Task Manager API

With this project we will learn to setup and connect to the cloud database. So effectively we will learn how to persist our data to the cloud and will perform CRUD operations.

CRUD – Create, Read, Update and Delete

Instead of a regular to-do app which stores data in local storage, in this project the data is stored in the cloud database and persist data to the cloud.

Assume a frontend were,

First API Call – on load – get all the request and display on the screen. (GET Request)

Second API Call – Adding a new task into the list and getting all the tasks so that UI can display all the tasks. (POST Request)

Third API Call – Editing the present task, getting single task with id, and displaying all the details. (GET Request with ID).

Forth API Call – Saving the details of the edited task in the cloud. (PUT request with ID)

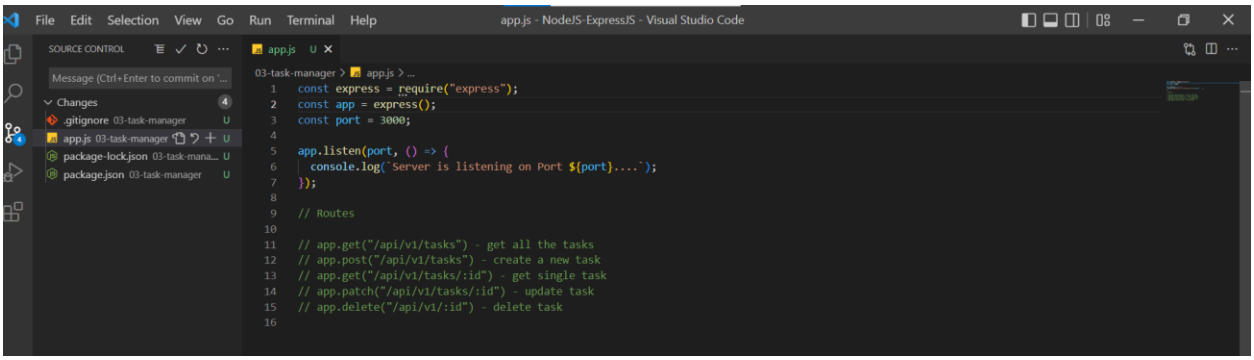
Fifth API Call – Deleting a task from the list. (DELETE with Request ID)

Version naming is important because whenever a new development comes and when accommodate that development in our existing API and route, we tend to create a new route with different version name.

The basic naming structure for an API would be

http://Domain-Name/api/version-name/items/:id

Routes in the Task Manager API Project

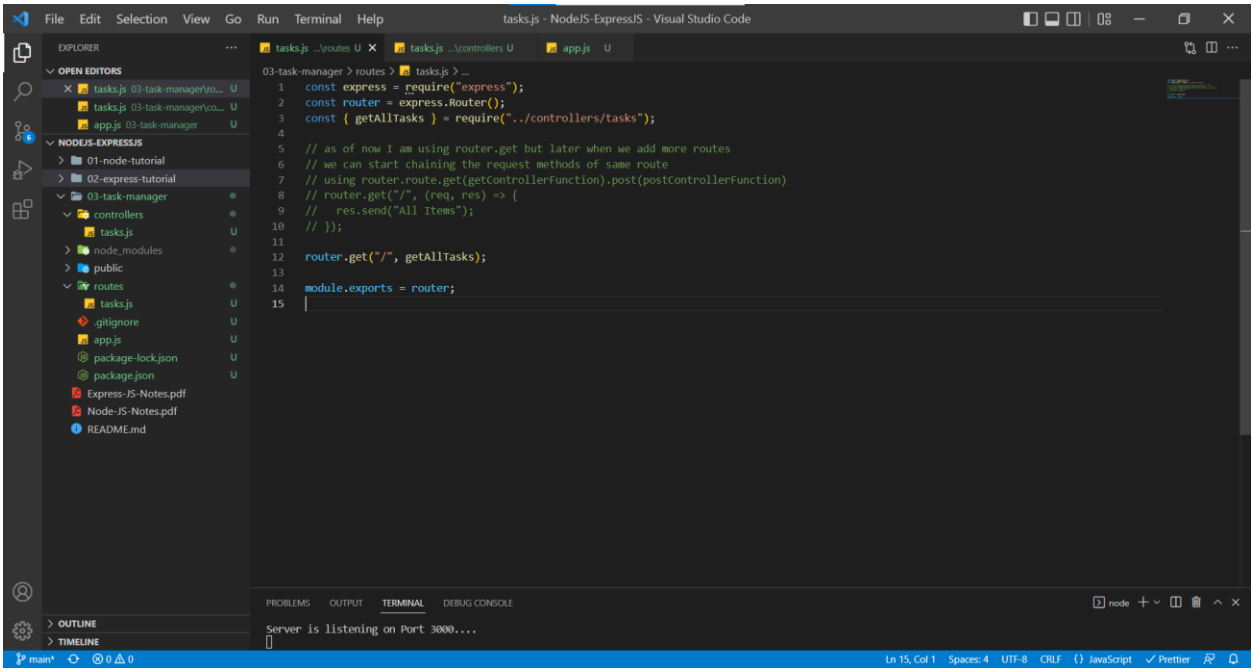


To maintain the app.js lean and clean, we need to use the MVC pattern and hence we follow that pattern throughout the project.

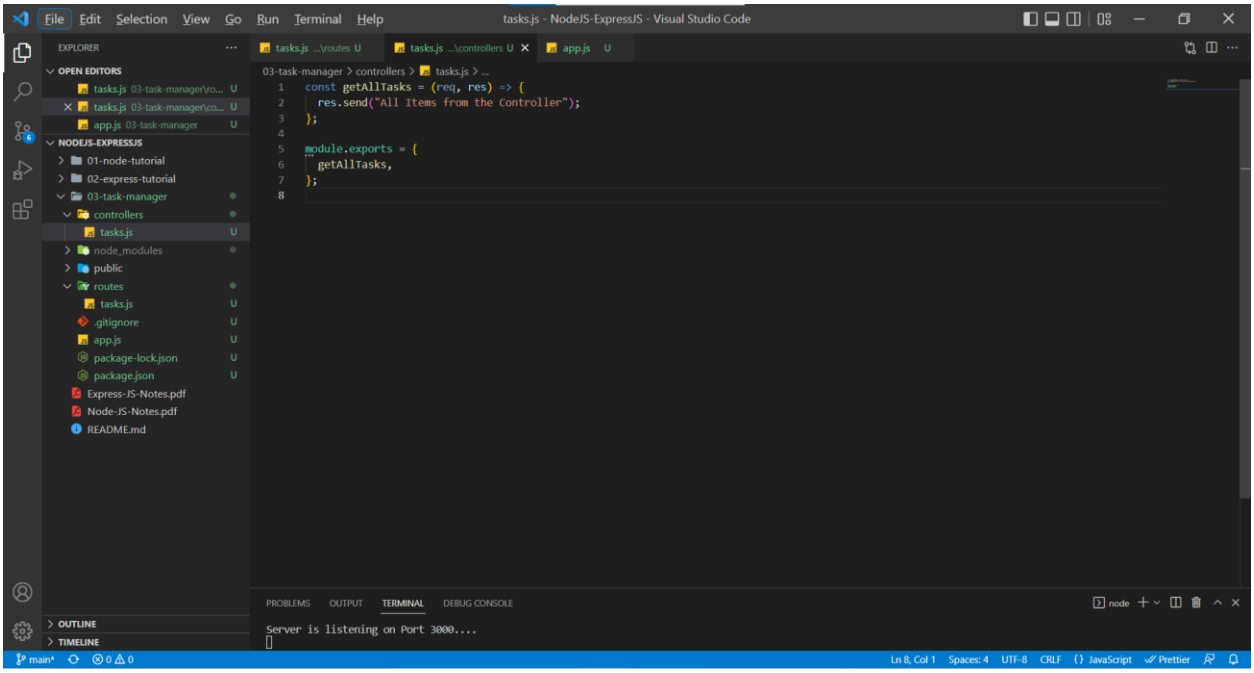
As of now, we are creating two folders namely controllers and routes.

Let us start with creating a route

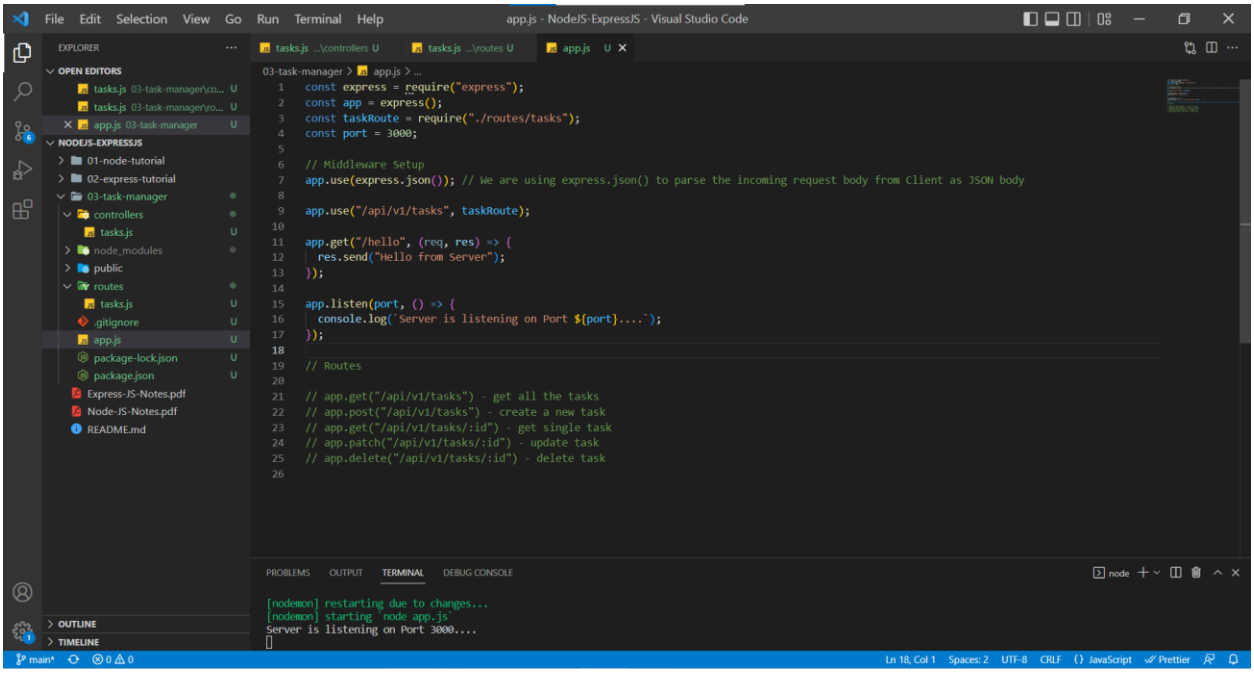
tasks.js file in routes folder



tasks.js in controllers folder



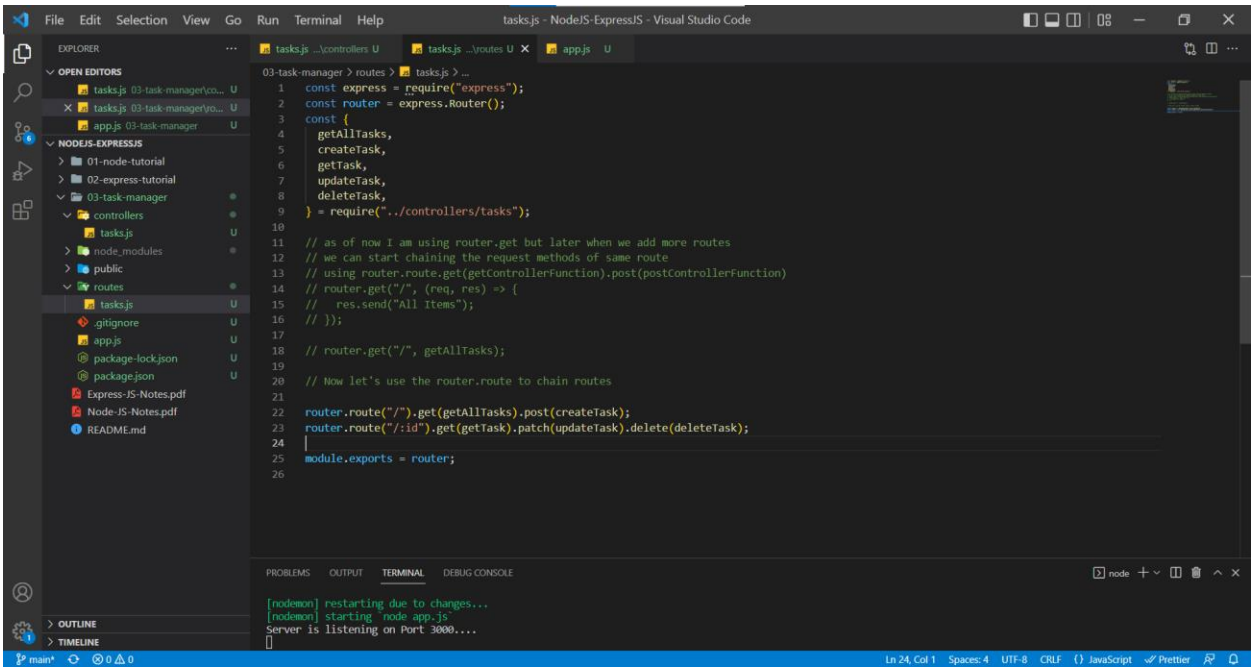
app.js



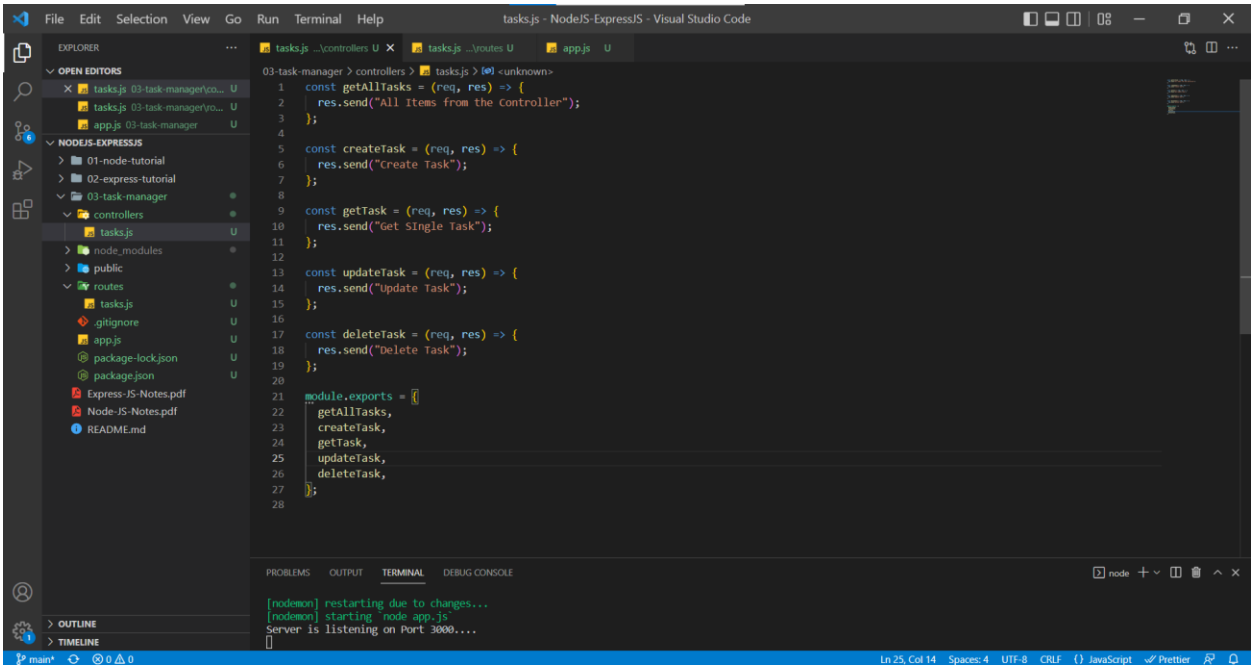
Now let's create controller functions and set up routes for other API routes.

There won't be any change in the app.js

tasks.js in routes folder



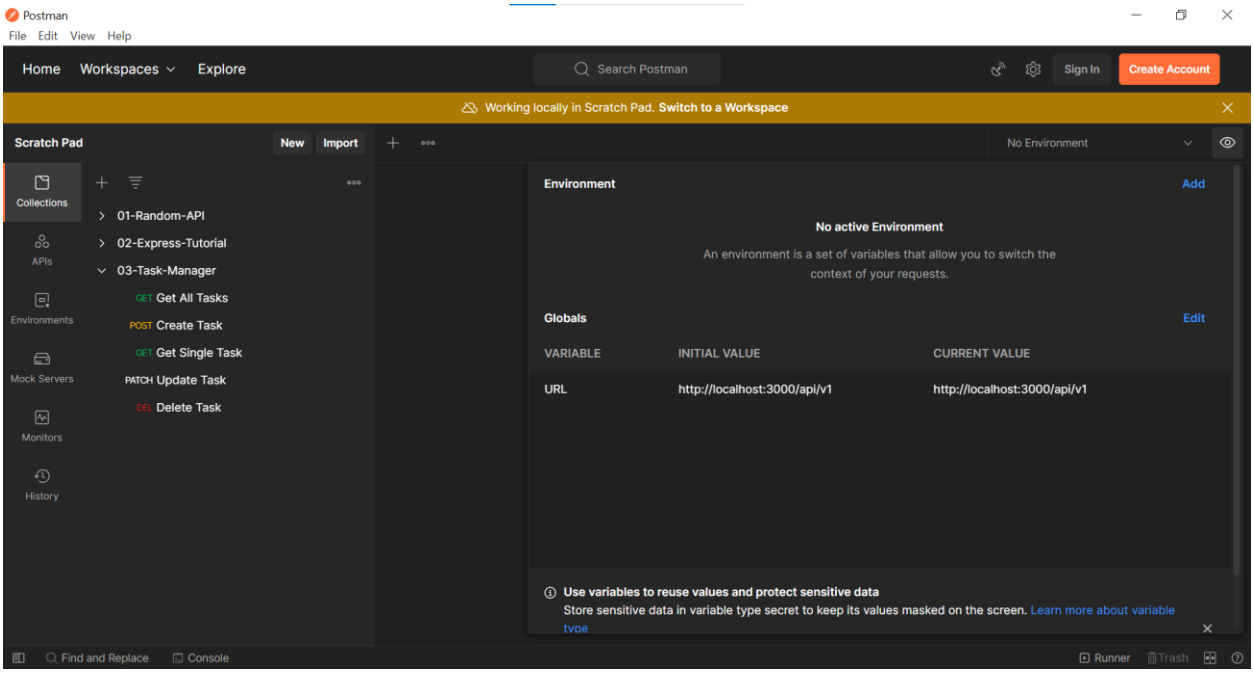
tasks.js in controller folder



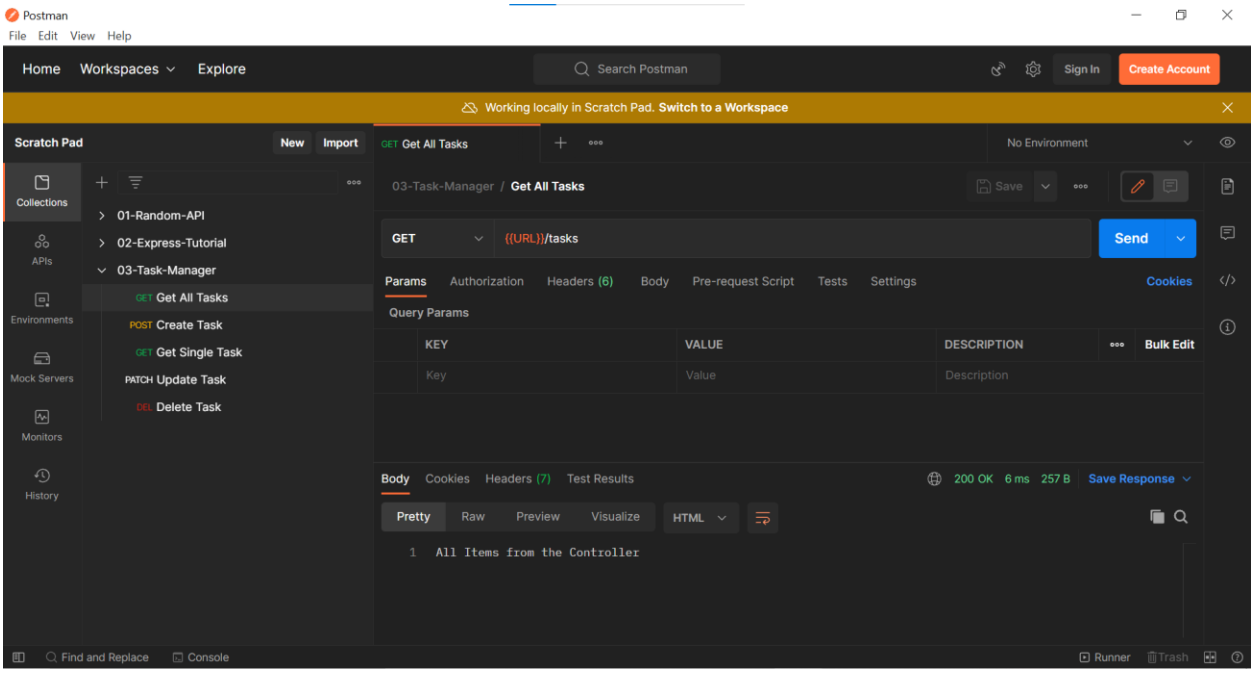
Now let us test our routes in Postman

Let us create a collection in Postman because we will be setting up multiple routes and they will reference the same application which is going to easier as we create more and more applications.

Created a collection of Task Manager in Postman as well set up a Global Variable which can be accessed throughout the postman. (Instead of always writing localhost:3000 for every request, we can setup that link as a global variable)



Get All Tasks Request in Postman



API Routes that have been setup

We are using the above structure because we are building the REST (Representational State Transfer) API. We want to create an HTTP interface where other apps mostly frontend can interact with our data.

REST is arguably the most popular API design pattern and essentially it is a pattern that combines HTTP verbs, route paths, and our resources aka data. So effectively REST determines how our API looks like.

Since JSON is a common format for receiving and sending data in REST API, we will us that approach as well. Right now we are send() method to send the request but eventually we will use json() method to send the response body.

Mongo DB

- No SQL, Non-Relational Data base
- Stores JSON
- Easy to get Started
- Free Cloud Hosting – Atlas

Unlike traditional database where we store data in rows and columns, in No SQL we store data as JSON, and it basically doesn't care how data relates to each other. Instead of tables we have collections which represent group of items and instead of rows we have documents which represent single item.

