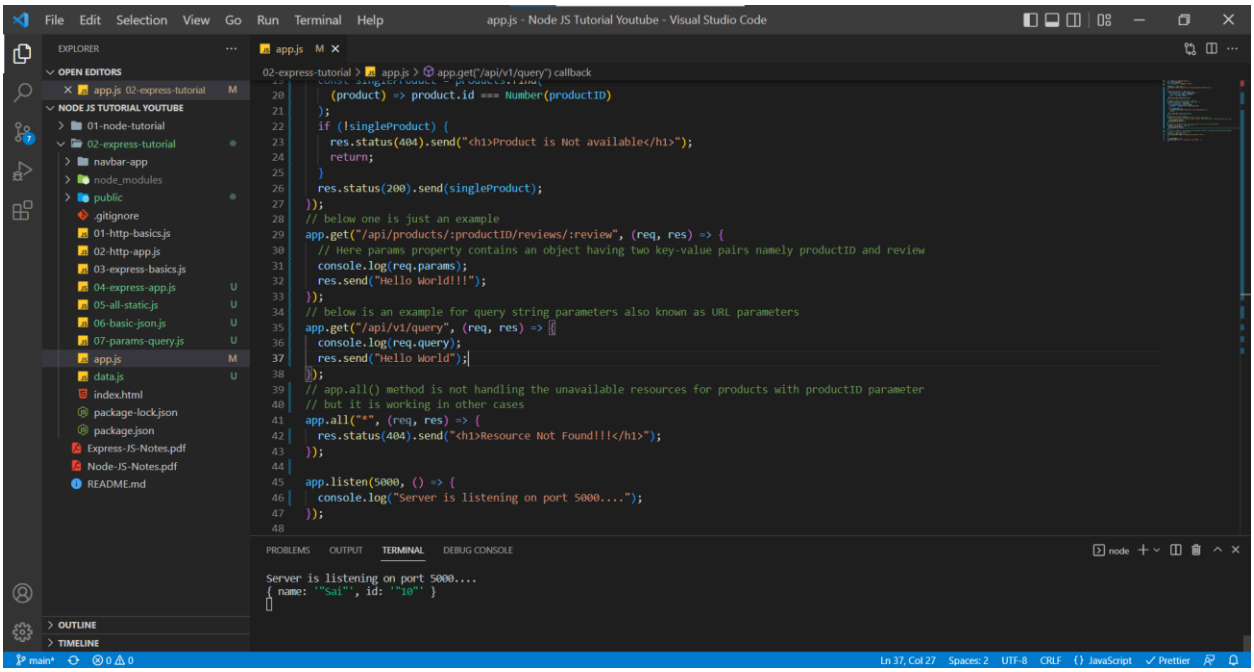
We can get the parameters as part of the request body, and we can retrieve them using the property named ***query***.

http://localhost:5000/api/v1/query?name=”Sai”&id=”10”

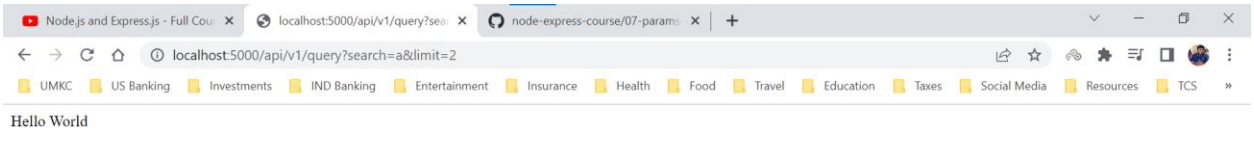


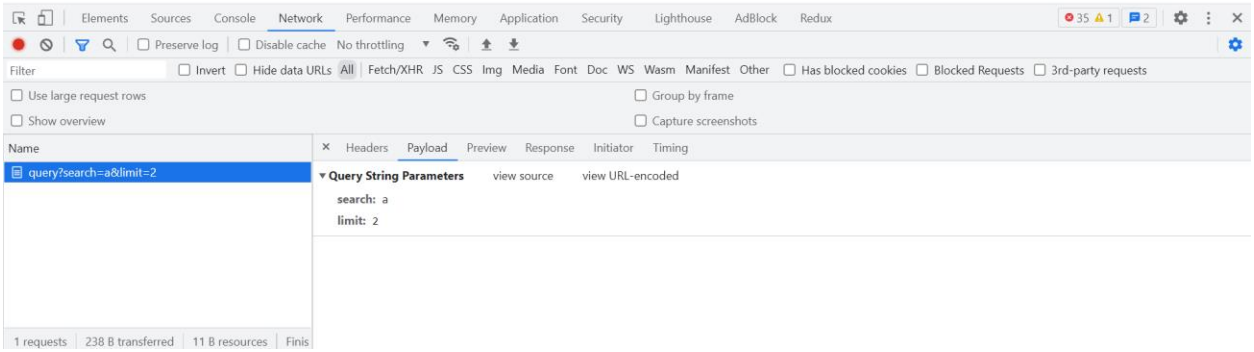


We are console logging the query parameters in the terminal and can see the query parameters by **req.query** property.

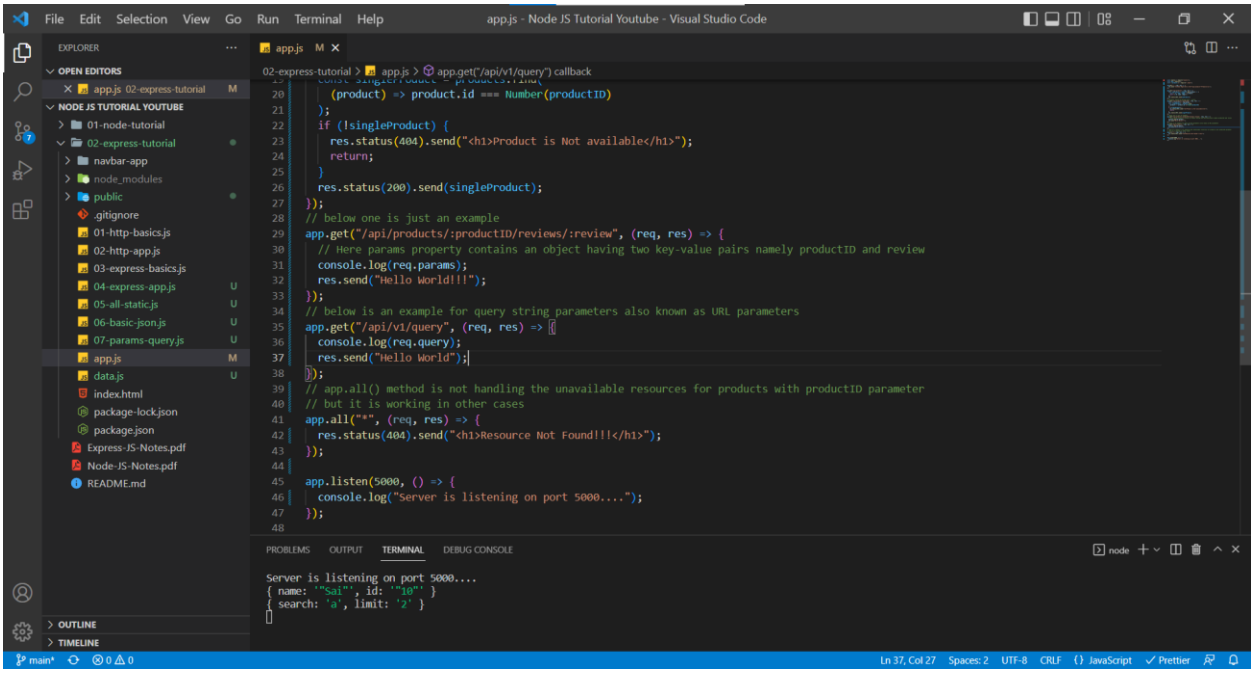


localhost:5000/api/v1/query?search=a&limit=2





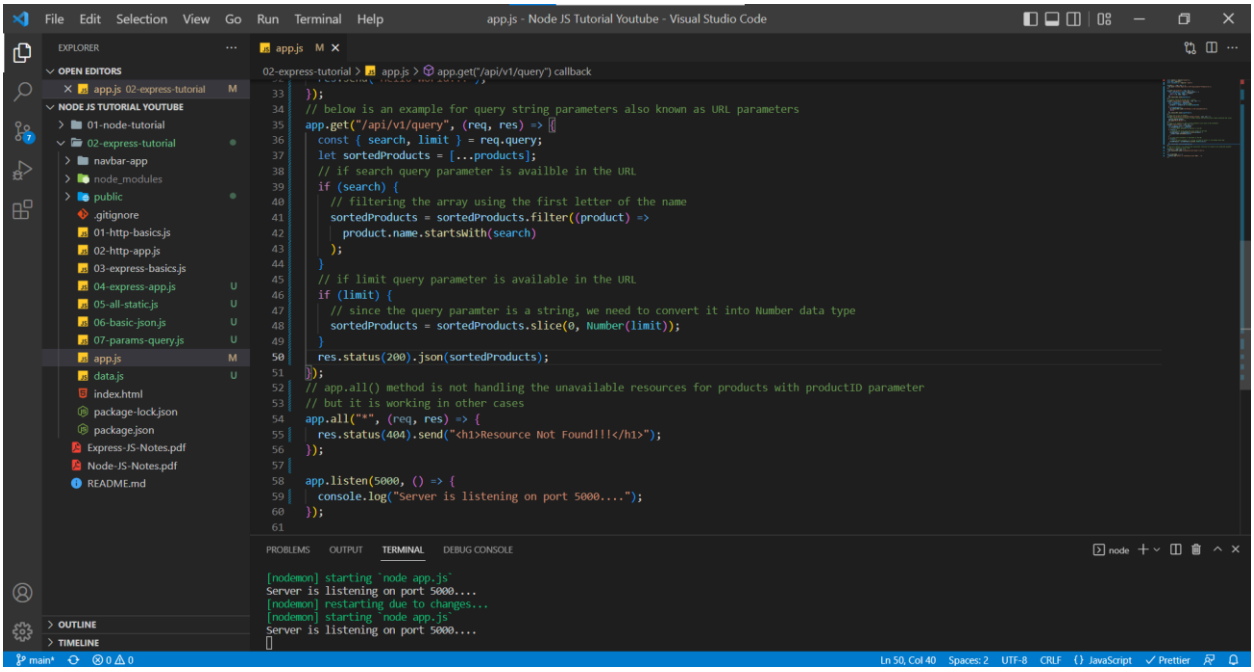
We can see those query parameters in the terminal



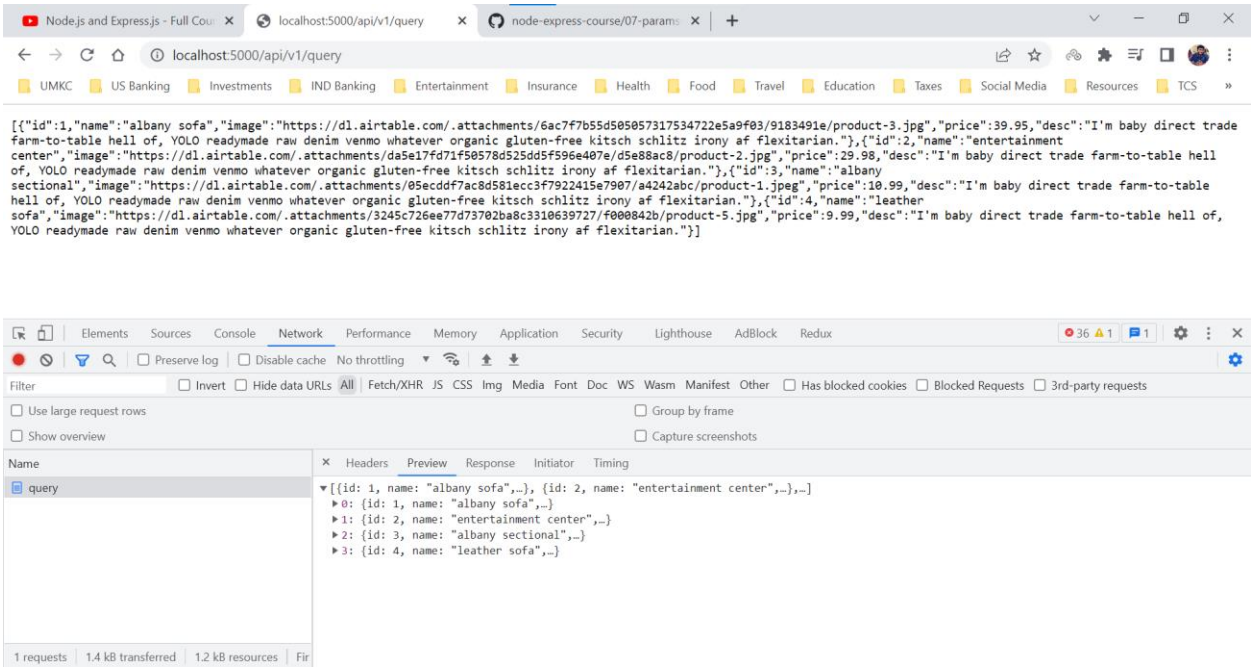
Note: The data we pass through query parameters are always a string, we don't to send them as string they will be automatically converted to string and sent to the server.

Here we are creating a scenario where we expect the user to send two query parameters namely search, limit and if we don't get any of those parameters, we will display all products but if we receive those query parameters, we will display the data accordingly.

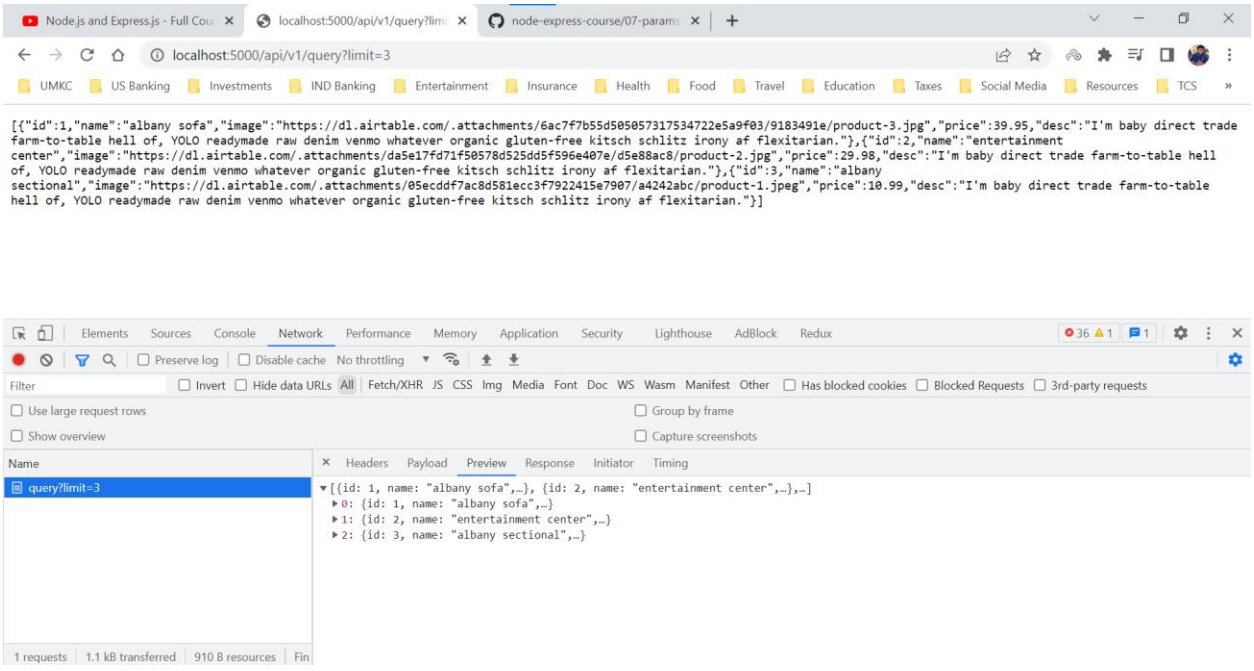
app.js



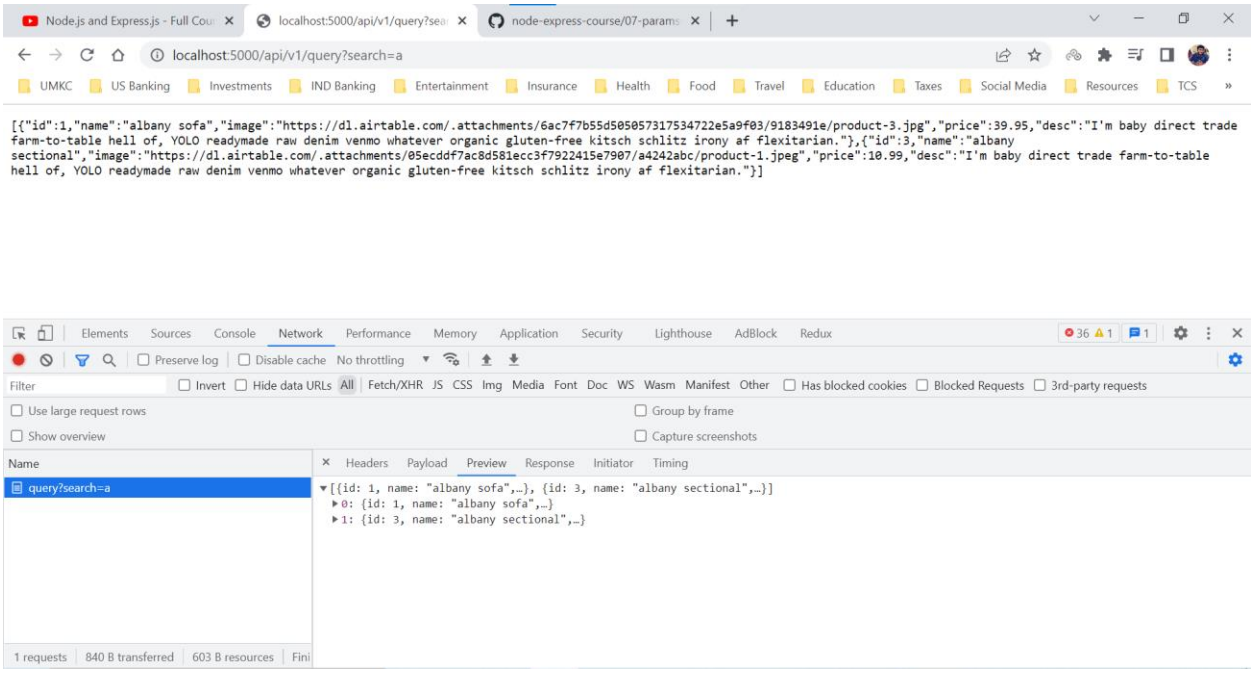
http://localhost:5000/api/v1/query - With No Query Parameters



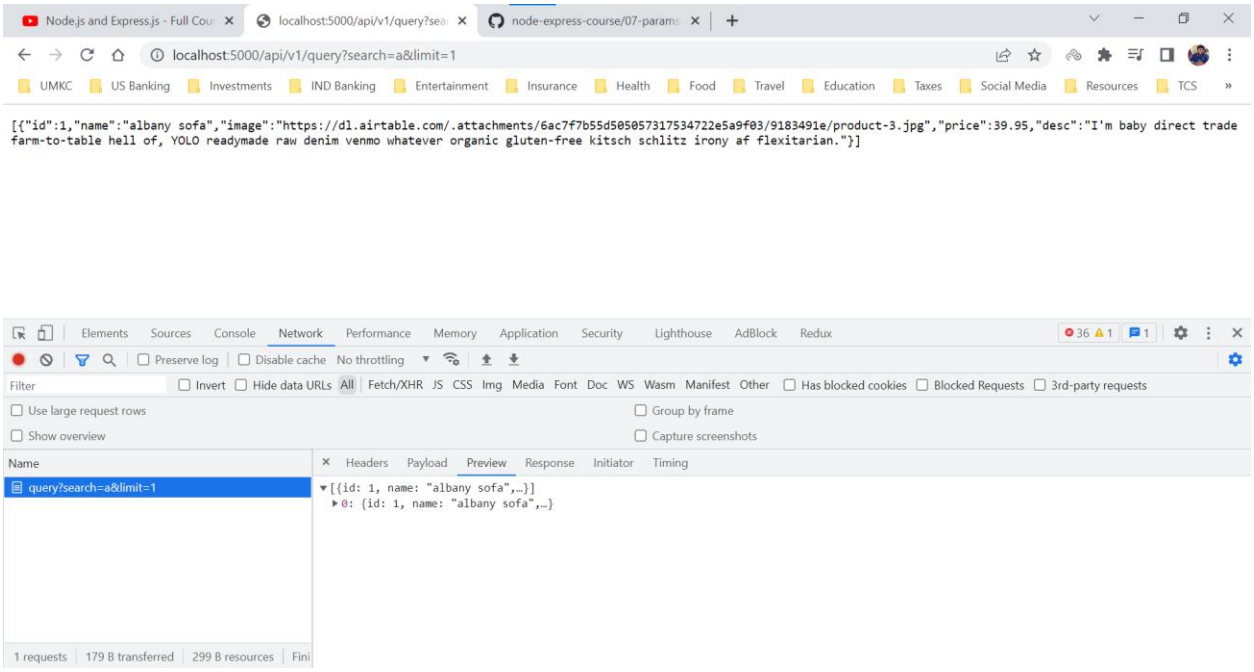
http://localhost:5000/api/v1/query?limit=3 - With only limit query parameter



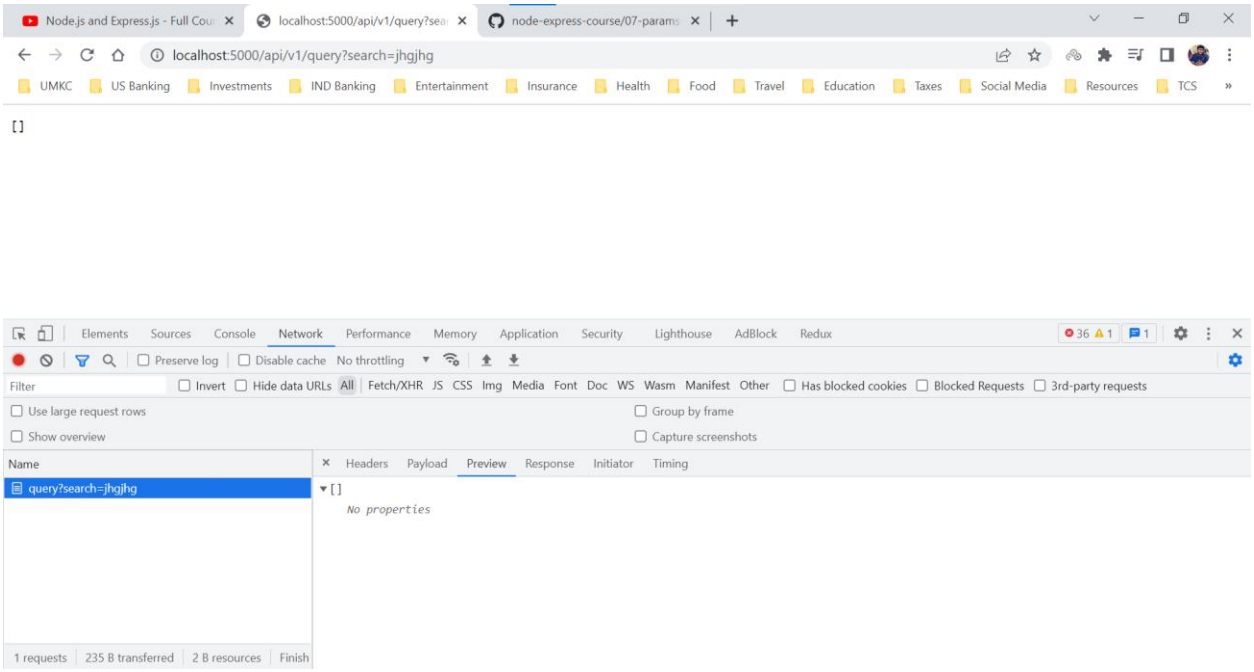
http://localhost:5000/api/v1/query?search=a – With only search query parameter



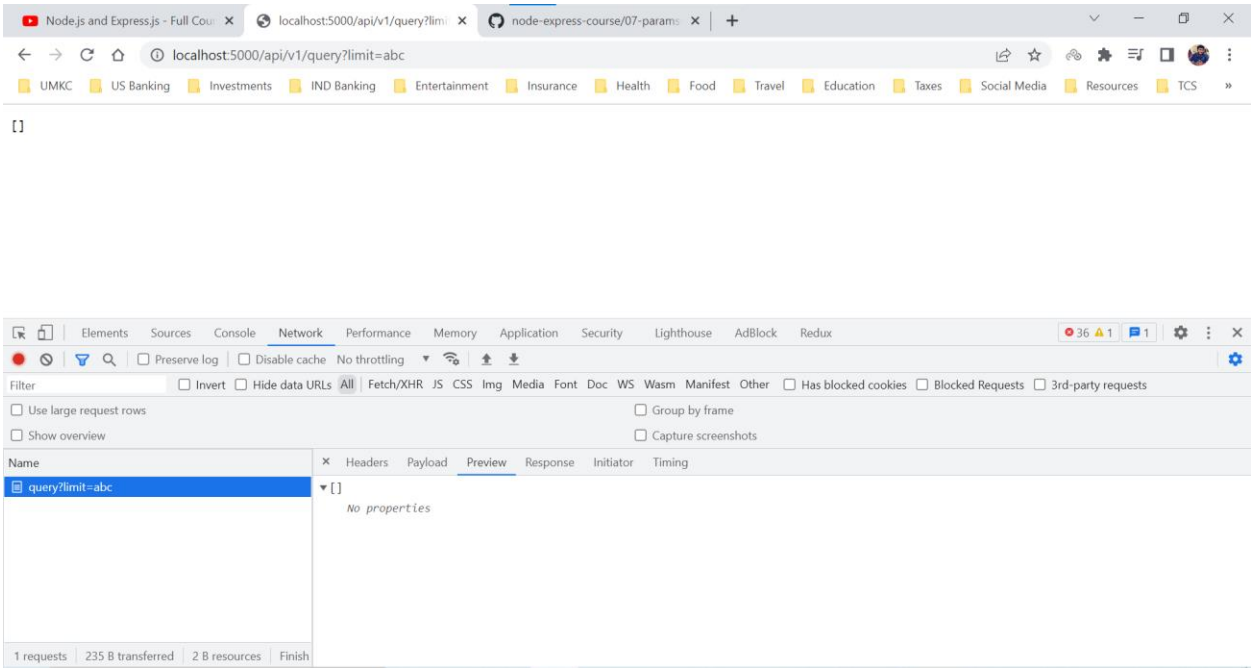
http://localhost:5000/api/v1/query?search=a&limit=1 – With both search and limit query parameters



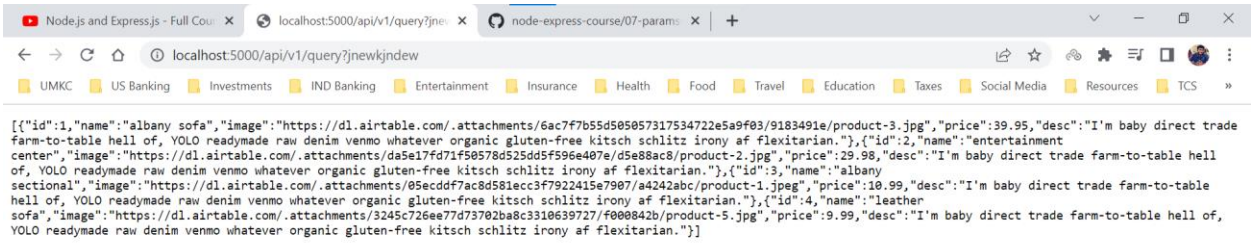
http://localhost:5000/api/v1/query?search=jhgjhg – with unavailable search parameter

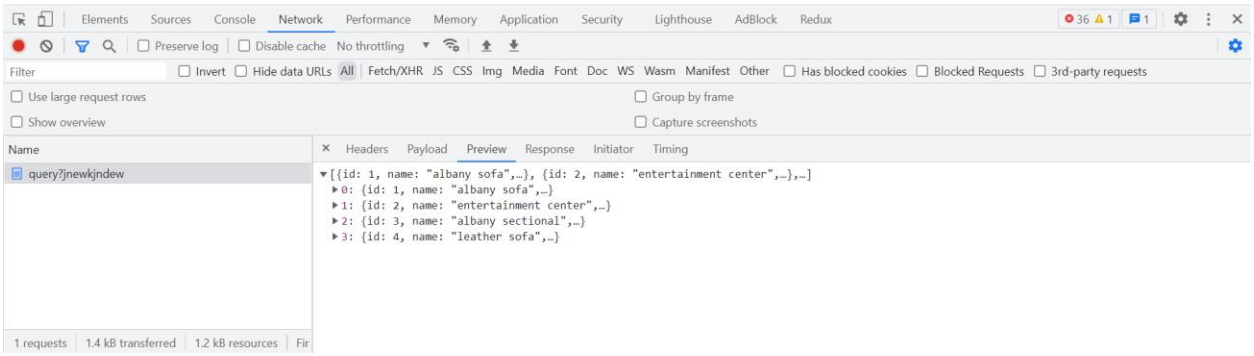


http://localhost:5000/api/v1/query?limit=abc – With unavailable/ wrongful limit parameter

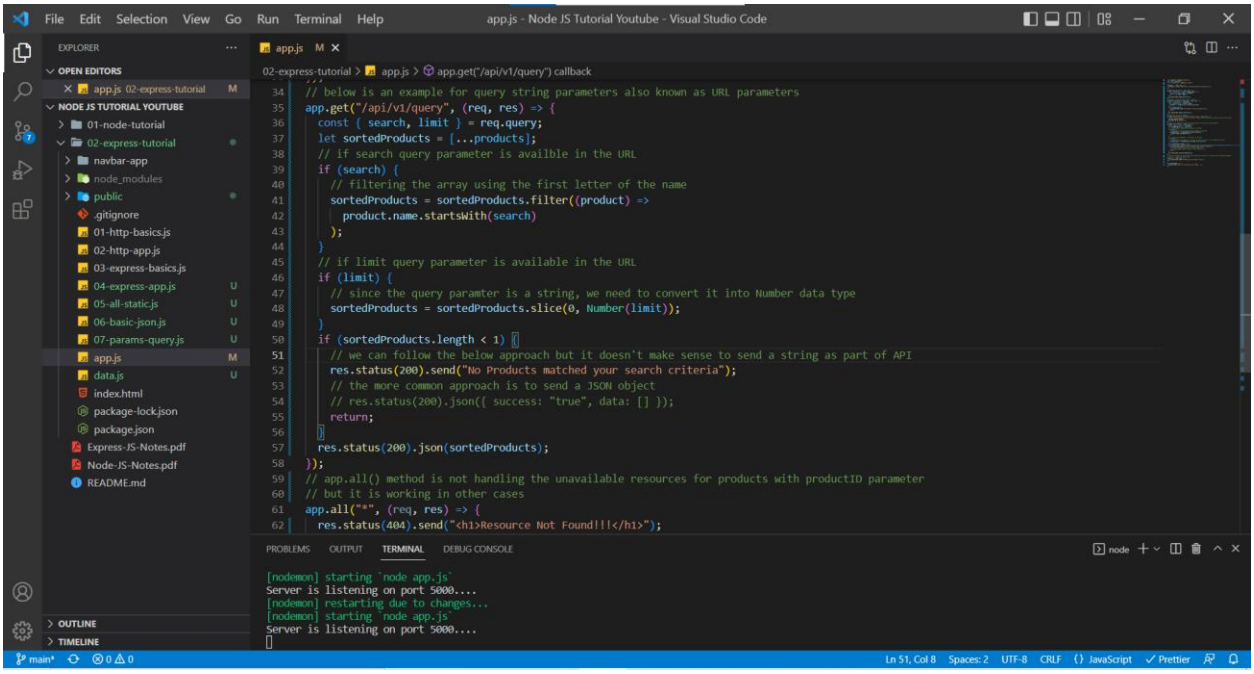


http://localhost:5000/api/v1/query?jnewkjndew - with random value after question mark(?)

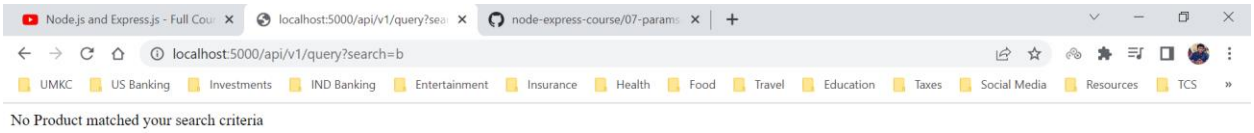


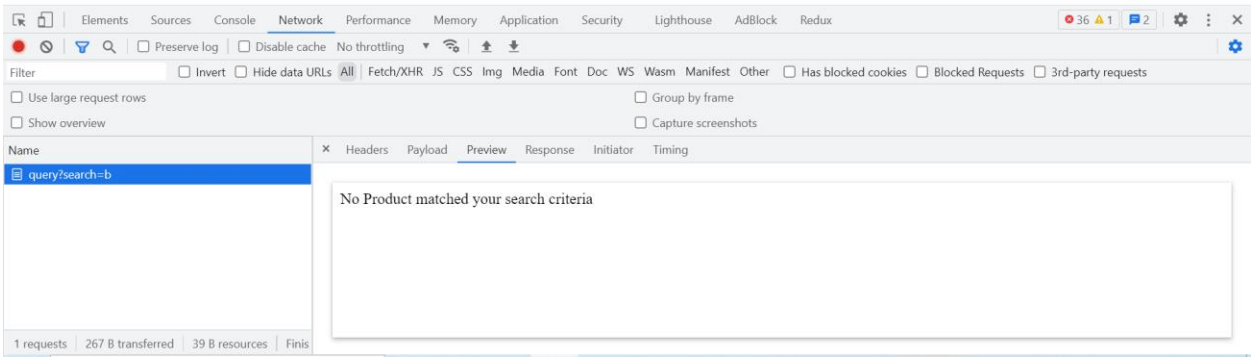


We can handle the display of empty array by adding a small piece of code. We can check the length of the array we are sending as a response.