A document is set of key value pairs and as far as data types we can use String, Arrays, Numbers, Array objects and may more.

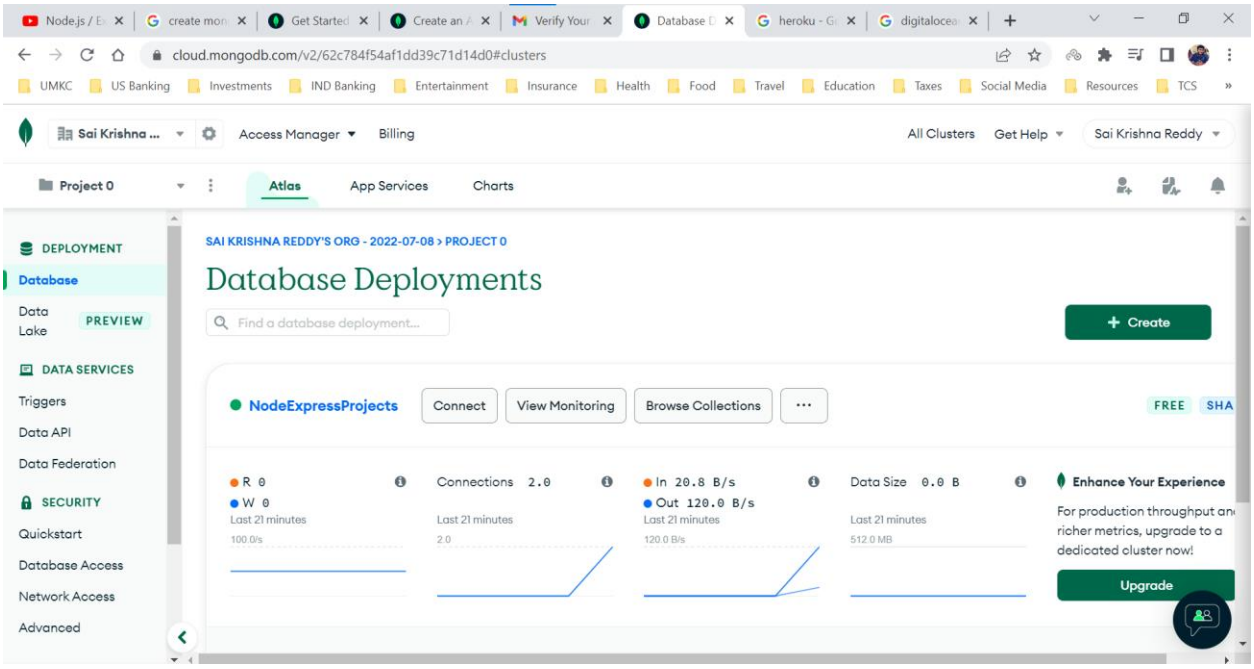
In this project we are using MongoDB Atlas, which is an official option, basically it is created by the people who created MongoDB. We are using the free tier for which we need to create an account.

Let's setup and configure MongoDB atlas so we cam host and manage our data in the cloud.

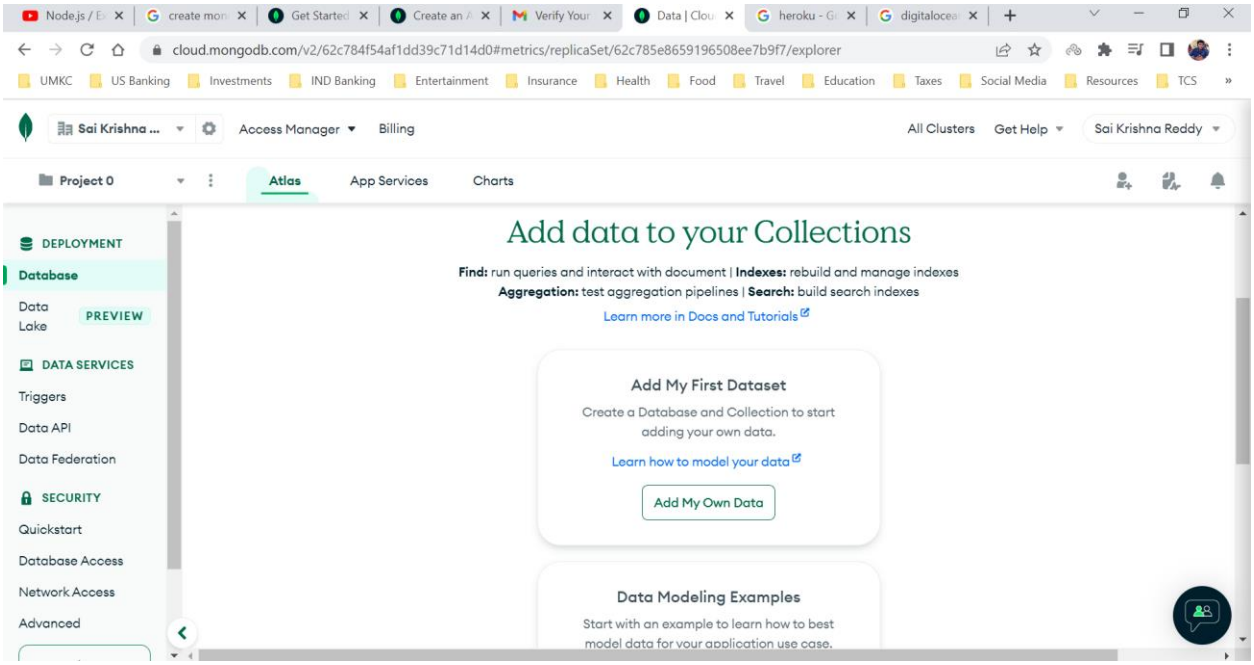
Heroku and Digital Ocean are the most popular platforms to host Node JS applications. When Hosting with Heroku, we need to set our IP address as anyone access ***(Allow from anywhere option)*** so that our application won't break, or we don't get any error. In digital ocean, we need to change from our Local IP address to Production IP address.

We have created a Cluster in MongoDB and added few accesses like user and IP address restriction. We copied connection string from MongoDB and added it ***db*** folder of our application.

Let us create a database in MongoDB with dummy data.

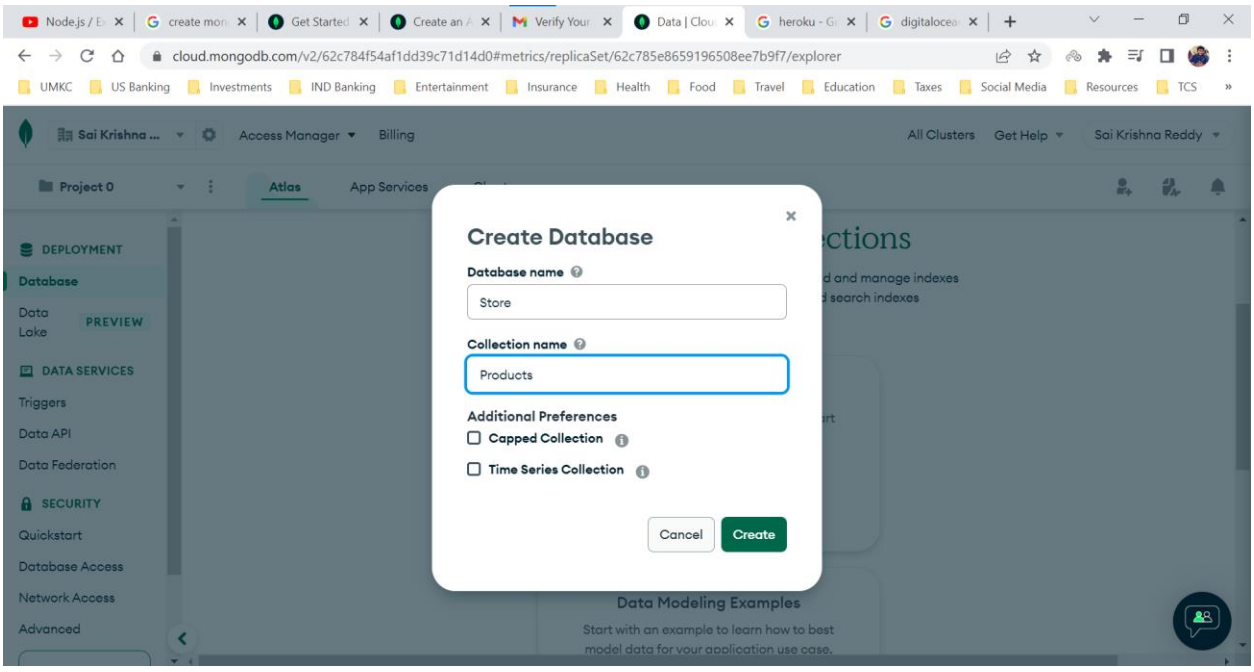


Click on Browse Collections and below screen will appear



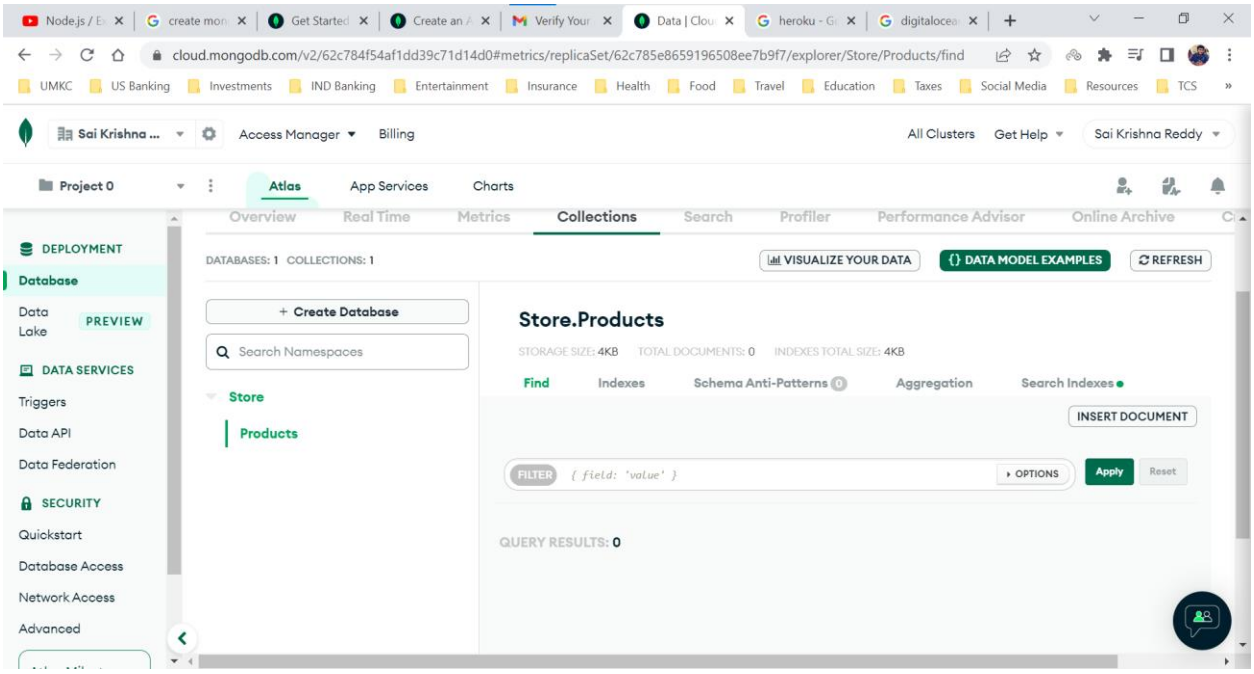
Click on Add My Own Data and below screen will appear.

We are creating a database named Store which contains a collection called Products (In SQL equivalence, Store is the database and Products is the table) and click on Create.

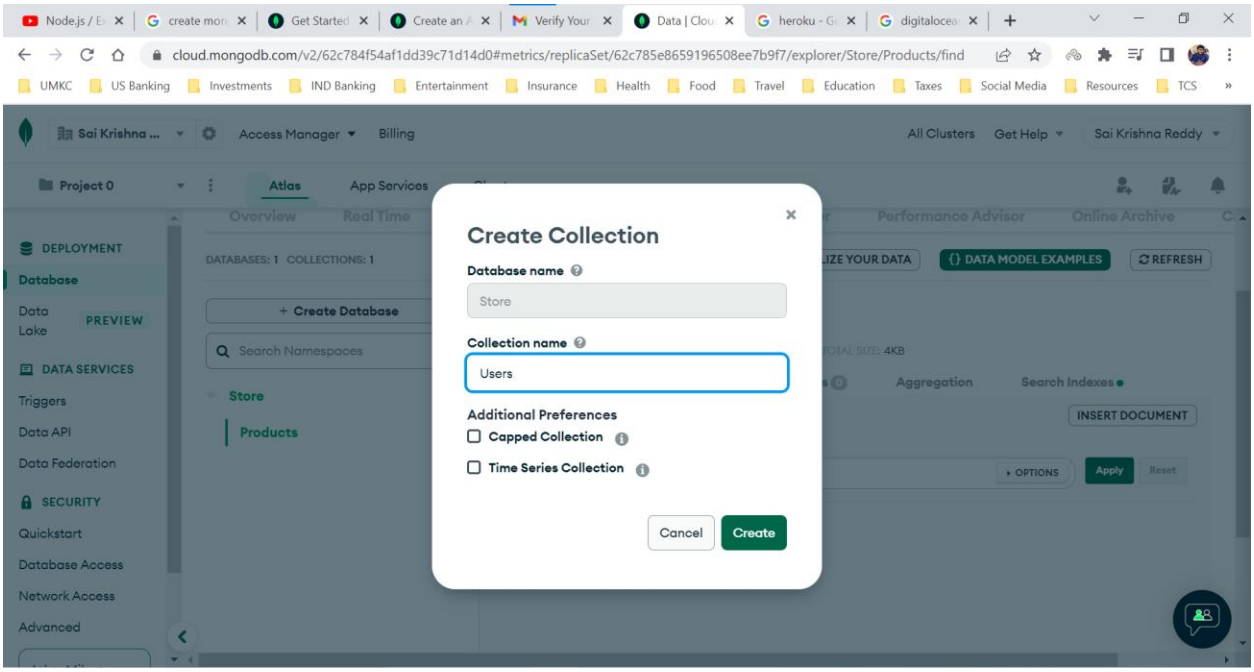


We can have as many collections (tables) as we want. In general SQL database we will be having many tables under one database.

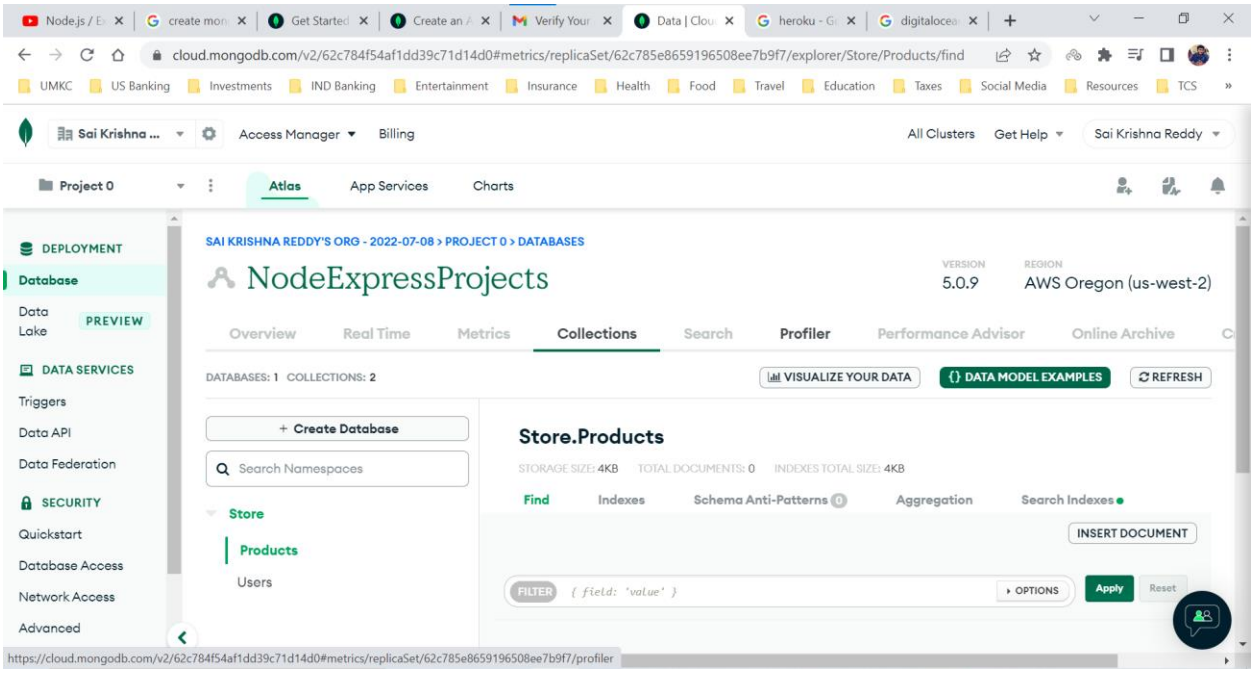
Click on the + (Plus) icon beside the database Store to create another collection.



We are creating a collection (table in SQL equivalent) in database named Store.



After Creation of collections (Products and Users) in table Store.

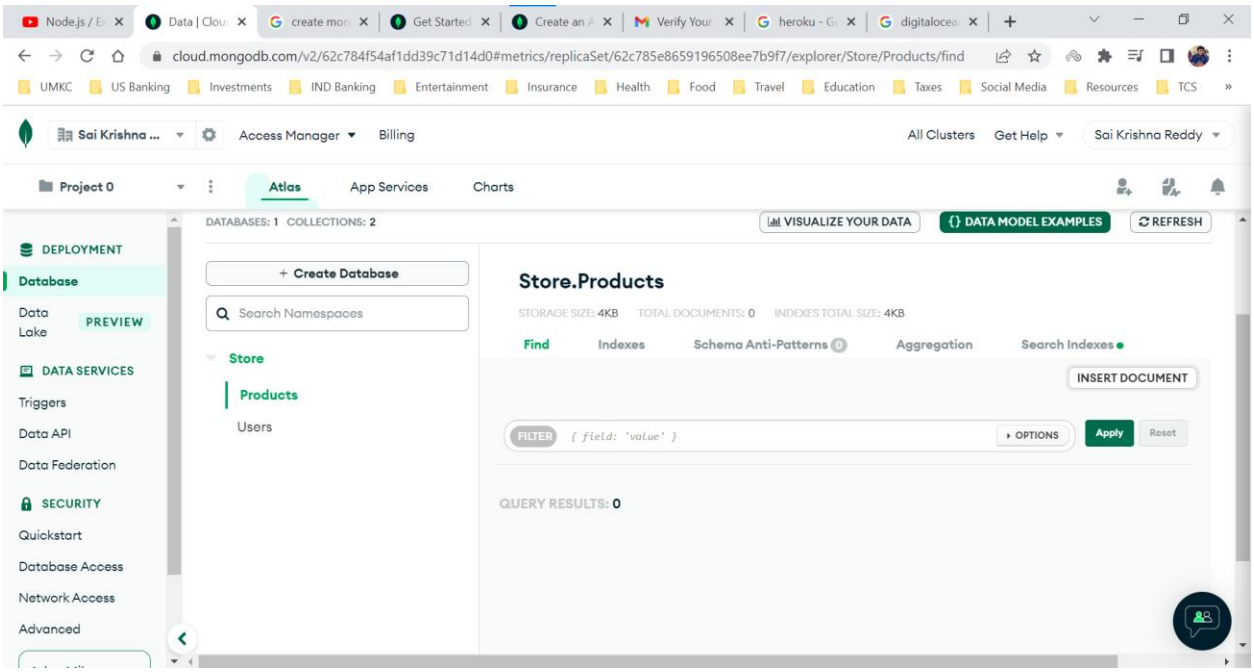


After creating collection, we need to add documents to the collection. In MongoDB a document effectively refers to a single item. Document is a set of key value pairs.

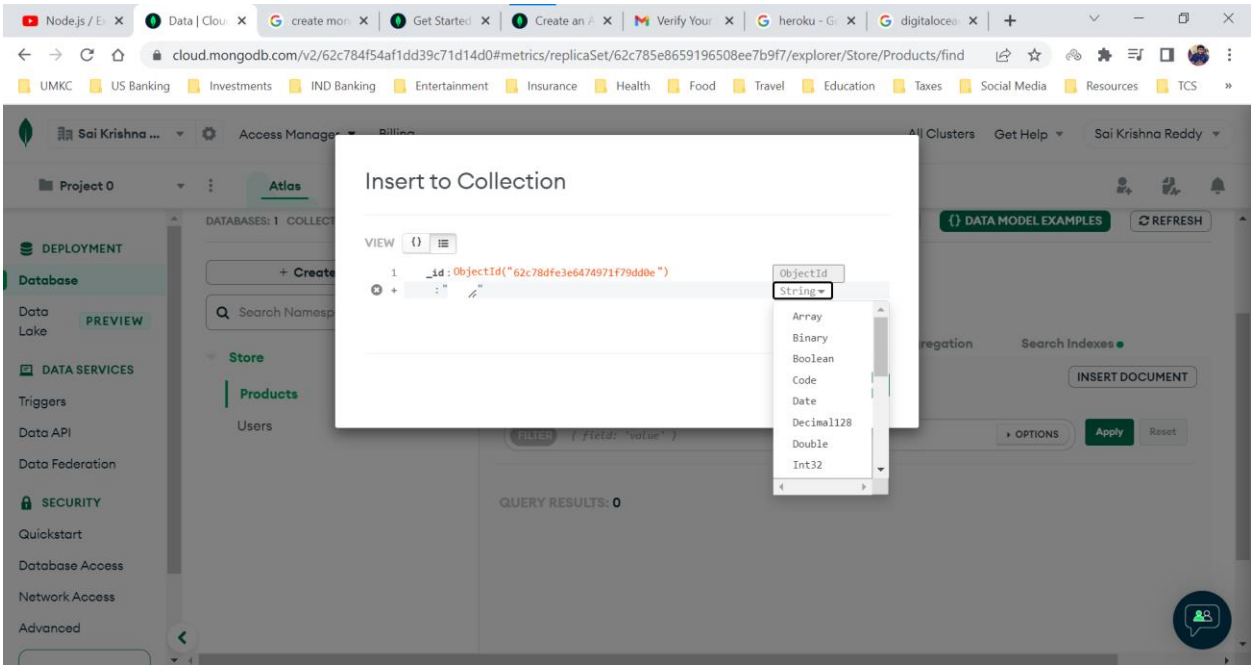
Now we are using the Manual Process to get a gist about MongoDB but once we start developing from server, we do all the operations using a tool called ***mongoose***.

Let us insert our first document (a single item) into the Products collection.

Click on Insert Document

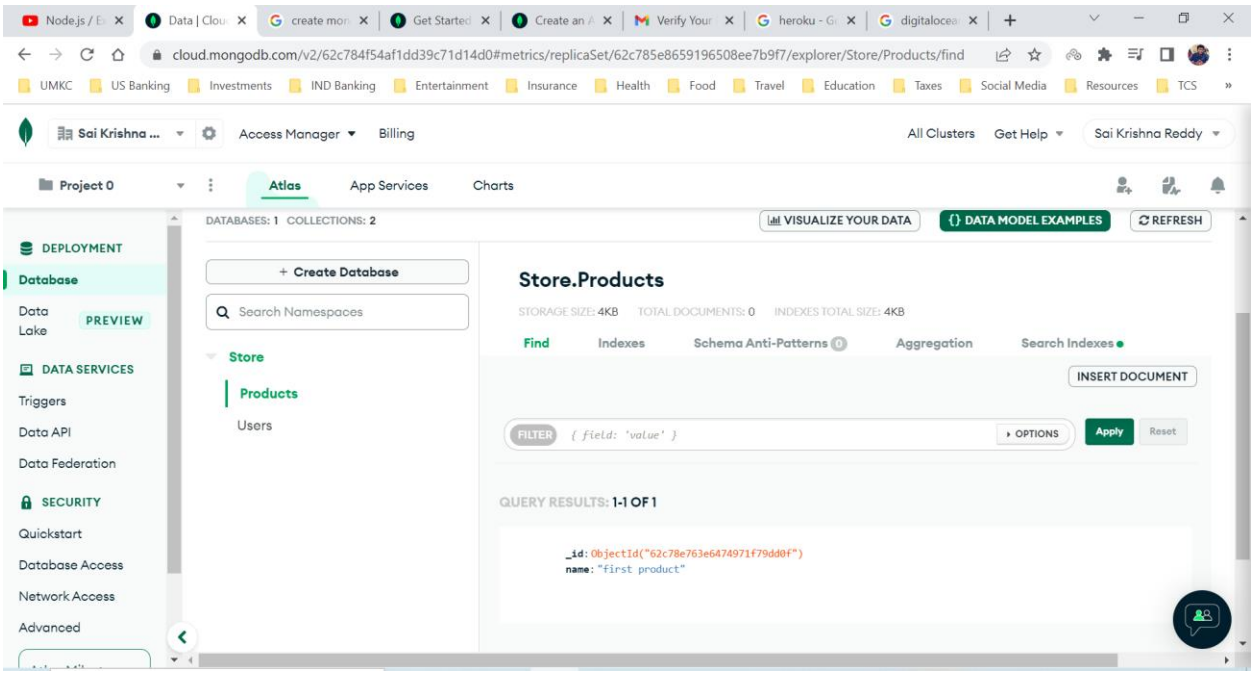


Here each document contains two details (key-value pairs), the first one being the id which is given by MongoDB by default.



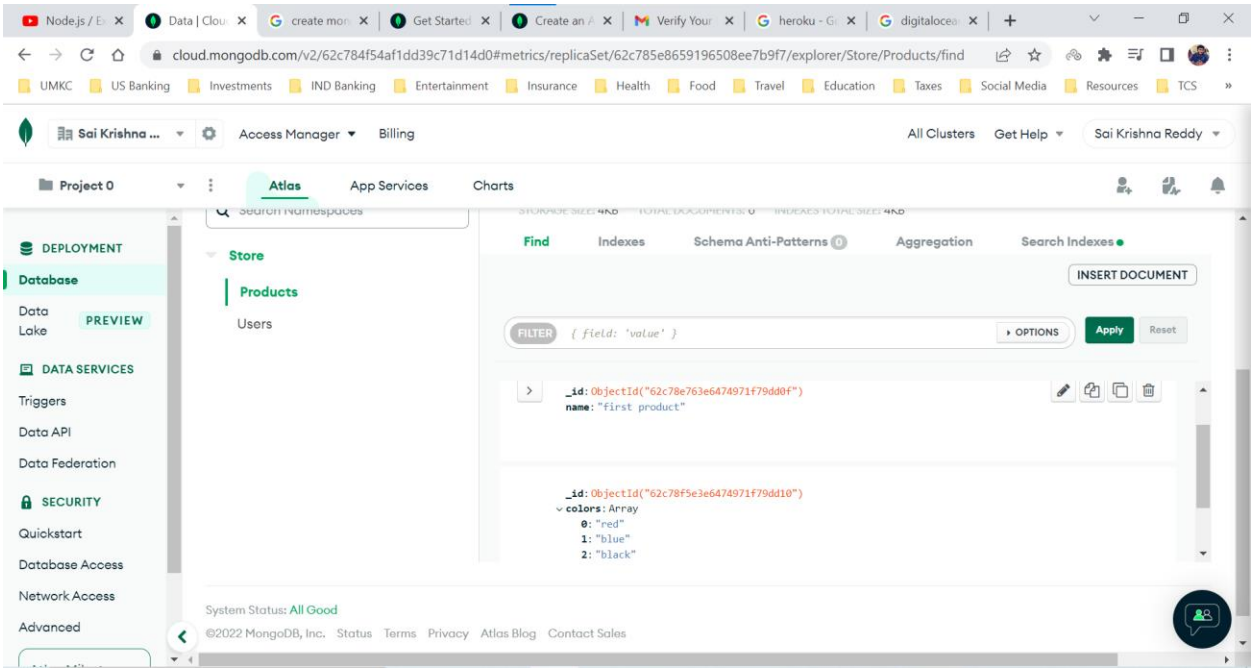
Now the second the property can be of any data type.

We have a created a document (a single item) with a key named name and value being first product.



In MongoDB, documents have a dynamic schema, which means that documents in same collection doesn't need to have same set of fields or the structure.

Since MongoDB has a dynamic schema, we are able to create two documents with different schema for each document (equivalent to row in SQL).



Just because MongoDB is allowing us to do that doesn't mean that we should create different schema for each document in a collection.

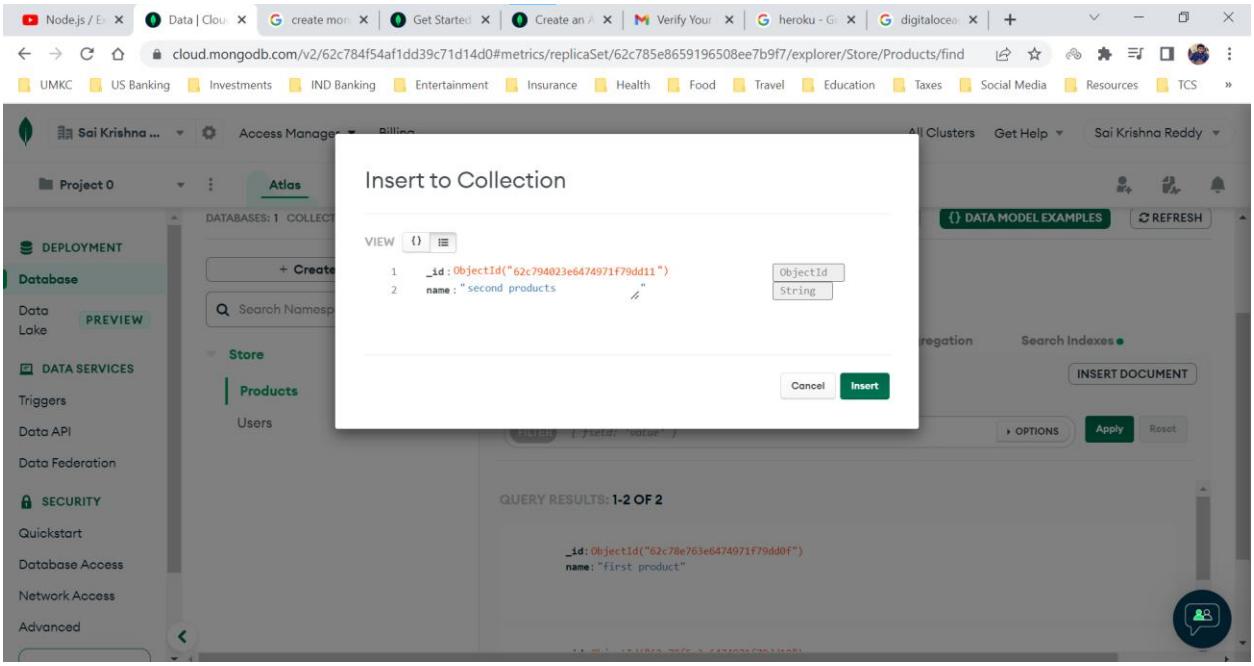
We will be using additional library named Mongoose which will setup that structure for us.

CRUD Operations using our Manual Setup

CRUD stands for Create, Read, Update, Delete

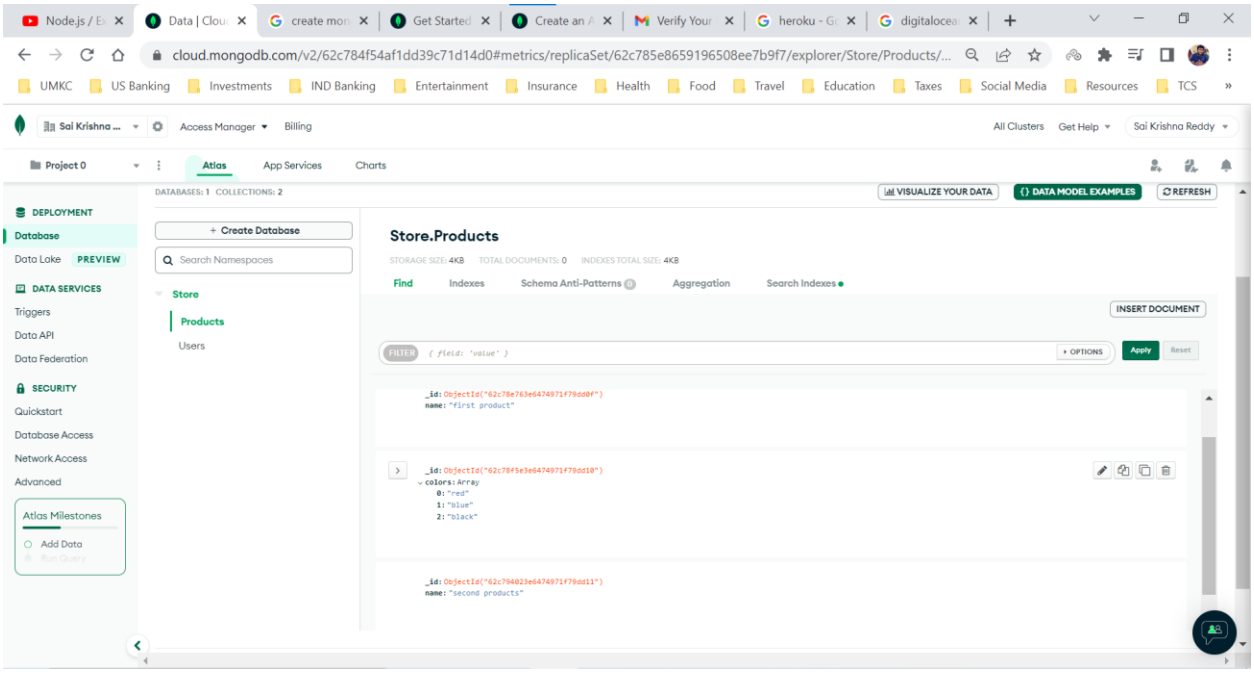
Create

Inserting a Document



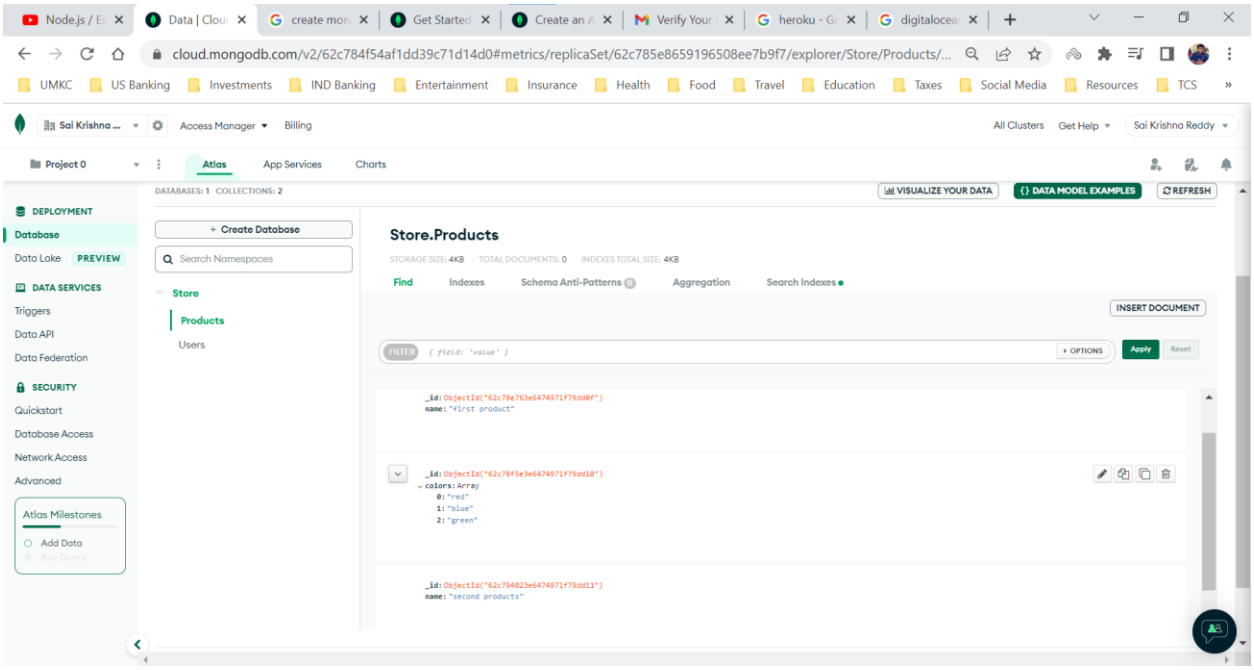
Read

Reading all the documents in the Products collection



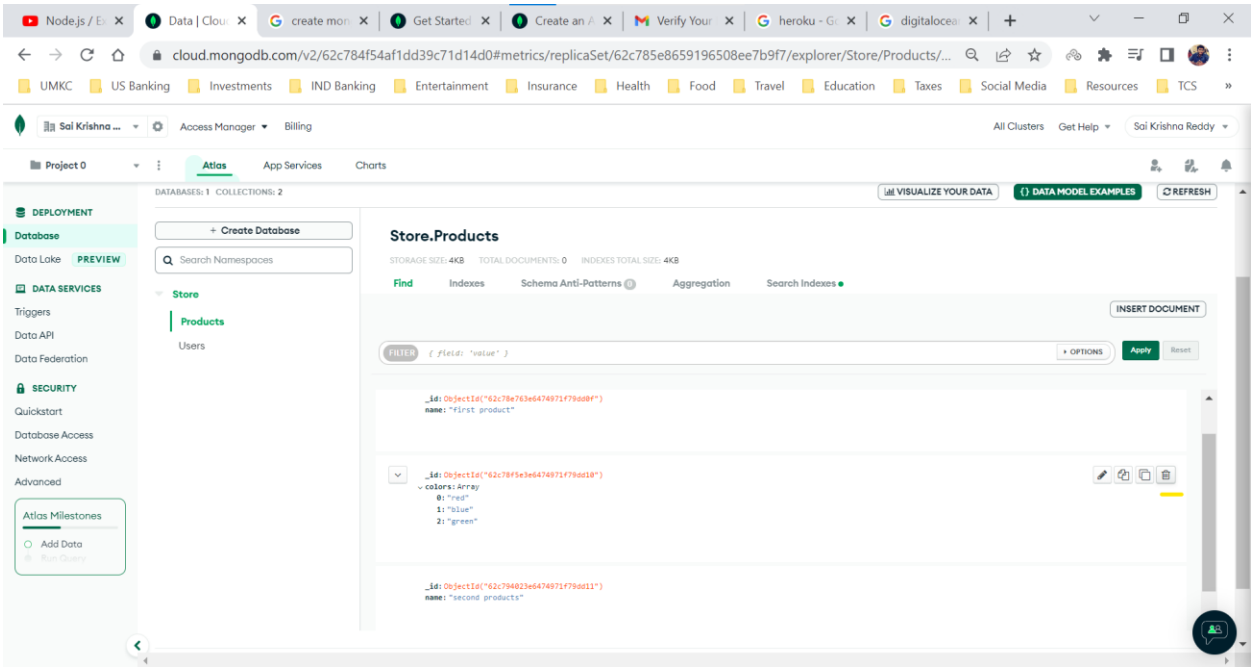
Update

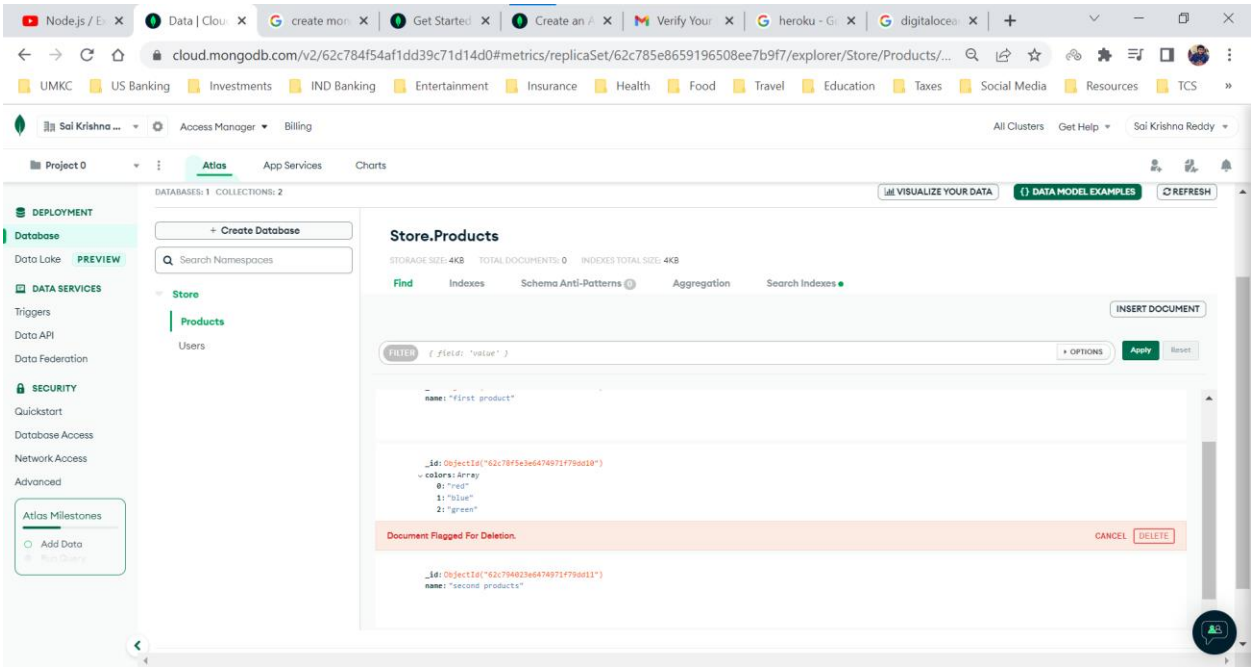
Updating a document in collection. (Updated a item in the color array from black to green)



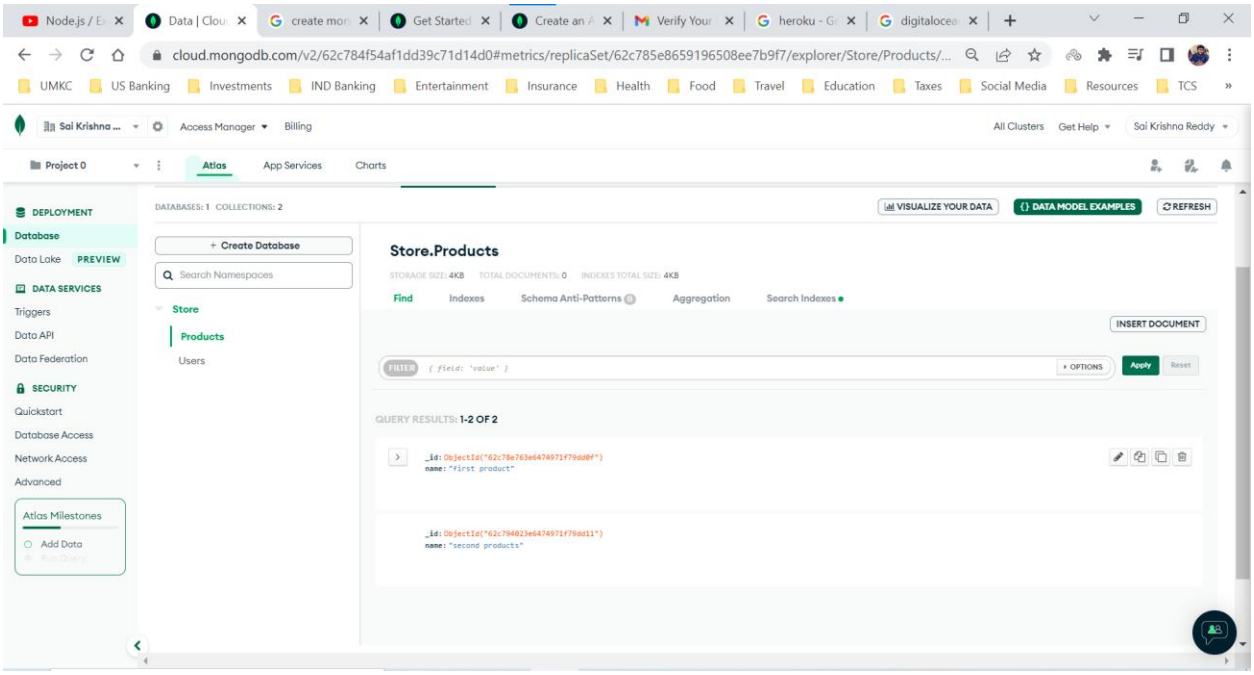
Delete

Delete a document from the collection (click on the delete icon to delete the document)





After deleting the documents



The above CRUD operations are done manually which is not the preferable way to perform the CRUD operations when working frontend and Node JS applications.

CRUD Operations using Mongoose

Once our database is ready, we need to connect to the database from our server. We can use the native MongoDB driver package name MongoDB, but a very popular alternative is to use by the package name ***Mongoose***.

Mongoose is object data modeling library and is popular because it comes with bunch of goodies which makes our development faster. We can see the use of goodies in our Node JS projects.