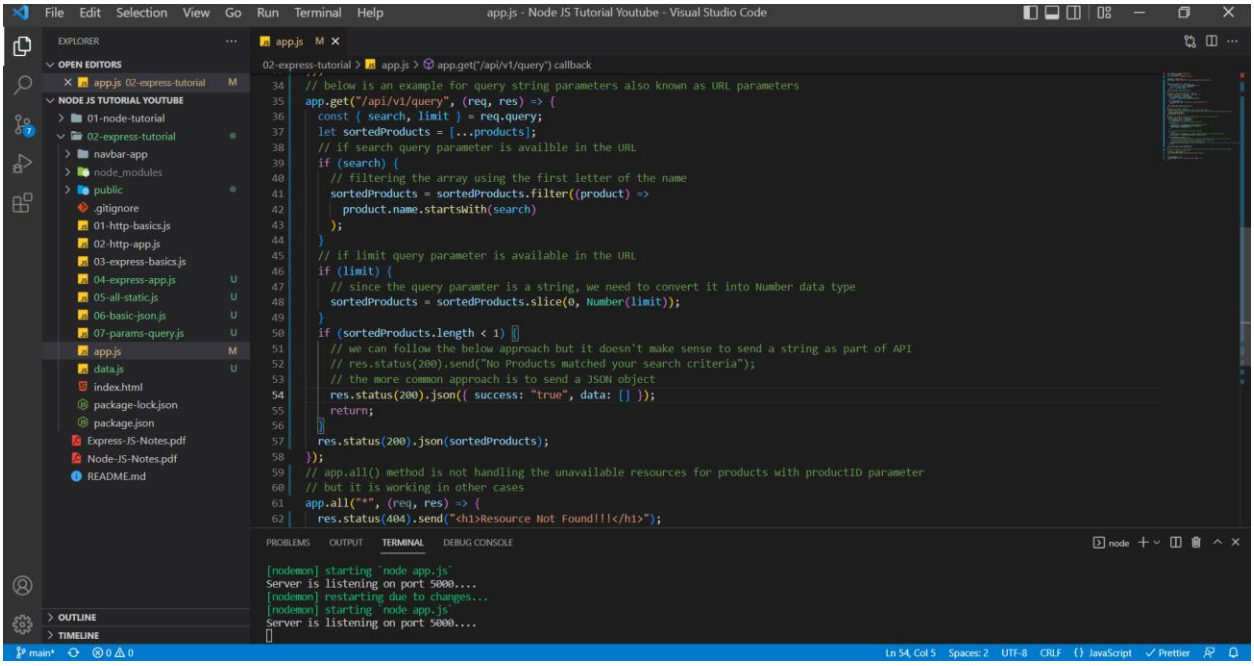


http://localhost:5000/api/v1/query?search=b – when no products match the criteria

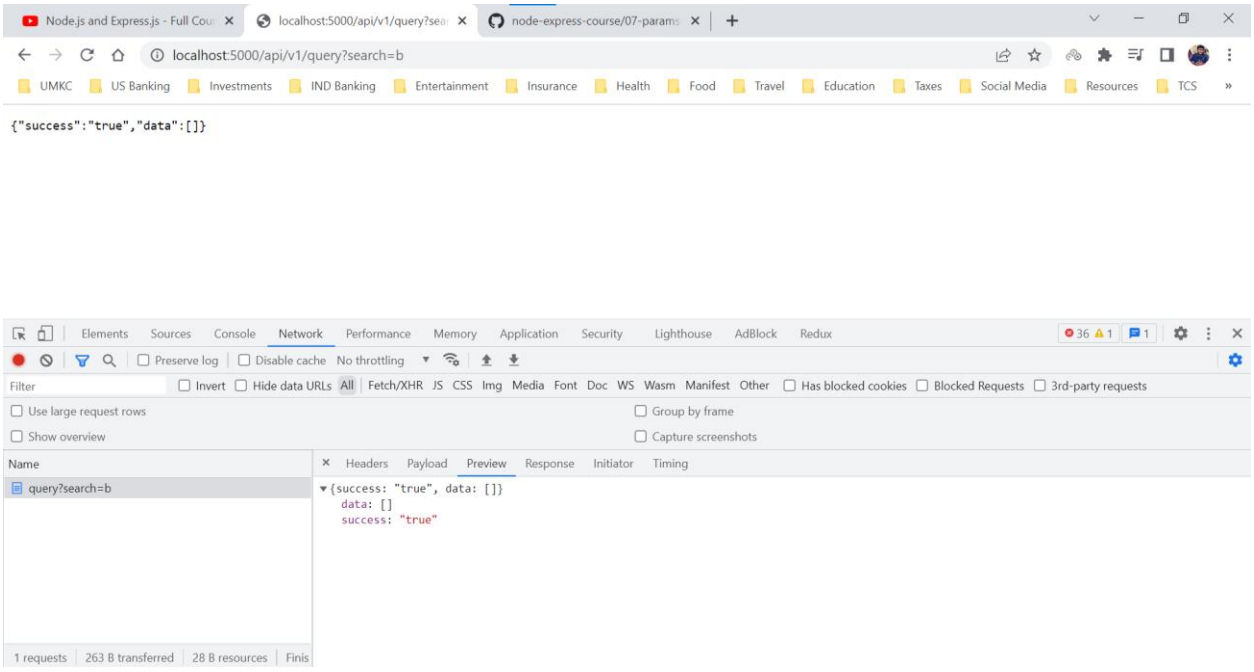




The more common approach is to send a JSON object even if we couldn't find the data for the request. We do follow this format because our request was successful but we don't have any data.



http://localhost:5000/api/v1/query?search=b - when no products match the criteria



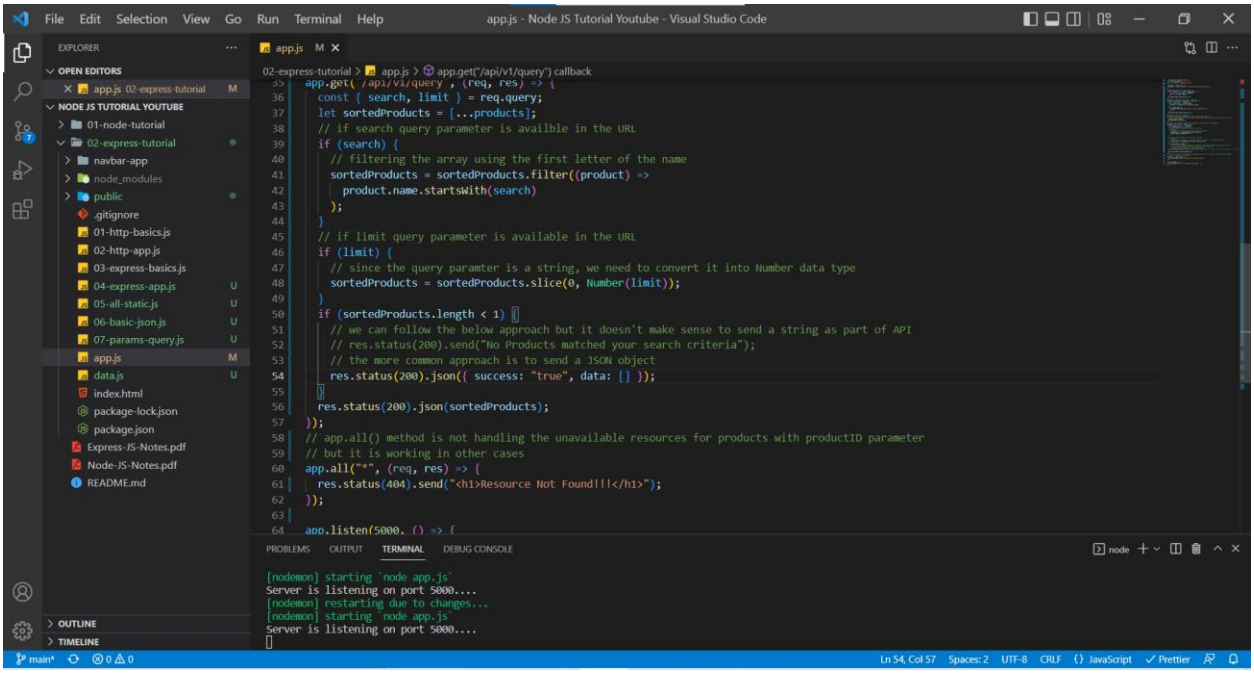
Additional Params and Query Info

We may have a doubt that why are we returning the res.status().json() in the if condition where we don't find any items using the search criteria.

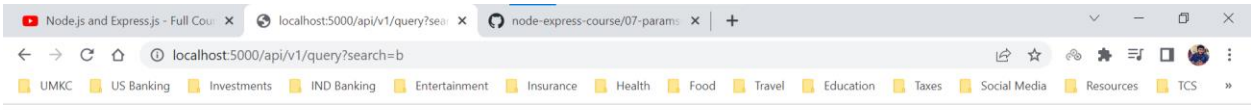
In JavaScript if we don't explicitly return then javascript keeps reading the code, so if we emit the return, we get a server error where we send back one response and javascript just keeps reading the code and the express is confused because it already sent a response so why we are sending another response in the same request.

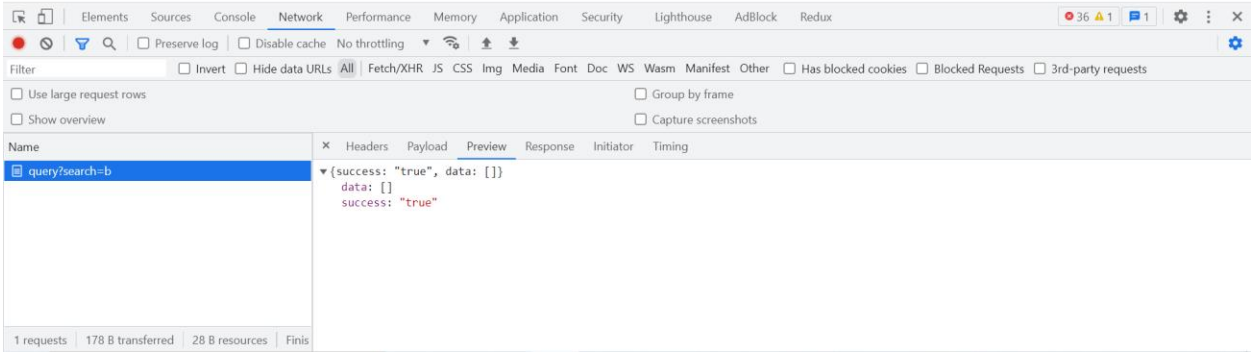
So basically, we cannot send two responses for the same request one after the another. Yes, we can send one response based on the conditions, but we can't send one after another in the same request.

app.js

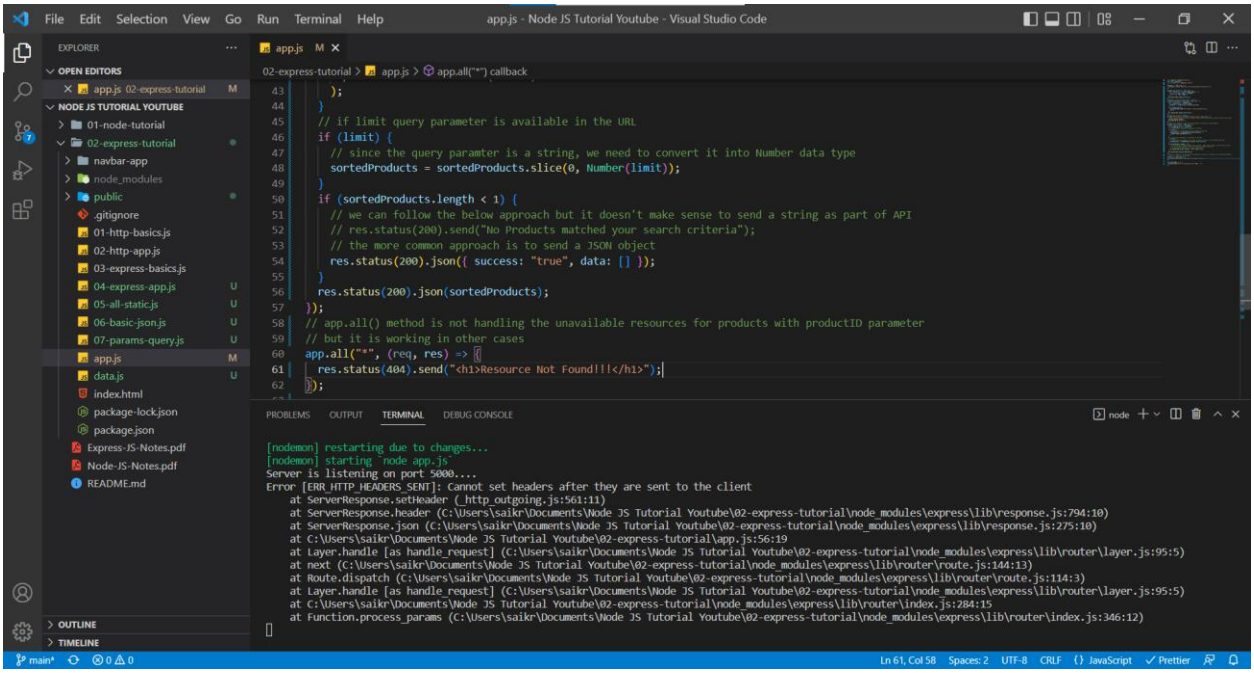


http://localhost:5000/api/v1/query?search=b - when no products match the criteria



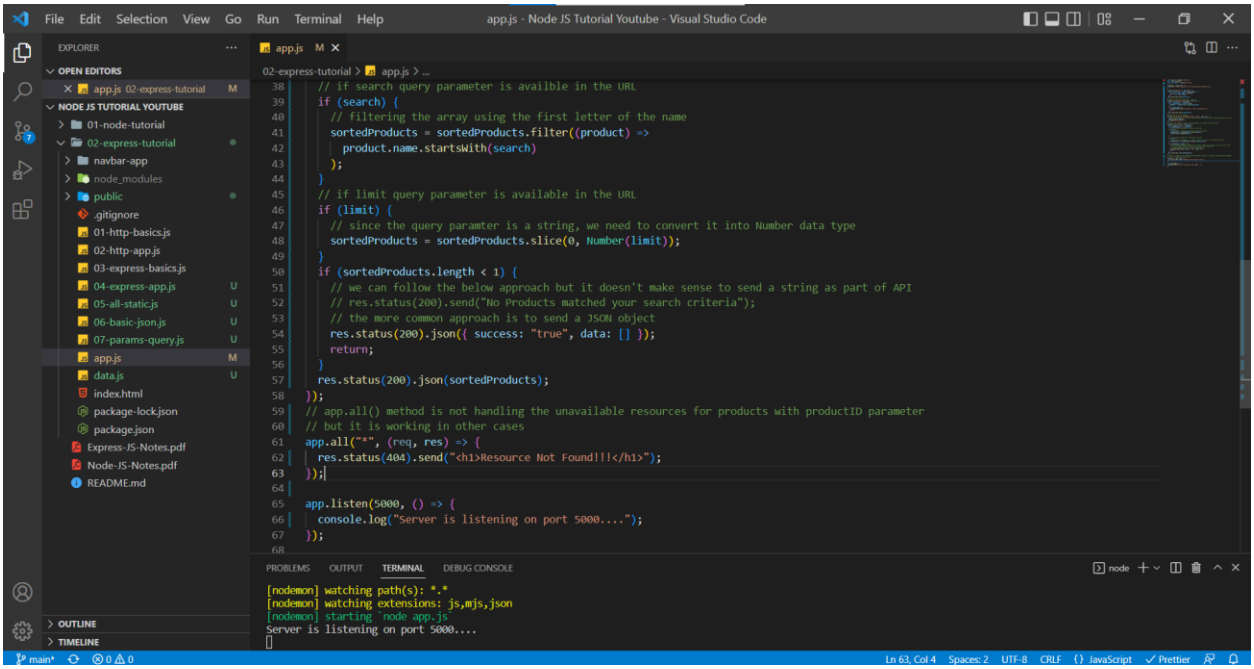


But at the once the request comes in at the same time our server breaks down in the terminal.



So ,we can only one response per request. In order to avoid the above mishap, we always go with the return statement inside the explicit conditions.

app.js



Middleware – Setup

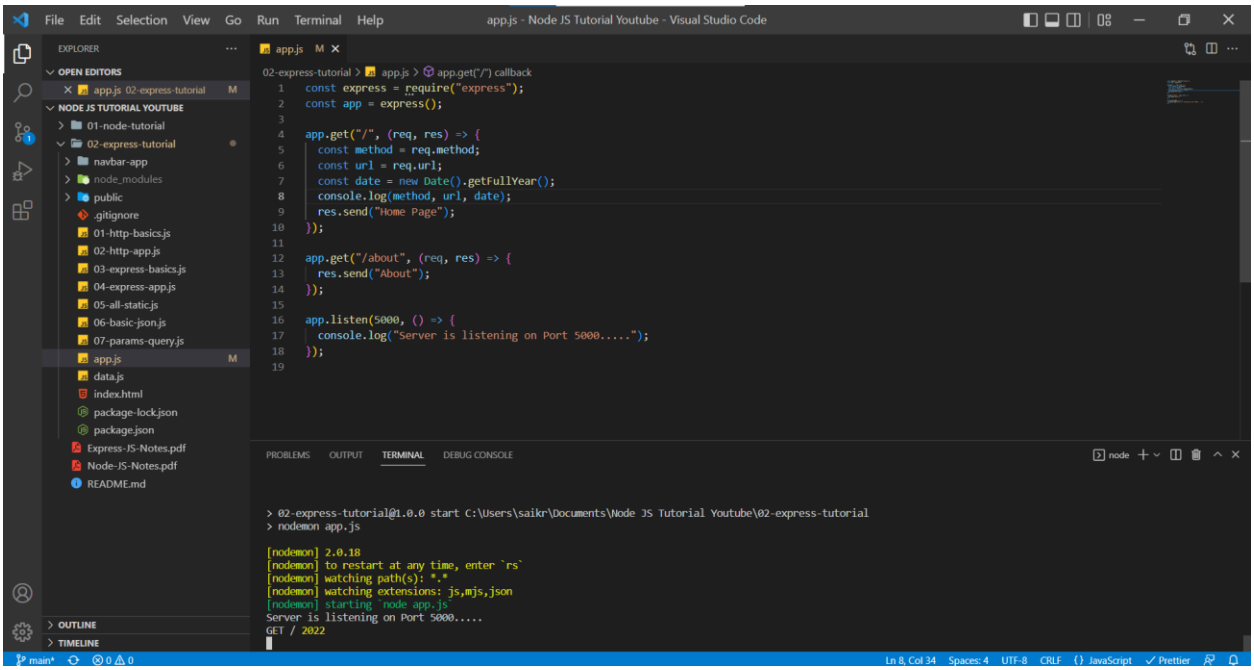
Express Middleware are functions that execute during the request to the server. Each middleware function has access to request and response objects and when it comes to functionality (literally sky is the limit).

Middleware is everywhere in Express. Express is nothing but a bunch of middleware functions stuffed together to make one nice express cake or desert. It is heart and soul of the express.

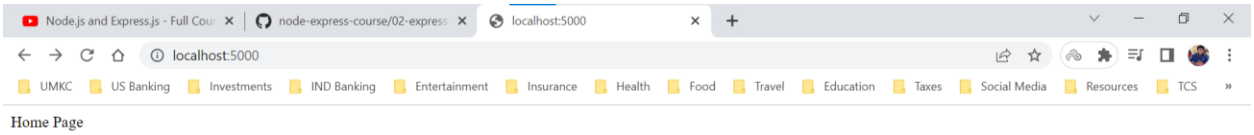
Middleware sits between request and response. When the request comes in, we do some functionality and then send the response.

In this scenario, we are going to have a two routes namely Home Page and About Page.

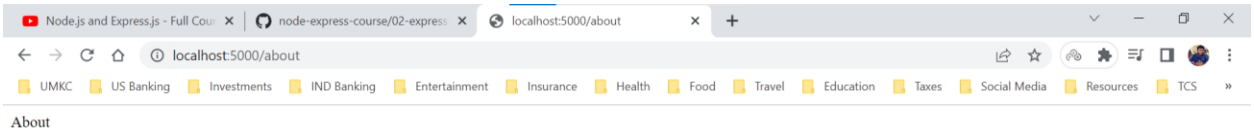
app.js



localhost:5000/

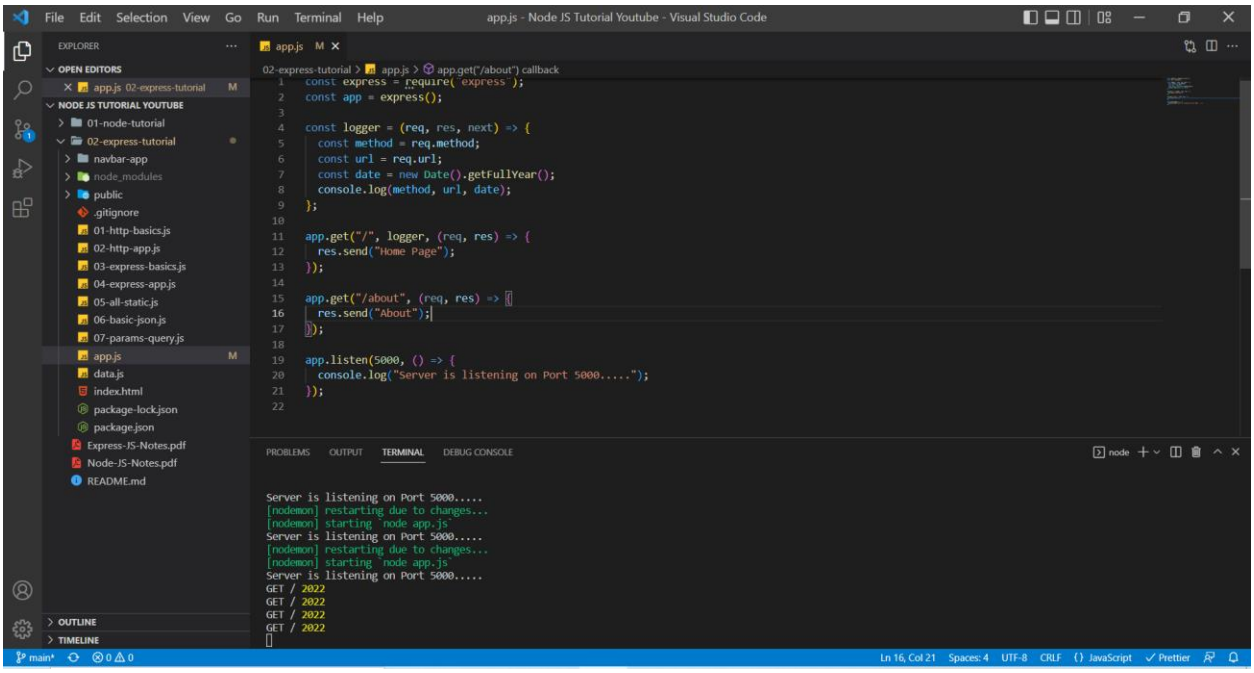


localhost:5000/about



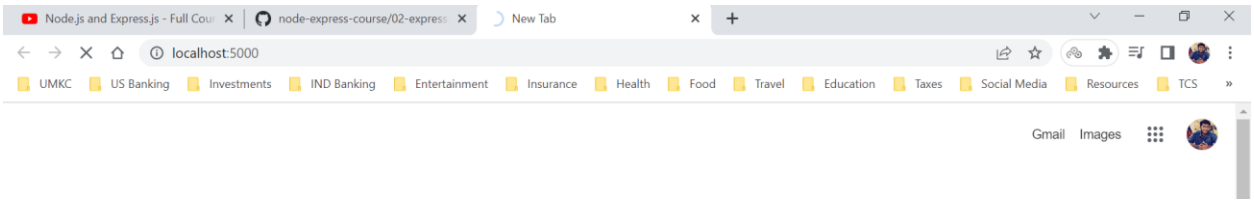
Assume, in this scenario we want to have the same logic of displaying method, URL and full year we can write the same logic for each URL request. To solve this issue, we can create function and write the logic over there and use that function in each URL request.

app.js



We created a function named logger and shifted the logic into that function. We are using that function as parameter in the app.get() method.

localhost:5000/

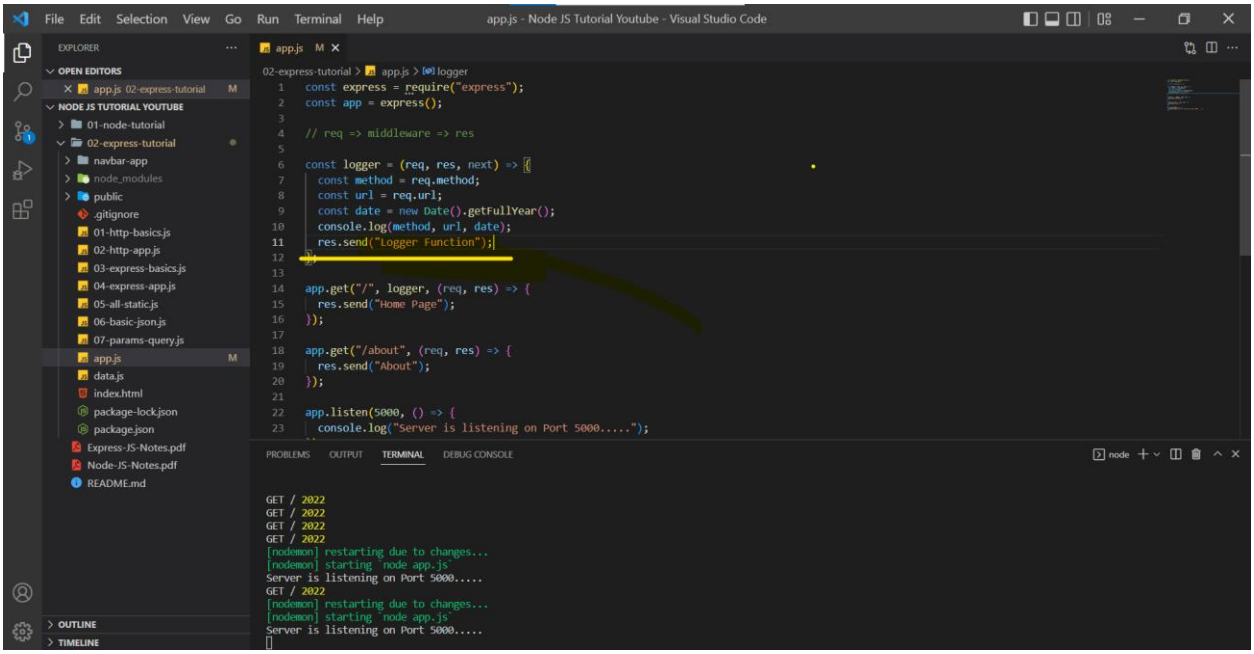


But here we can see the page is still loading and not providing the data. This is because Logger function that we are using is not terminating the response or it is not passing to next middleware in the server.

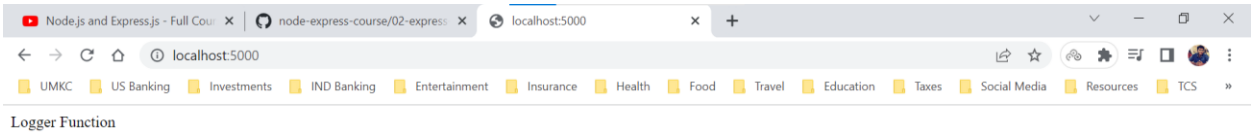
This issue can be solved in two ways

1. Terminate the response in the function (logger function) itself by sending response object from that function.

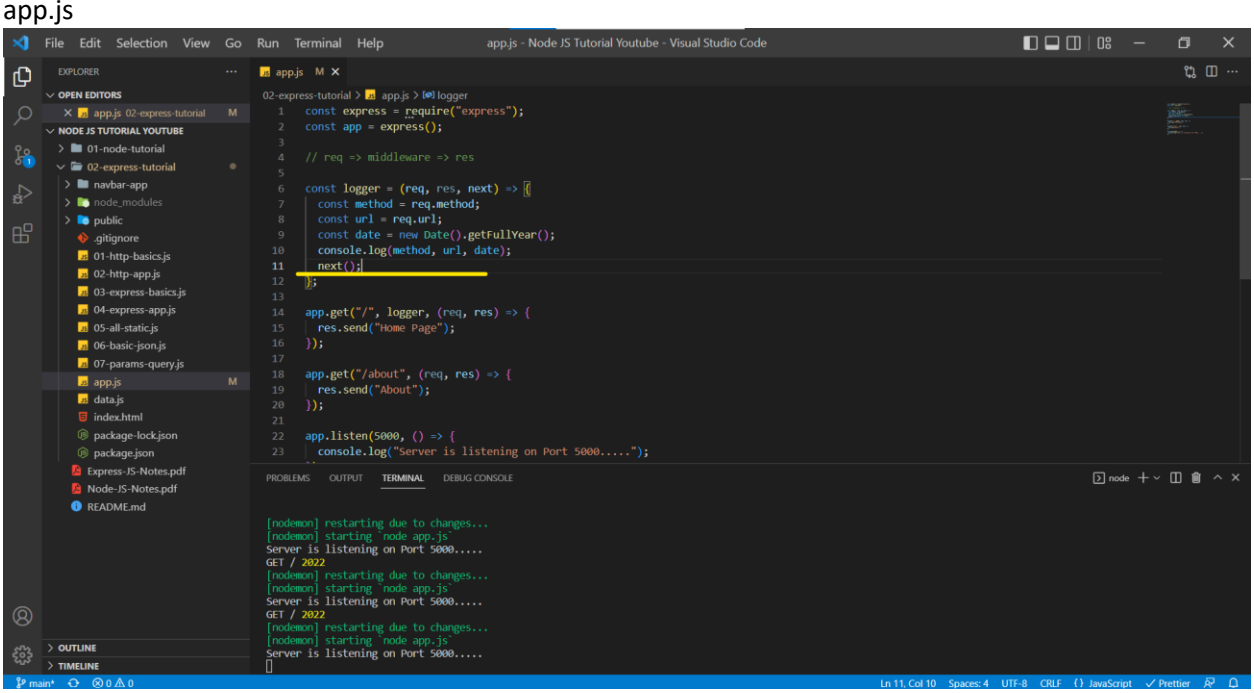
app.js



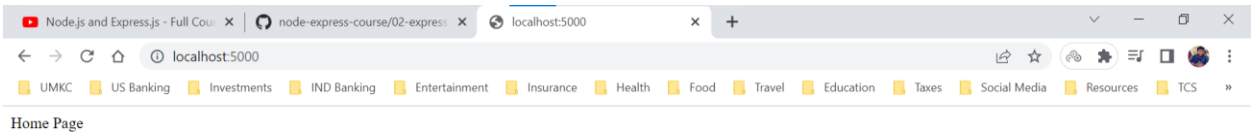
localhost:5000/



2. We execute the function (logger function) and can pass it over to next middleware (in this case it would be app.get("/") method since it is using the logger function as parameter)
app.js

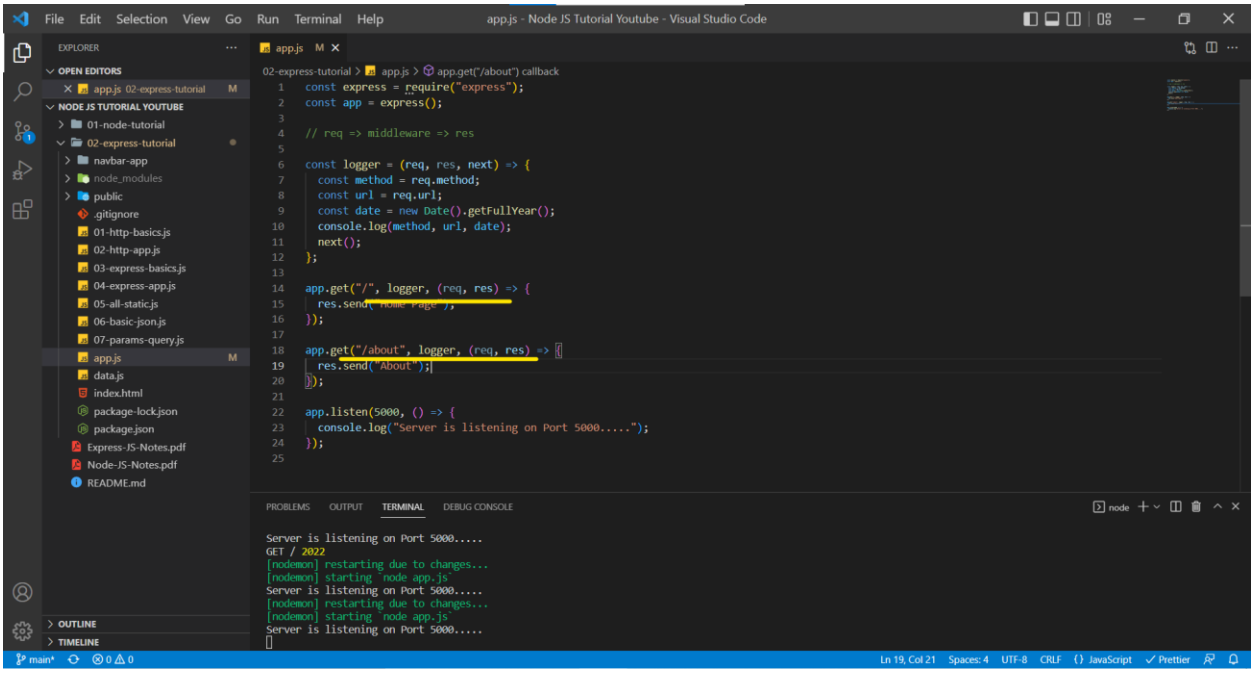


localhost:5000/



We can use the same logger function for any request from the user.

app.js



Express provides request and response object to logger function or any other function in the server. Logger function written above is the first middleware function we have ever written.