Yes, we can also setup our applications just with native MongoDB package, but the reason mongoose is extremely popular because it has extremely straight forward API and basically it does all the heavy lifting for us.

Command to install Mongoose

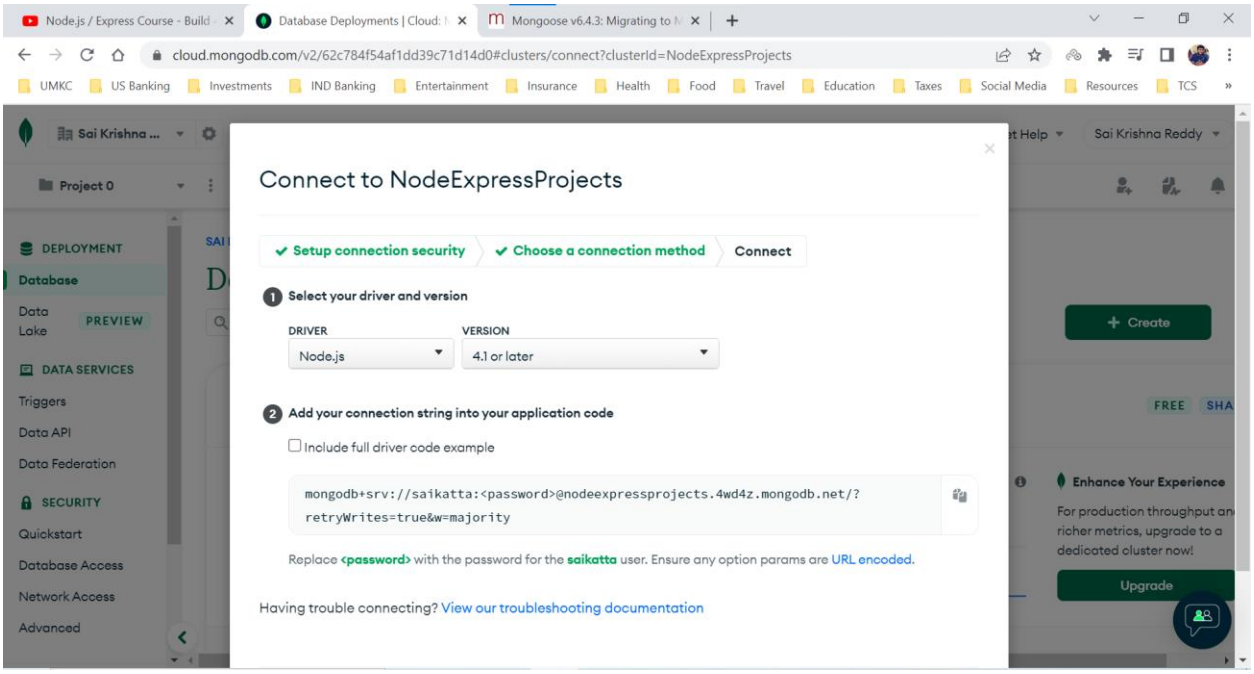
npm install mongoose

Using Database on Server

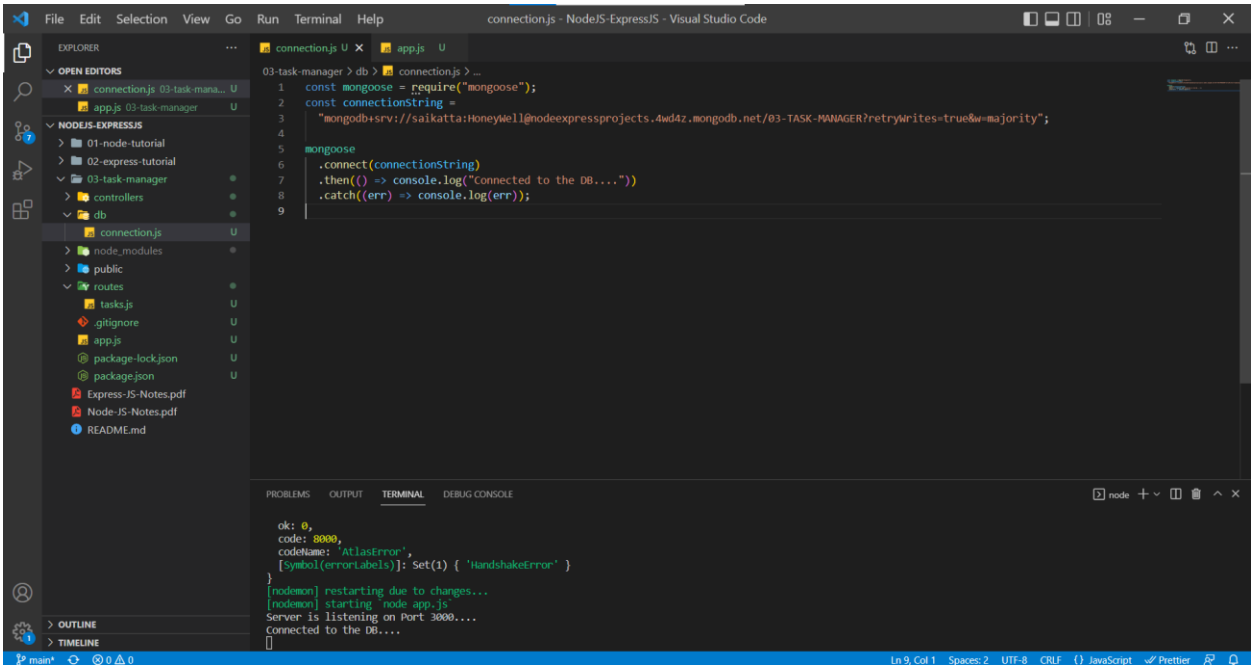
First thing we need to do is to setup a connection. To setup the connection with database we are using mongoose library.

Mongoose library is already installed in our code but since here we are using version 6 which has few changes when compared to version 5 (which was used by the instructor in the video).

To setup the connection we need to get the connection string from the MongoDB. We need to use it as a connection string in our code.



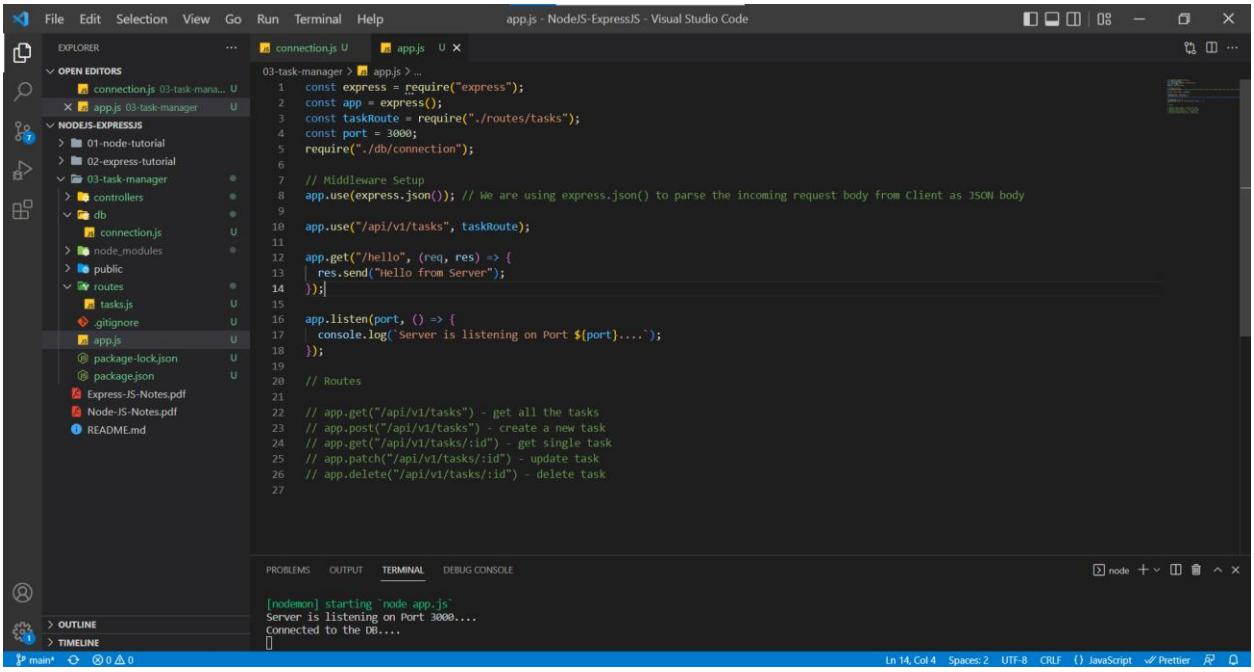
connection.js file in db folder



We need to add our database name in between the / and ?. I have added the database name as 03-TASK-MANAGER. connect() method in mongoose library returns a promise hence we try to catch it using the then() and catch() methods.

As we have learned in the basics of Node JS, when we import some files into app.js we are going to those imported files. We want to start the database connection when we want to start our server. Hence, we are importing the connection.js file and executing it in the app.js file.

app.js file



We have completed the database connection setup but here we do have two problems

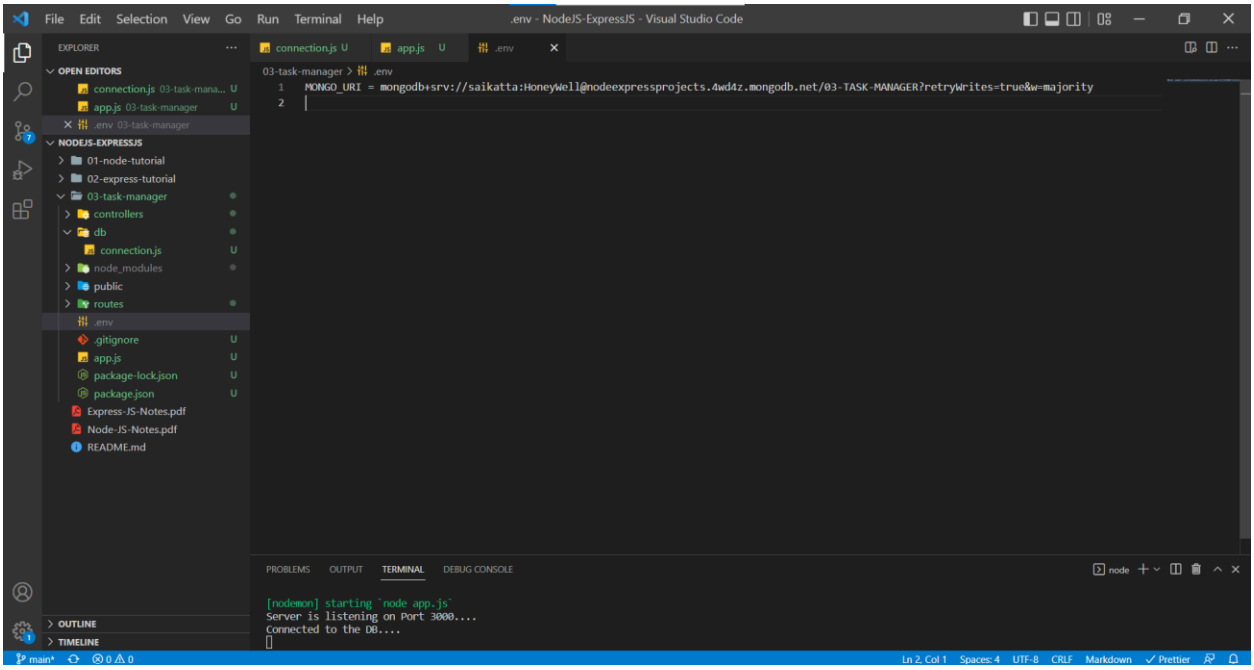
1. Exposing our database credentials to others when we try to push our code to GitHub.
2. We are starting the server and then checking the Database connection but ideally, we should check the database connection and then start our server.

The first problem can be solved by using a file containing our important details about the project. To do that we use a library named ***dotenv***.

Dotenv is a zero-dependency module that loads environment variables from a .env file into process.env. Storing configuration in the environment separate from code is based on The Twelve-Factor App methodology.

Let us create a file named .env for the whole project, so that we can those secret codes throughout the project. We created the file and added the required variable.

.env file

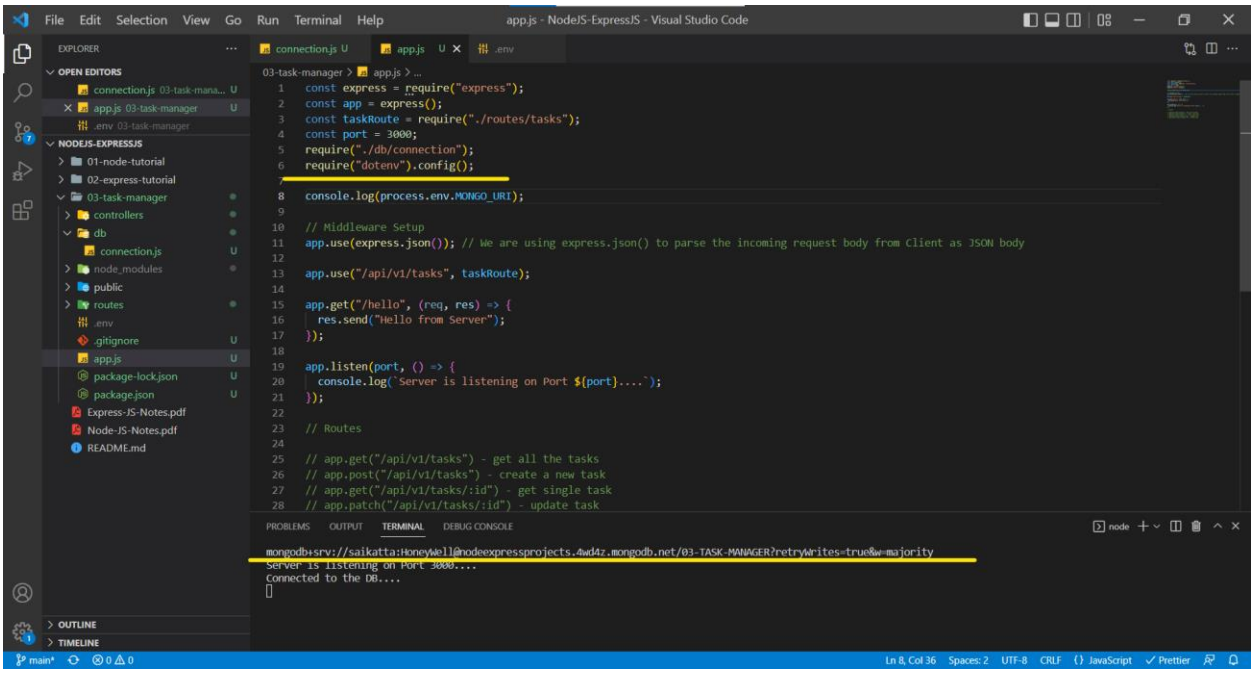


Don’t forget to add .env extension in .gitignore file

Now we need to get access to the .env file in our app.js file

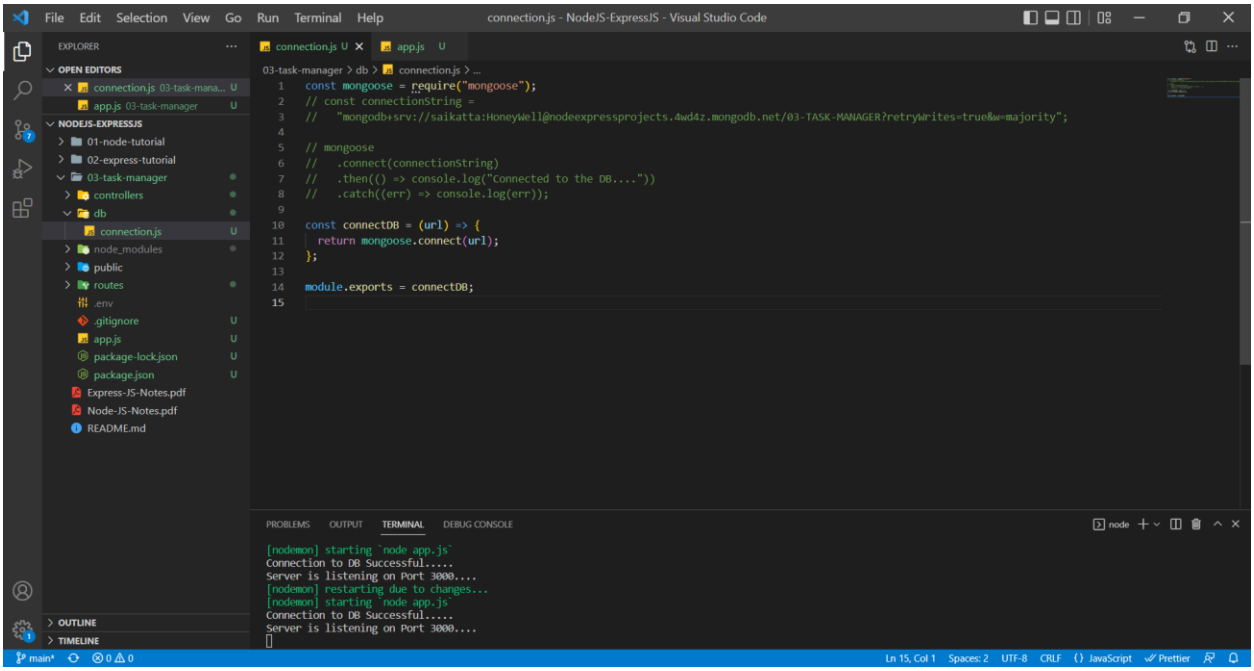
To get access, first we need to import the dotenv library into app.js and then configure it using ***config()*** and then we can use the global variable ***process*** to get hold of the ***env*** and variable where we stored the value.

app.js

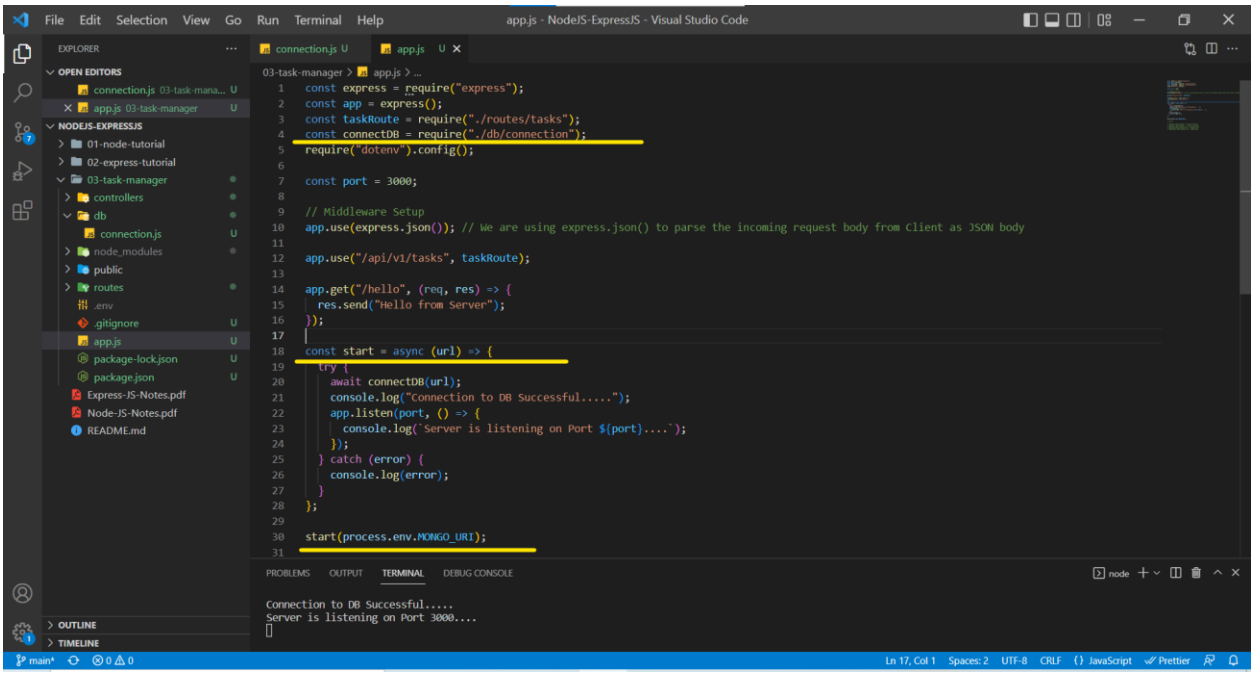


The second problem can be fixed by writing a logic in the app.js where we need to check the database connection and then start the server also, we will make the code a bit cleaner.

connection.js



app.js



Now, let us set a structure for our future documents and assign them to collections. We are going to do that using schema and model from mongoose.

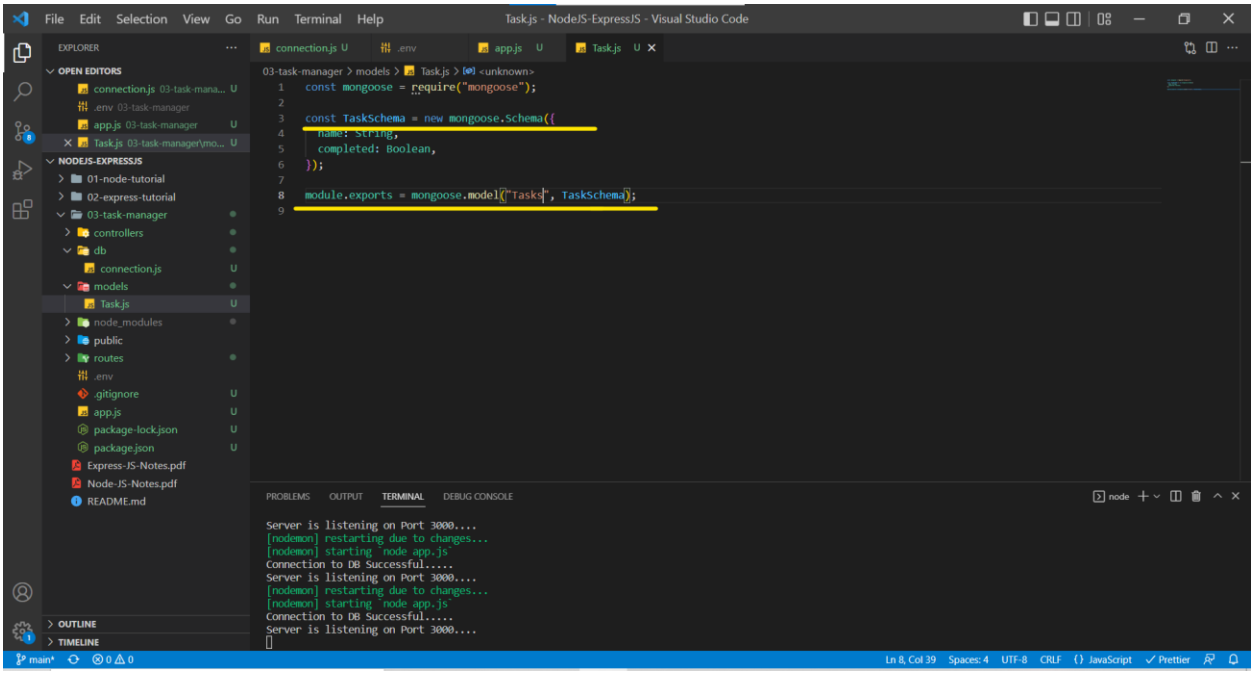
We will create a folder named models and inside of models we create a file named Task.js. With the help of mongoose schema, we are going to set up the structure of the documents in a collection.