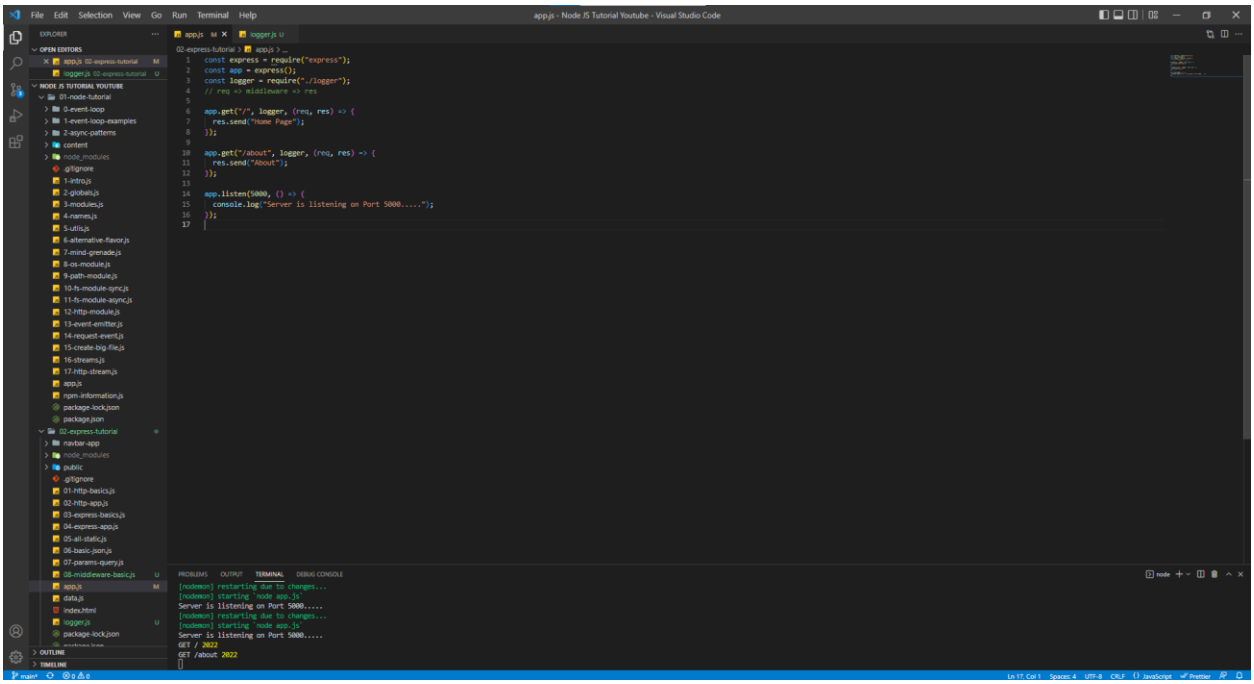
APP.USE Method

There are two issues with the current setup. The logger function is the first middleware function we have ever written.

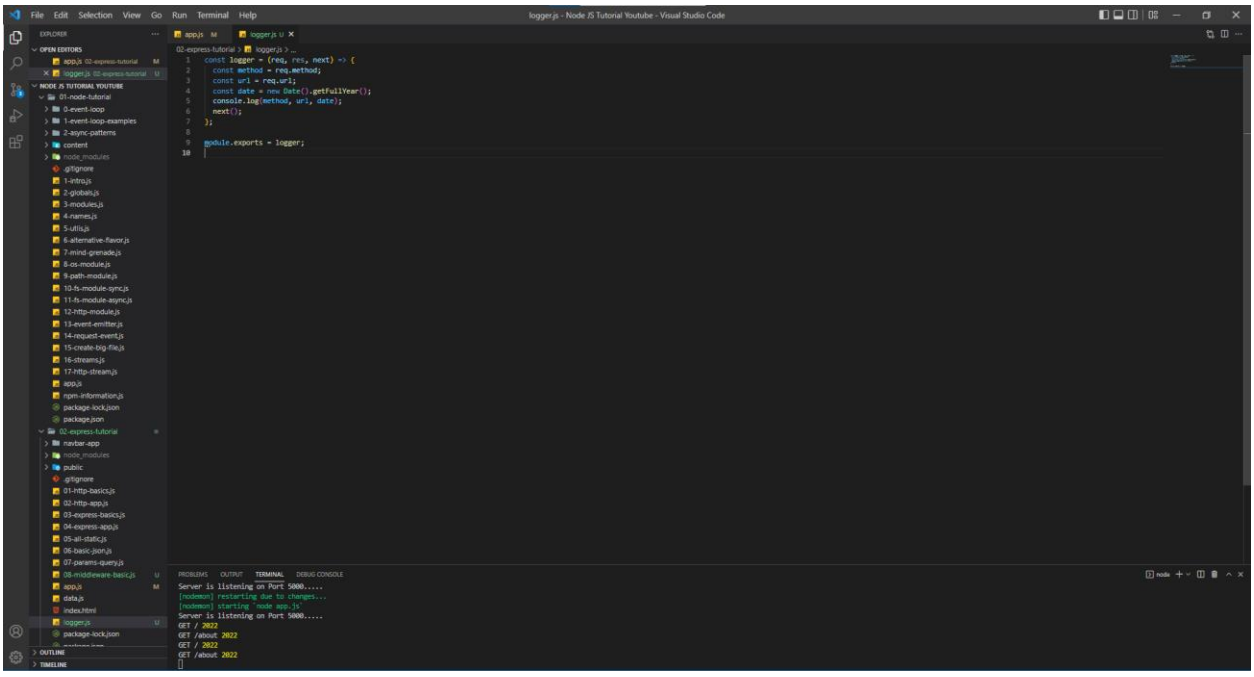
1. Our app.js is getting clunky because we have logger function and other requests in one page. (We can have a logger function in a separate file). This approach will keep our app.js file lean.
2. Assume a scenario where we 50 more routes and we don't want to add this logger function to each route manually. (It would be nicer If there would be a method that just adds my middleware function to any route.

To solve the first problem, we create a separate javascript file and post the logger middleware function code over there and export the code through ***module.exports*** and in the main application we import it using the ***require*** global variable.

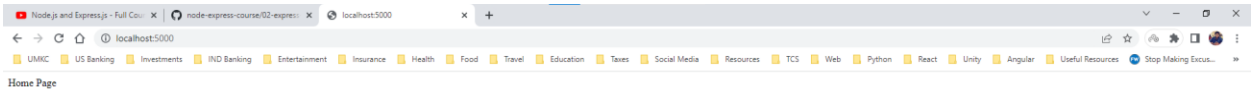
app.js



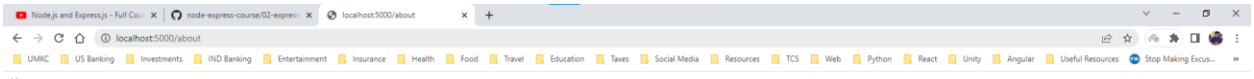
logger.js



localhost:5000/

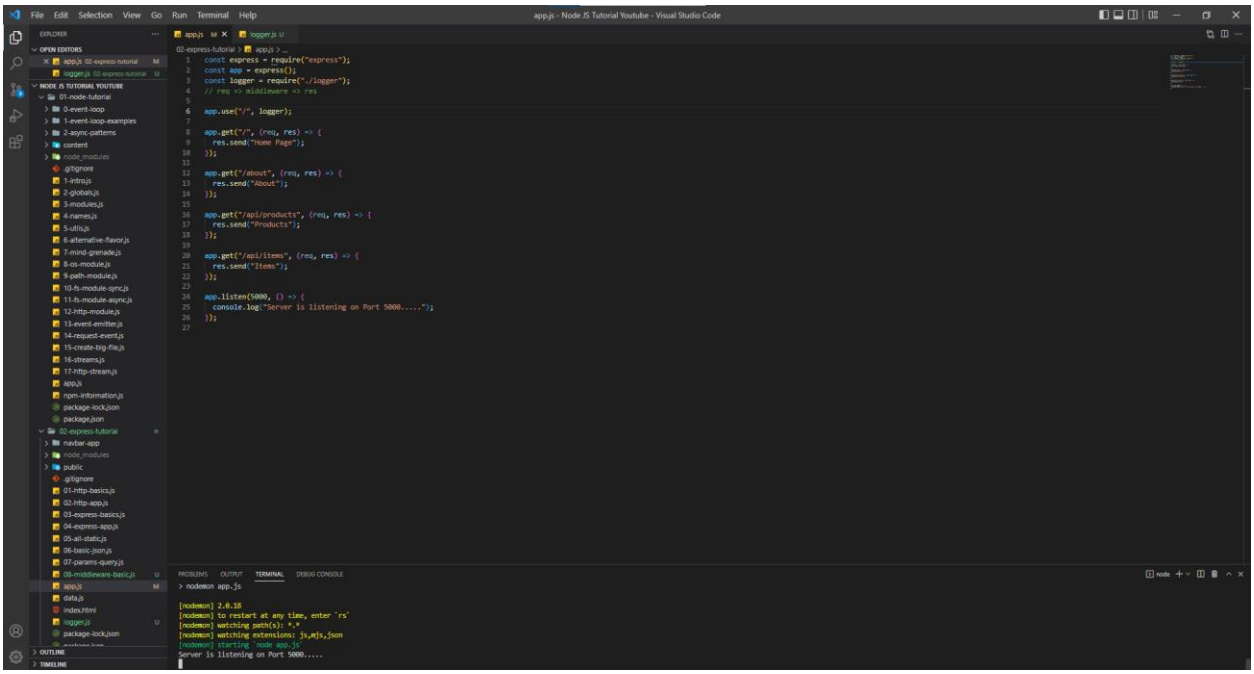


localhost:5000/about

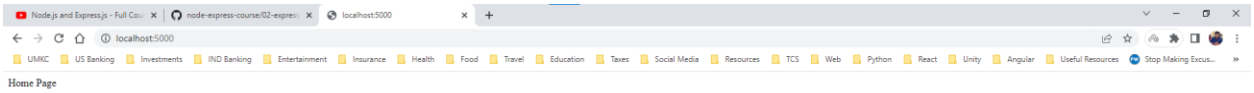


The second problem can be solved by using the app.use() method which we applied to all the requests the server receives. If we add app.use() method at top of the requests, then it will add to all the request but if it as added after one request then it won't apply for the top request, but it will apply for the bottom requests.

app.js

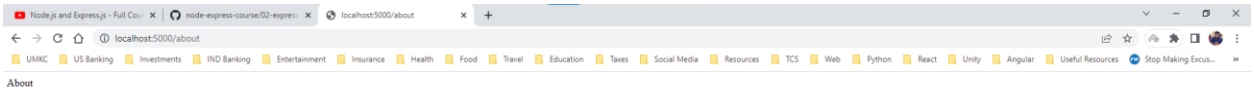


localhost:5000/



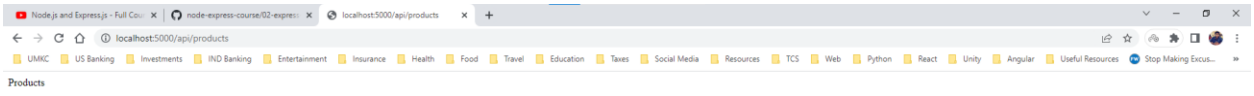
Terminal (Here we get console.log statement from the logger middleware function)

localhost:5000/about



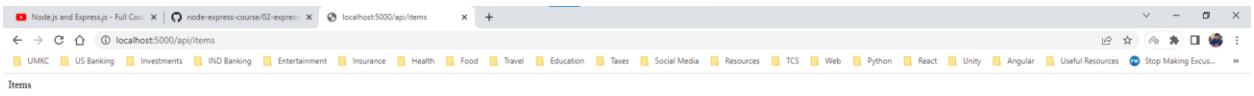
Terminal

localhost:5000/api/products



Terminal

localhost:5000/api/items

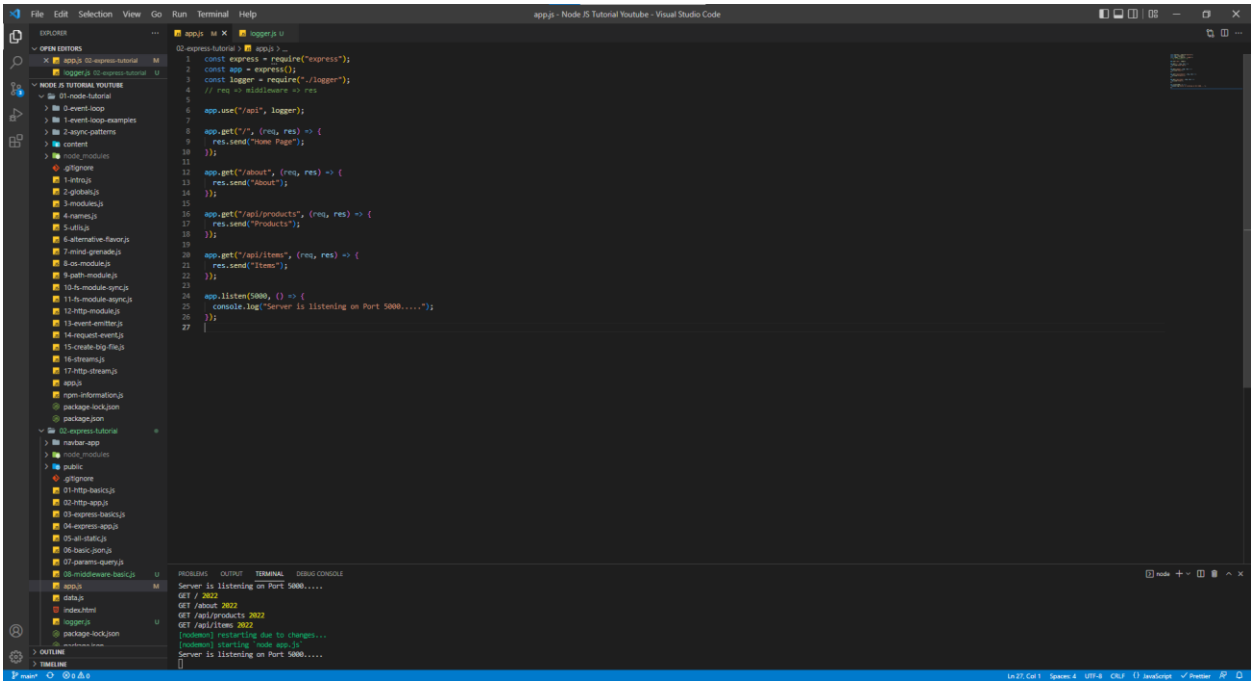


Terminal

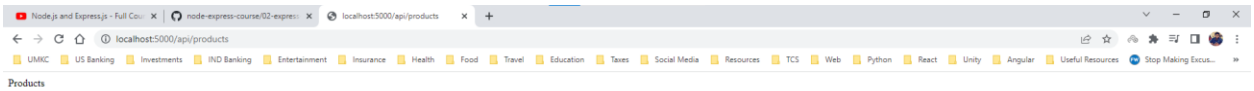
We can also add the parameters of the URL path to which the app.use() method can be applied, and it contains a callback function having request and response object and we need to terminate the request by sending the request by sending by response else browser waits for response and it will be in a loading status.

Now we have mentioned **/api** in the parameters of app/use() method it will provide logger middle function to the request containing the URL **/api**

app.js

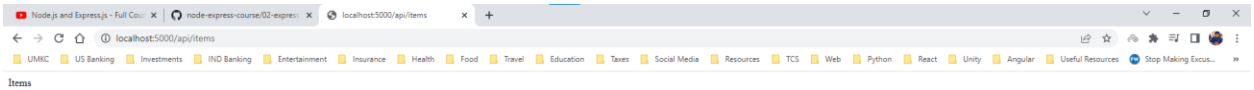


http://localhost:5000/api/products



Terminal (We can see the console log statement of the logger middleware function)

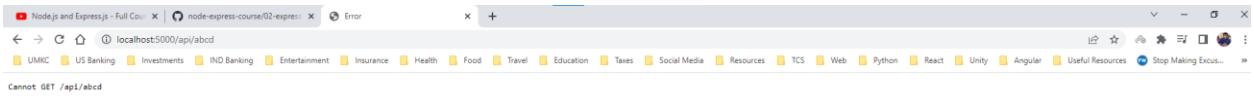
http://localhost:5000/api/items



Terminal

Even though ***http://localhost:5000/api/abcd*** doesn't have a resource in our server but still we console log the statement because we are using app.use("/api") hence it will apply for all URL requests starting with /api.

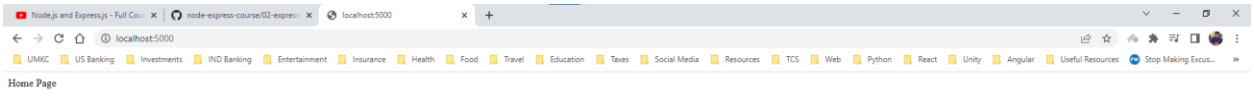
http://localhost:5000/api/abcd



Terminal (We will console log the information)

But if we try to use the home Page route, we will send the response to the server, but we won't be console logging the information because the logger middleware function won't be available for requests whose URL doesn't contain /api.

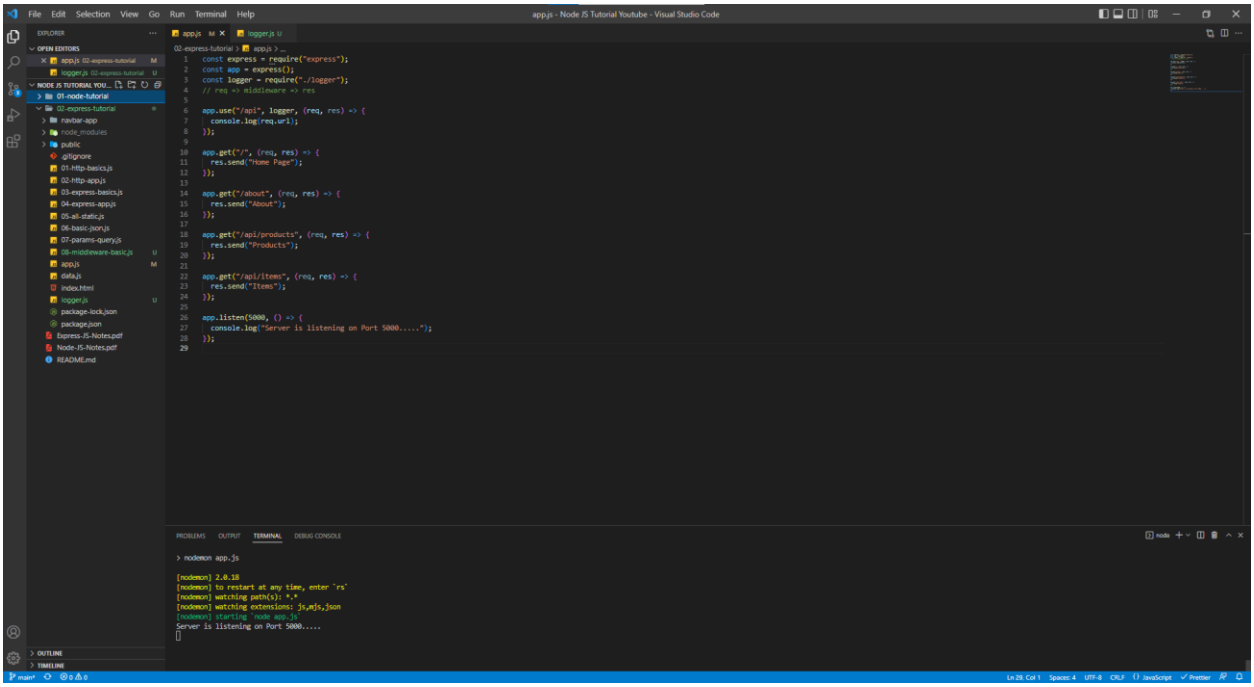
http://localhost:5000/



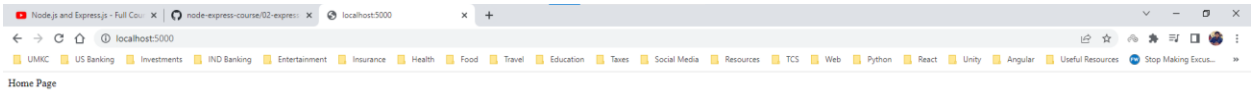
Terminal (We won't see the console log information)

app.use() method can also have the request and response object but if we are using the request and response object, we need to handle them as well.

app.js



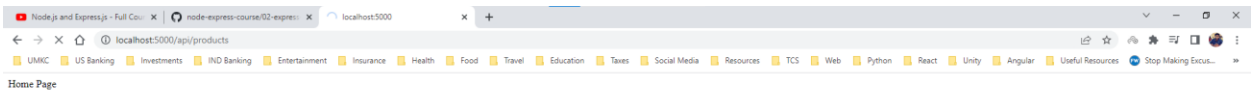
http://localhost:5000/



We won't see anything in terminal because logger middleware is not applied to Home and About routes.

Since we are not sending the response in app.get() method, browser will wait response and it will be loading state itself.

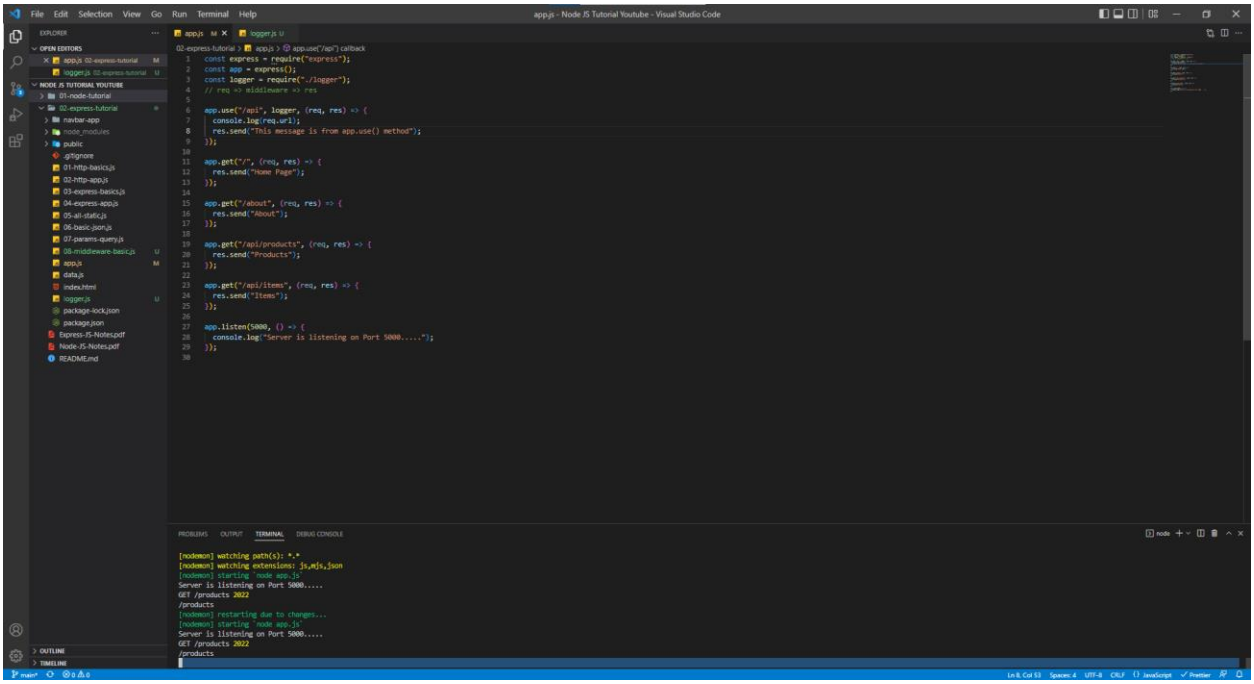
http://localhost:5000/api/products



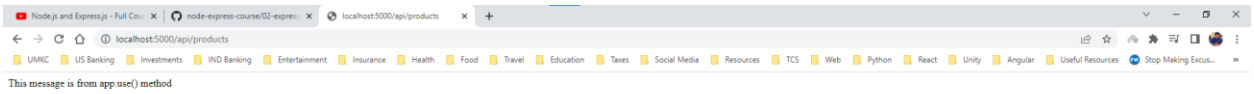
Terminal

We can solve this problem sending the response, but it will the override the other responses.

app.js

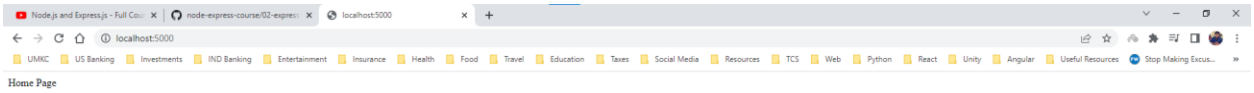


http://localhost:5000/api/products



Terminal

http://localhost:5000/ (app.use method can’t override the home route since it isn’t being applied to Home and About route)

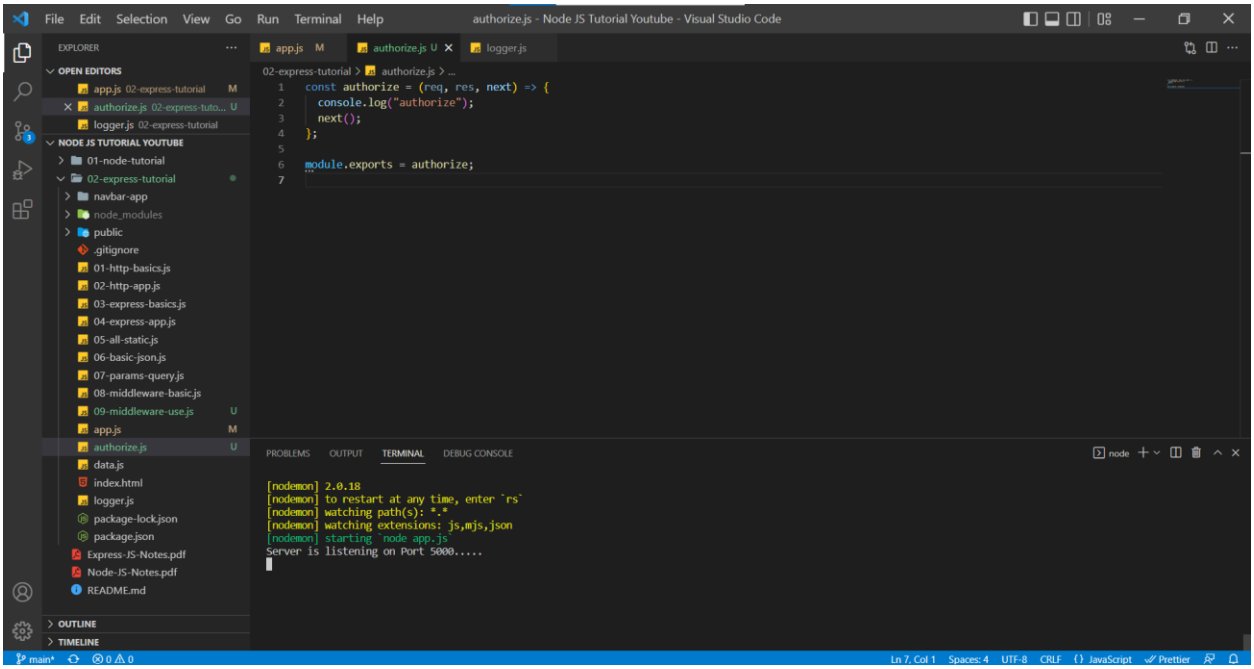


Multiple Middleware Functions

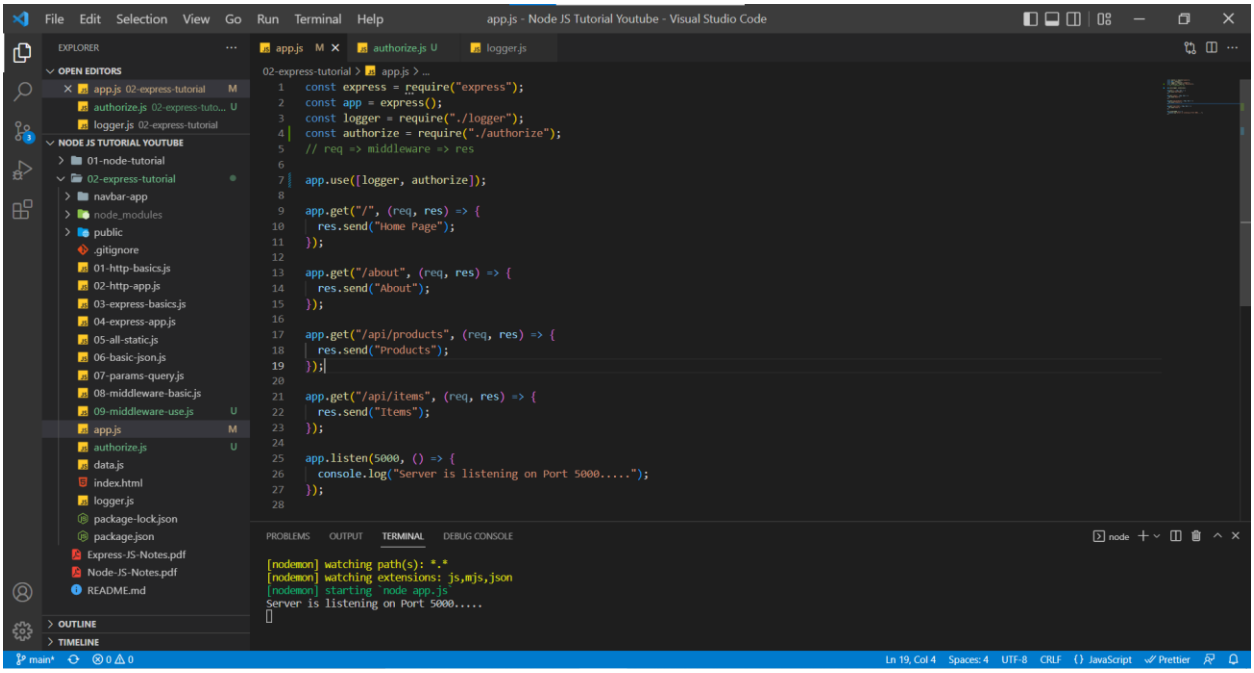
In the earlier scenario, we have seen only 1 middleware function being written but sometimes we need multiple middleware functions which needs to apply to the requests that coming into the server.

Let us create another middleware function named authorize.

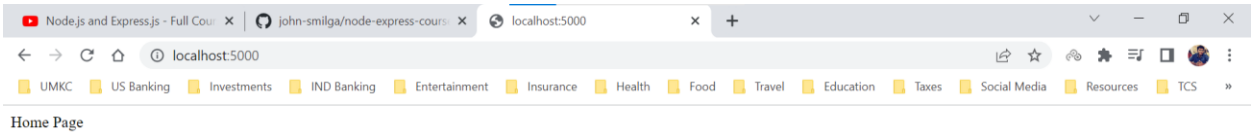
authorize.js



app.js



http://localhost:5000/

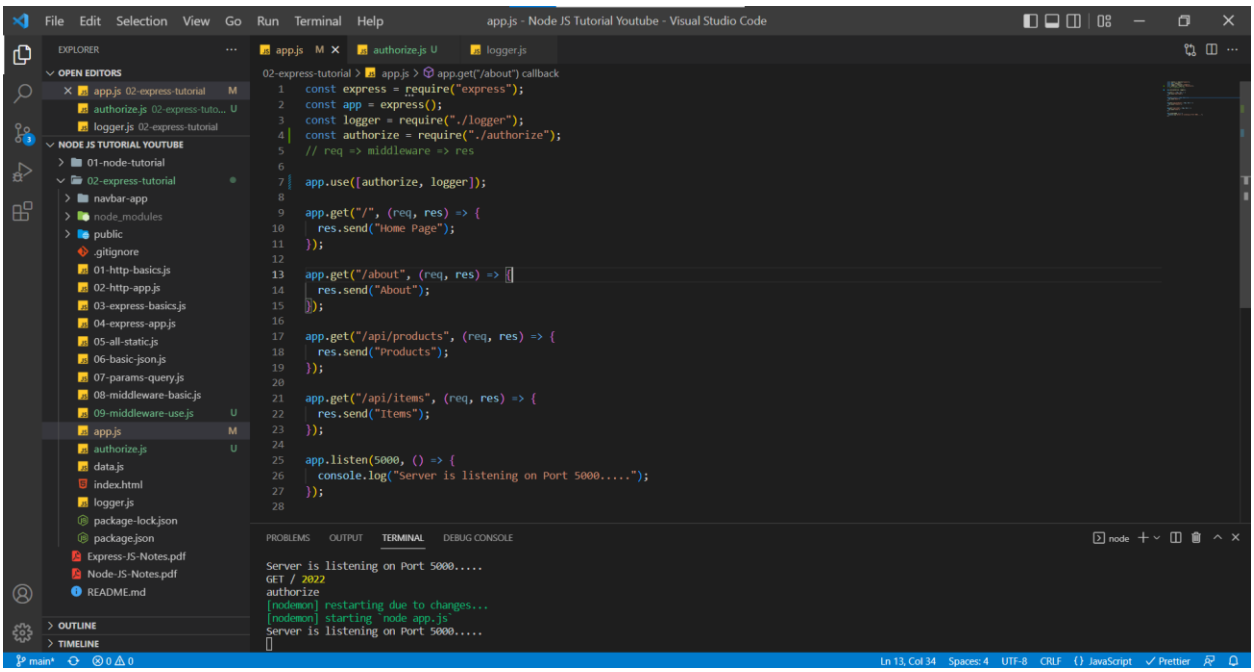


Terminal (Here we can the see two console statements, one from logger.js and another from authorize.js)

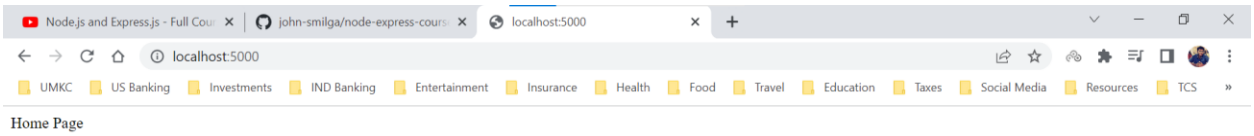
Order is important in middleware functions, the order we enter in the code is the order that Express JS follows. In the above terminal you can see statement of logger.js and then statement of authorize.js.

In the example, we can find the reserve order when we change the order in the app.use() method.

app.js



http://localhost:5000/



Terminal (after changing the order and restarting the server, we can see the console log statements in the order we mentioned in app.use() method).

Order of middleware functions is important in app.use() method.

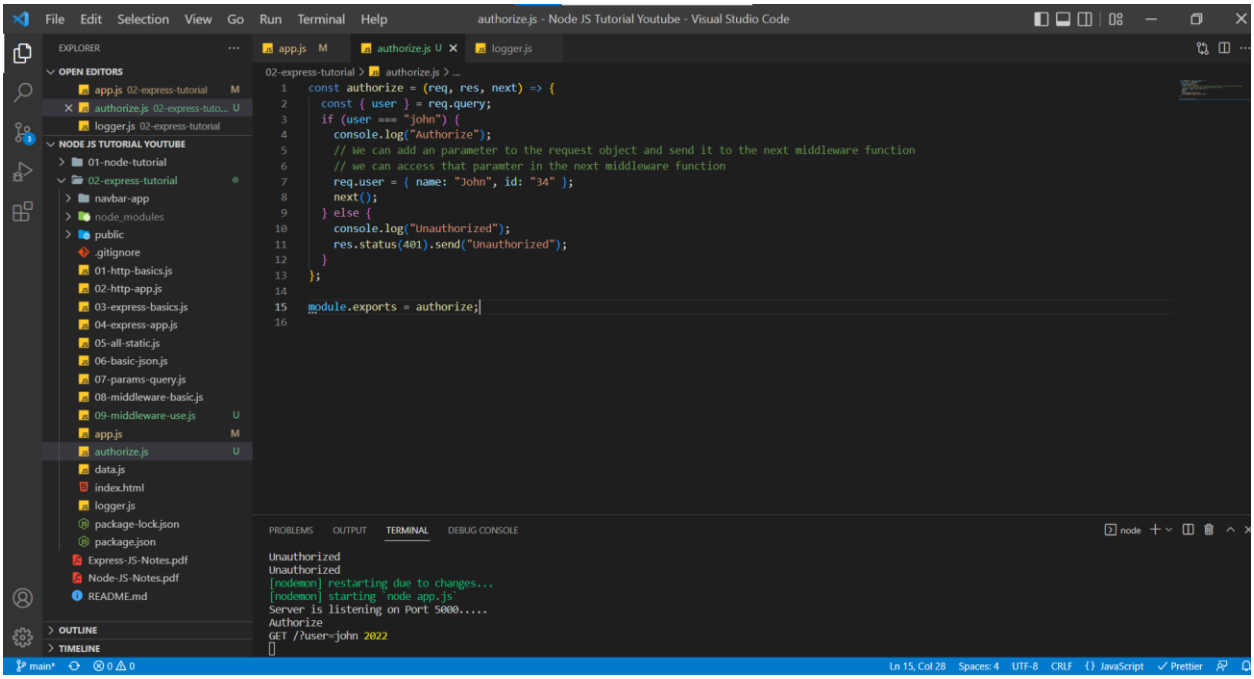
In General, that is not how we authorize the webpages, we use the JWT token to authorize. In the below example we will try to authorize using the query string (this is also an example).

authorize.js

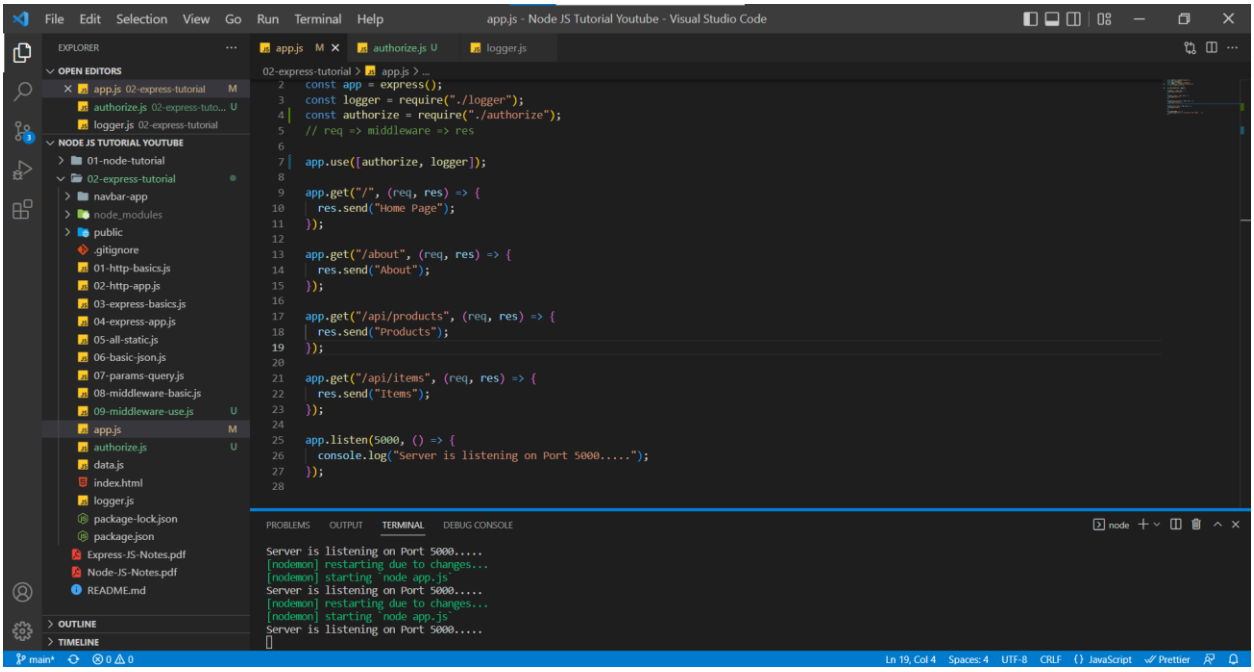
Lessons Learned:

We cannot use next() method once we sent back the response

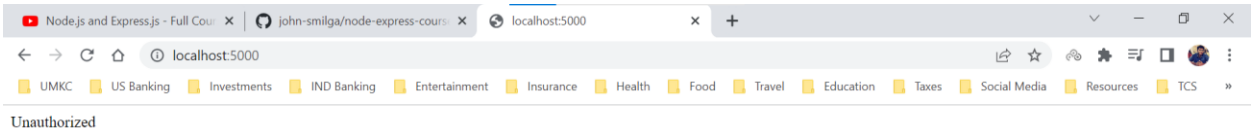
We must extract the query string from req.query property.



app.js

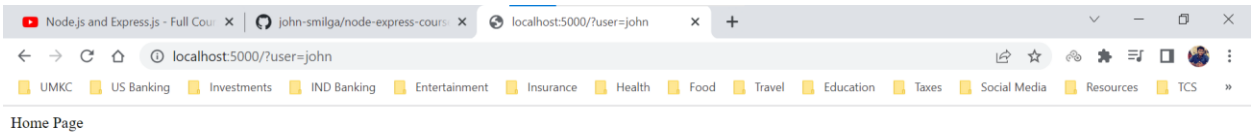


http://localhost:5000/



Terminal (We can see the console log statement from authorize middleware function)

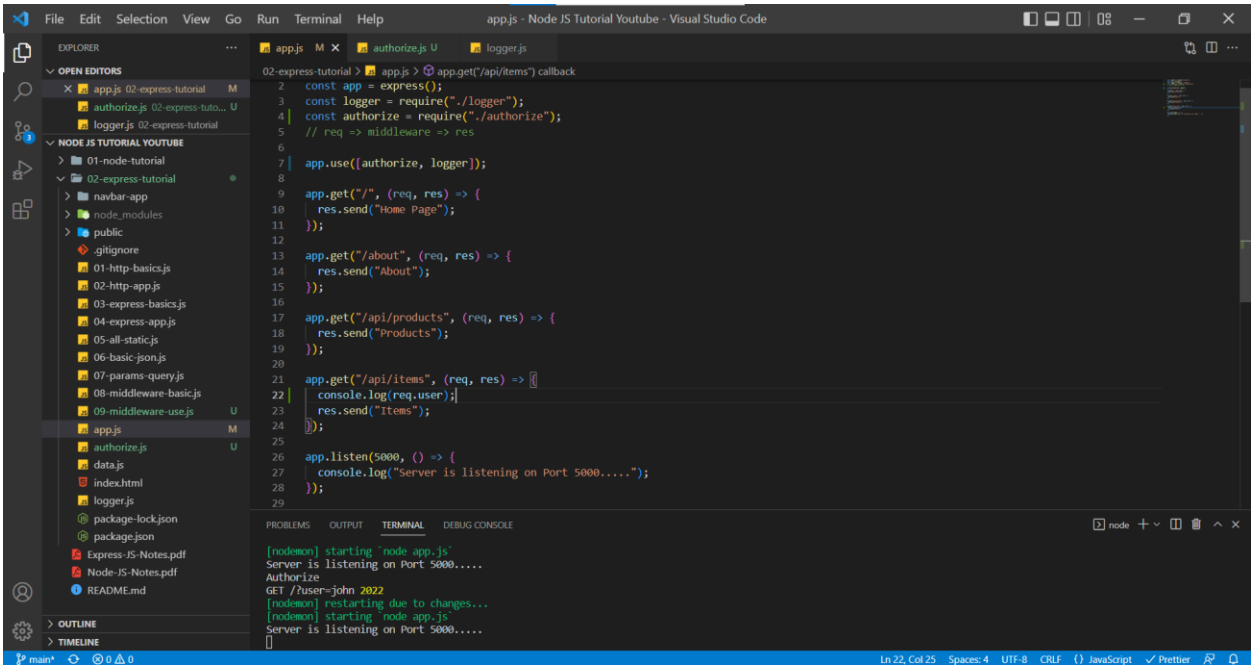
http://localhost:5000/?user=john



Terminal (We can see authorize middleware function and then the logger middleware function)

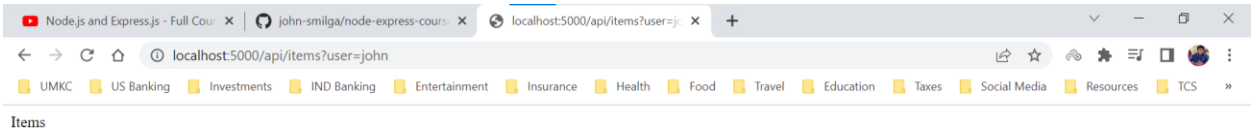
We can now access the user parameter that we added to the request object in authorize middleware function.

app.js



We are retrieving the user information for the route /api/items hence we need to visit the route.

http://localhost:5000/api/items?user=john



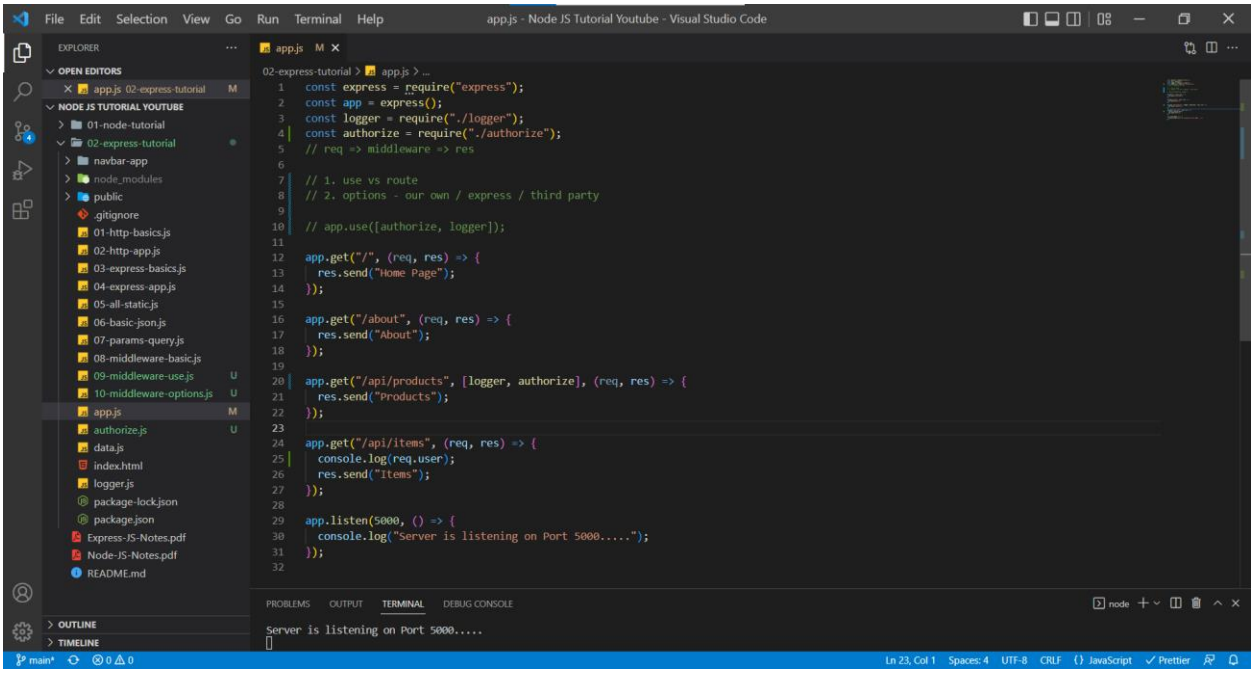
Terminal

In the terminal, we can see 3 console log statements. The first one being the authorize middleware function, second one being the logger middleware function and then third one being the console log statement from /api/items route.

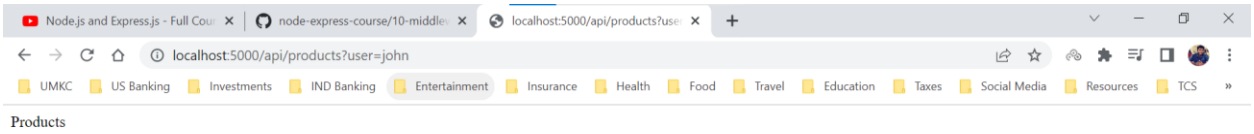
Additional Middleware Info

Assume a scenario, where we don't want to add app.use() method to all the routes. In the above scenario we applied app.use() method to all the routes in the app.js but we will apply only to /api/products route.

app.js

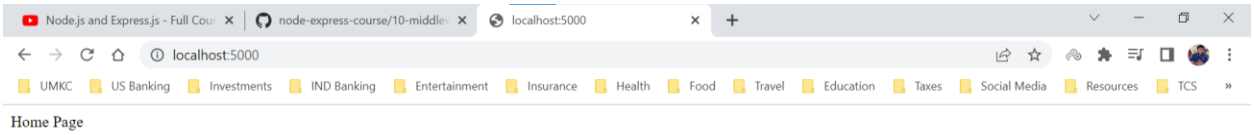


http://localhost:5000/api/products?user=john



Terminal

http://localhost:5000/



Terminal (we won't see any console log statements because middleware functions are not applied to home route)

The options when we come to middleware functions

1. Own Middleware Functions (Ex: logger.js, authorize.js)
2. Express Middleware Functions (Ex: static)
3. Third Party Middleware Functions

We have created our own middleware functions like logger.js and authorize.js

There are many complicated Express middleware functions which we can handle but as of now we have an example of static middleware function which requires all the static files of the project that needs to send to the browser.

To use the third—party middleware functions we need to install the package.

In this example we are going install the package ***morgan*** and we need to run command ***npm i morgan***.

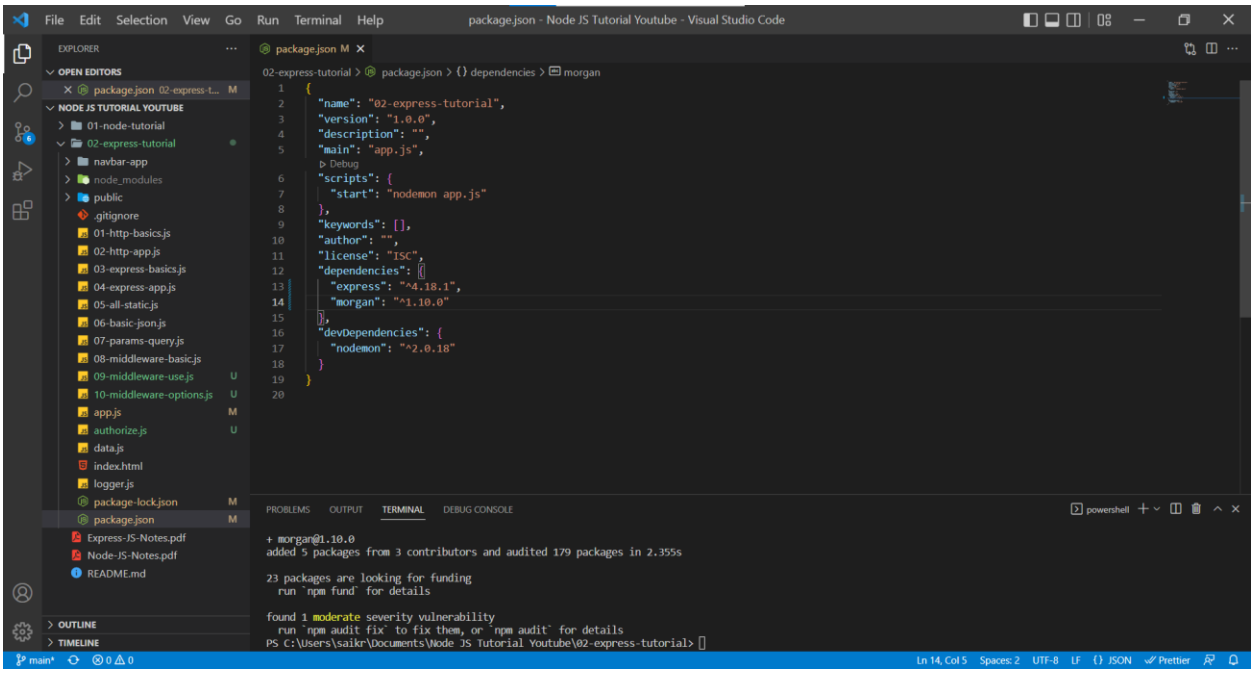
morgan is a HTTP request logger middleware for node.js

In a default way, they provide information of request came to the same server with a format, we can edit the format, but the default format is

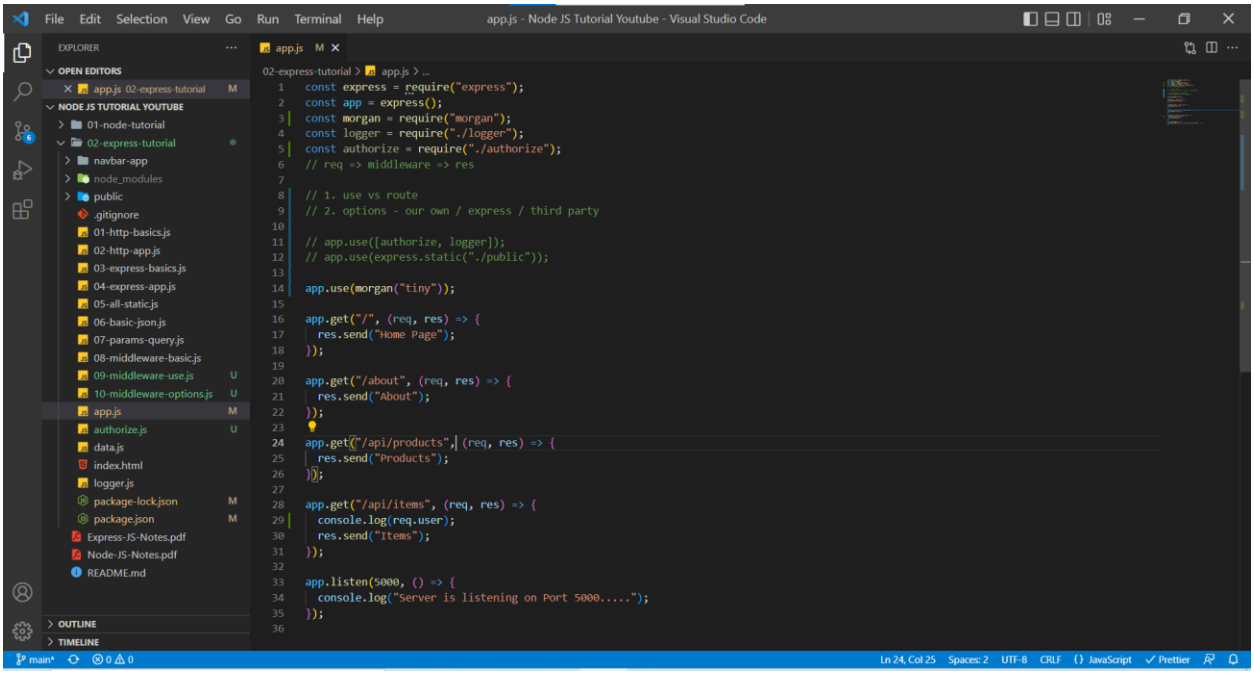
':method :url :status :res[content-length] - :response-time ms'

We installed morgan package.

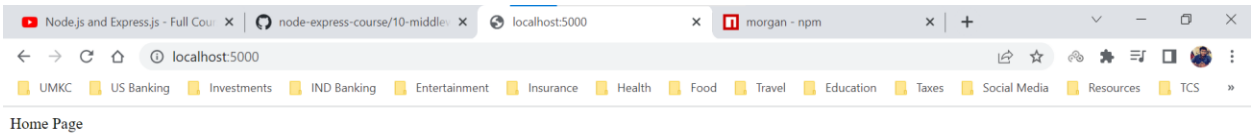
package.json



app.js



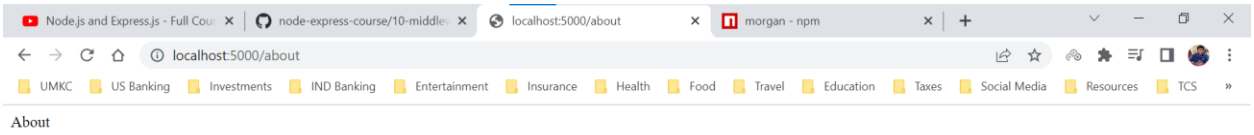
http://localhost:5000/



Home Page

Terminal

http://localhost:5000/about



About

Terminal

Methods – GET

GET is the default method for a request body to the server.

Below are the HTTP methods.

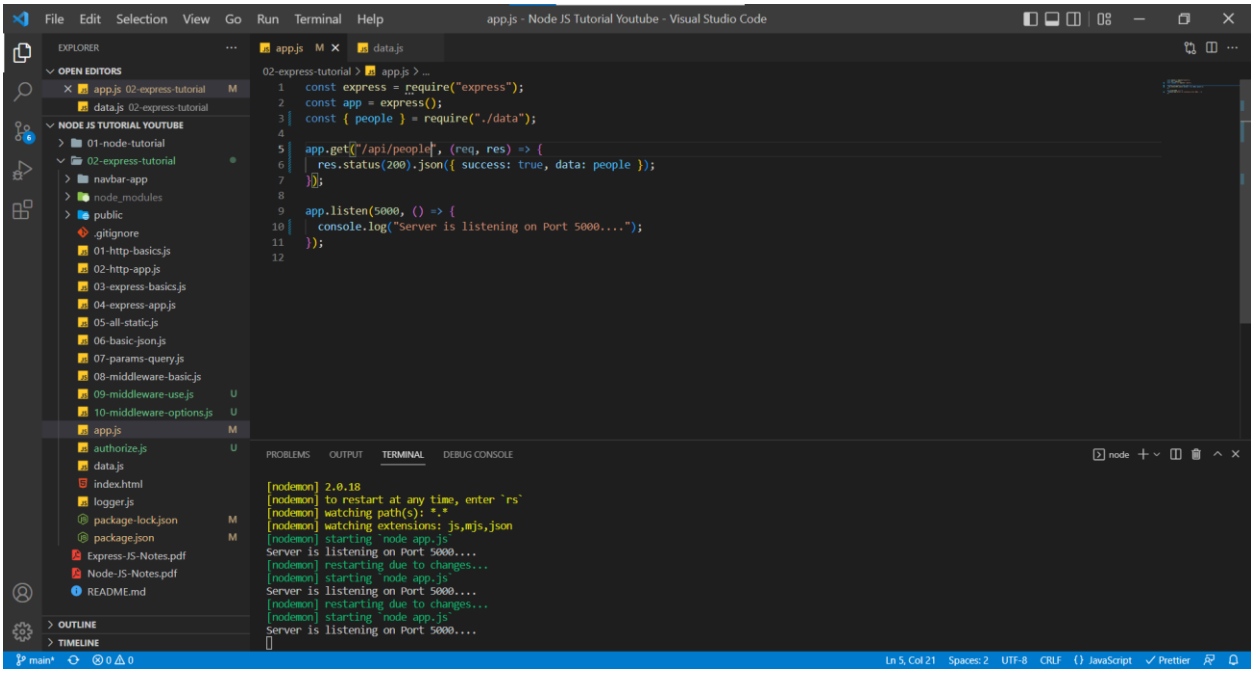
HTTP METHODS

GET	Read Data
POST	Insert Data
PUT	Update Data
DELETE	Delete Data

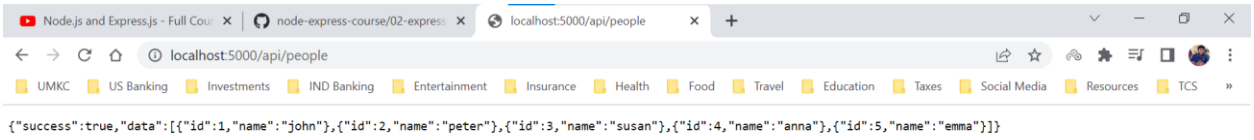
GET	***www.store.com/api/orders***	get all orders
POST	***www.store.com/api/orders***	place an order (send data)
GET	***www.store.com/api/orders/:id***	get single order (path params)
PUT	***www.store.com/api/orders/:id***	update specific order (params + send data)
DELETE	***www.store.com/api/orders/:id***	delete order (path params)

Generally, we would the contact the database to get data and provide the data back to the user but in this case, we are getting the data from a file named data.js and sending the JSON data back to the user.

app.js



http://localhost:5000/api/people



We are not logging anything in the terminal in this example.

Methods – POST

POST method is used to add/insert data on to the server.

We can setup the POST requests using two methods

1. Traditional Form Example
2. JavaScript option

In this case, we will a little bit of static data. In the GitHub (https://github.com/john-smilga/node-express-course/tree/main/02-express-tutorial) we have a folder named methods-public.

We are this folder because we cannot simply perform a POST request from the browser.

We cannot simply configure our browser to perform a post request. Either we need to use a tool named ***Postman*** or another tool named ***Insomnia***, or we need simply set up a working application.

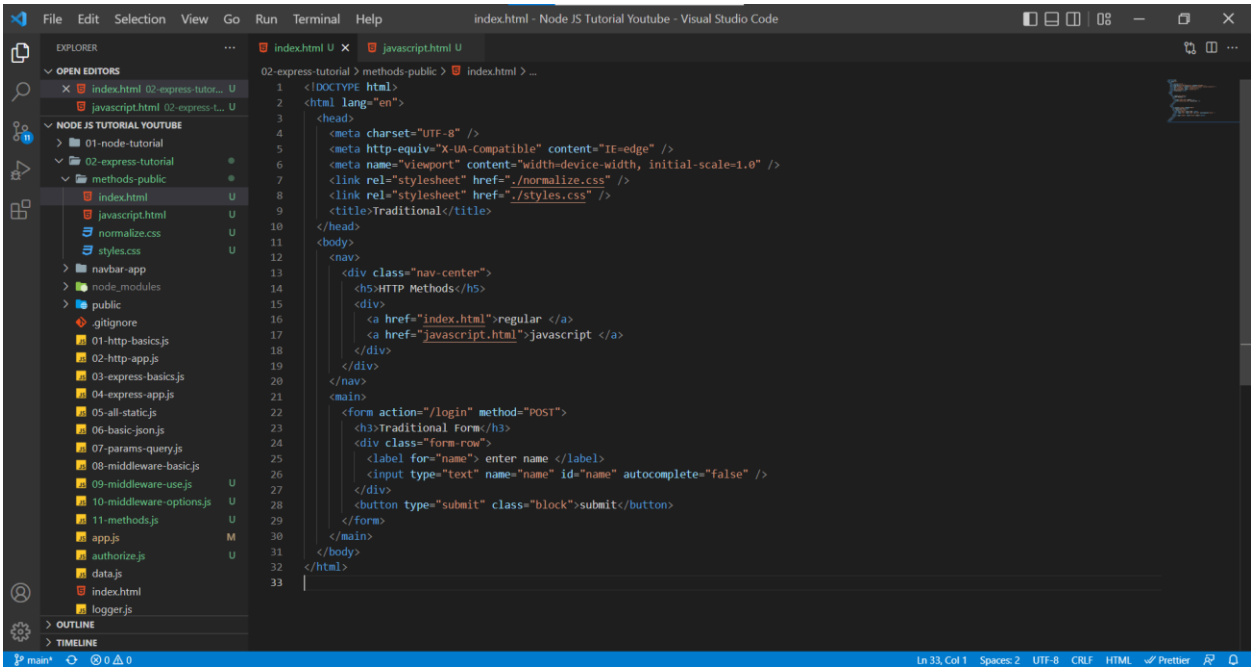
In this case we are going with the Traditional Form Application.

Methods – POST (Form Example)

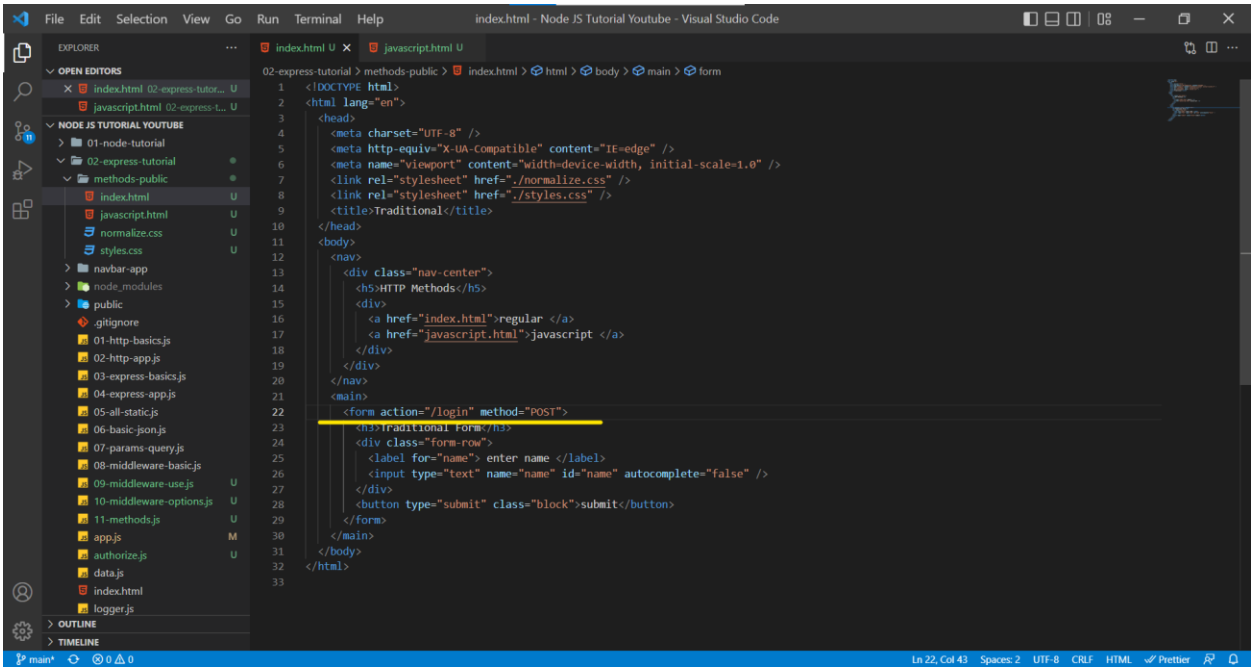
In this example we are using the traditional form application. We have route name /login and method is POST.