In schema we define the name of the key and Schema Type the key is going to store. Once we have the schema/structure for the data now we want to set up the model.

Model is representation for collection. In Mongoose a model is a wrapper for the schema so if the schema defines the structure for the document, a mongoose model provides an interface to the database. Using the model, we will be able to CREATE, READ, UPDATE and DELETE our documents with great ease.

While creating a model, the first argument we use is the singular name of the collection your model is for. Mongoose automatically looks for the plural, lowercase version of your model name.

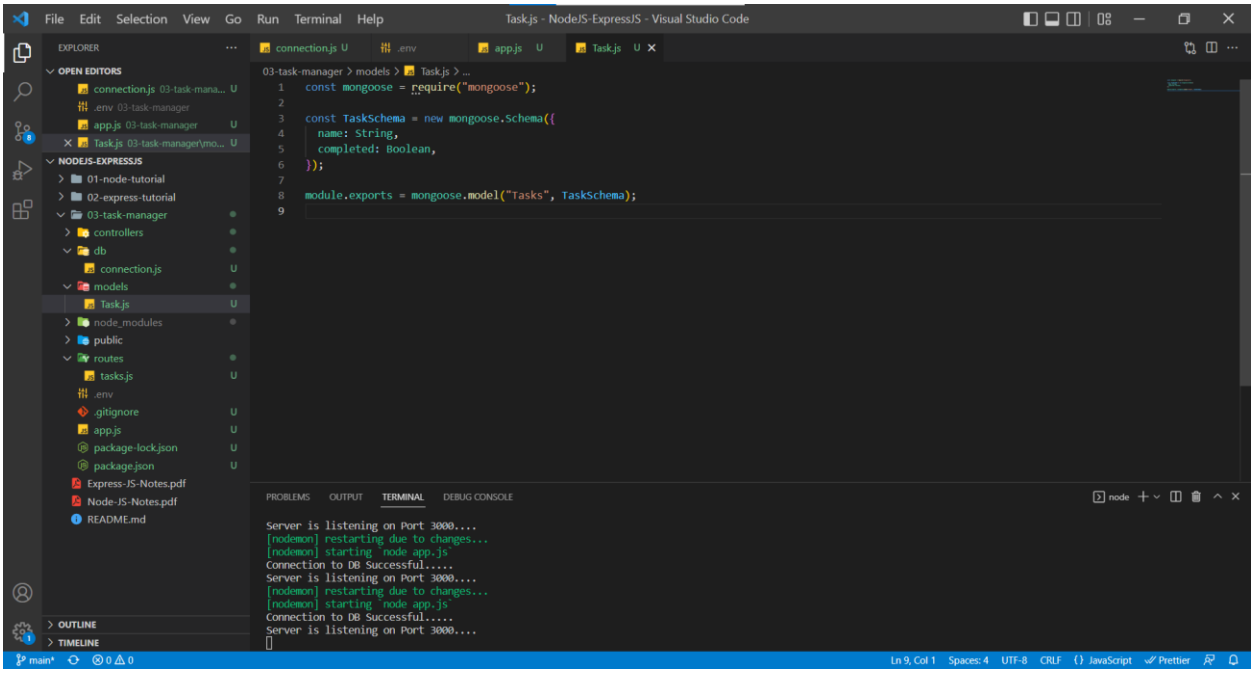
Task.js in models folder



Now we have created a model and we just need to use those models in our controllers to perform the CRUD Operations.

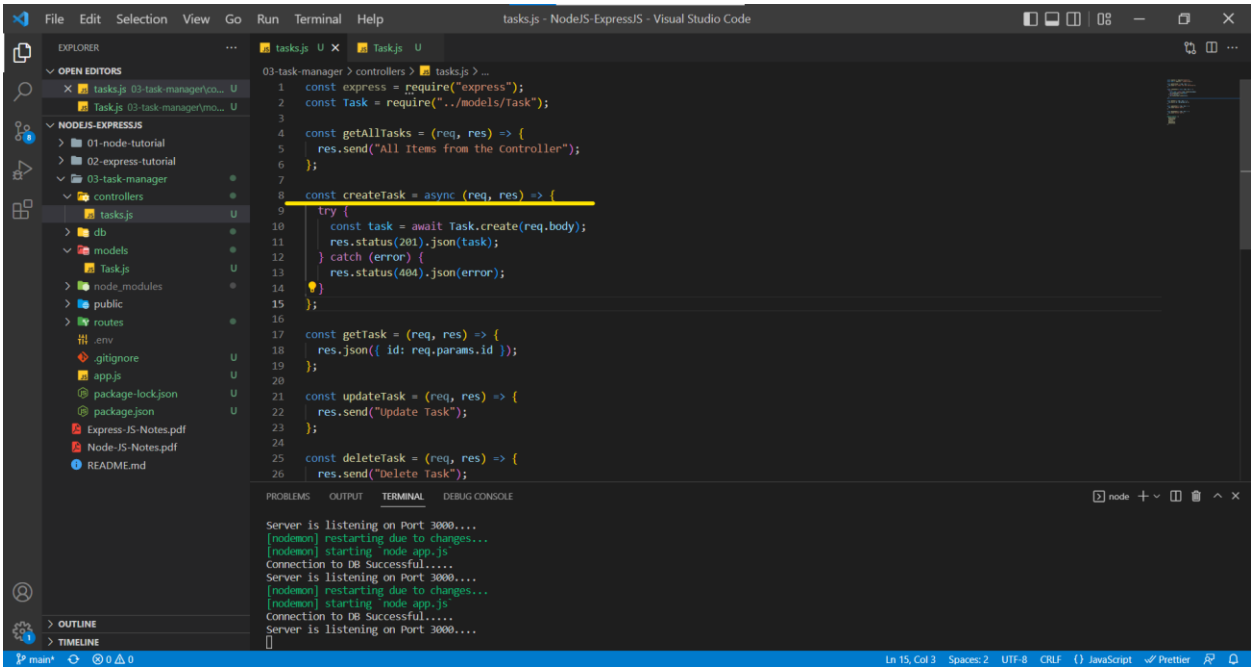
An Instance of a model is called document. Creating them and saving them to database is so easy.

Task.js form models folder.



A Schema also acts like a validator which ignores other properties that are being sent in the request and will only add the properties which are mentioned in the schema to the database.

app.js

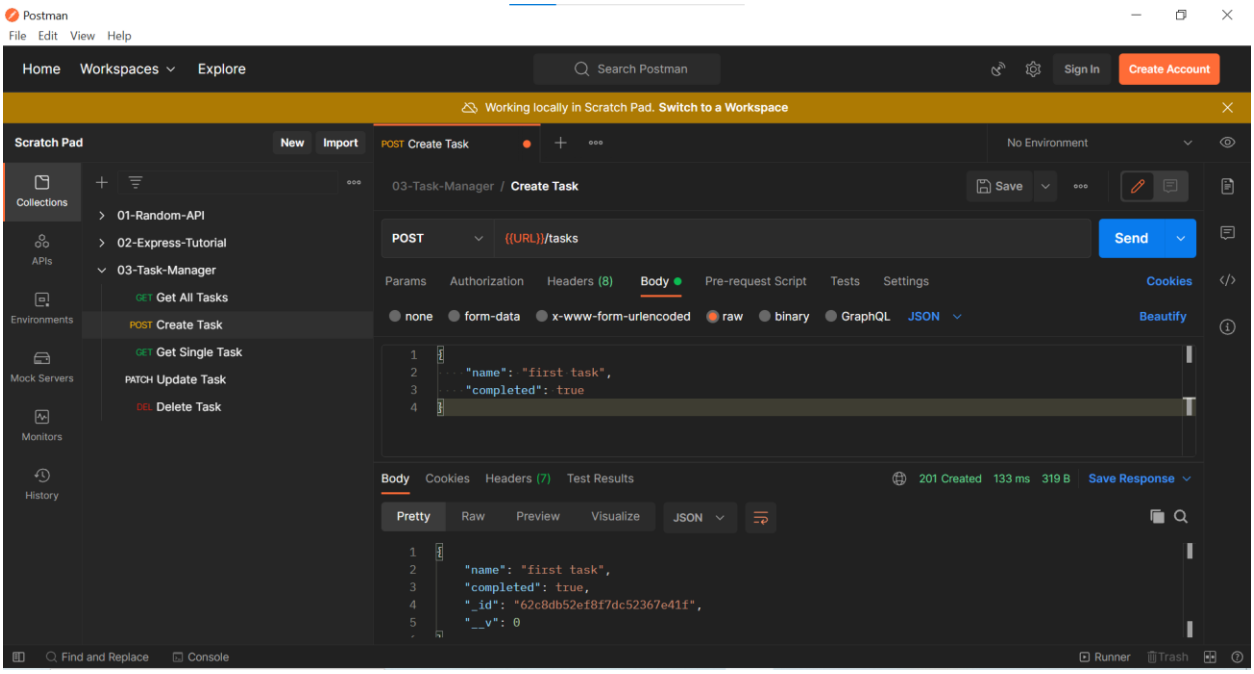


As we know that each instance of model is a document, we are using the **create()** method available in mongoose library to create a document and save it in the database. **create()** method in background calls the **save()** method to save the document in the database.

create() method returns a promise hence we are **async** and **await** to handle the promises. We are using the status 201 because it indicates the success using a POST method.

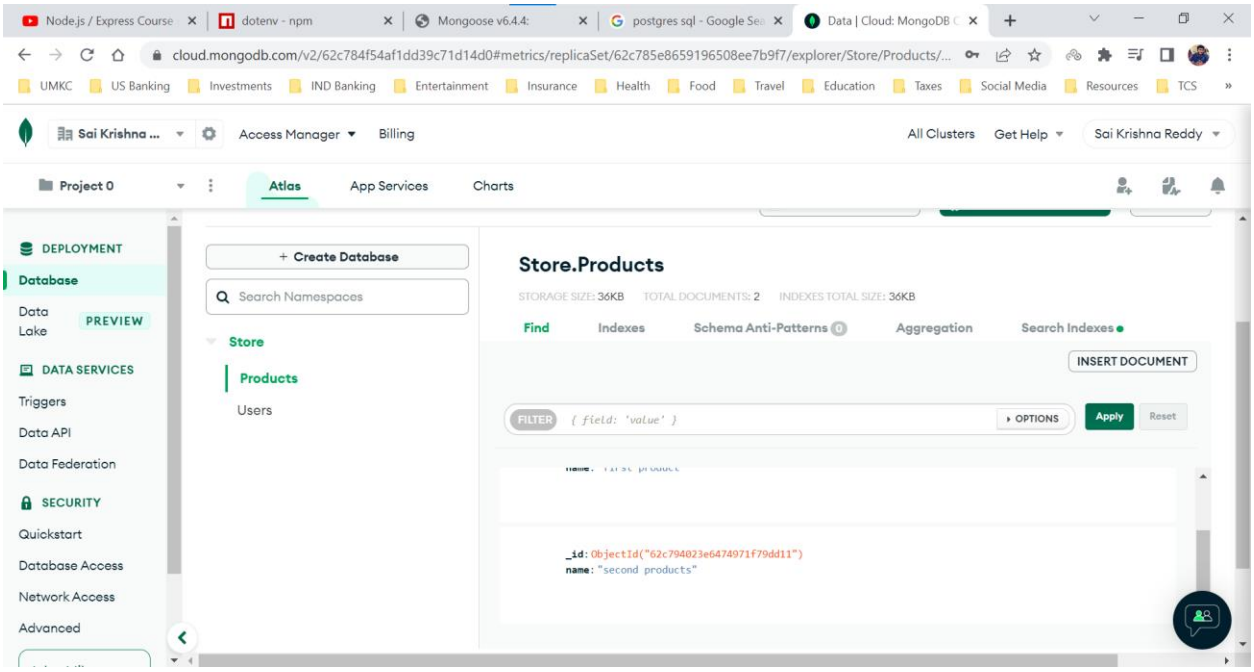
We are using the try catch blocks to handle if promise is rejected due to any database connectivity issue or any other issues.

POST Request from Postman

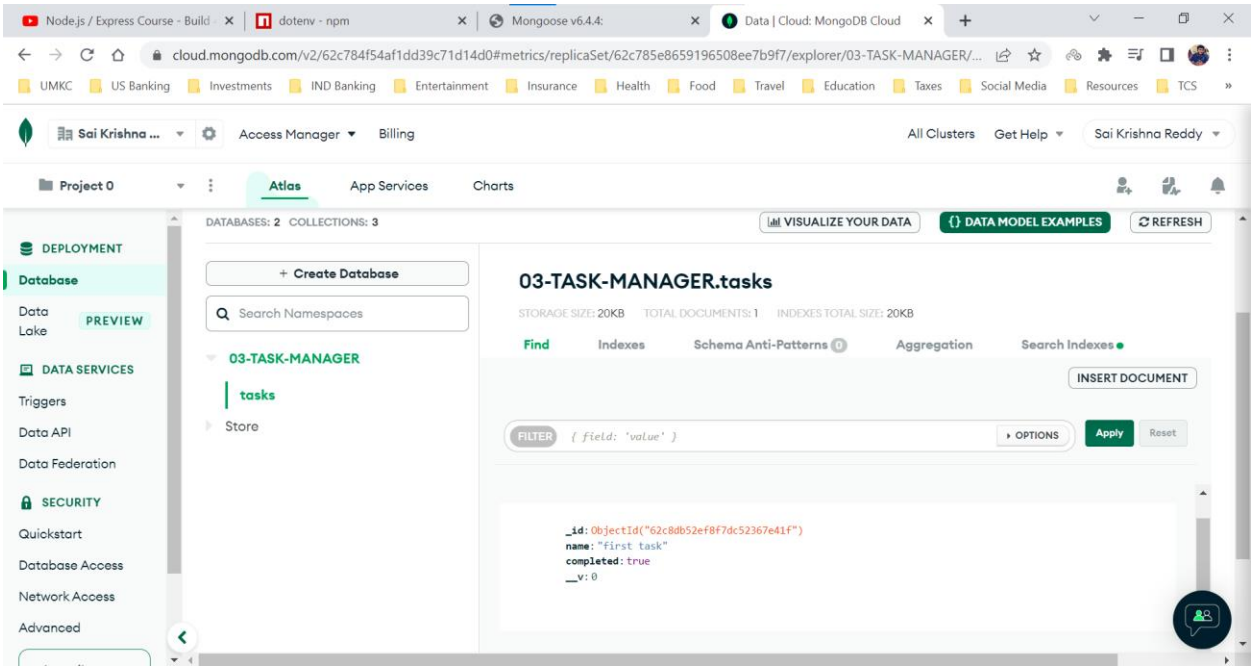


We received the status 201 as expected and a JSON object having data of the document which we inserted with two parameters **(_id and _ _v).** Those two parameters are added by MongoDB by default.

Before refreshing the screen in MongoDB

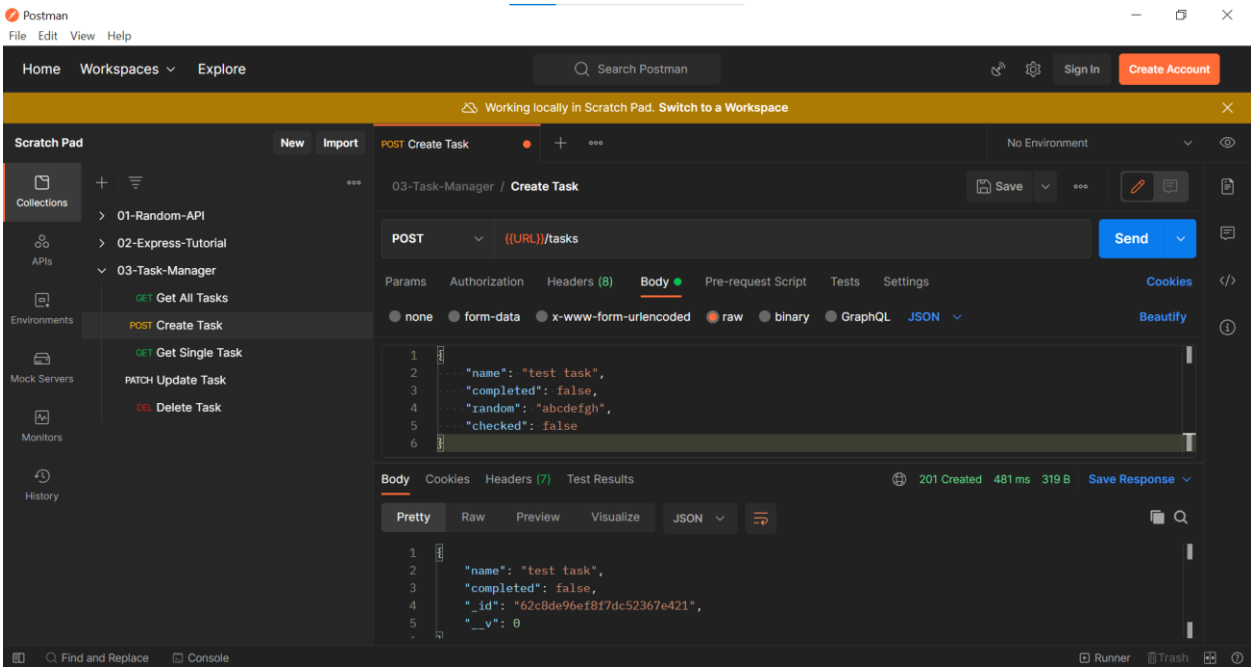


After refreshing the screen, we can see the database **03-TASK-MANAGER** being created and having collection **tasks** with one document.

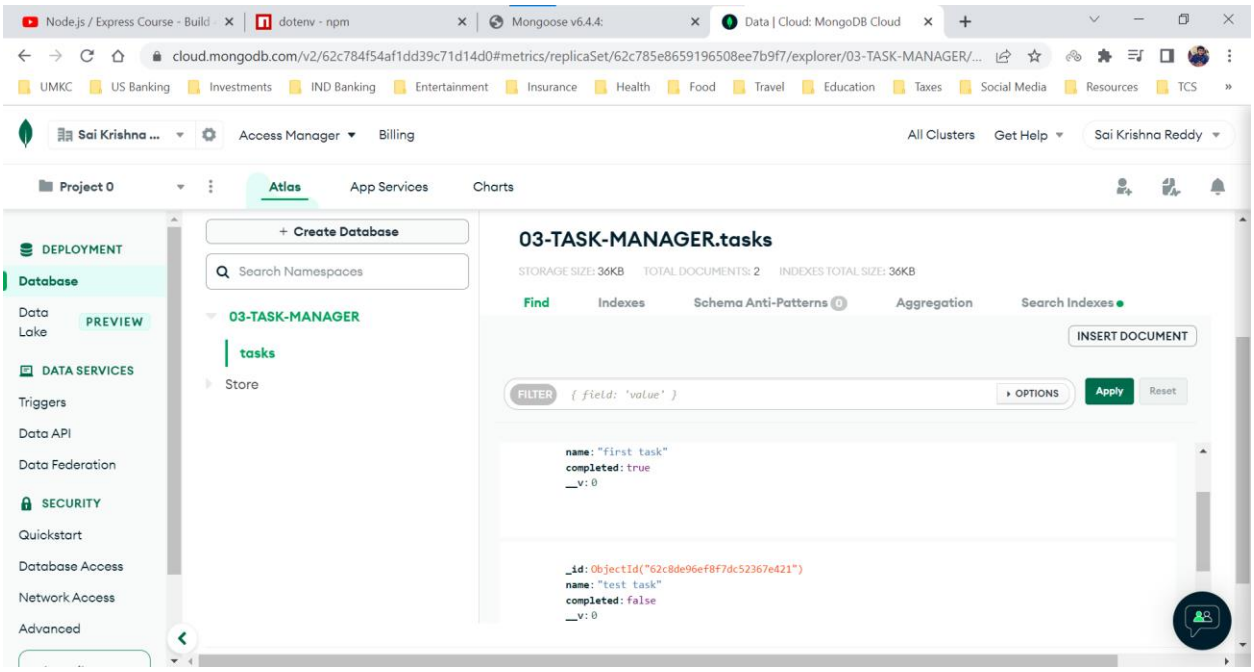


When we try to add additional parameters into the request body, Mongoose will strictly ignore the additional parameters and will only add the parameters mentioned in the Schema into the database. Mongoose performs validation using the Schema.