First, we have imported methods-public folder code from GitHub.

methods-public folder (it contains two html and two css files)



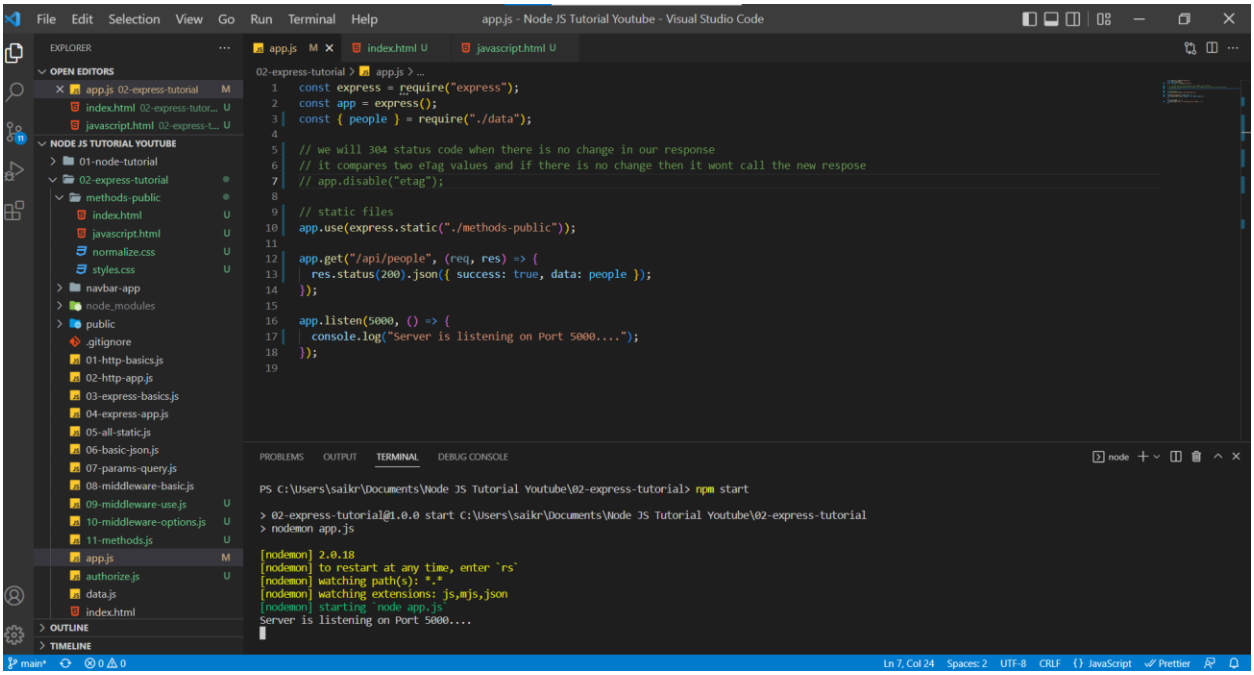
index.html



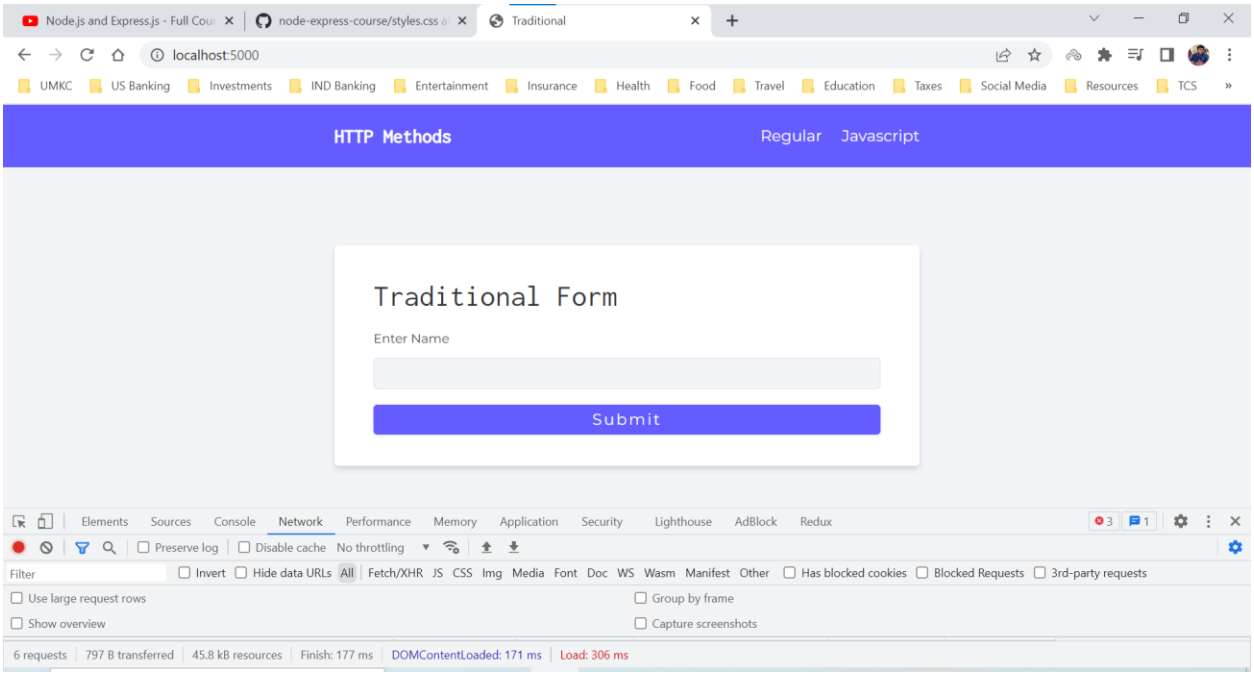
If we see at the line 22 action defines the route path whereas the method defines the type of HTTP method for that request.

As we know the body is optional in HTTP messages but it crucial when it comes to POST because we need to add data on the server hence, we require the body from request object.

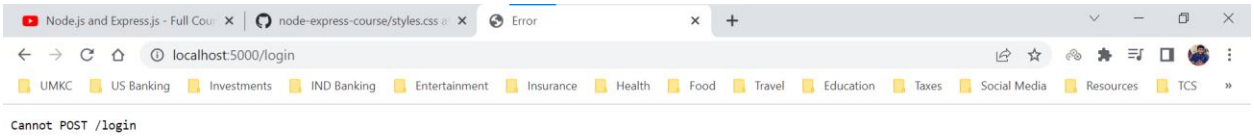
app.js

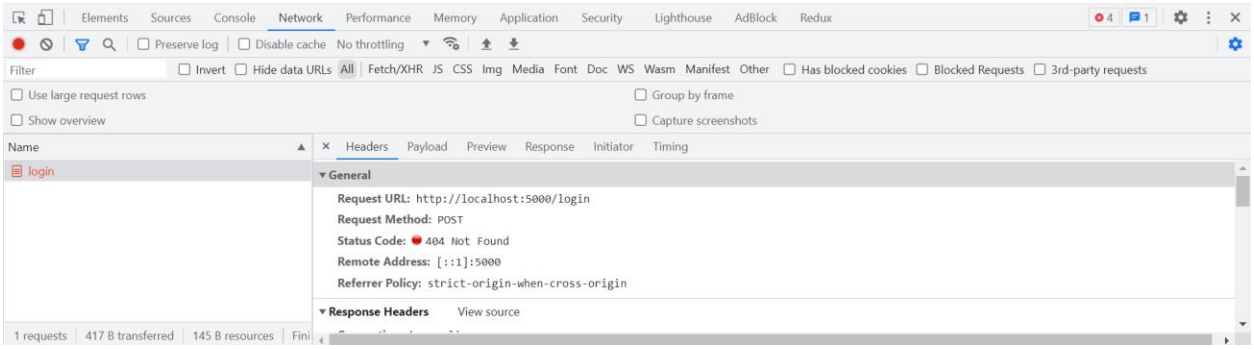


http://localhost:5000/

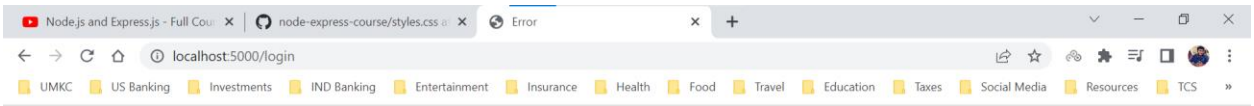


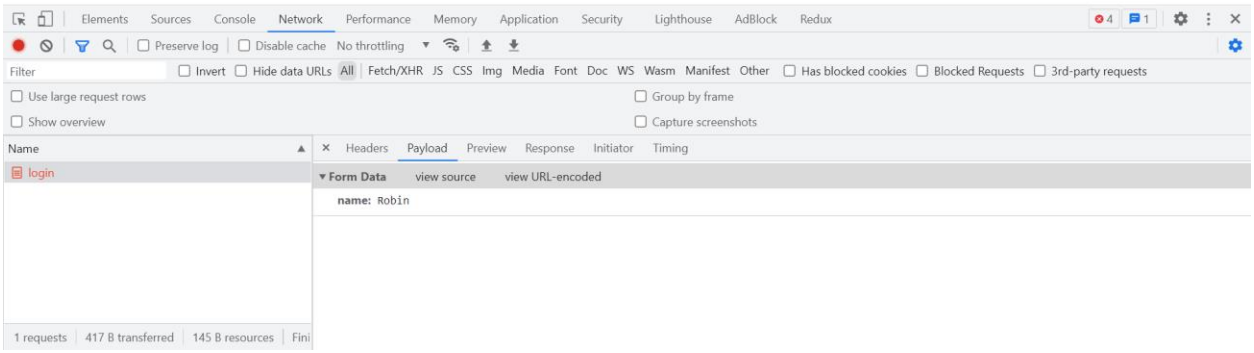
When we enter some value and click on submit, we are redirected to route ***http://localhost:5000/login*** and since we are not handling that route in app.js, we are getting 404 error. Also we can see the HTTP method being POST.





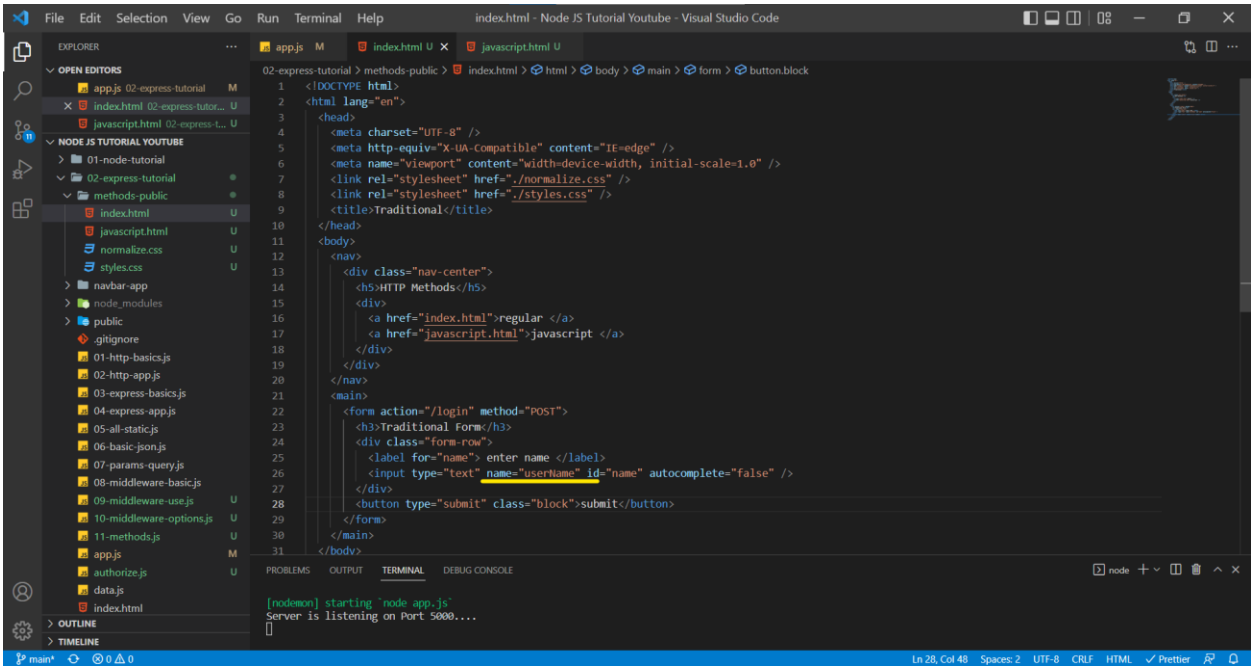
We entered names as Robin and clicked Submit button and hence we see the name Robin in Payload.



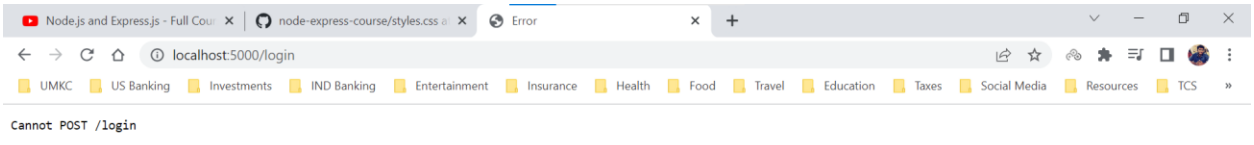


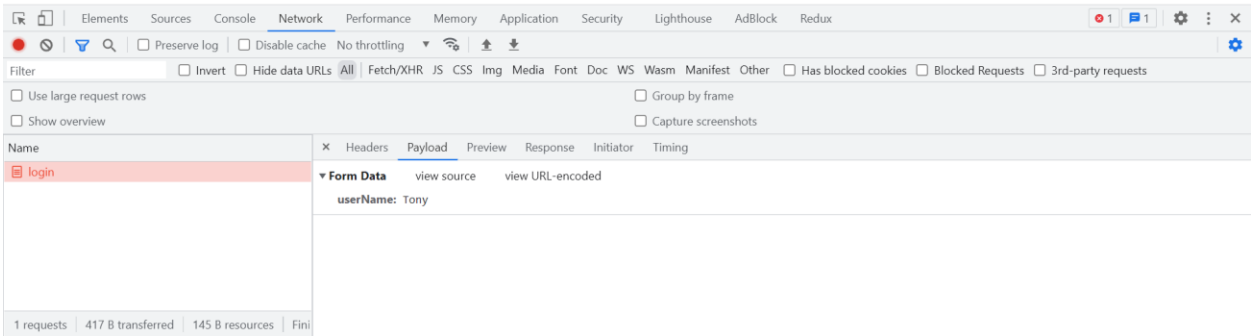
We have changed the name attribute in input type from name to userName.

index.html



http://localhost:5000/login



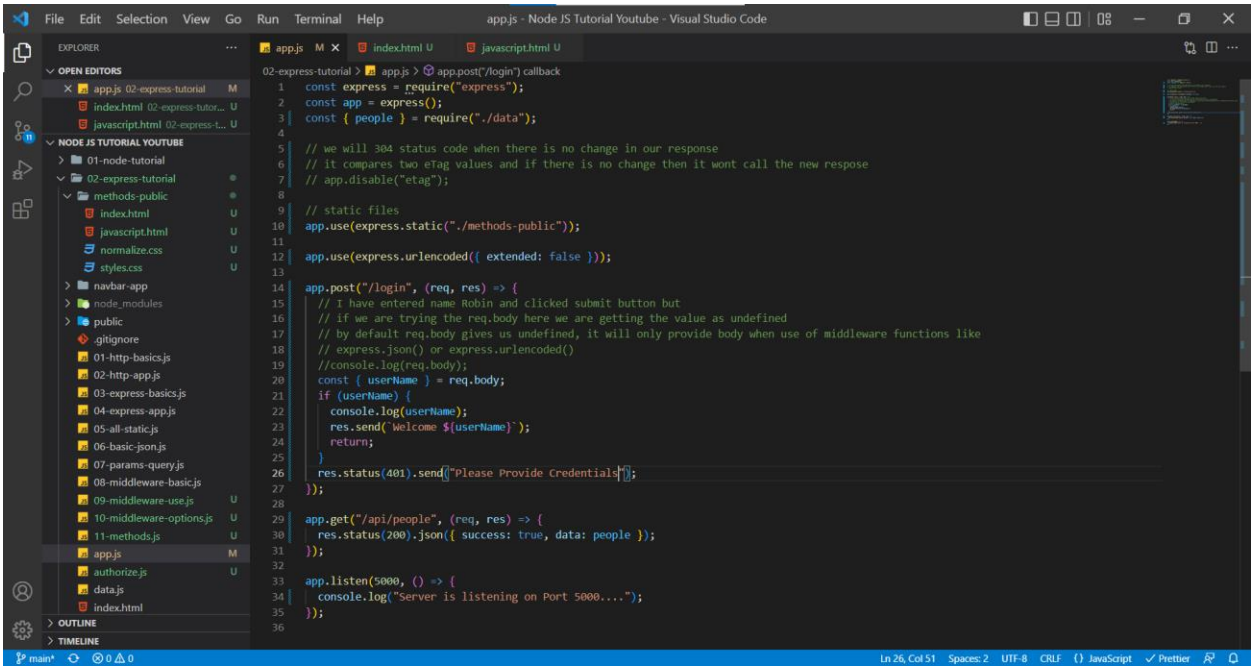


The attribute name in the input will be key and the whatever enter in the form would act like value and that is the reason we see them as a key value pair the request body of /login route.

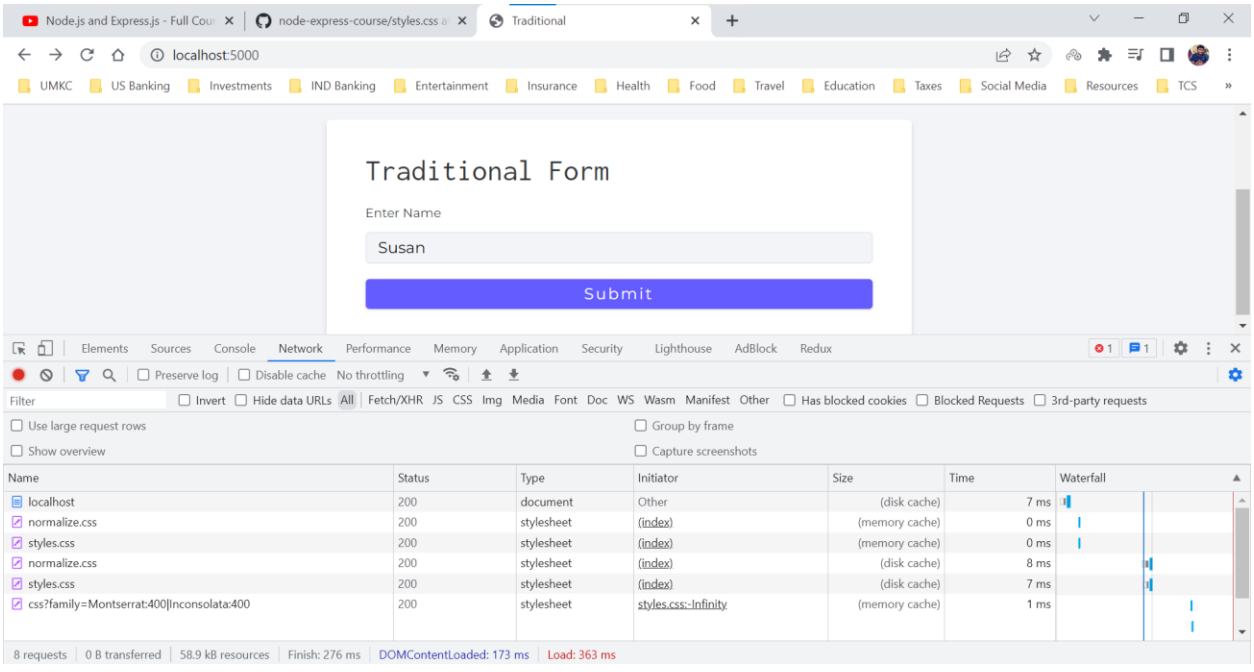
We now must create a app.post() to handle route /login and to get access to request body we need to use inbuilt middleware function named express.urlencoded.

We can have a though using the req.body but by default it will provide us undefined, it will provide the request body only when we use middleware functions like express.json() or express.urlencoded().

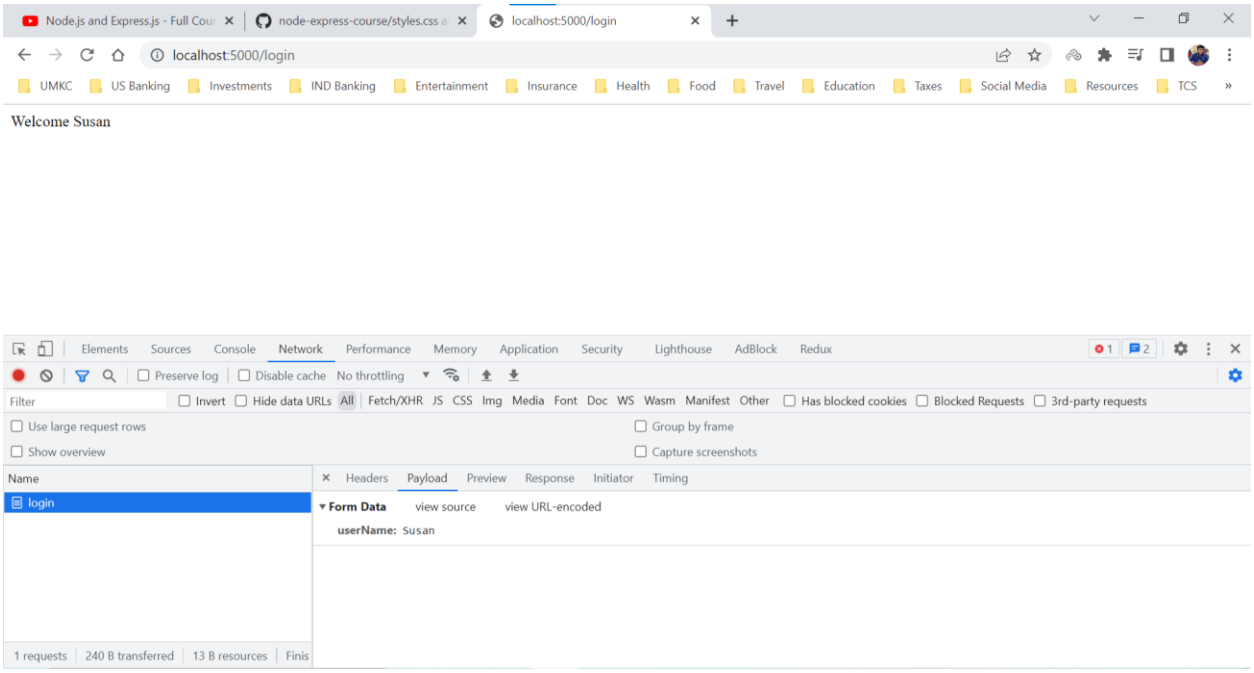
app.js



http://localhost:5000/

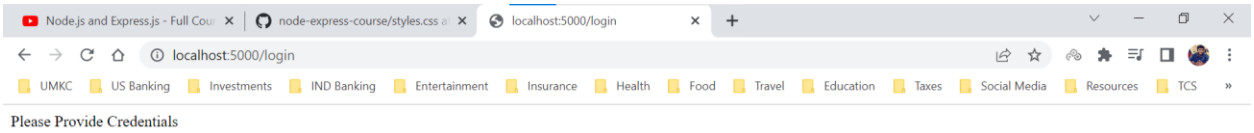


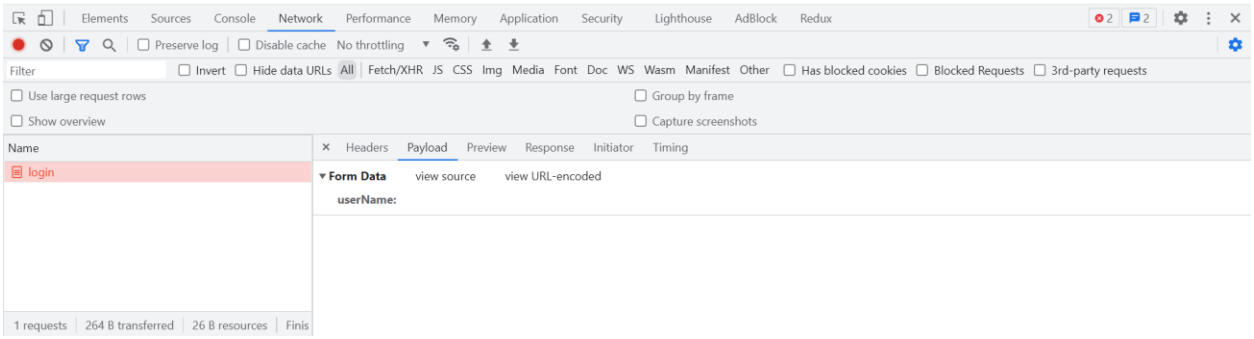
After click on submit button



Terminal

If we don’t enter name in the form, then we display the message “Please Provide Credentials”





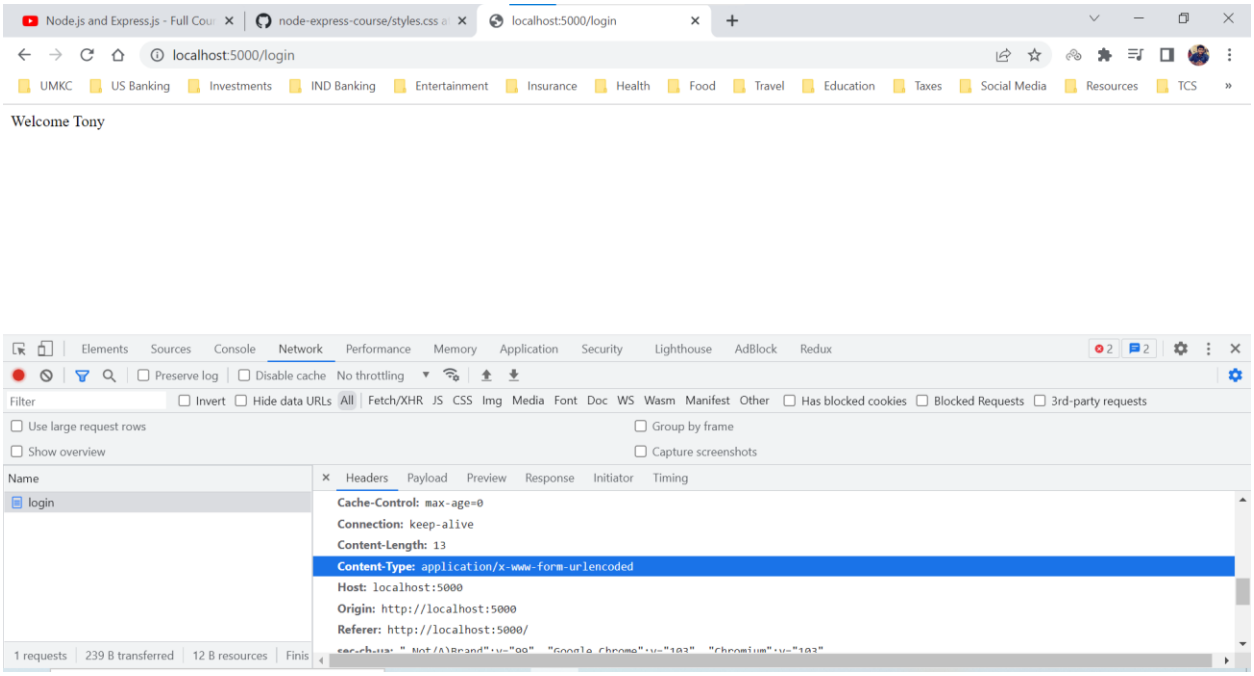
extended property inside the express.urlencoded() middleware helps to choose the format of encoded data.

By default, it will be **true**,

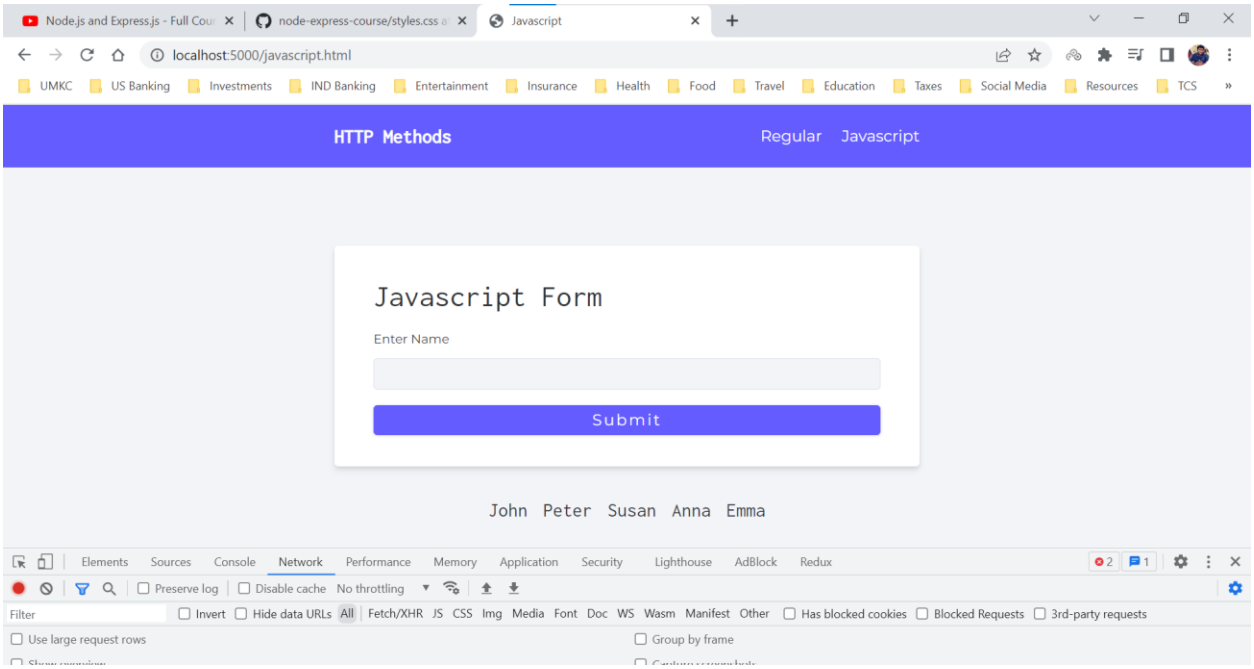
> true- parsing the URL-encoded data with qs library
>
> false - parsing the URL-encoded data with querystring library

Methods – POST (JavaScript Example)

If we look at the Content-Type in request headers, we find something interesting we have **application/x-www-form-urlencoded**.



We are showing that now because for javascript that is going to be different.



In this case we are handling the form strictly using the javascript. We strictly use javascript to send requests and of course the Content-Type is going to be different.

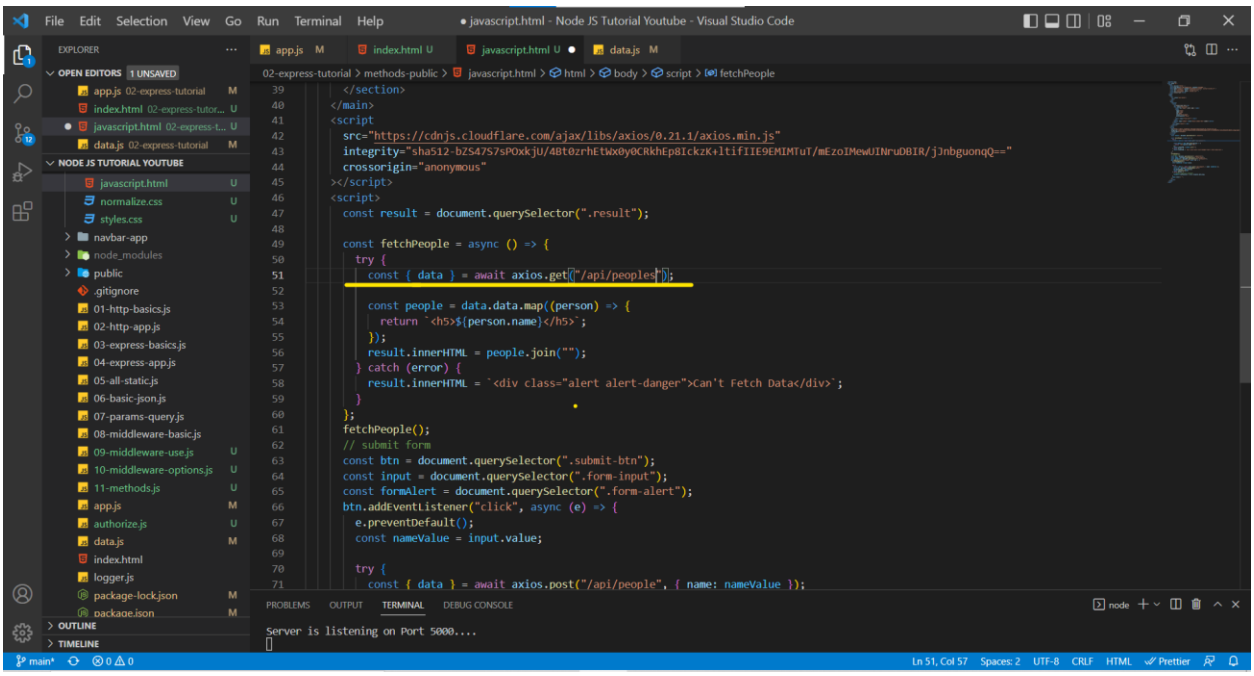
In javascript.html file, we don’t have any onClick method for submit button.

Axios is one package which makes HTTP requests easier to setup. Instead of using in-built fetch we will be using axios because it provides clean API and better error messages.

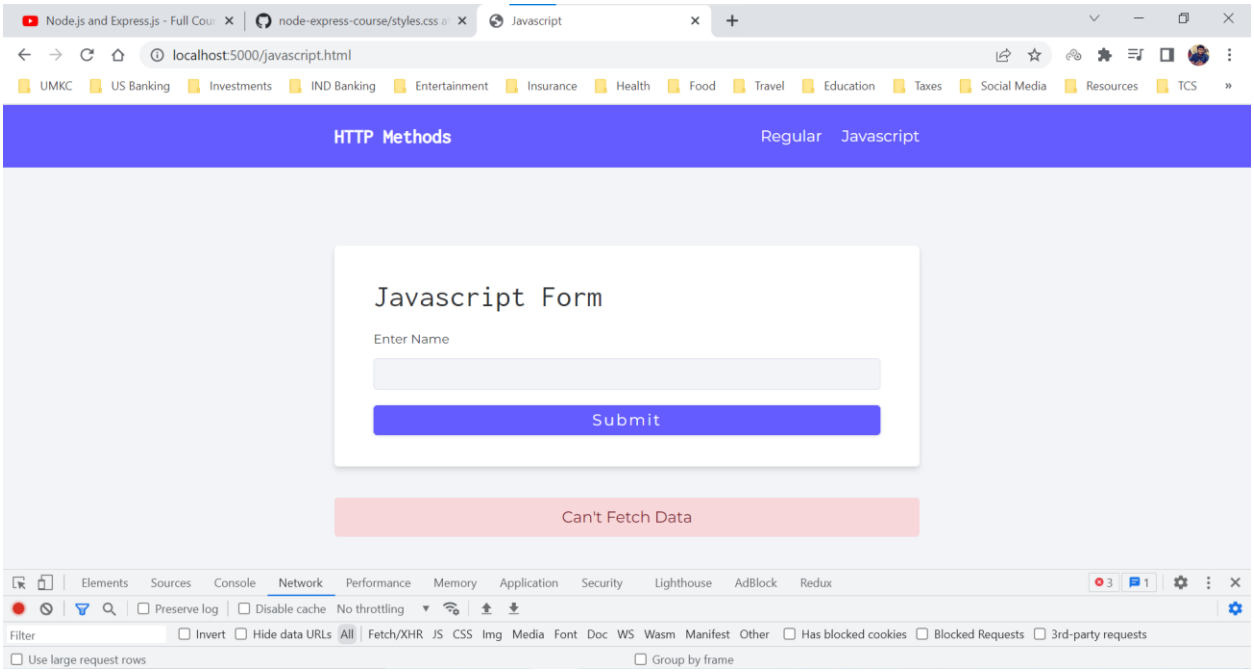
We are using a function fetchPeople in frontend which is going to fetch people from the server and the path is /api/people.

On load itself we are trying to fetch data from people object in data.js

if we don’t provide the proper route or api we will get an error.



We aren’t handling the route /api/peoples and hence the output can be

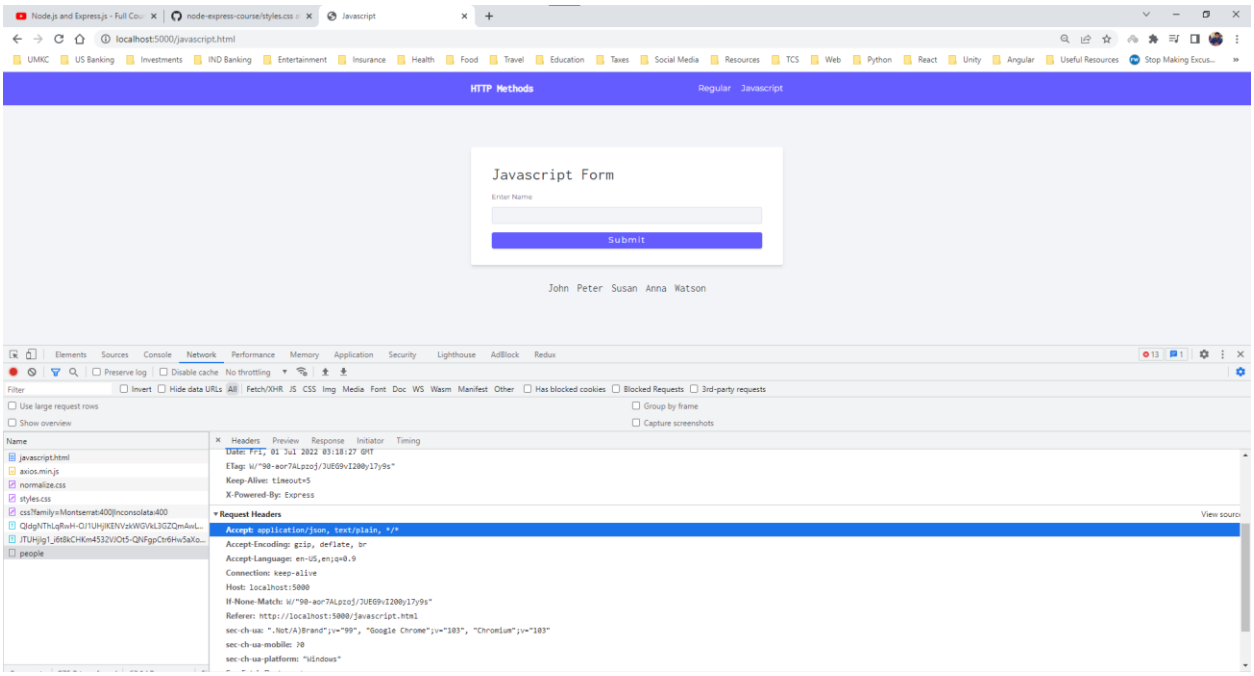


201 is the status code that we send back the user when we success executed a POST HTTP method request.

400 is the status code that we send we want to send an error to the user.

Generally, we need add the Content-Type in request header since we are using axios, it will take care of it.

http://localhost:5000/javascript.html (as mentioned earlier the Content-Type will be different, since we are using the javascript it is application/json)

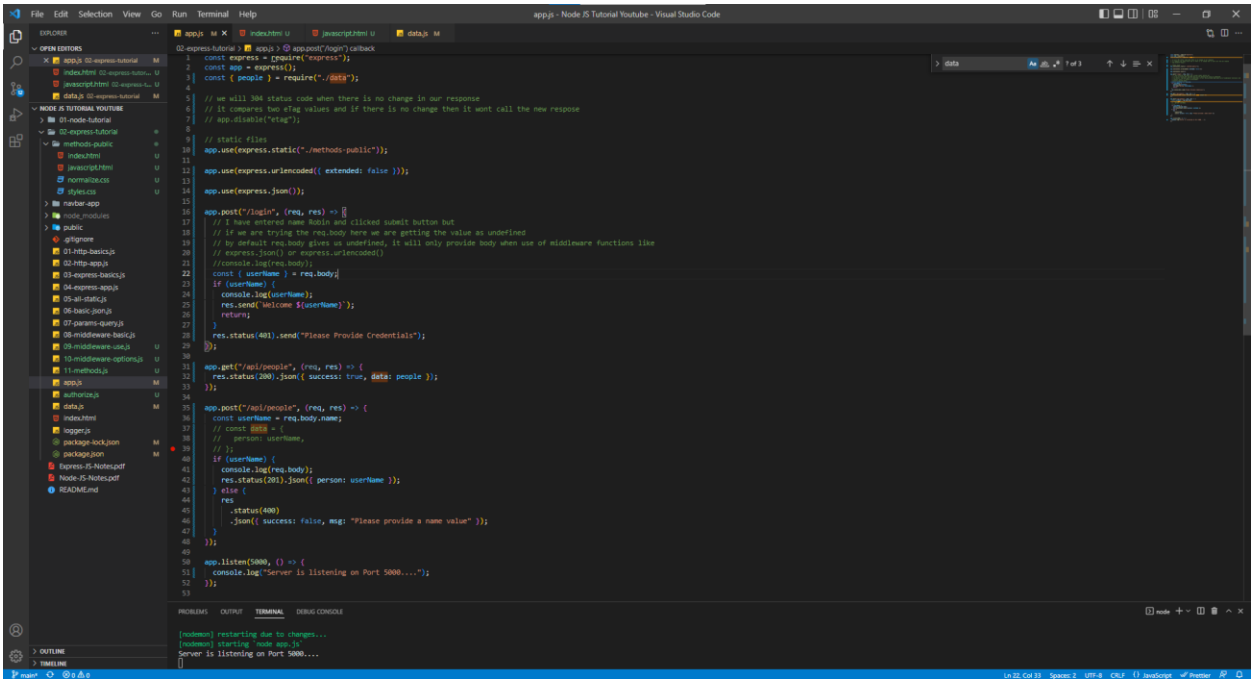


Since the Content-Type is application/json, to access the request body in the server we need to use a express middleware function named express.json().

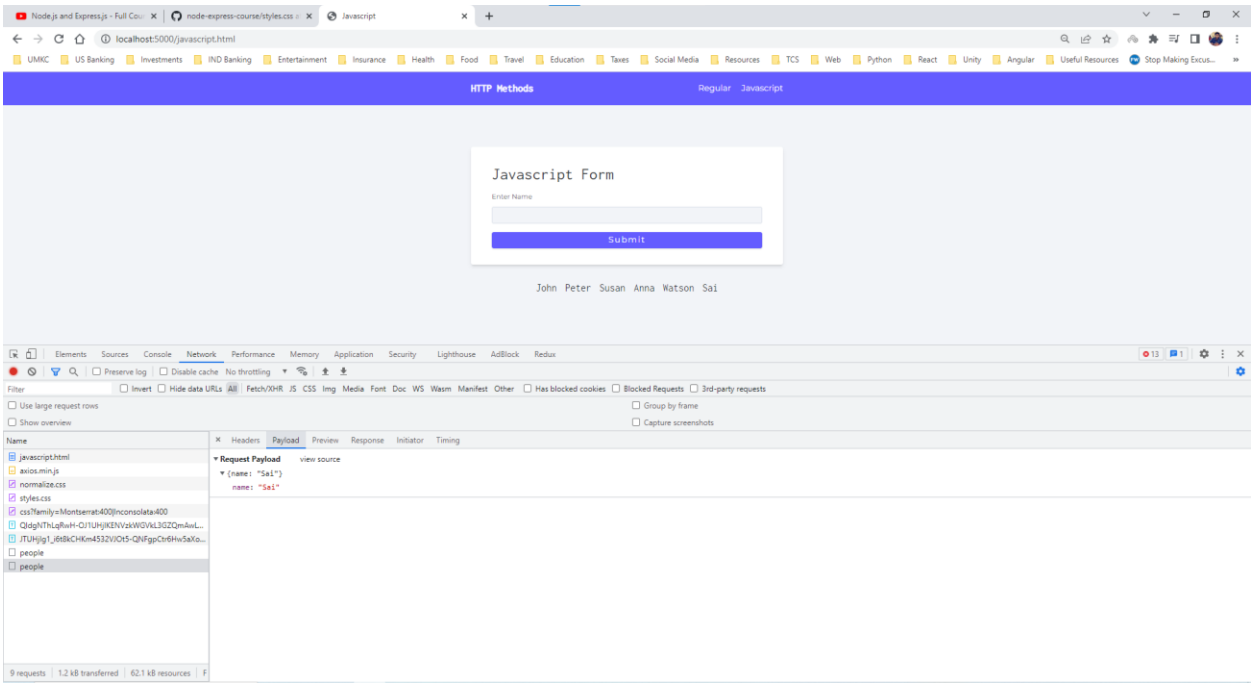
This is a built-in middleware function in Express. It parses incoming requests with JSON payloads and is based on body-parser.

Returns middleware that only parses JSON and only looks at requests where the Content-Type header matches the ***type*** option. This parser accepts any Unicode encoding of the body and supports automatic inflation of gzip and deflate encodings.

app.js

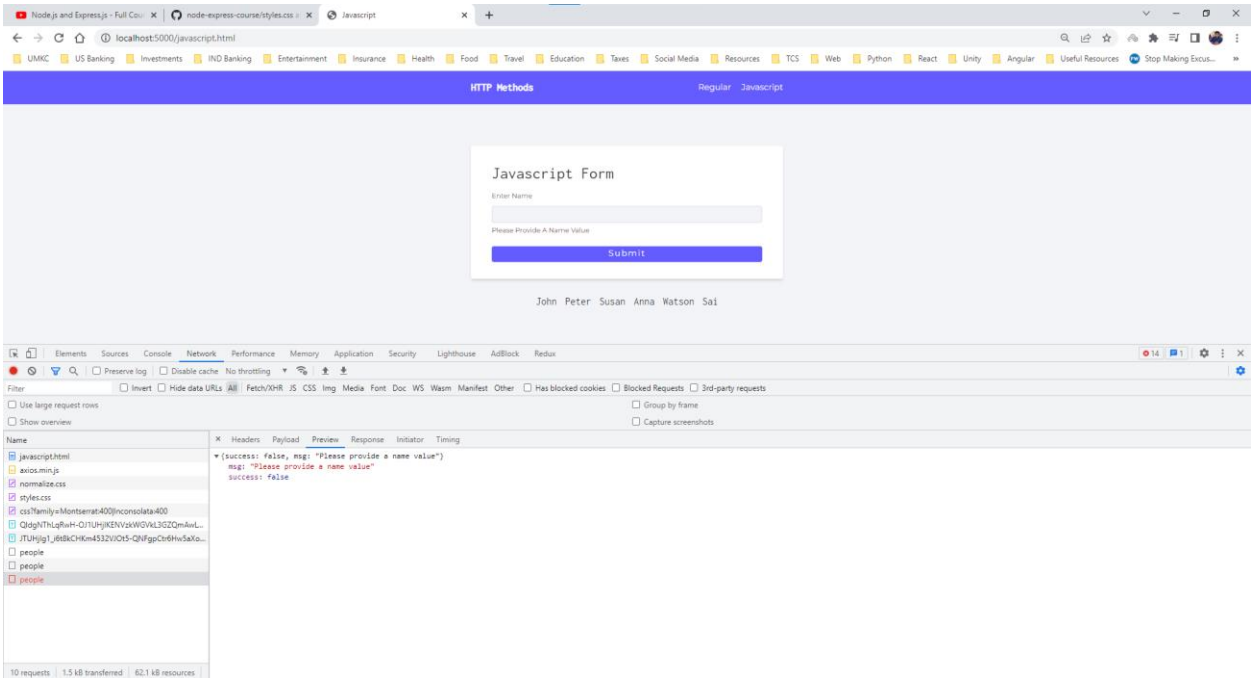


If username is entered by user, we will add it to the bottom list



Terminal

If submit is clicked without any username



In axios, whatever we receive from the API's it is of data object and we can add properties to that and then we can retrieve data from that.

Install Postman

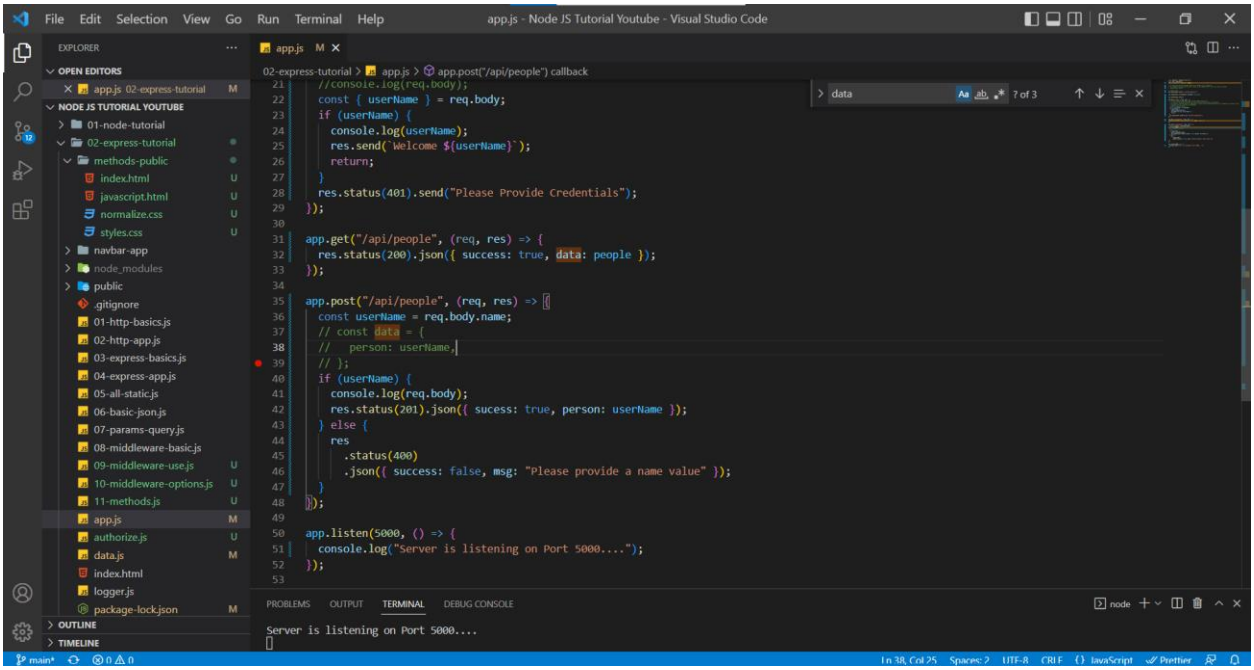
However, there is a major problem in our current setup and the problem is,

If we need frontend to create testing for every route, then our development will be extremely slow because it will take way longer time for us to setup the frontend than just setting up one simple route.

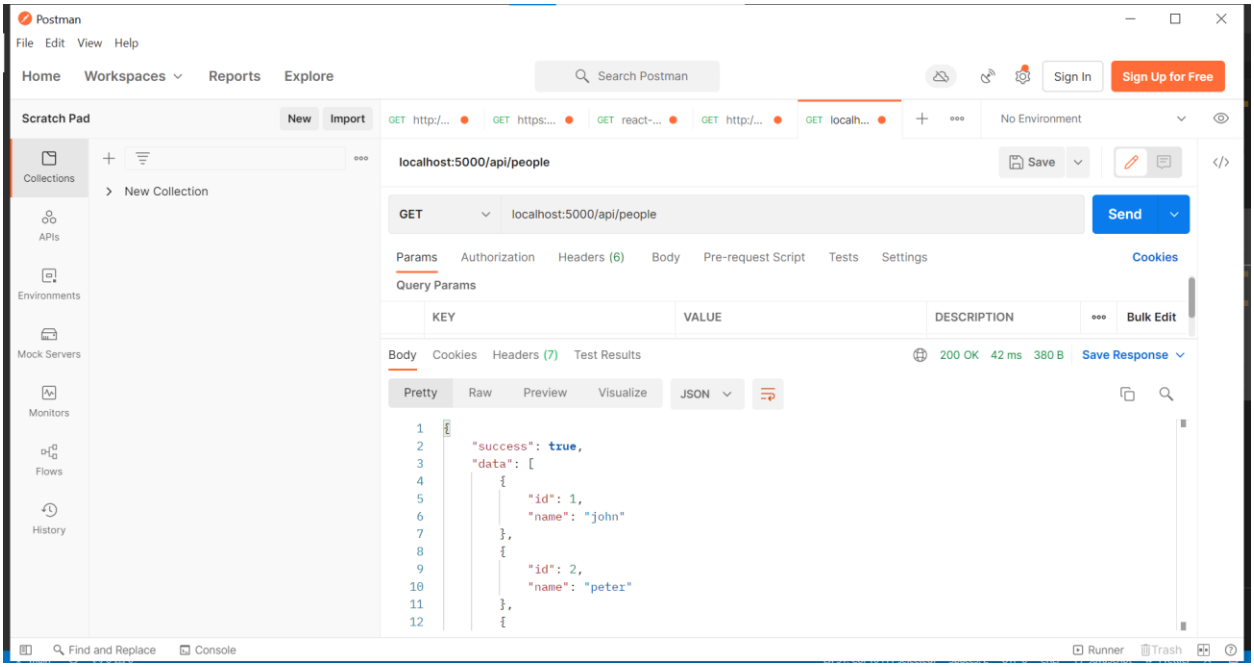
In order reduce the effort on developing frontend we have a tool named Postman which is used to quickly test our API's.

In Postman we can group our requests into Collection. Postman helps us to enter an URL and the type of HTTP method (GET, POST, PUT, DELETE, PATCH) etc. for request body.

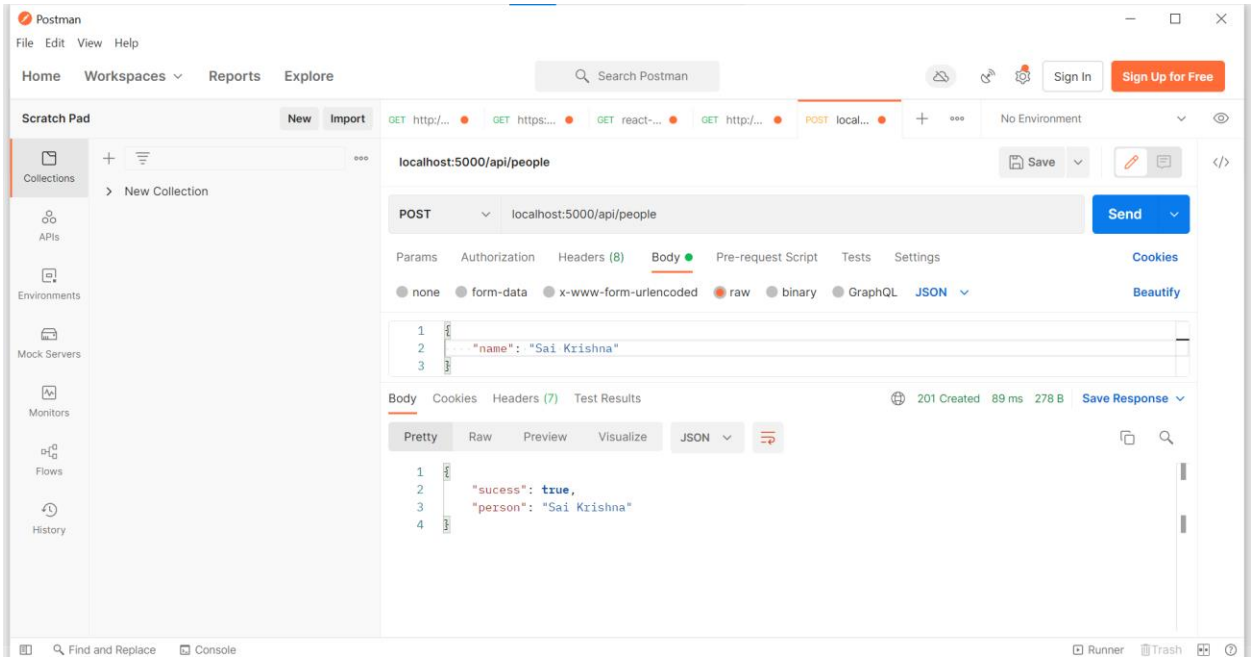
app.js



In Postman, localhost:5000/api/people with GET Method

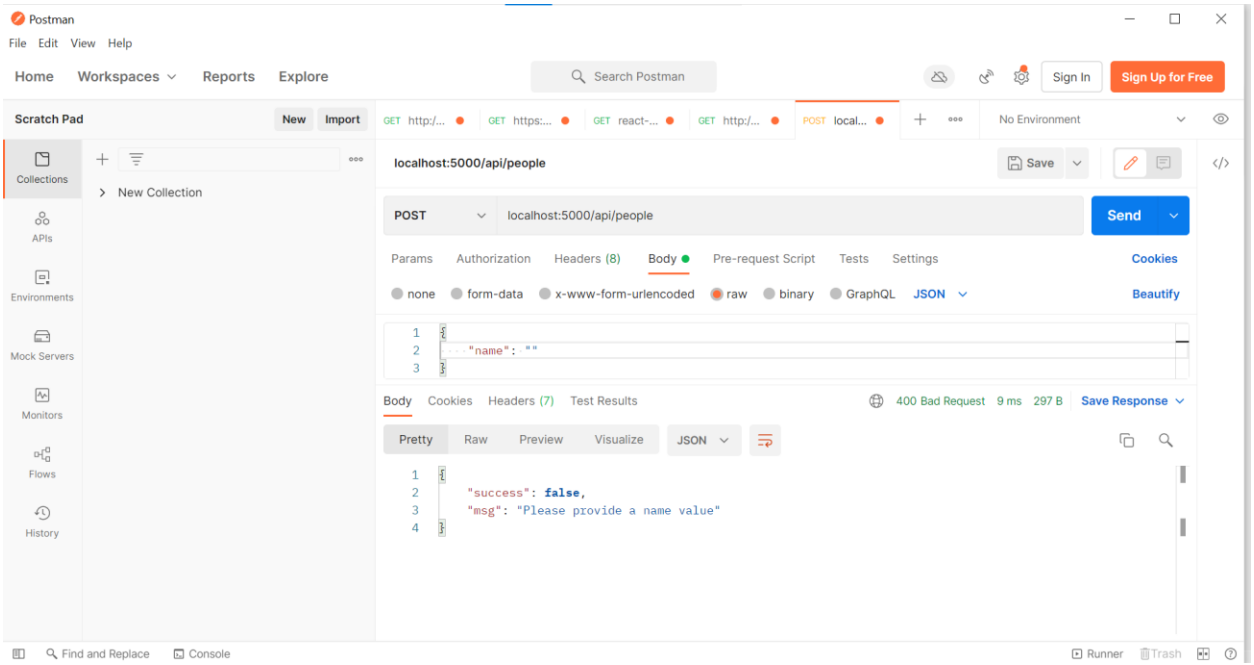


In Postman, localhost:5000/api/people with POST Method but for POST method, we need to send a request body with property named as ***name***.



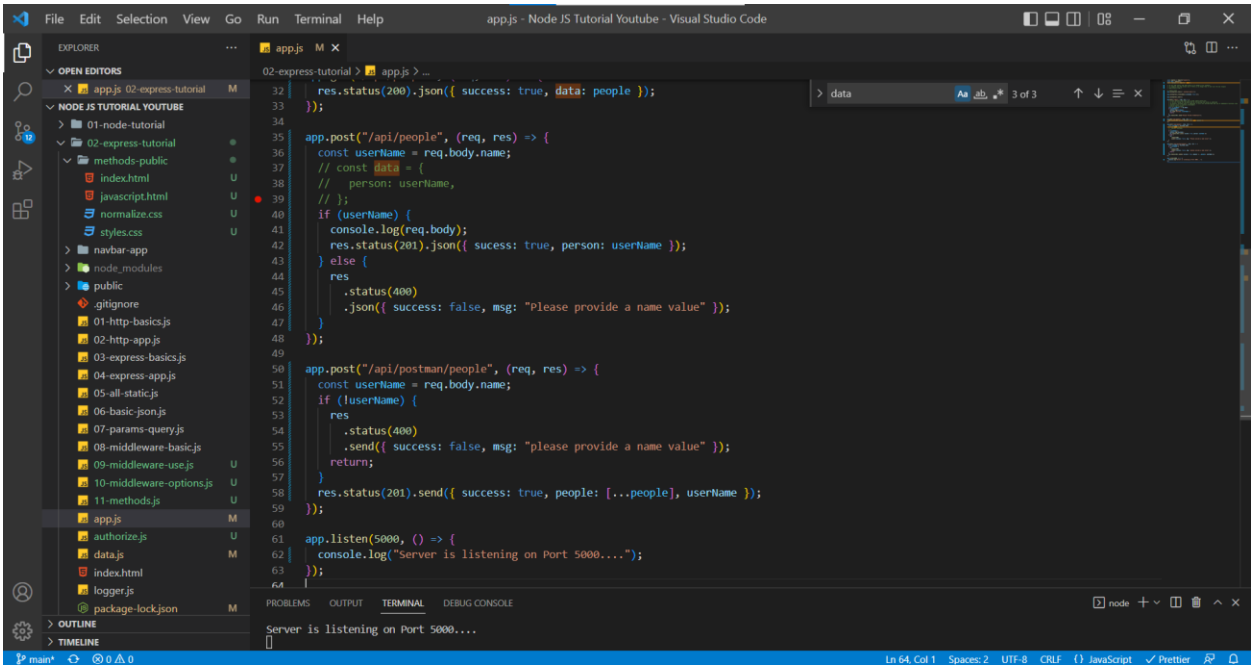
When we don’t send a value to server then we will receive a 400 error.

localhost:5000/api/people with POST Method.

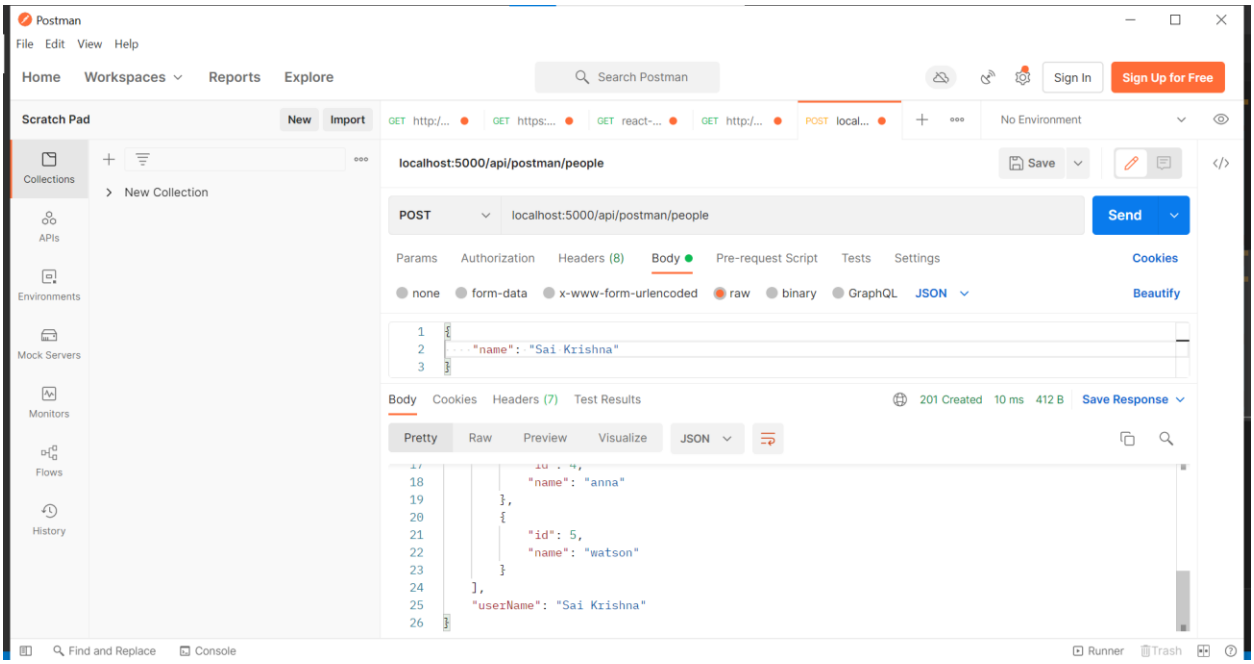


Let us create an endpoint for postman and test it. (localhost:5000/api/postman/people with POST method)

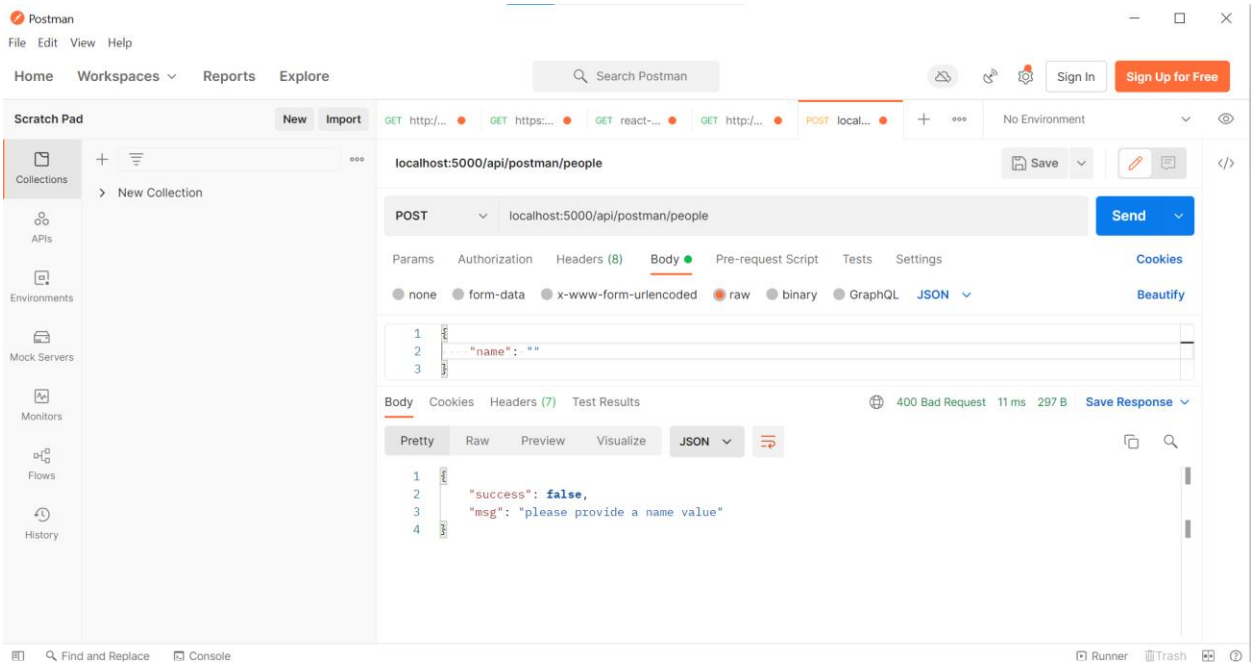
app.js



In Postman, localhost:5000/api/postman/people with PUT method with request body.



In Postman, localhost:5000/api/postman/people with POST method with no request body.