Additional Parameters in the Request Body in Postman



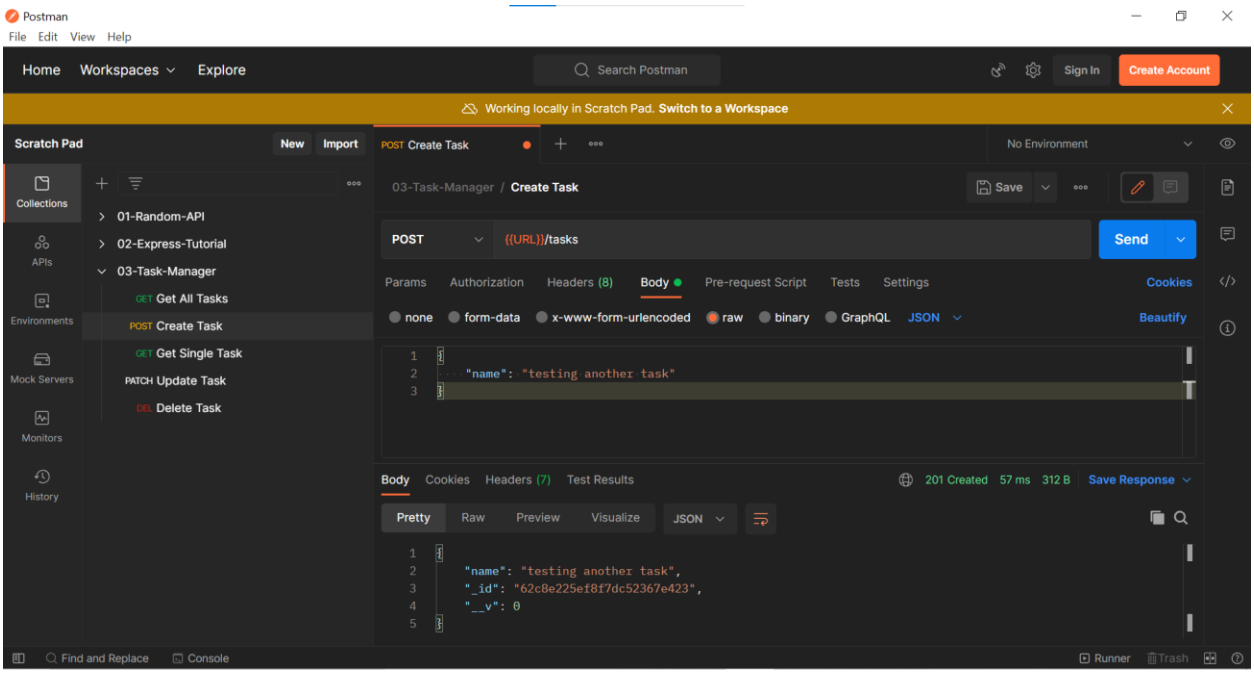
Document in the database after ignoring the additional parameters



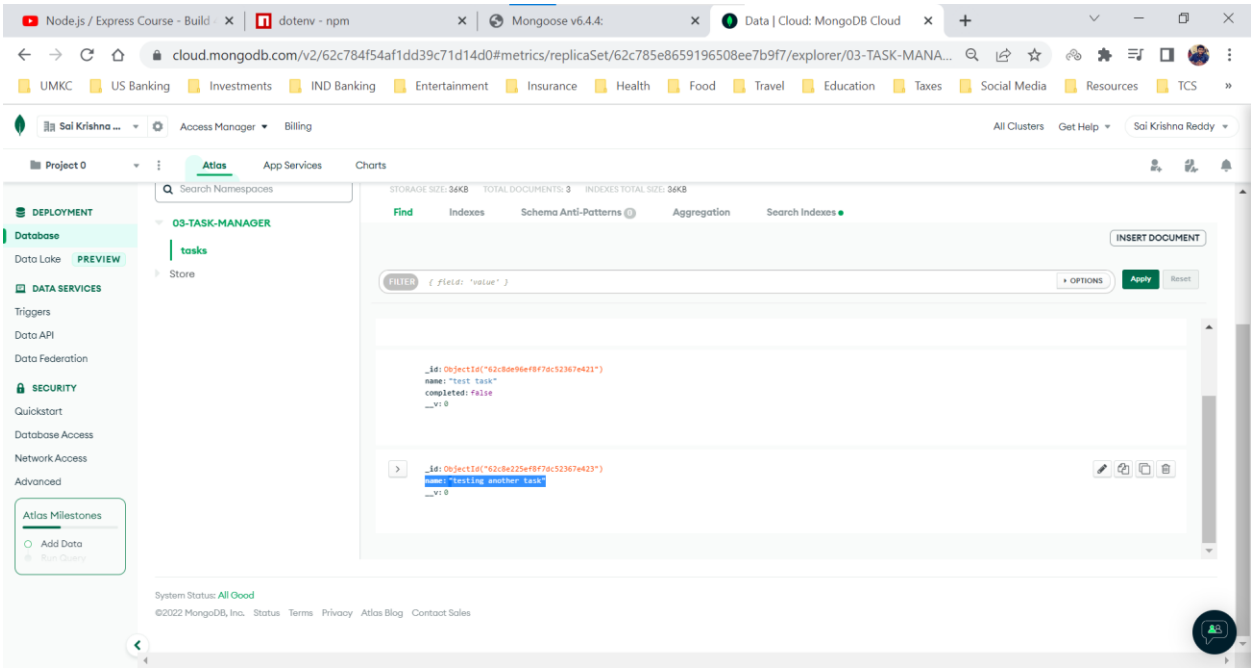
Even though Mongoose acts like a validator for ignoring additional parameters but it doesn't handle when we send an empty object or if we don't send the parameters mentioned in the schema.

To handle the validations, we provide validators for each key in the schema.

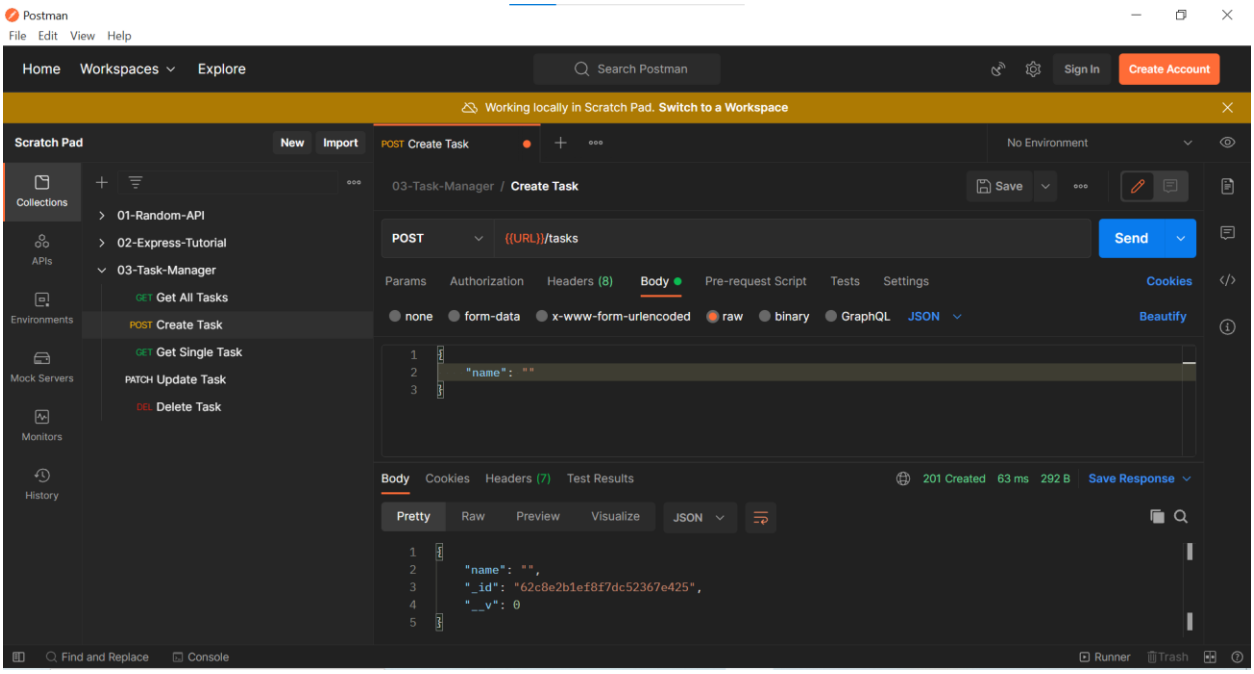
Let us see when we don't send parameters present in the Schema



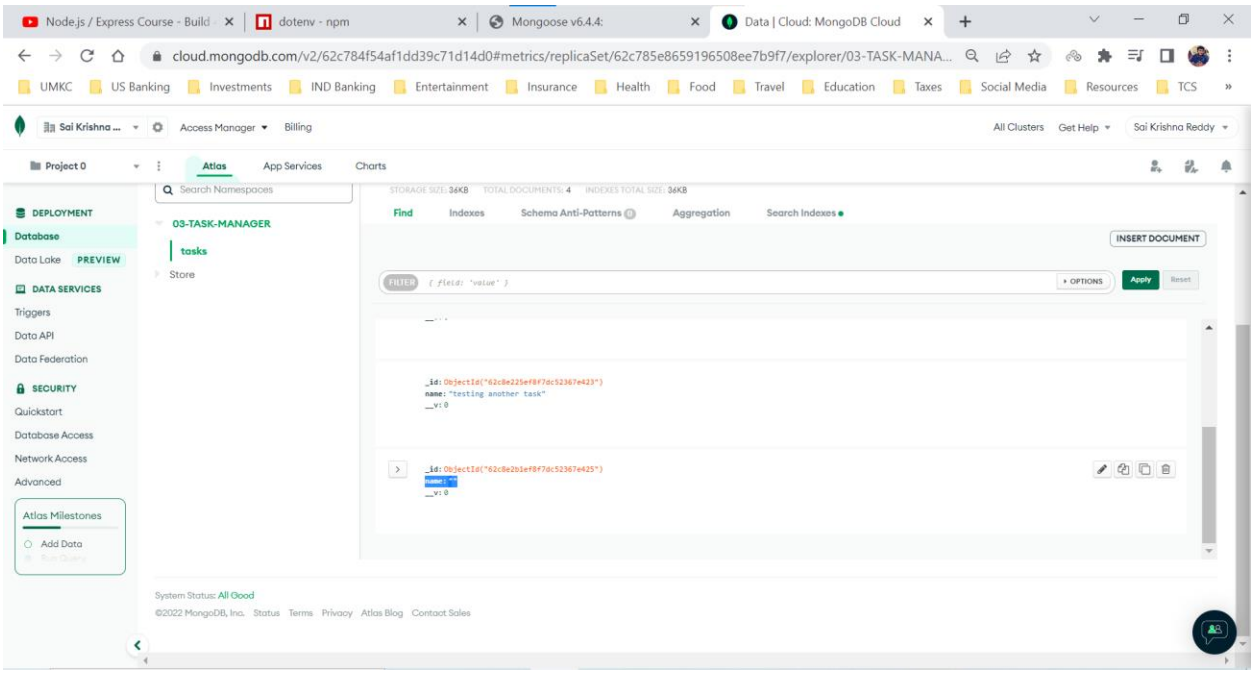
Document in created in the collection only with name property.



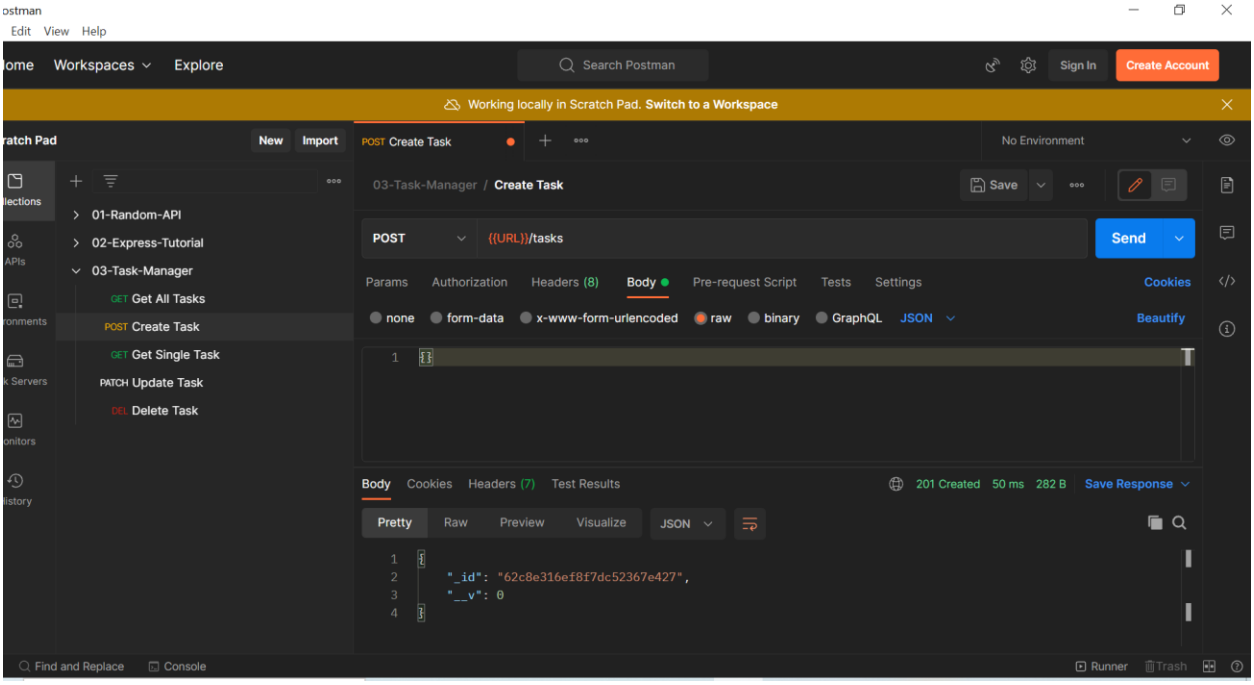
Let us see what happens when we send an empty value for key **name.**



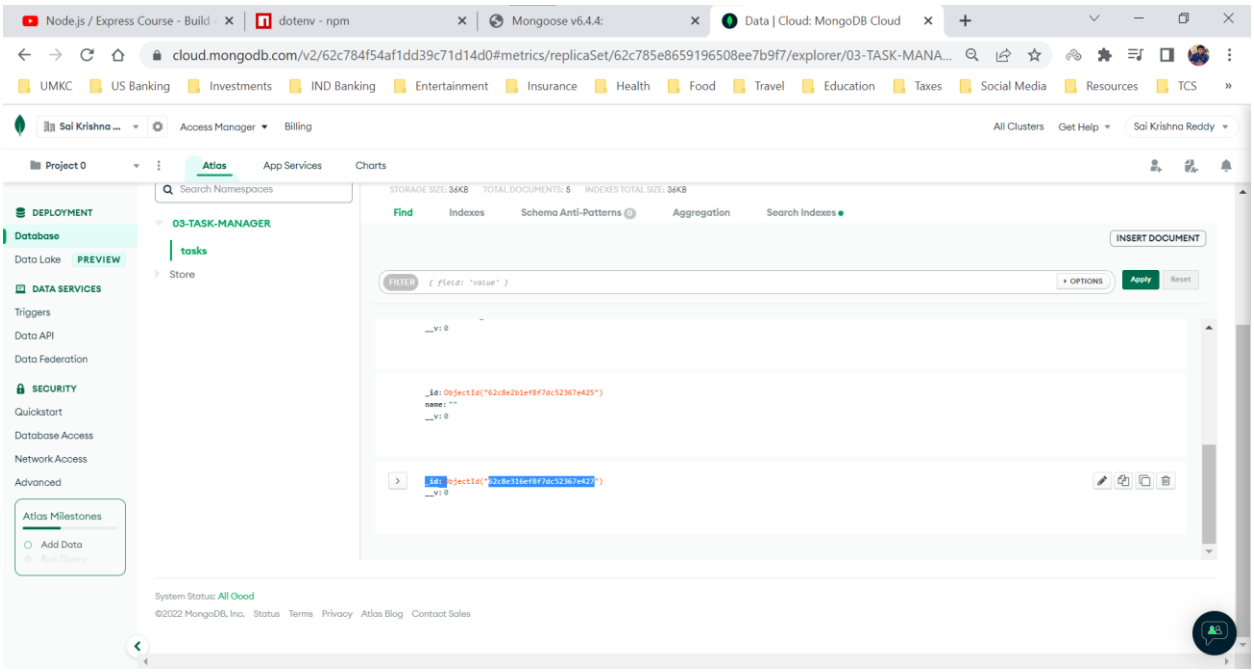
Document with name property is created but it contains an empty value



Let us see what happens when we send an empty object

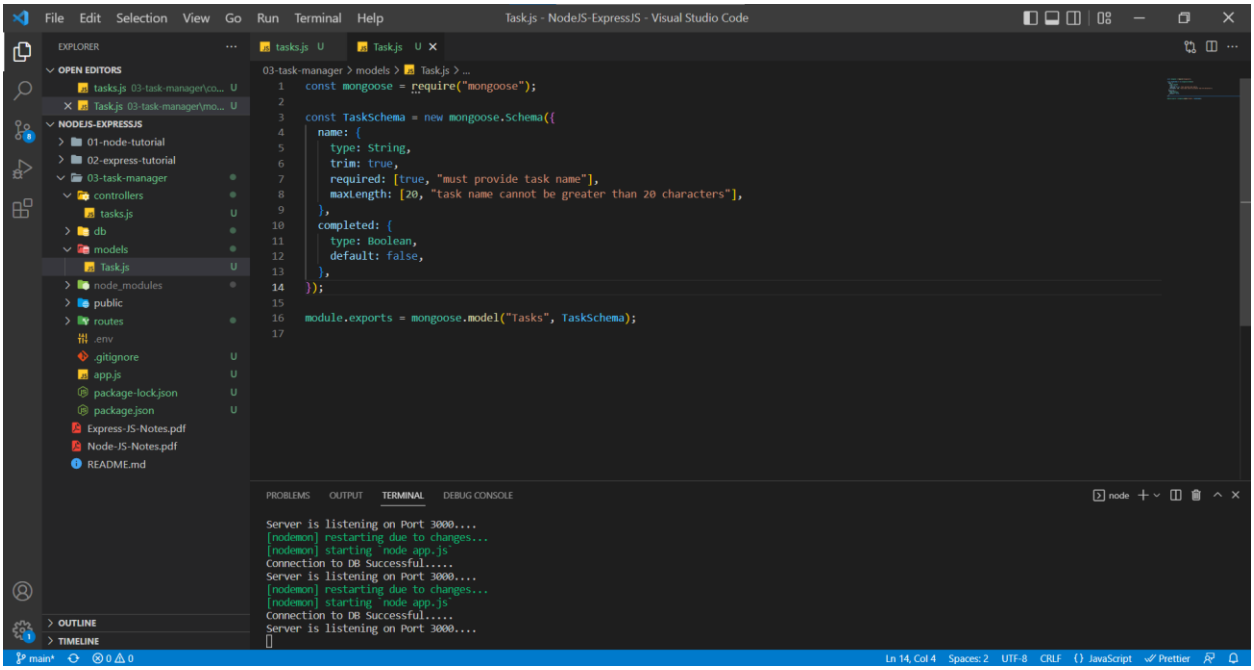


Document is created with no name and completed properties



To handle the above all issues we provide validators in the Schema

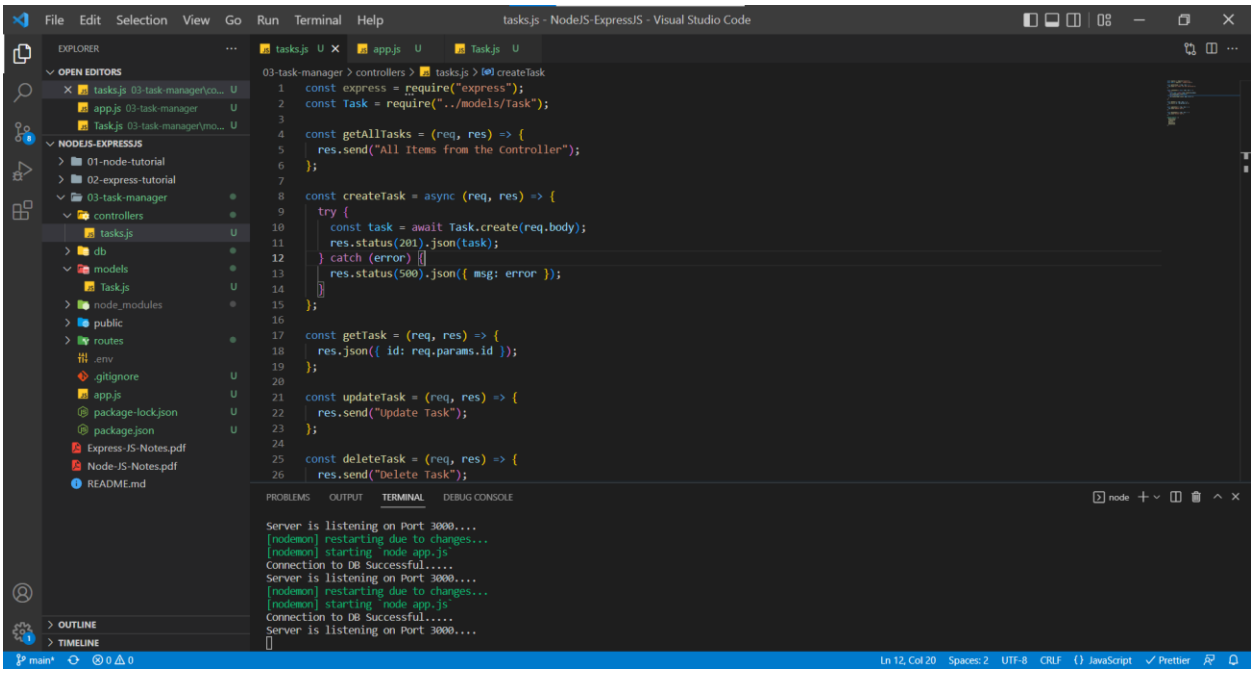
Task.js in models folder



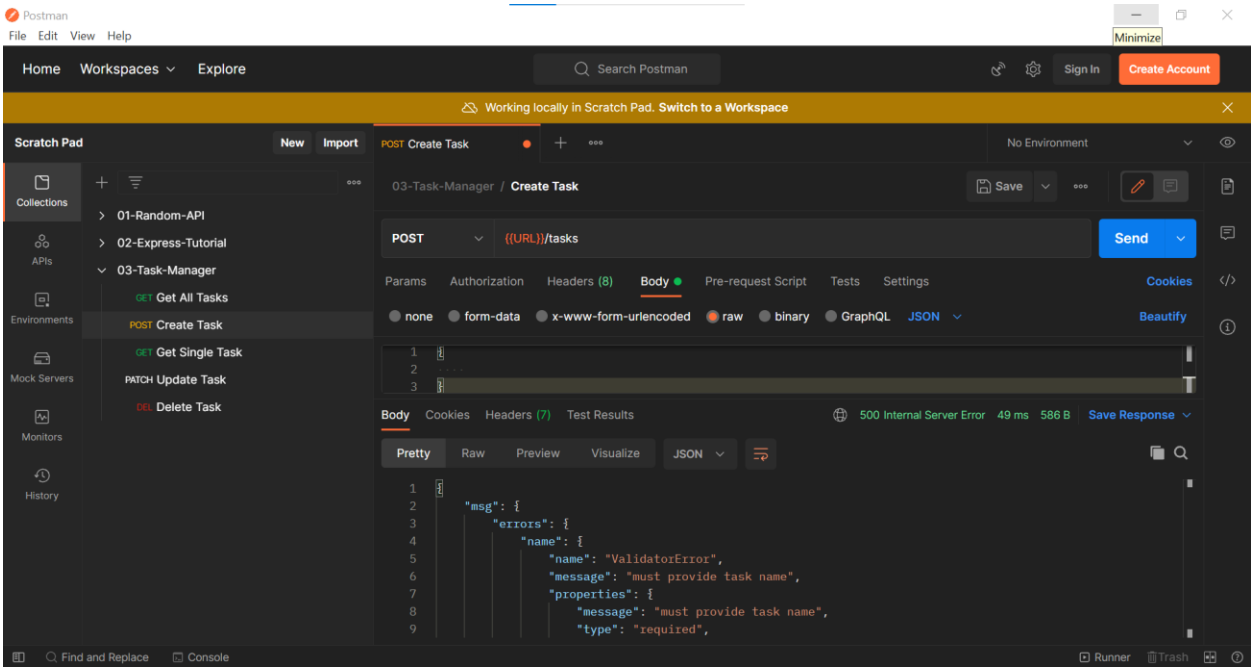
Now let's see how the validators work

For any custom message to be sent as a response, we need to add array with first index being the datatype and the second index being the custom message. We can see in the above example for required and maxLength.

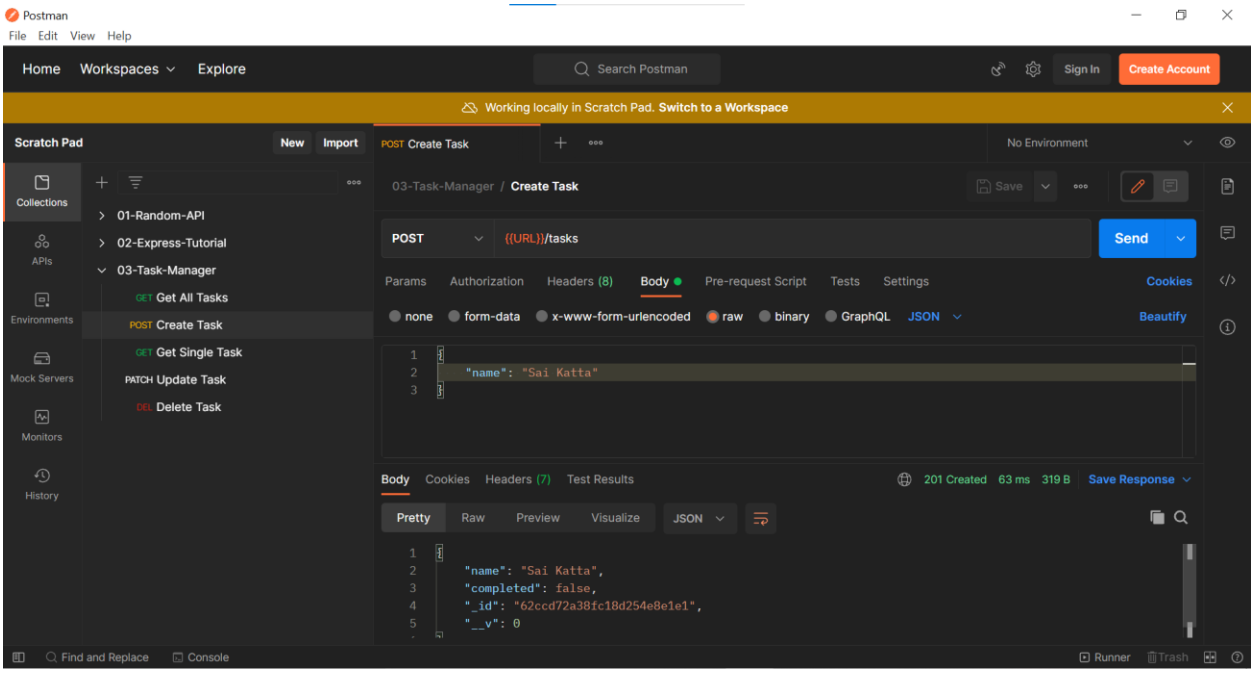
app.js



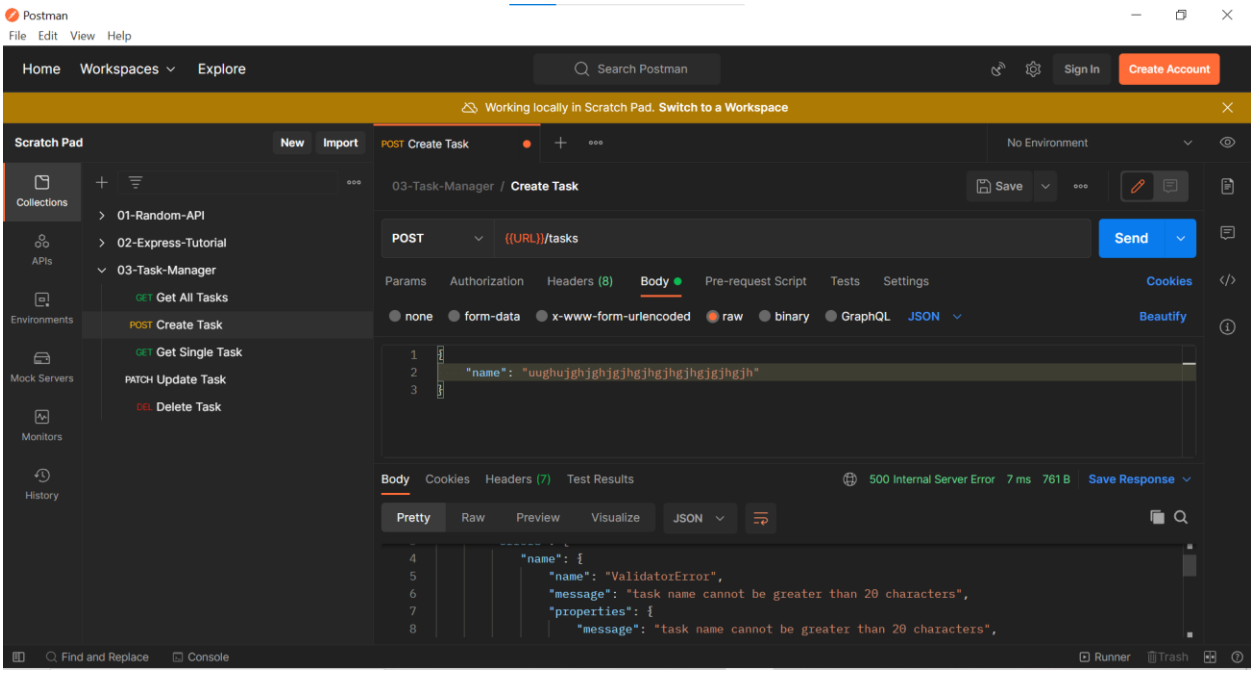
When we send an empty object



When we send only name, it is created because the second parameter completed is added as false by default in the validators in Task.js file



When we send name as more than 20 characters



Note:

Generally, when we send a request from Client, we will send the data in the form of JSON object but in the server side when we want to do any operation we cannot work with JSON object, hence we want to convert it from JSON object to JavaScript Object. We can do the conversion by using the Express Middleware function known as express.json() method. After conversion from JSON object to JavaScript object we do perform some operations on the JavaScript object.

Now, we have performed some operations on the JavaScript object, and we want to send it back to the Client. If data must be passed through the client, we need to convert the JavaScript object into JSON object. We can convert the JavaScript object to JSON object by using the express method in response object named json() method.

Ex: res.status(200).json({name: "Sai"});

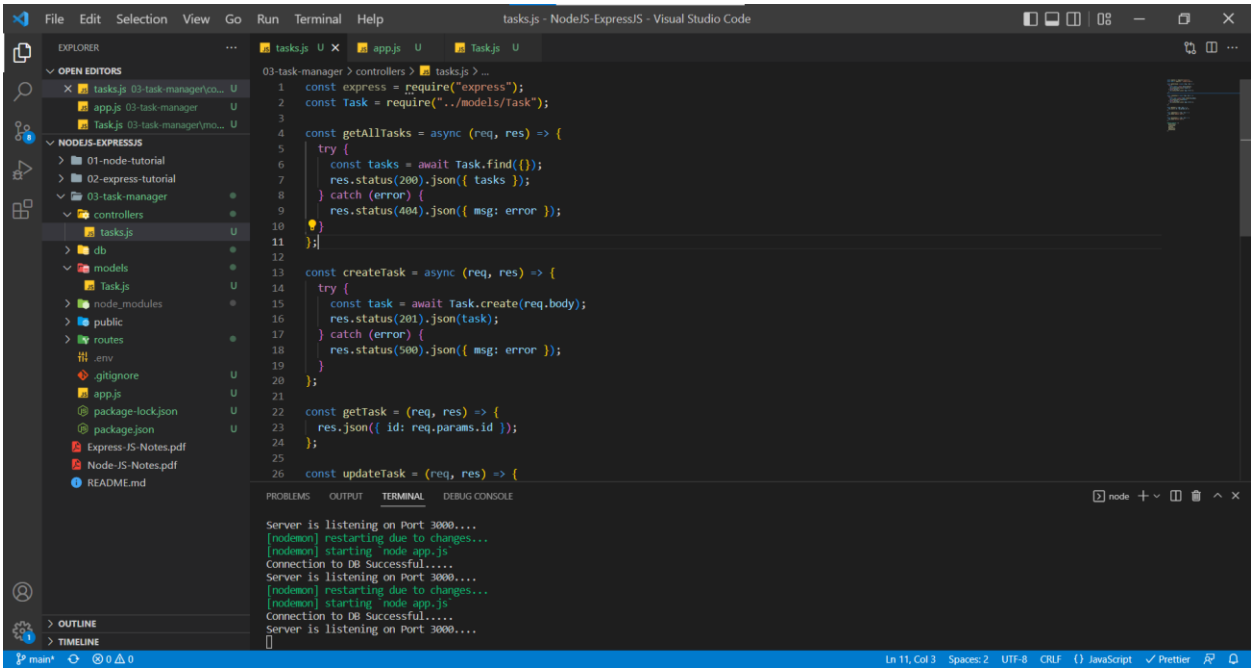
res.status(400).json({error});

Even Database sends a JavaScript object to the server. It's the server work to send an JSON object to the client and once the client receives the JSON object, the frontend engineer or developer converts the JSON object to JavaScript object and works on the data.

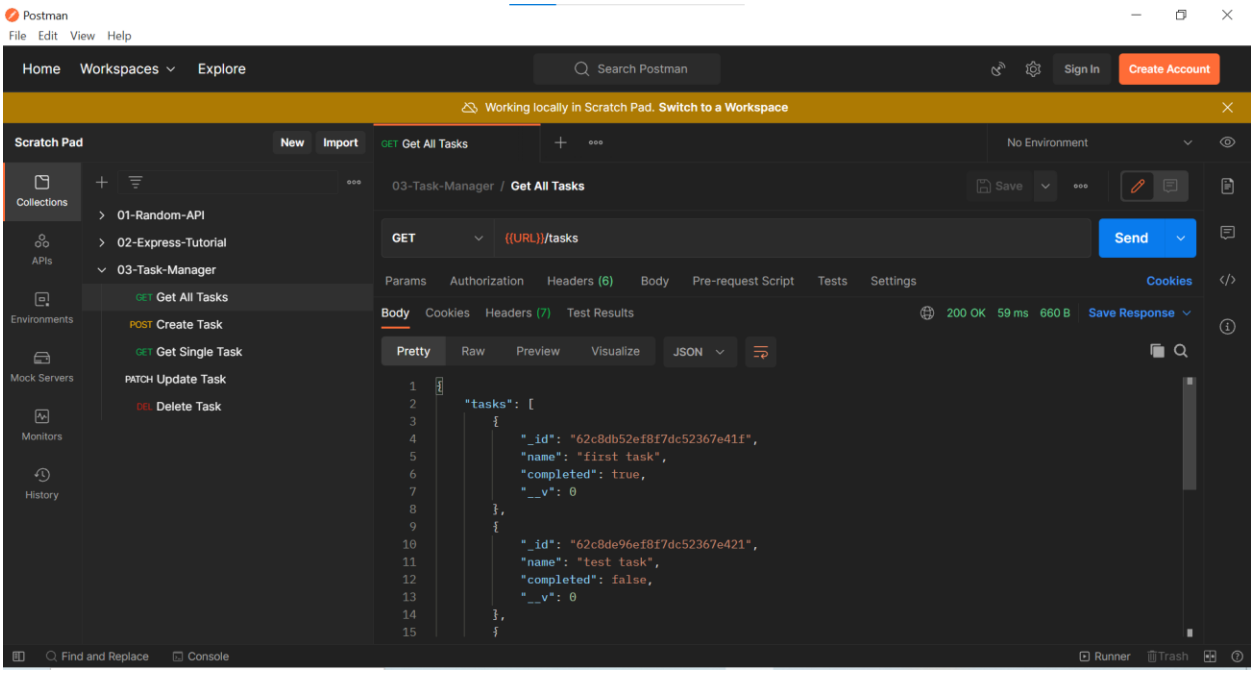
Till now we worked on creating a task, but now we will work to retrieve data from the database and send it to the client.

Mongoose models provide several static helper functions for CRUD operations. Each of these functions returns a mongoose Query object.

Get All Tasks



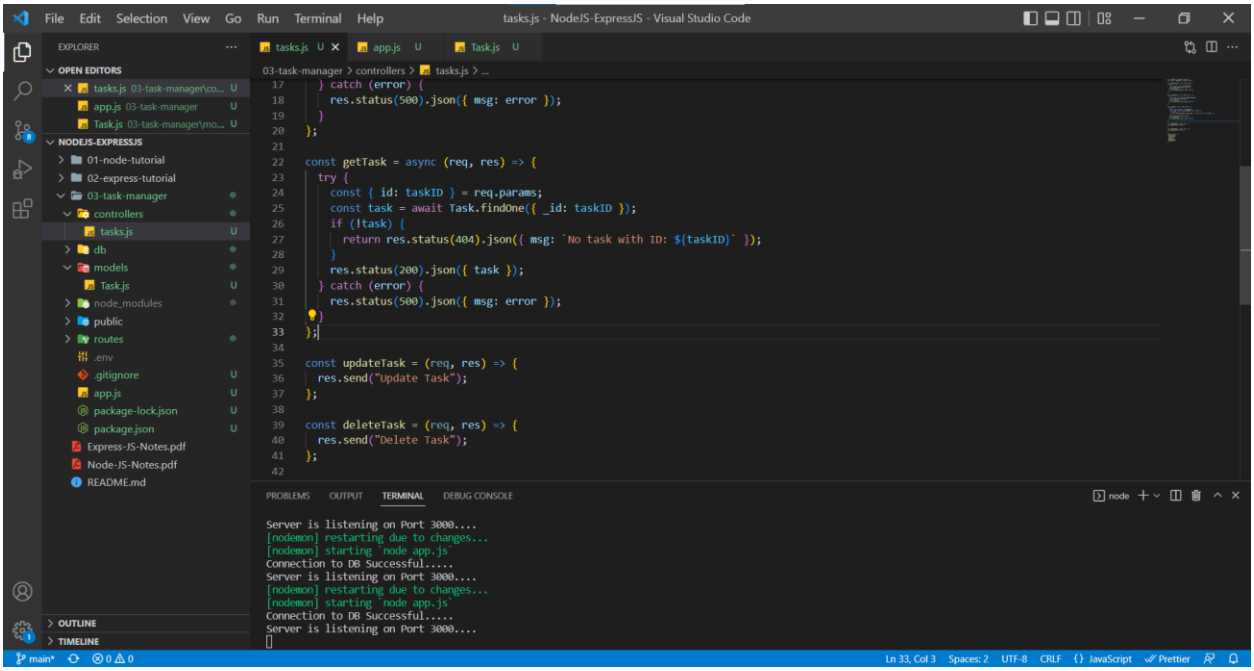
In Postman



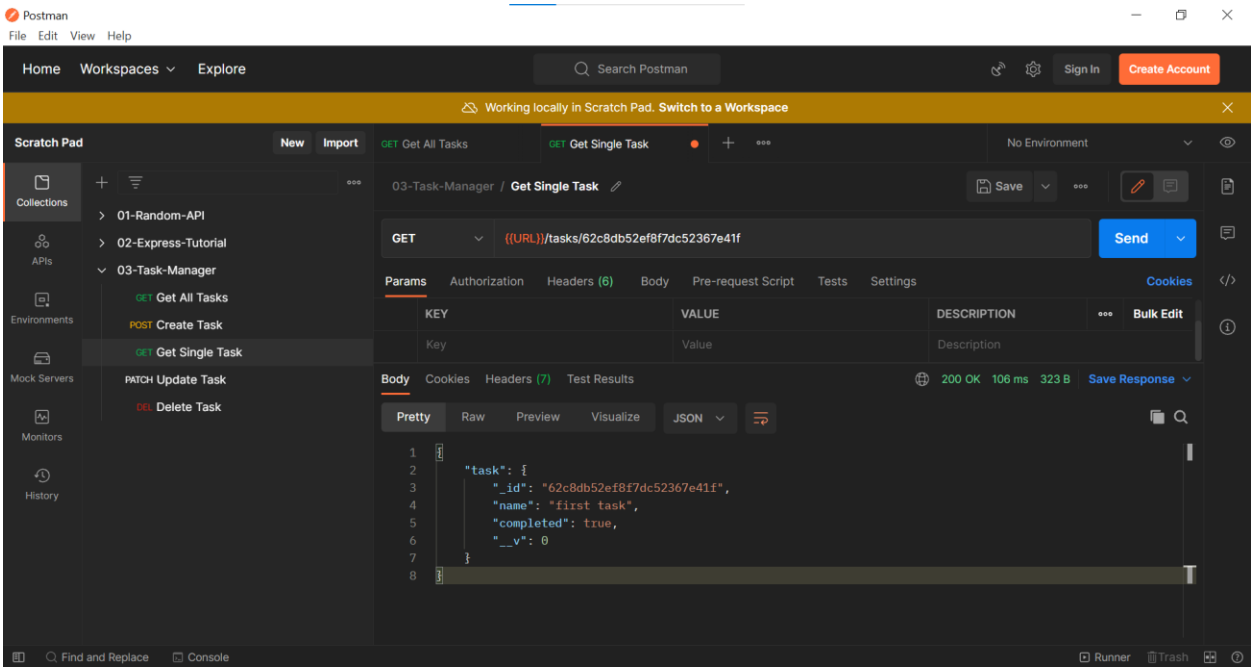
Get Single Task with Particular ID

Finding a Singular task in the database with ID from parameters.

tasks.js

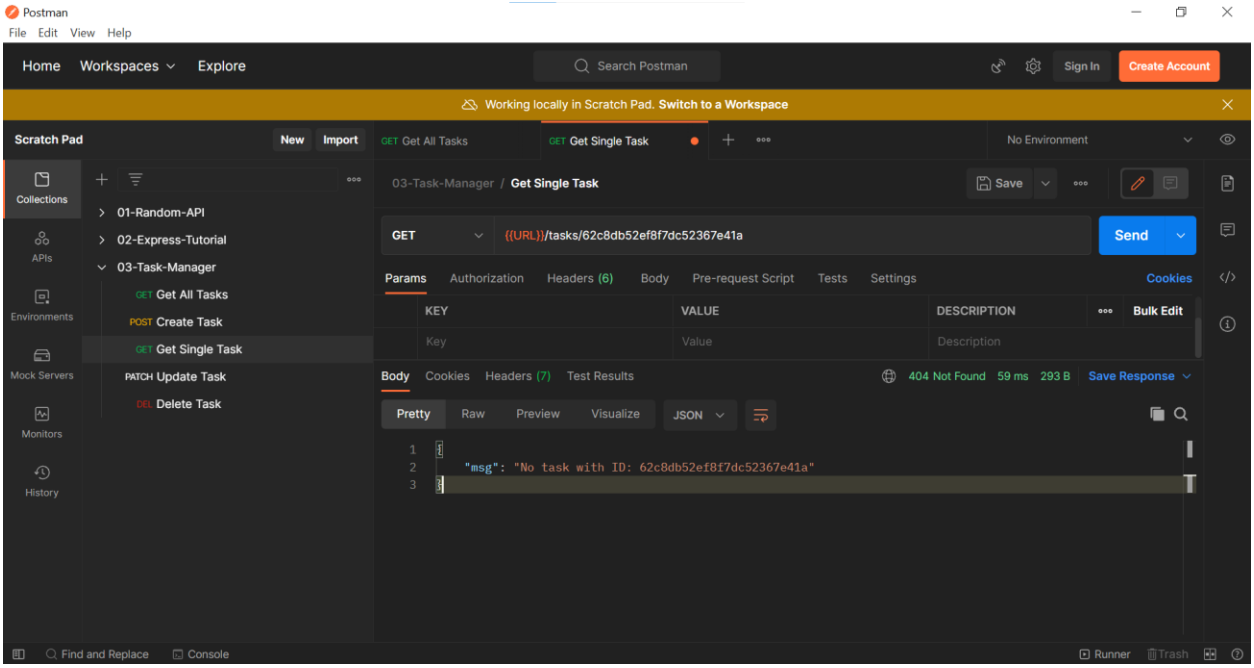


In Postman, with task ID



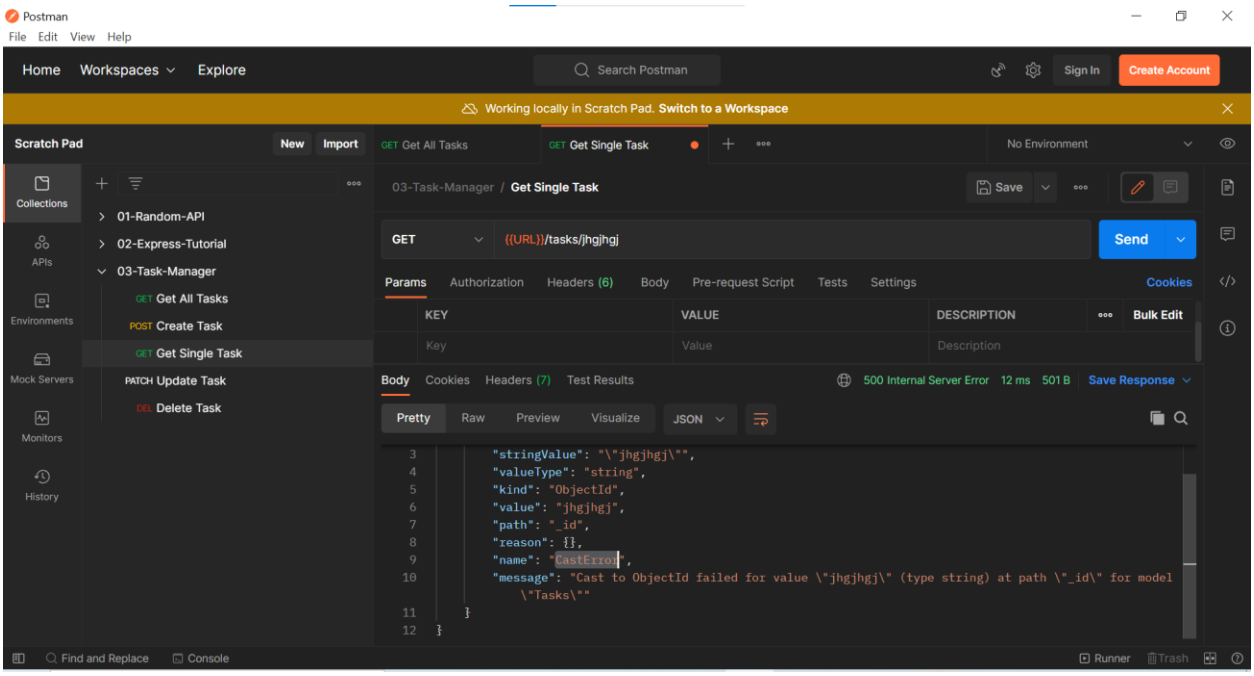
In Postman, no task with ID containing same length of characters equal to ID characters in Database

Response 404 Status Code



In Postman, when ID characters length is less than the ID character length in Database, we receive an error named Cast Error.