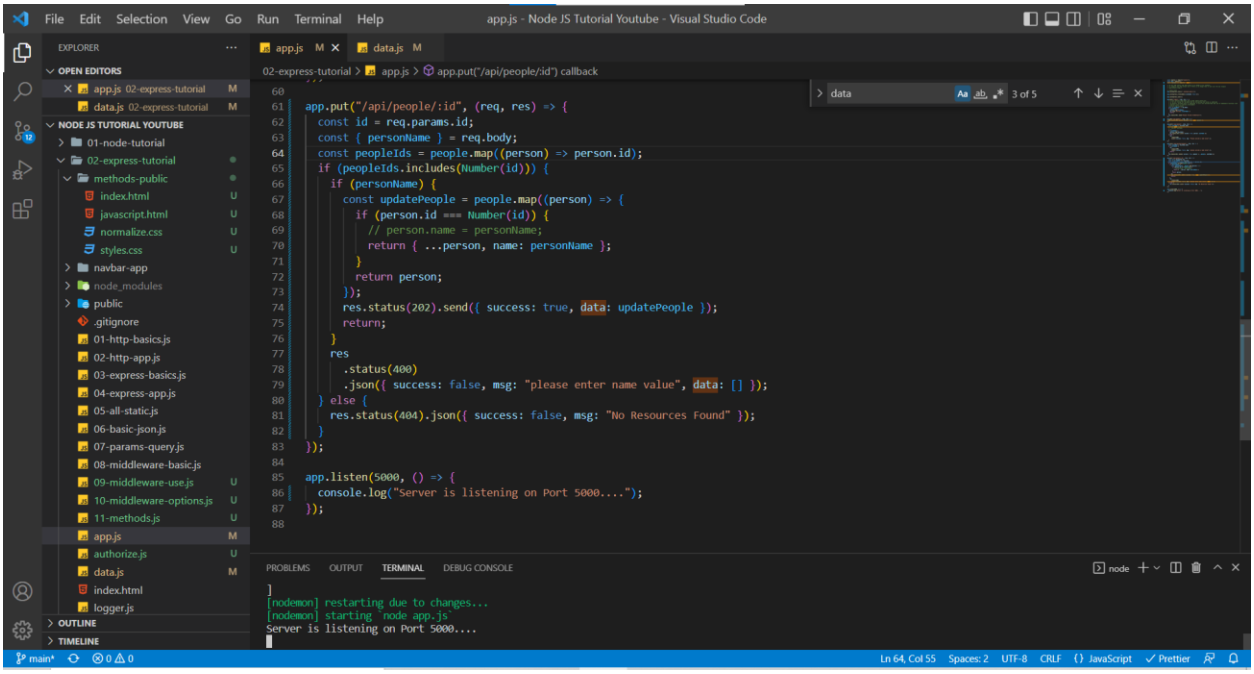


Methods – PUT

This method is for editing the data. So, PUT method is for update. If we edit or delete, we always go with the route parameter which gives us the ID to edit the exact specific item from the list.

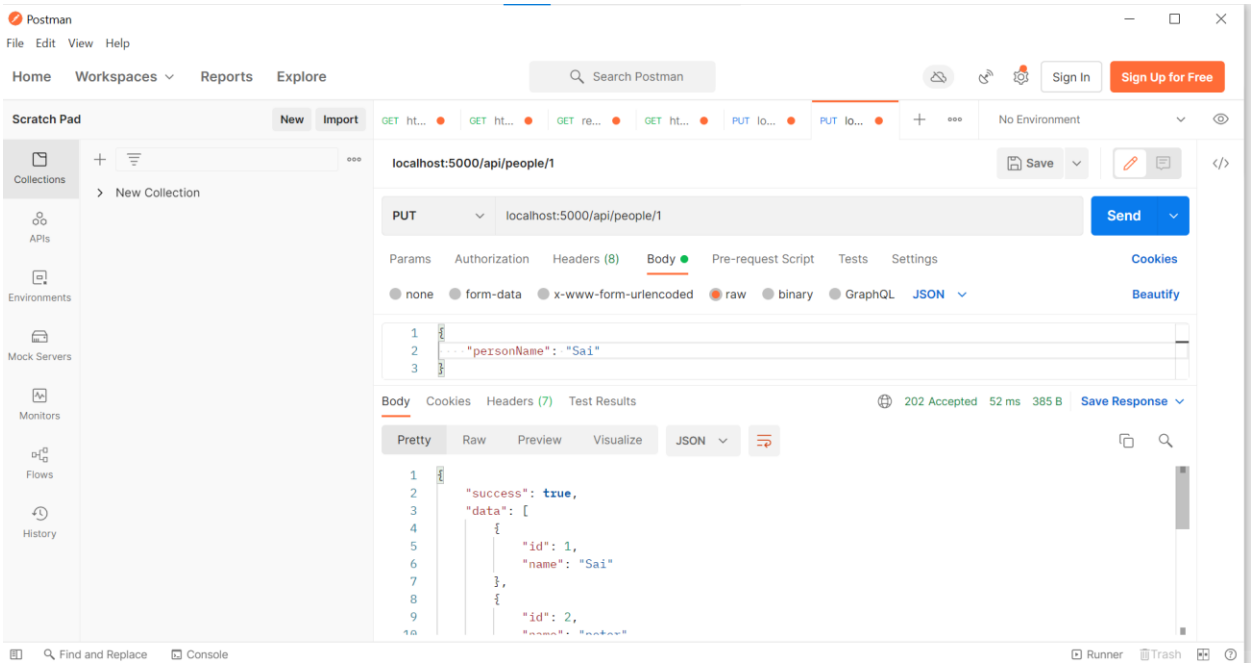
When we send PUT or DELETE requests, we will send it to a specific path with item id, if item exist then there is body which is used to update.

app.js

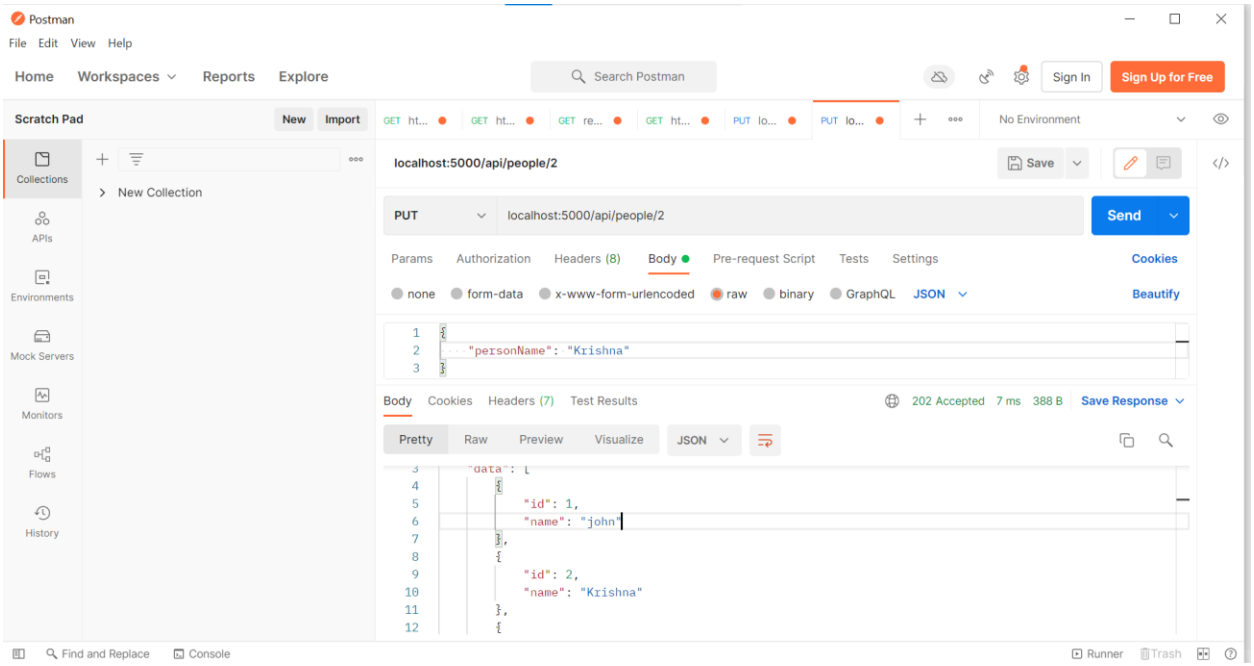


In this case, we are working with route /api/people/:id which requires an id to edit/update a particular item and it requires body containing the new value of the item.

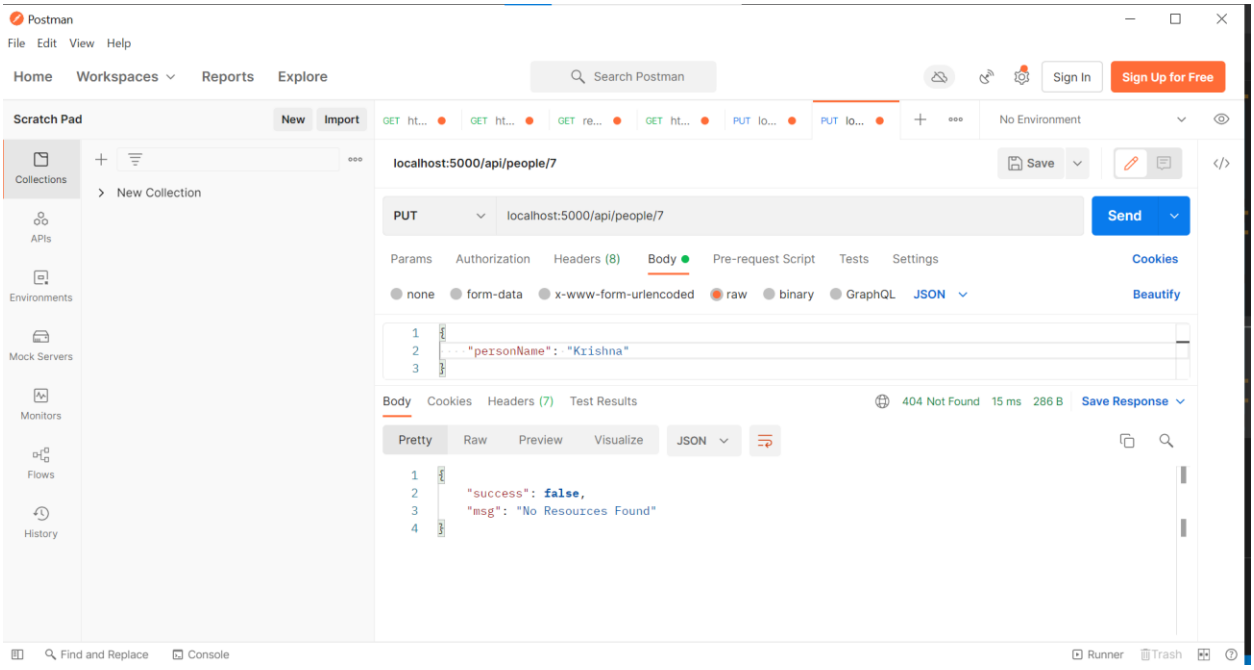
In Postman, localhost:5000/api/people/1. We are entering with Id and passing a request body



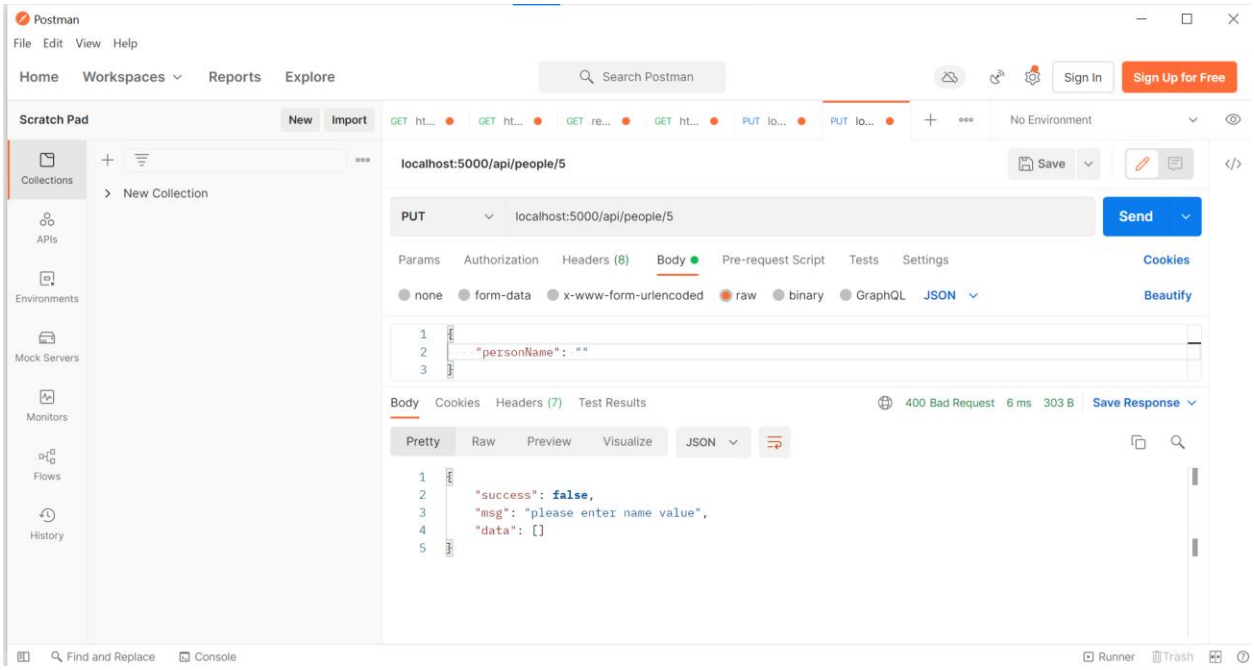
In Postman, localhost:5000/api/people/2



In Postman, localhost:5000/api/people/7 when there is no user with id 7



In Postman, localhost:5000/api/people/5 when there is no personName sent in request body

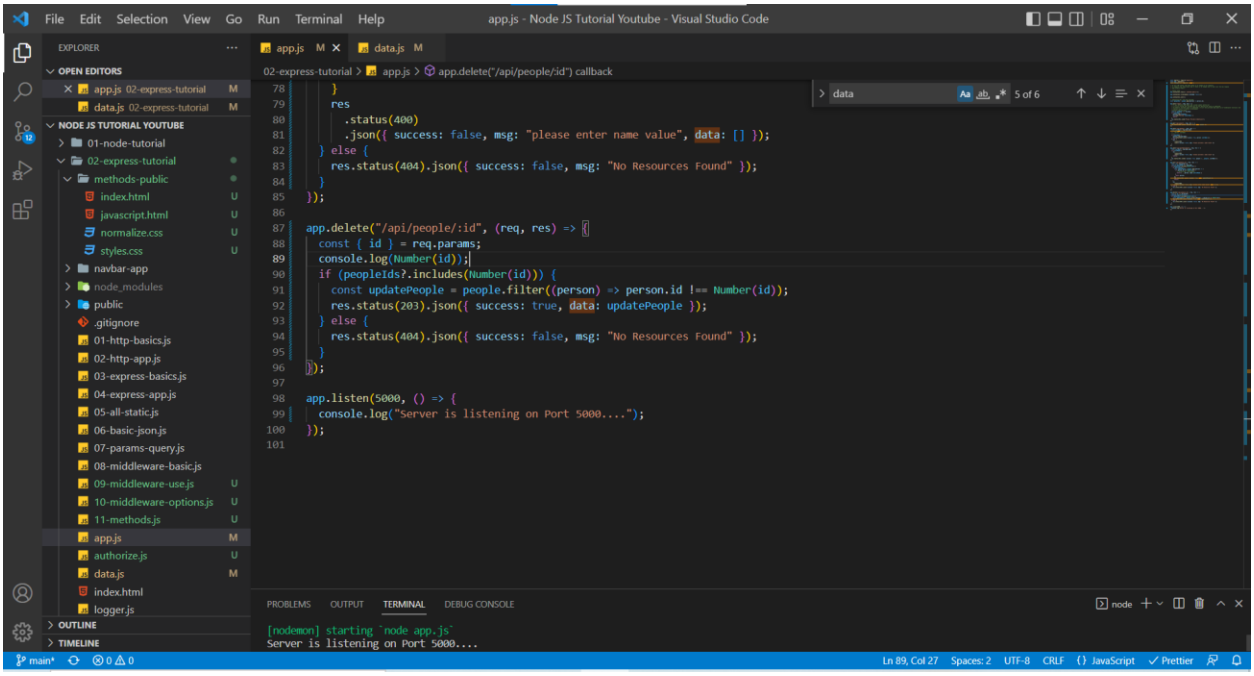


Methods – DELETE

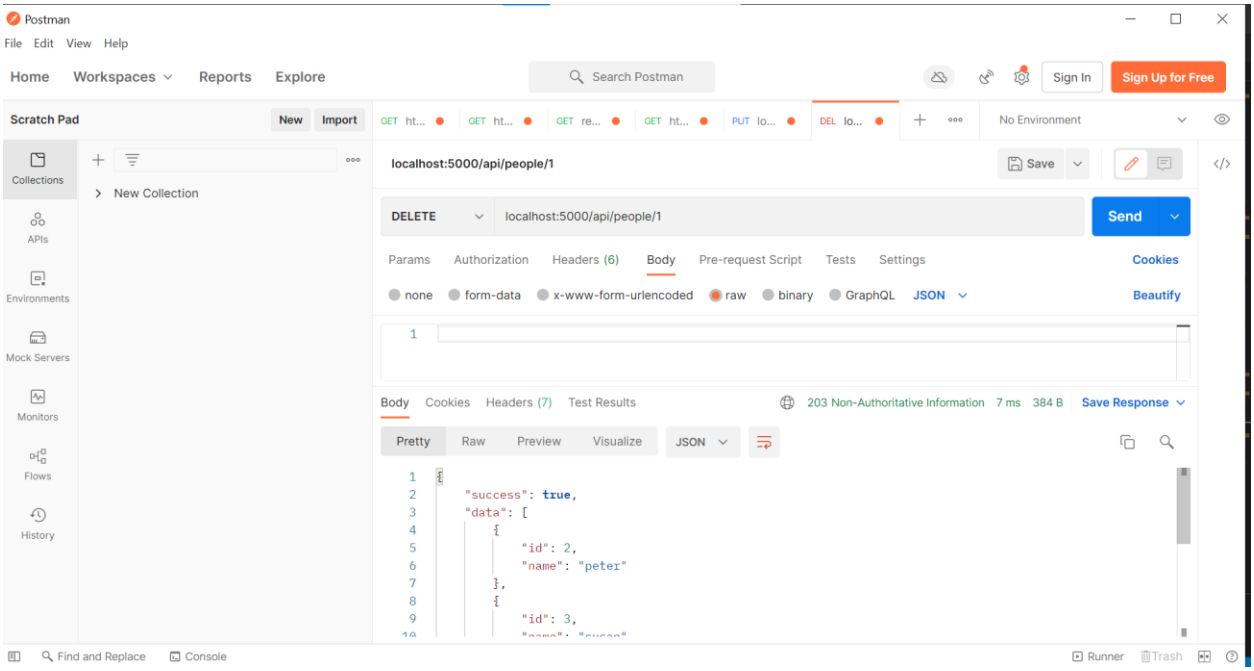
In this method, the setup would be extremely like the PUT method. API route requires an id to delete a specific item.

The only difference from PUT method is that we don't need send a request body for the DELETE request.

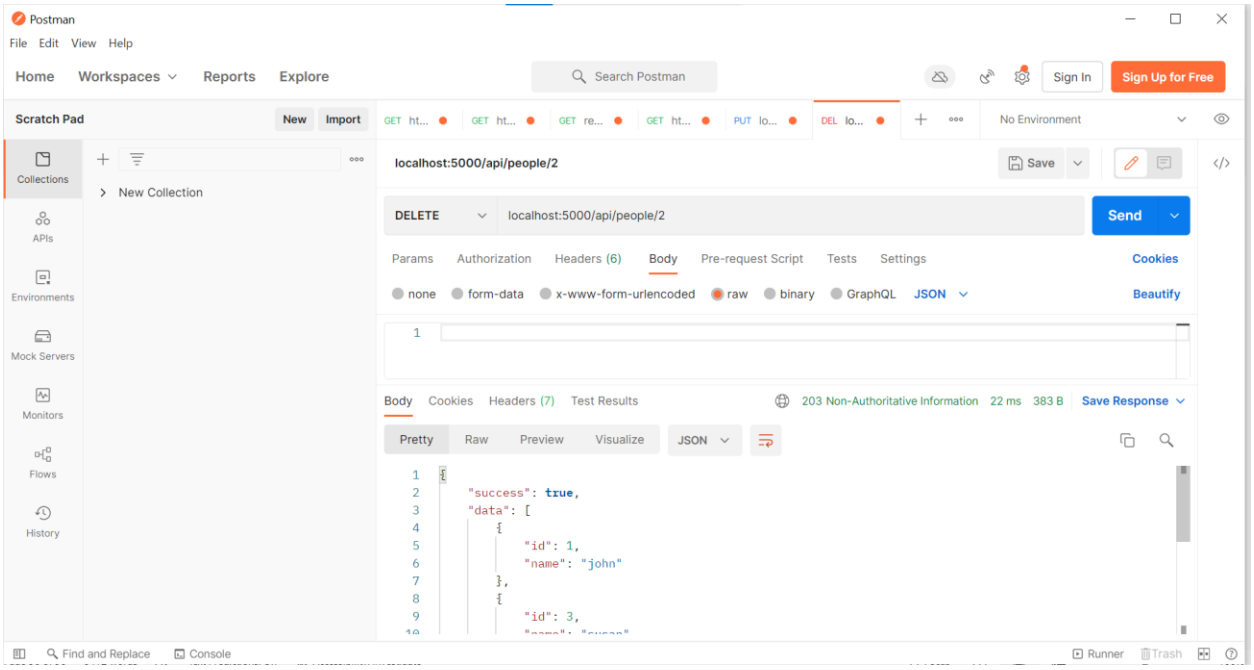
app.js



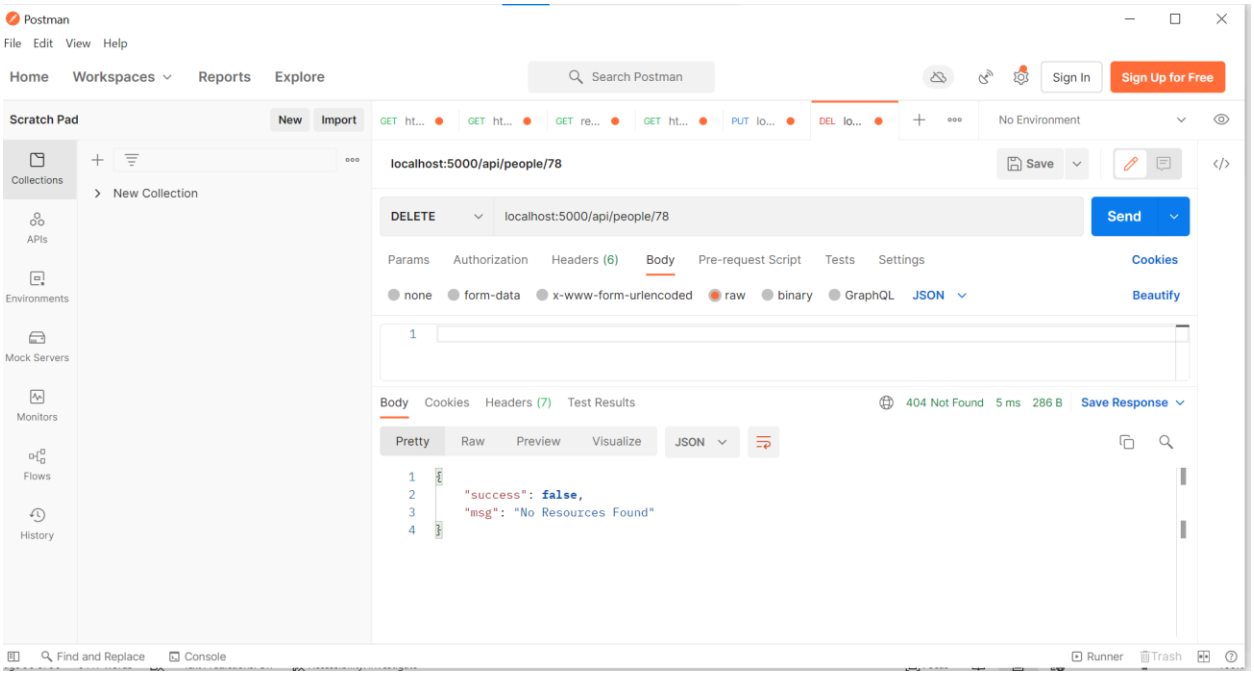
In Postman, localhost:5000/api/people/1 deleting the first user.



In Postman, localhost:5000/api/people/2 deleting the second user.



In Postman, localhost:5000/api/people/78 when user isn't available, we are providing 404 error.



Express Router – Setup

As we are having more routes and more functionality, we have another issue of app.js getting busy. The solution is using ***Express Router*** where we can group the routes together and then as far the functionality, we can set them up as ***controllers***.

Basically, when we connect with database in the next example, we use a pattern named MVC (Model View Controller). It is a nice pattern when we are setting up with the database. Here we are missing the model part in the MVC pattern because we haven't connected to the database.

In this example we are going to refactor to reduce the burden on app.js and make it lean.

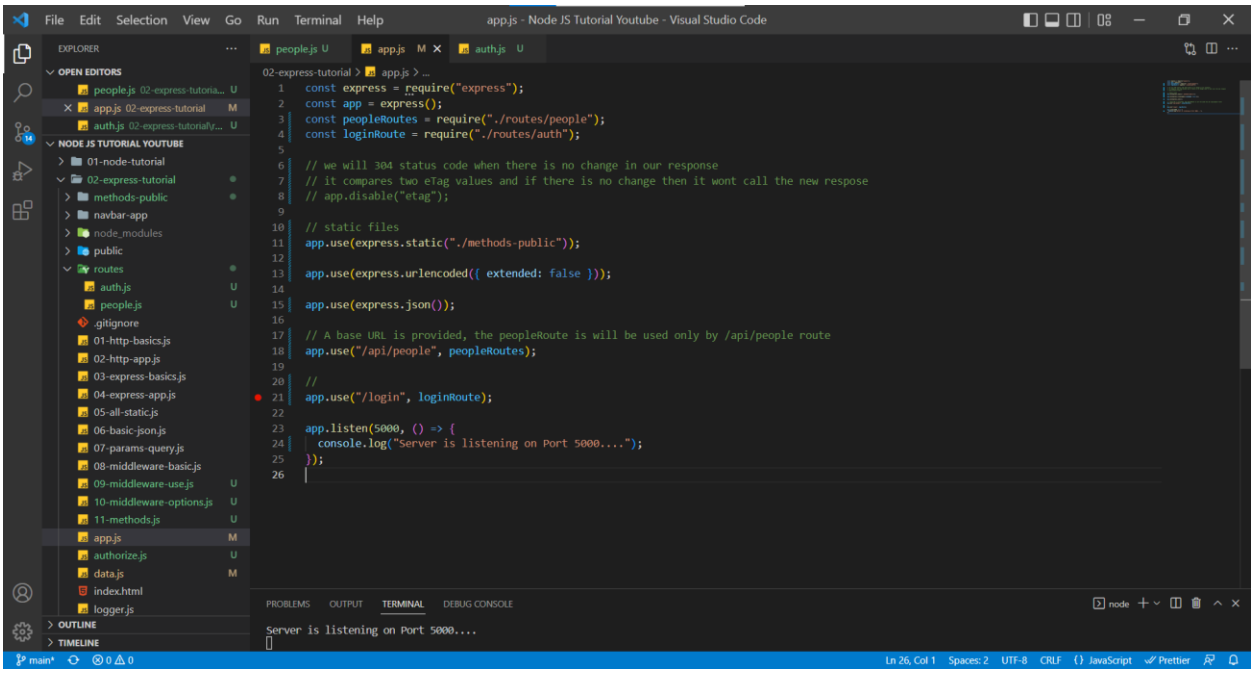
When we have a look at our code, we can most of the routes starting with /api/people and hence it would be a good idea to group those routes together and add one other login route (/login)

Let's create a folder and name it as routes as per common convention.

For login route, we are going with the filename auth.js and /api/people route we are using filename people.js

In people.js instead of setting up the app we go with router, and we explicitly get the router from express.

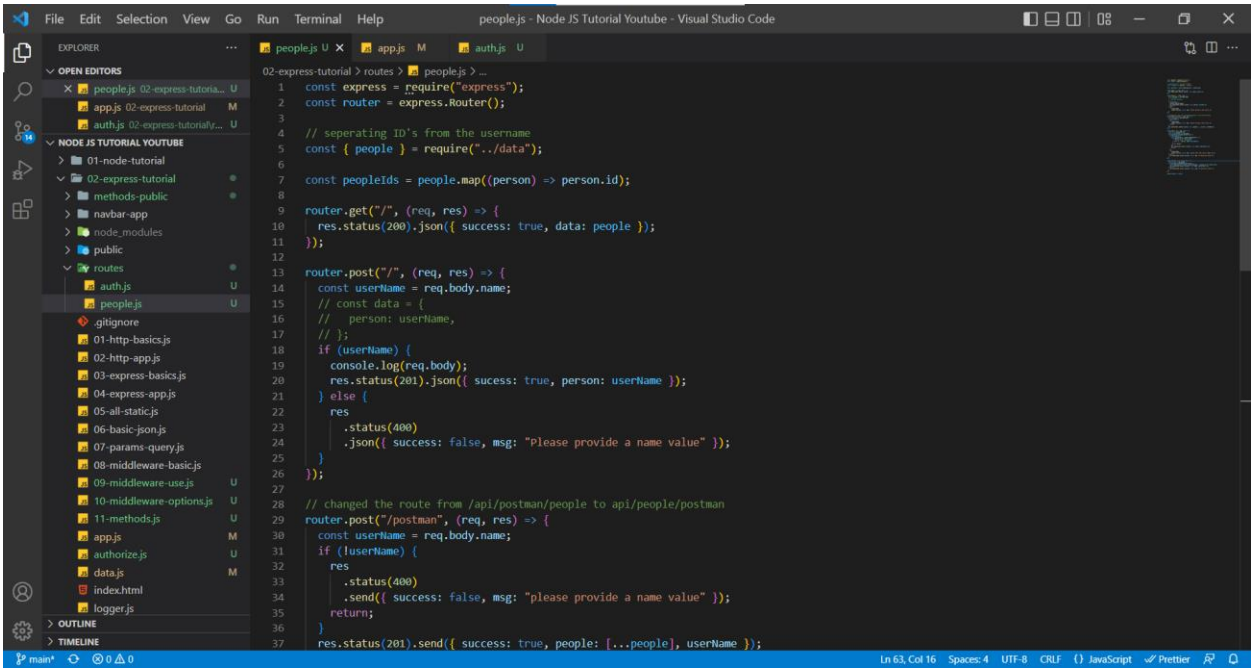
app.js

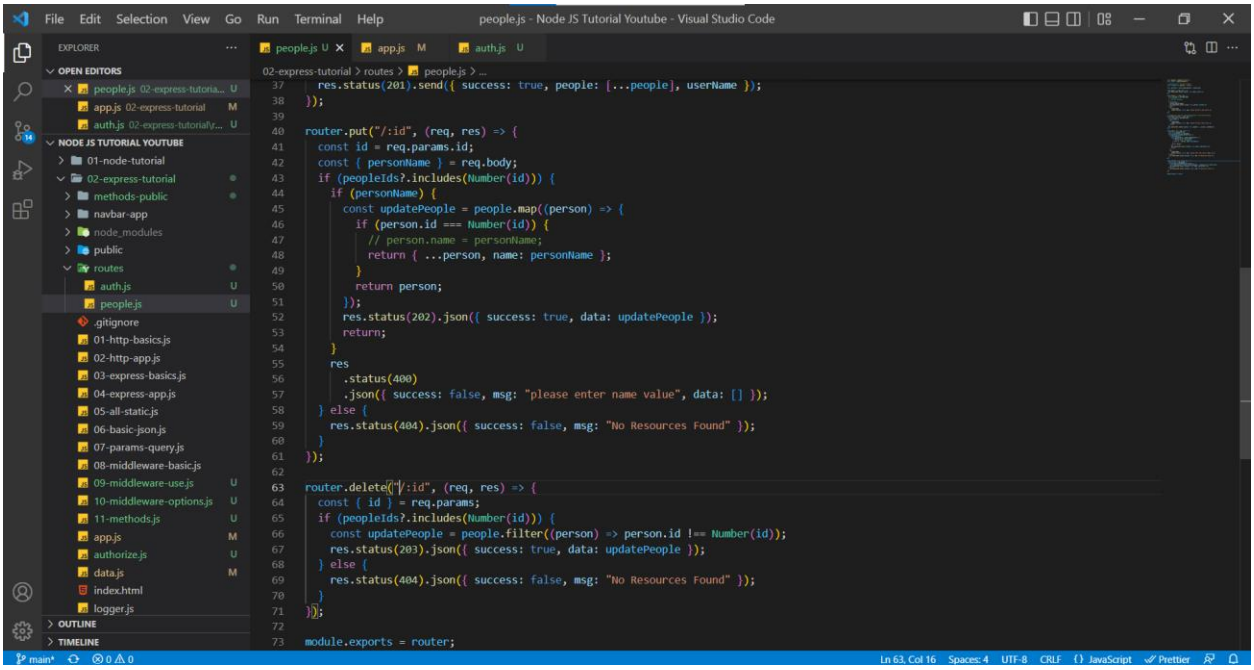


In app.js, we import the routes from each of the base route (/login and /api/public) and then we use the app.use() method to apply those routing routes and functionalities to the base URL.

people.js

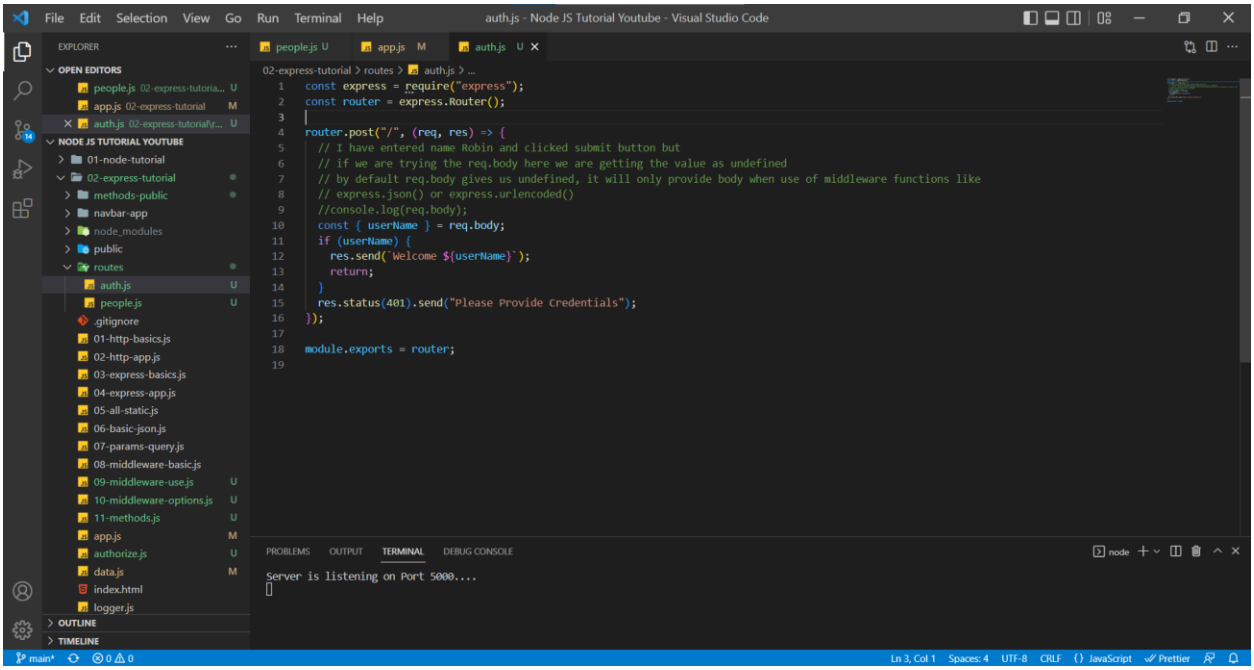
In people.js file we write all the routes starting with /api/people and their functionalities depending on the HTTP method.





auth.js

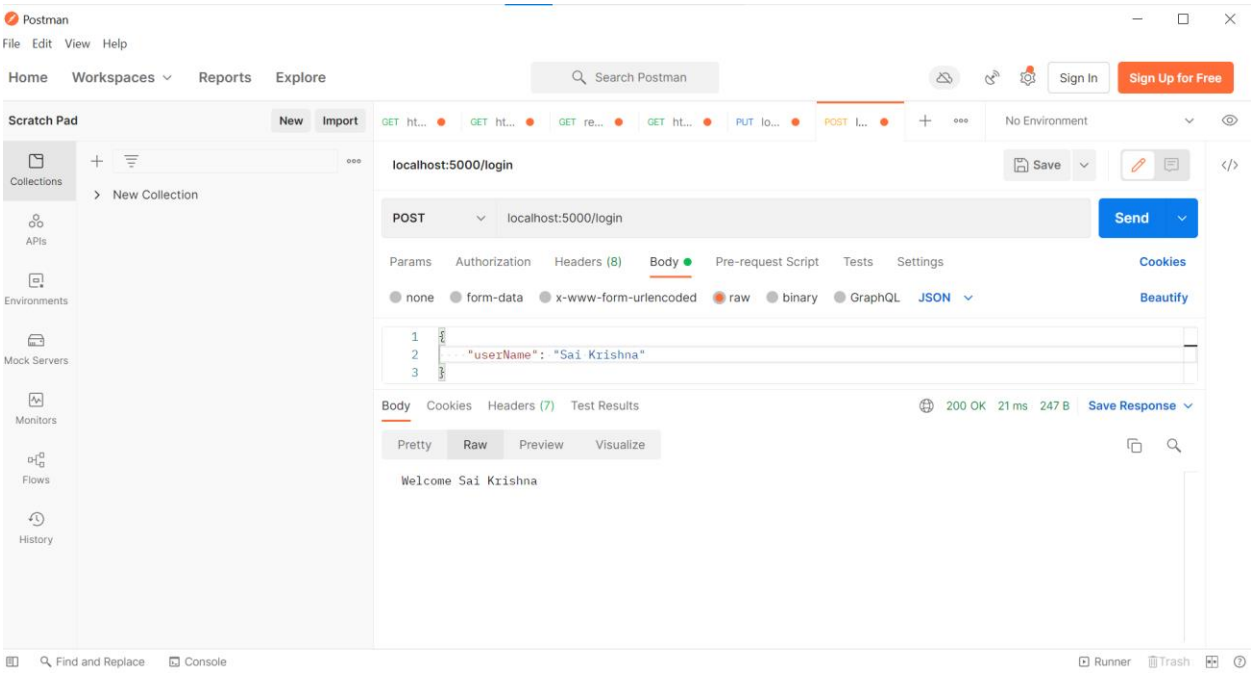
In auth.js file, we write the functionality for routes starting with /login.



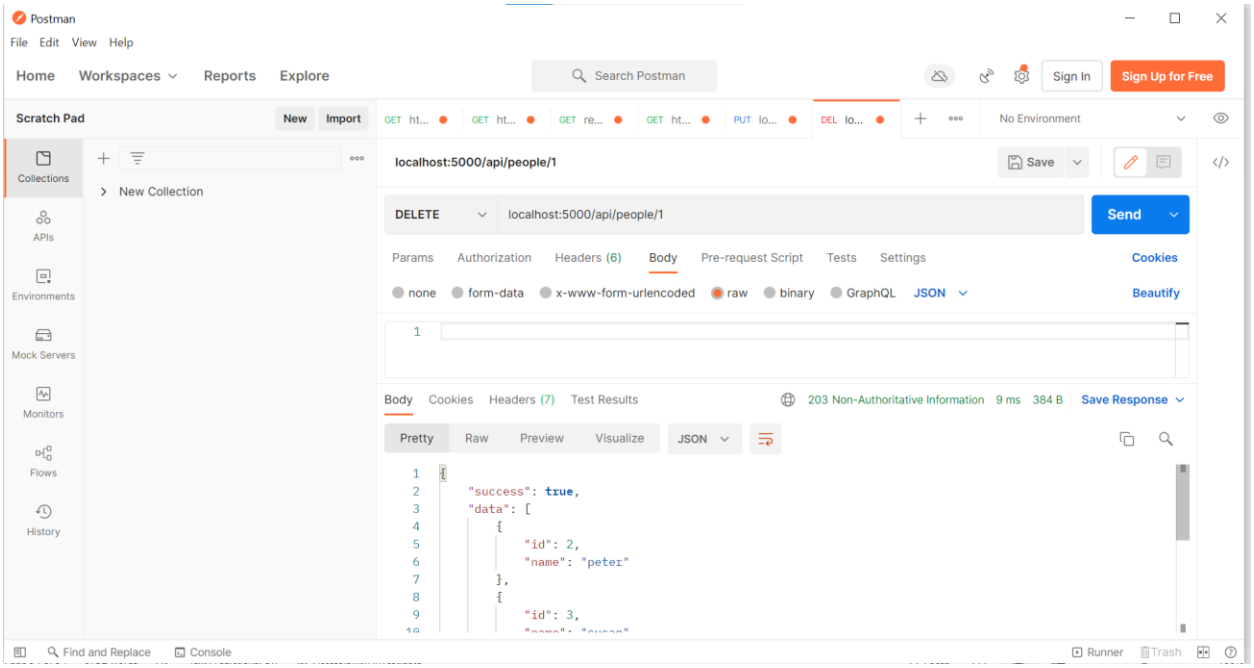
The process of using the routes and MVC pattern makes the similar routes to be grouped together and maintain lean and clean code in app.js file.

After Using Routing,

In Postman, localhost:5000/login



In Postman, localhost:5000/api/people/1



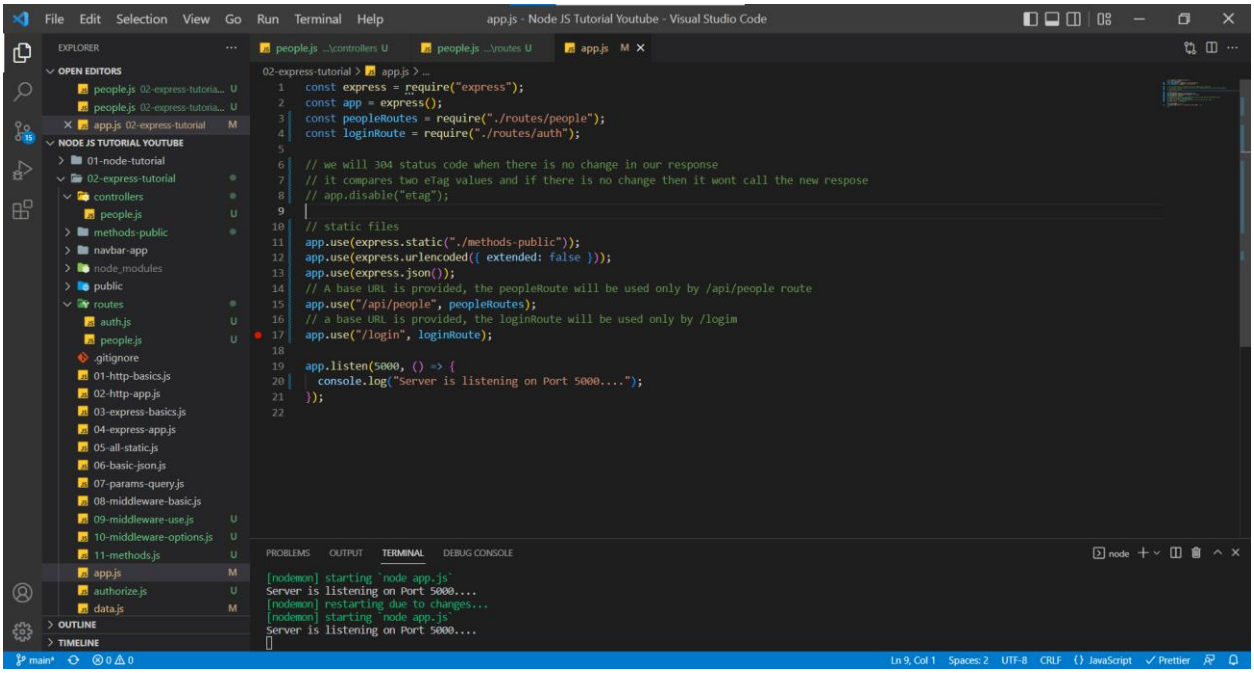
Express Router – Controllers

If we have a look at people.js it is still somewhat a mess because we have more routes and bunch of functionalities.

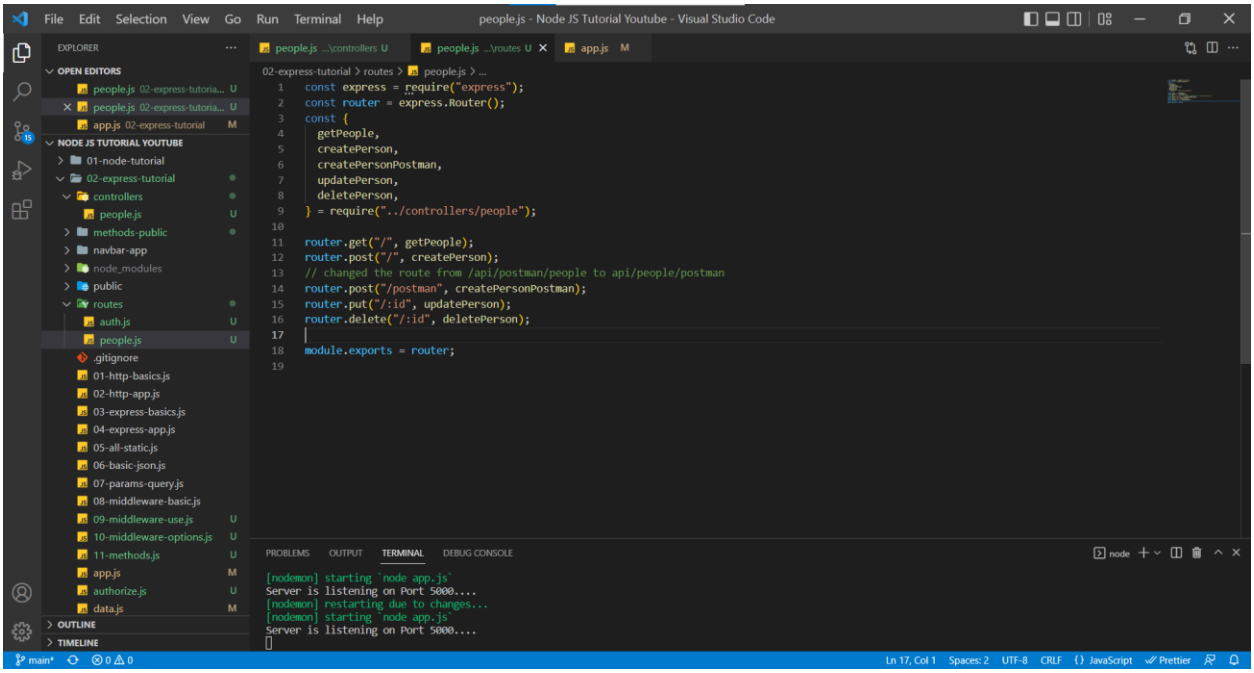
A better setup would be if we could split this into separate file (functions).

To maintain a clean code in routes/people.js we create a new folder named controllers where we refactor the code.

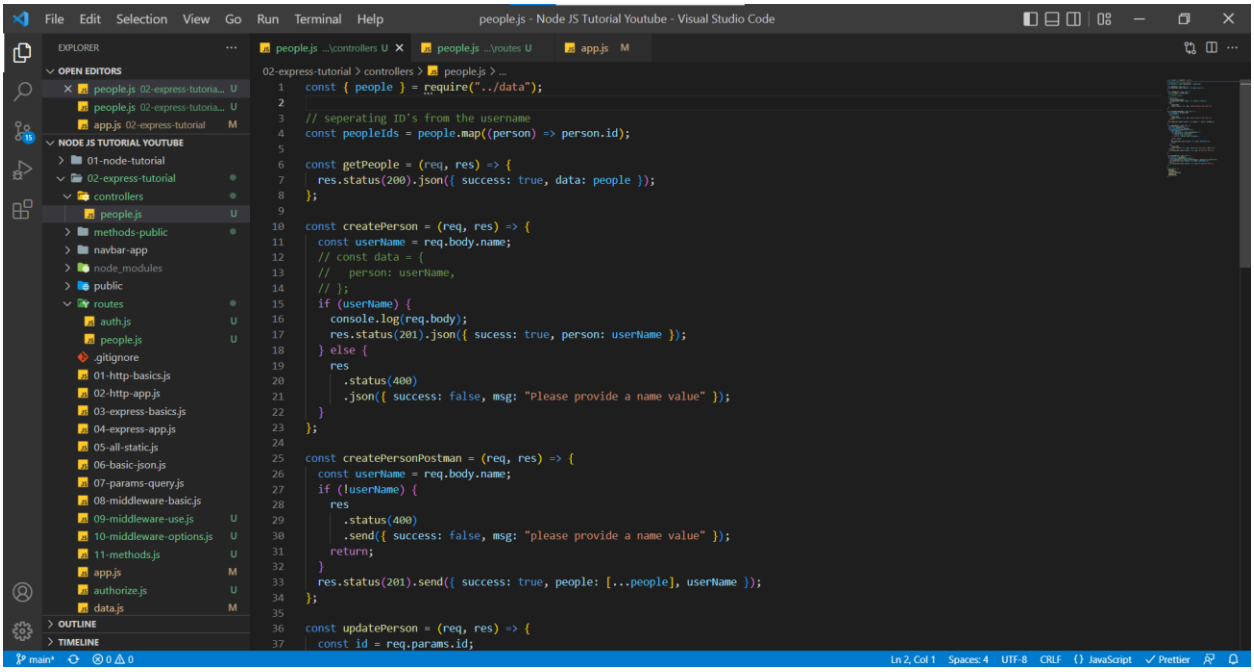
app.js

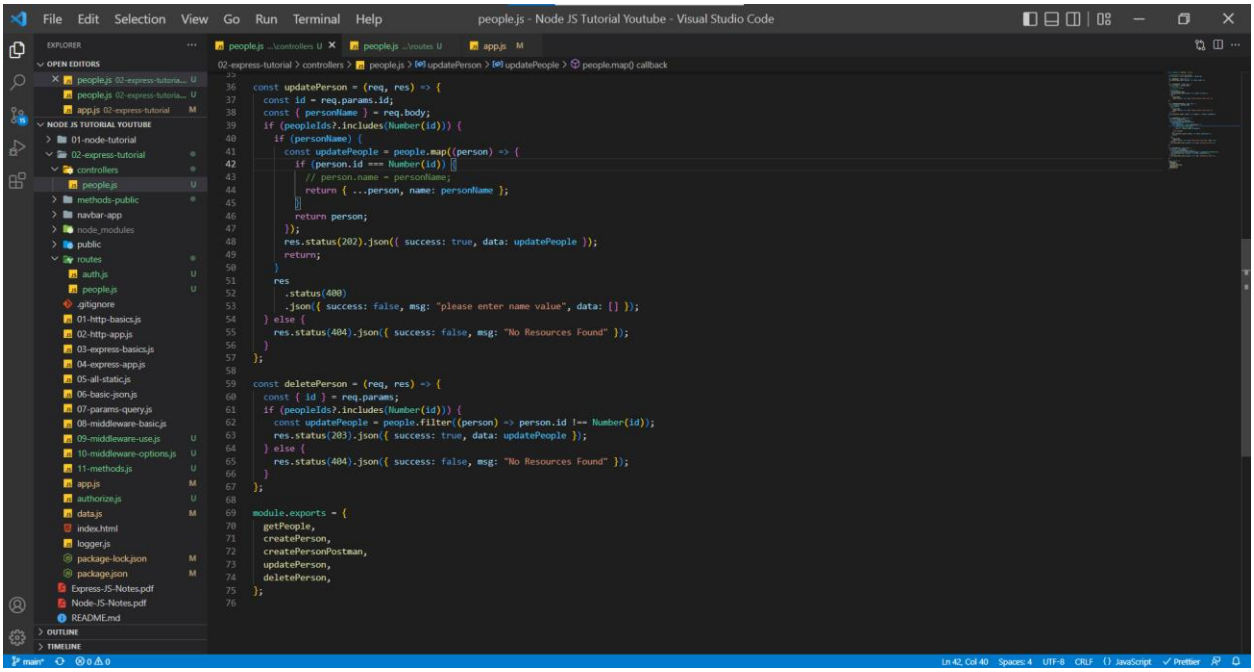


routes/people.js (Routes are also made lean and clean) we import the functions from controllers and then use them as per the route.



controllers/people.js

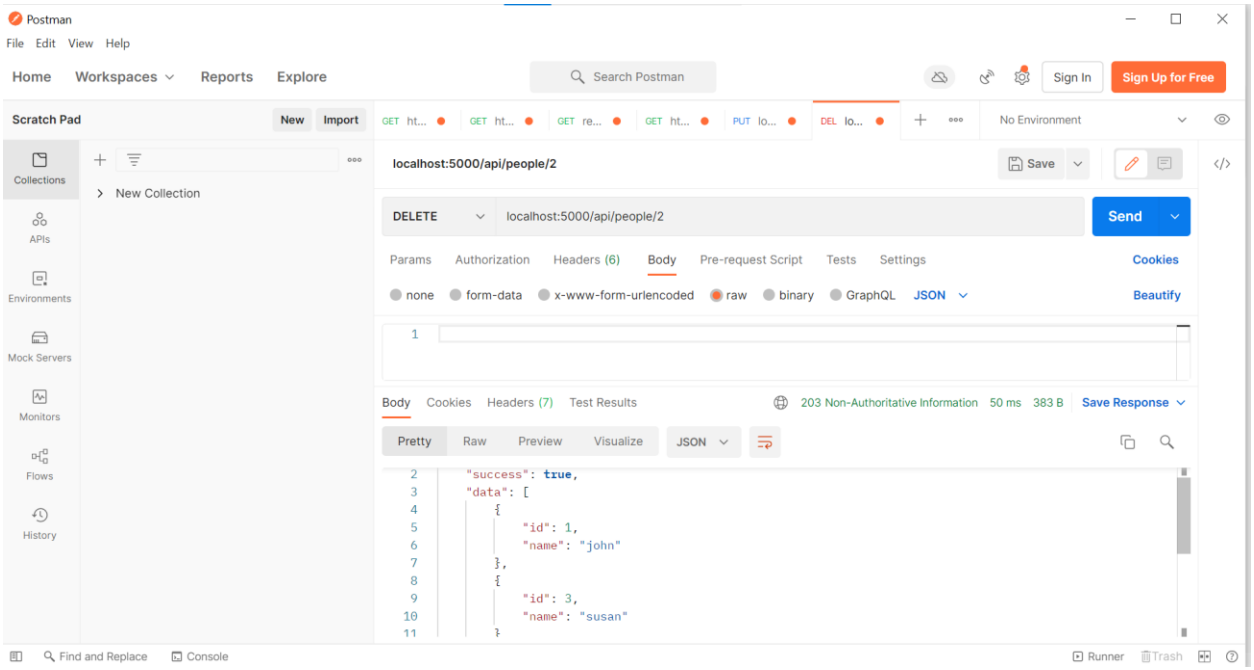




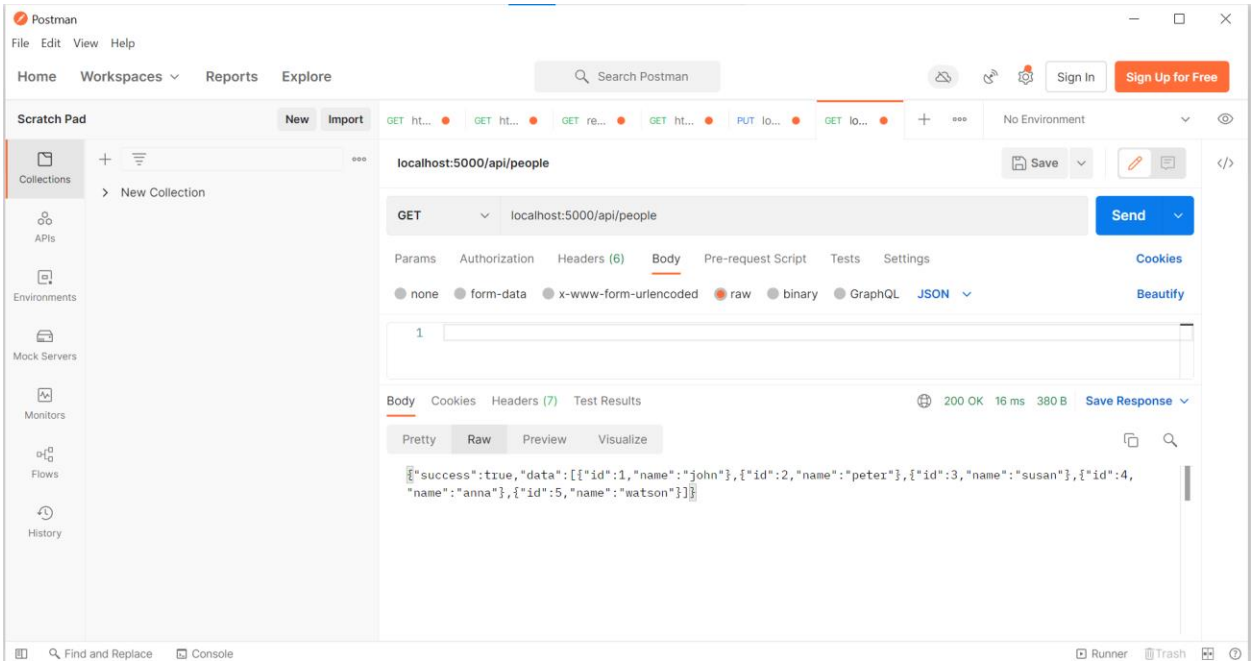
Controllers folder contains all the functionality code ensuring our code is lean and clean.

We have created controller for /api/people route because there are lot of functionalities but /login route has only one functionality, hence we didn't create a controller for that route.

In Postman, localhost:5000/api/people/2 with DELETE Method.



In Postman, localhost:5000/api/people with GET Method



We can set the routes in two methods,

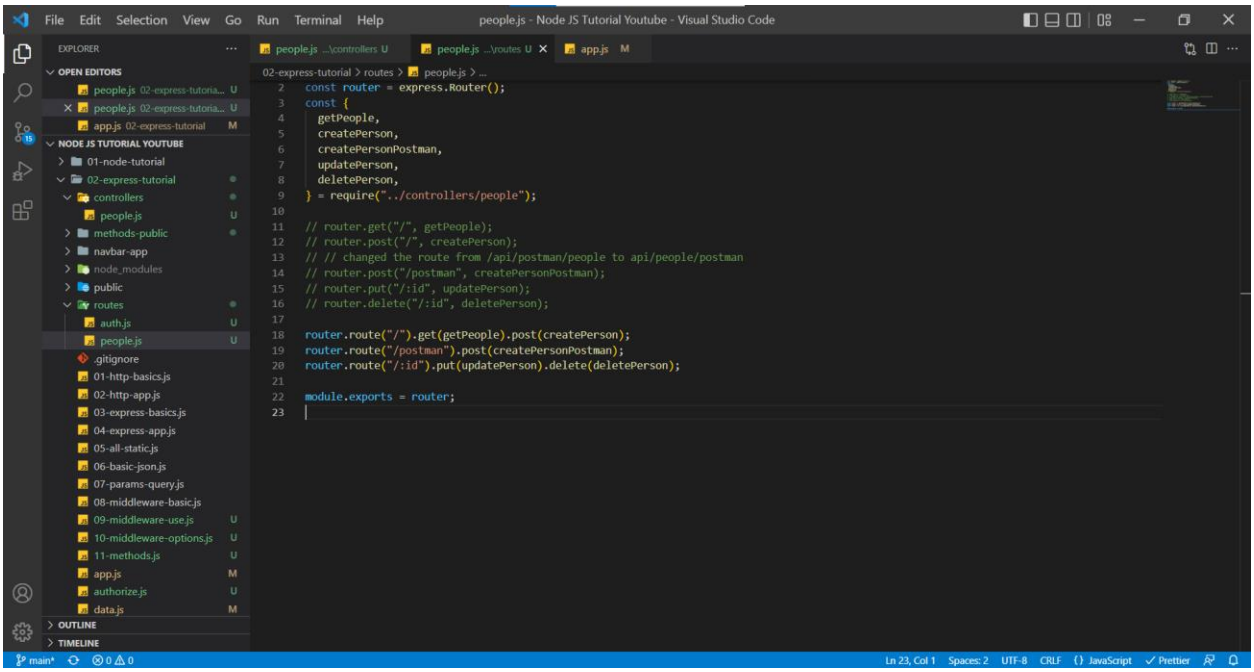
1. router.get("/", getPeople); // we can set the route for each different HTTP method

```
router.get("/", getPeople);
router.post("/", createPerson);
// changed the route from /api/postman/people to api/people/postman
router.post("/postman", createPersonPostman);
router.put("/:id", updatePerson);
router.delete("/:id", deletePerson);
```

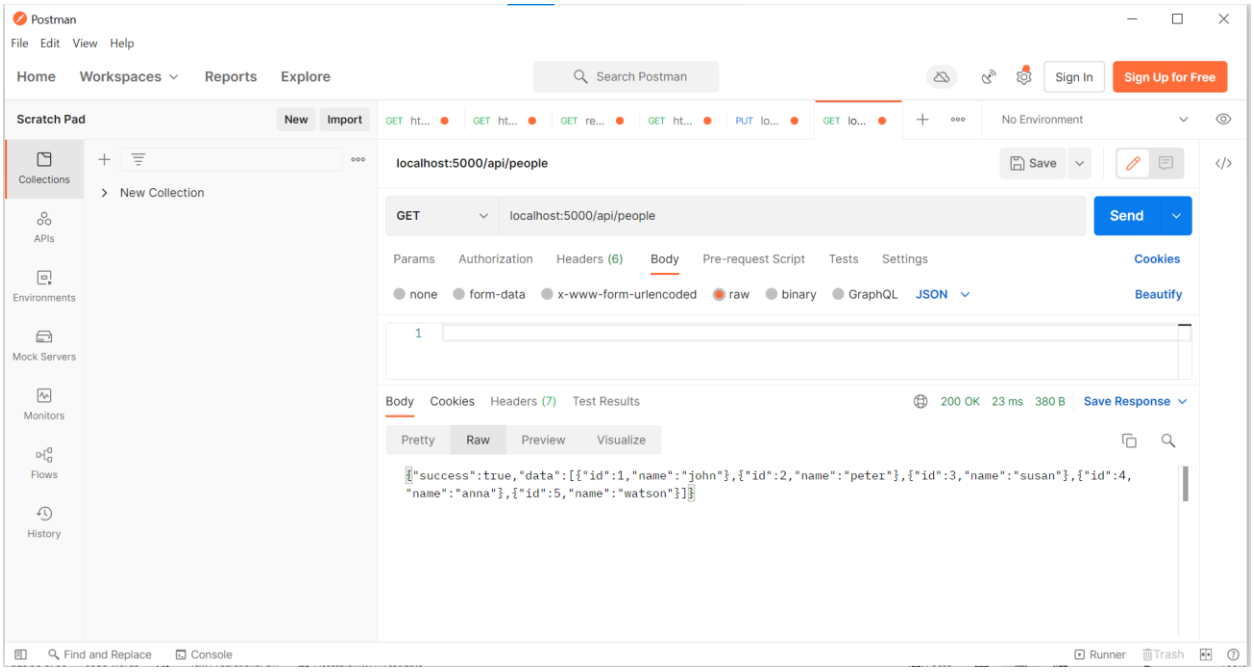
2. if we have same the URL with different HTTP methods then we can chain them using router.route() method.

```
router.route("/").get(getPeople).post(createPerson);
router.route("/postman").post(createPersonPostman);
router.route("/:id").put(updatePerson).delete(deletePerson);
```

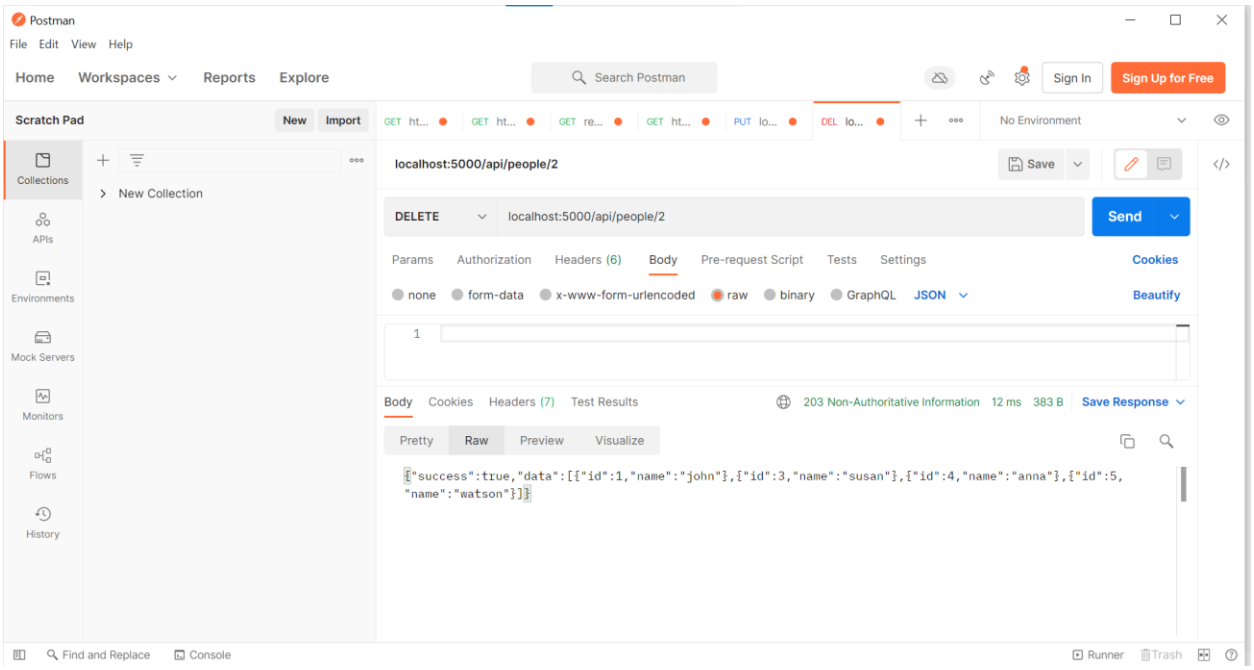
routes/people.js



In Postman, localhost:5000/api/people with GET Method



In Postman, localhost:5000/api/people/2 with DELETE Method



End of Express Tutorial Module

Until Here we are done with Express Fundamentals