Response 500 Status Code

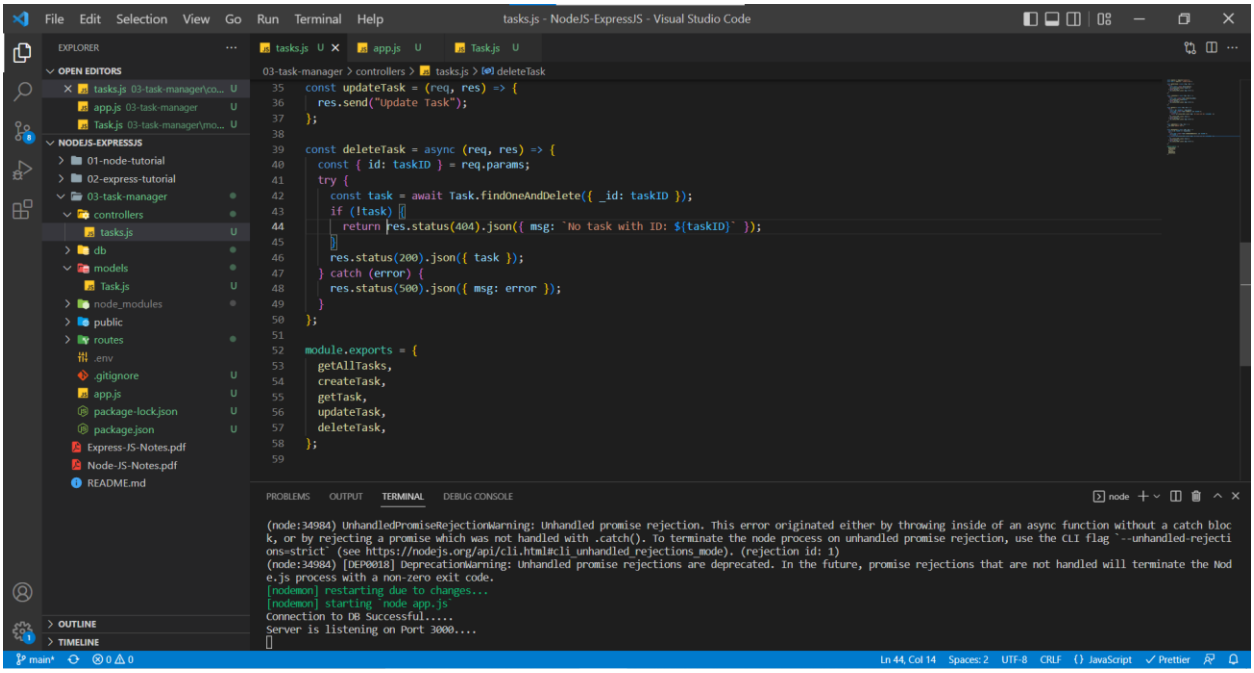


Delete Task

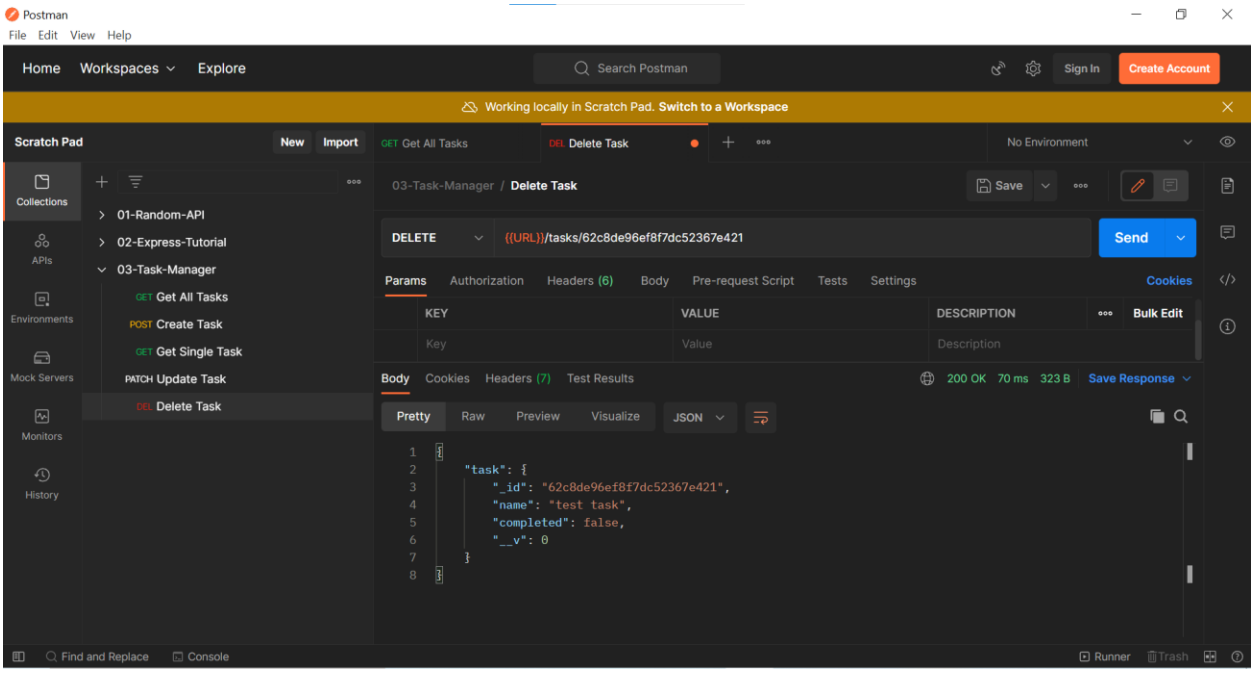
Deleting a particular task with ID

tasks.js

We shouldn't have two response in one try block, we have return only one so that is the reason we are returning the one inside the if block else we will return the response outside the if block.

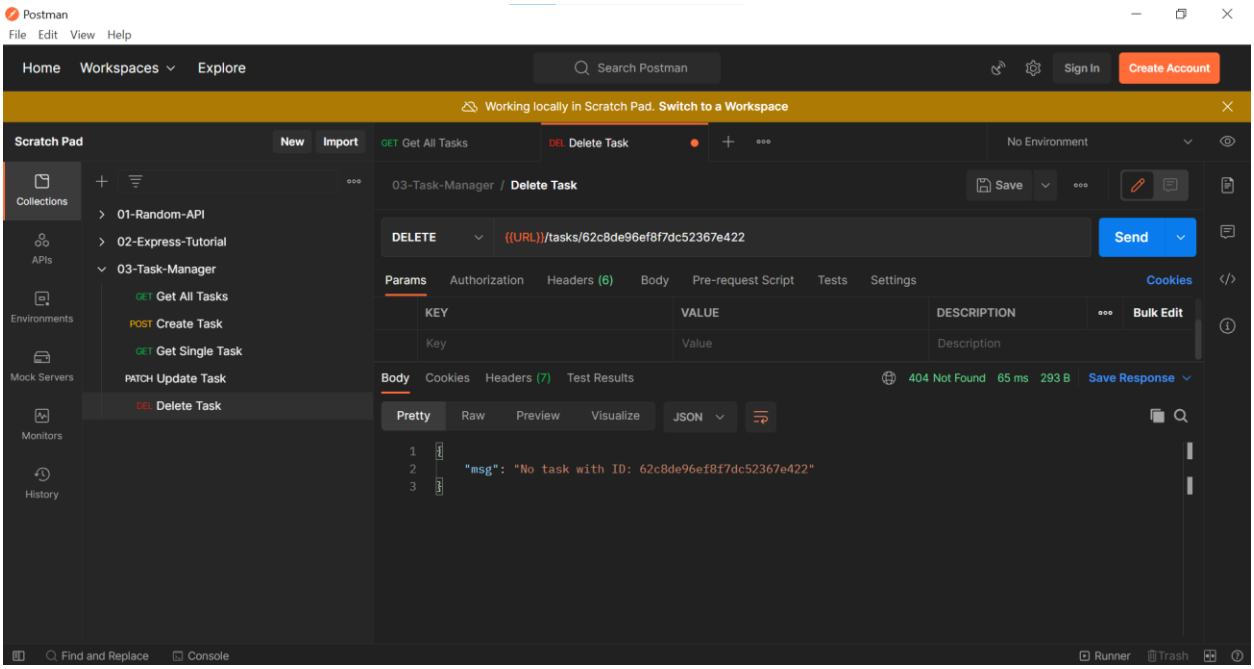


In Postman, deleting a task with Particular ID available in the database



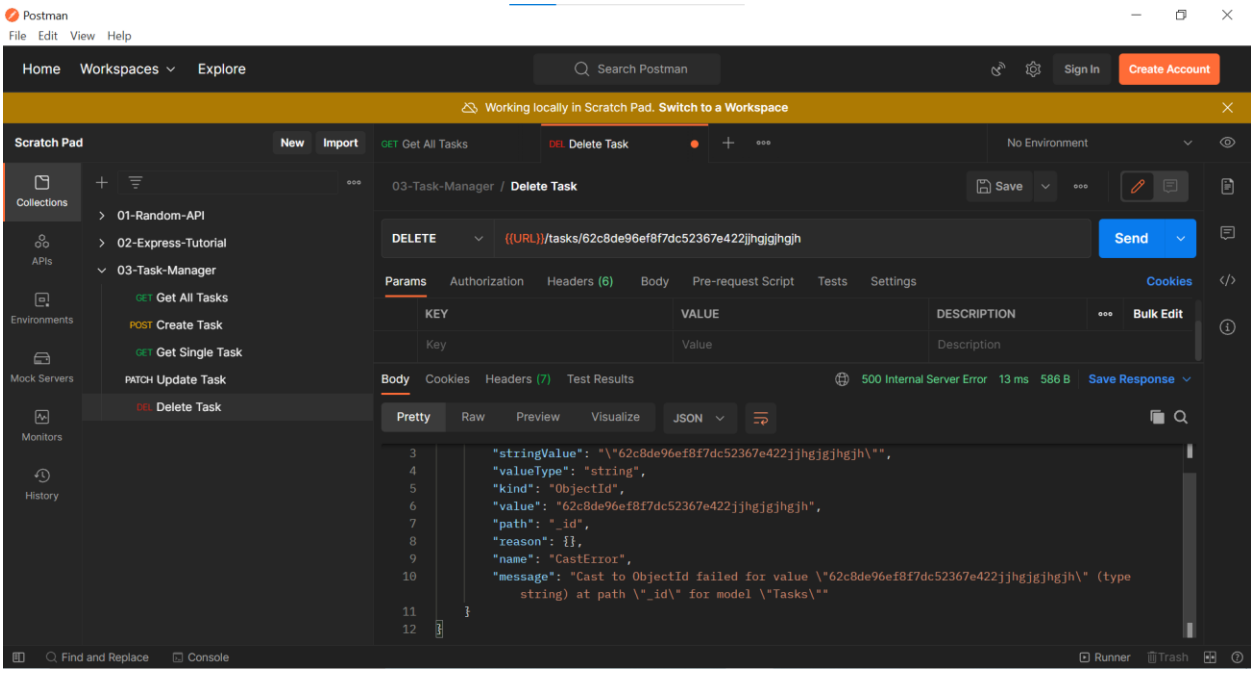
In Postman, deleting a task with ID does not present in database but has equal number of characters

Response Status: 404 – Not task found with ID



In Postman, deleting a task with ID not available in database and doesn't have equal ID characters with database.

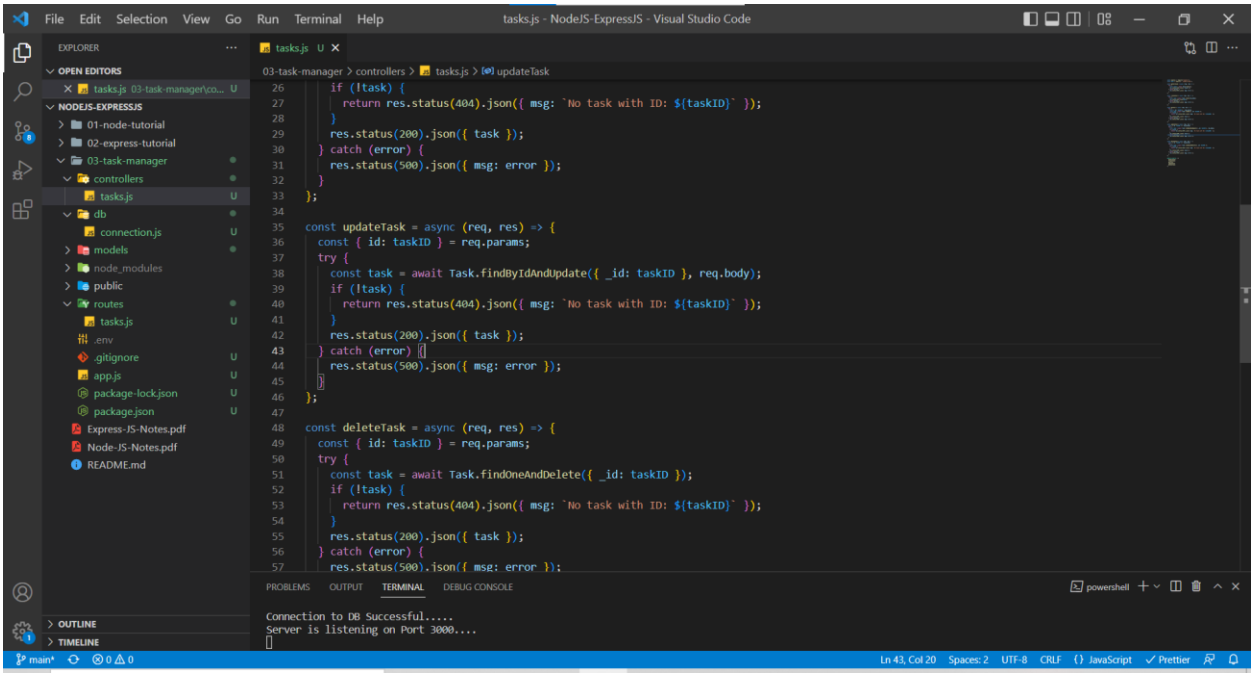
Response Status: 500 – Cast Error



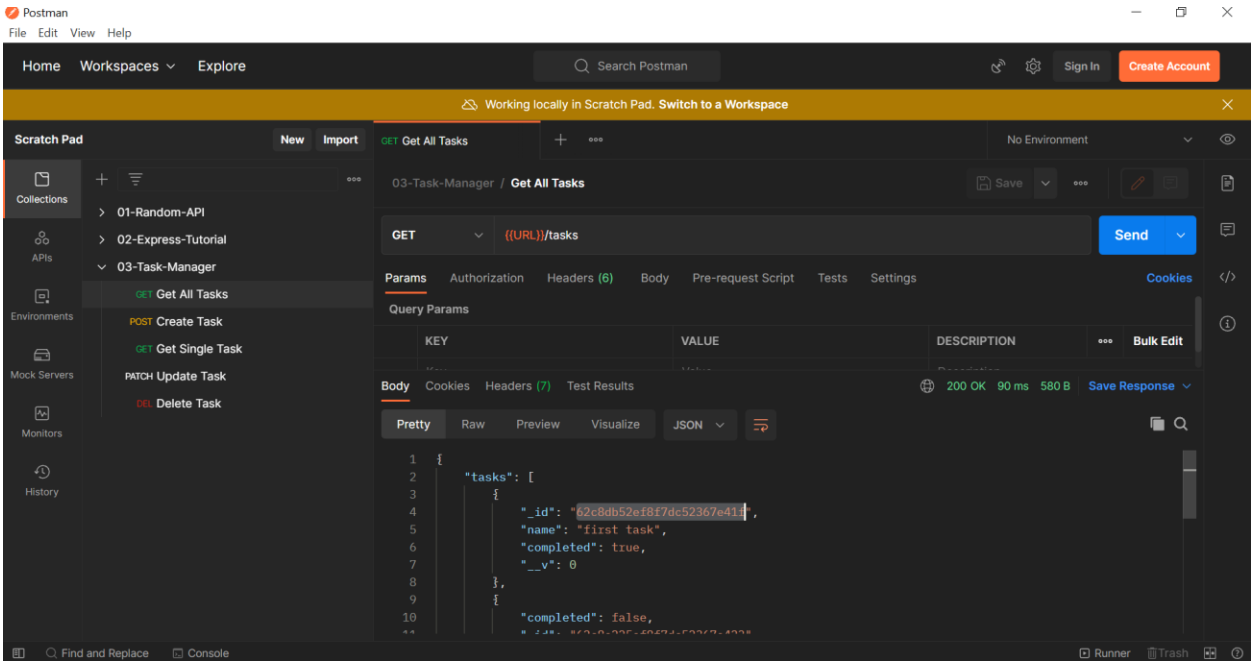
Update Task

Updating a task with certain data and ID

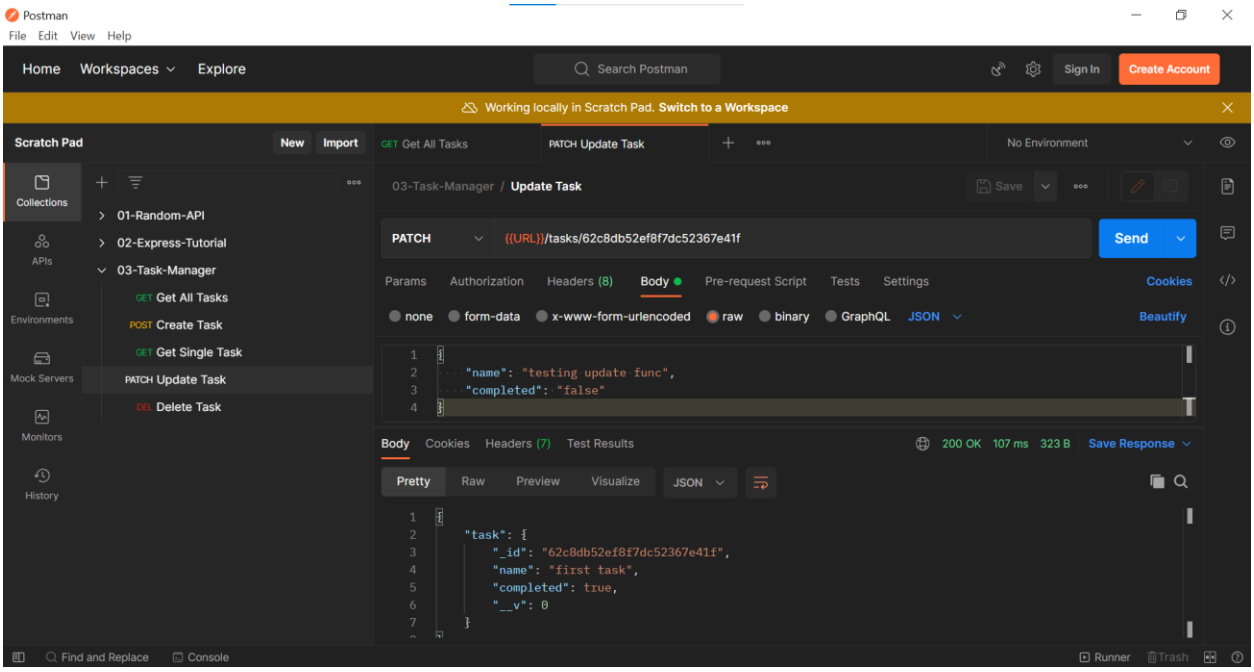
tasks.js in controllers folder



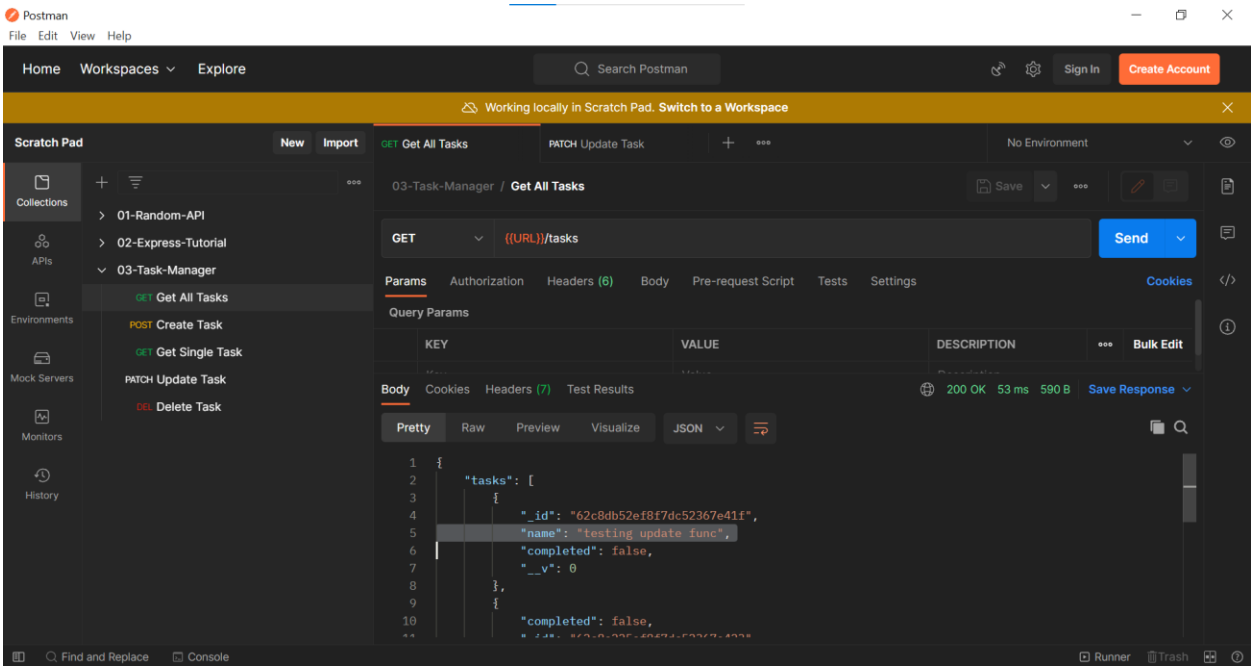
In Postman, we are going to edit the first task with ID 62c8db52ef8f7dc52367e41f



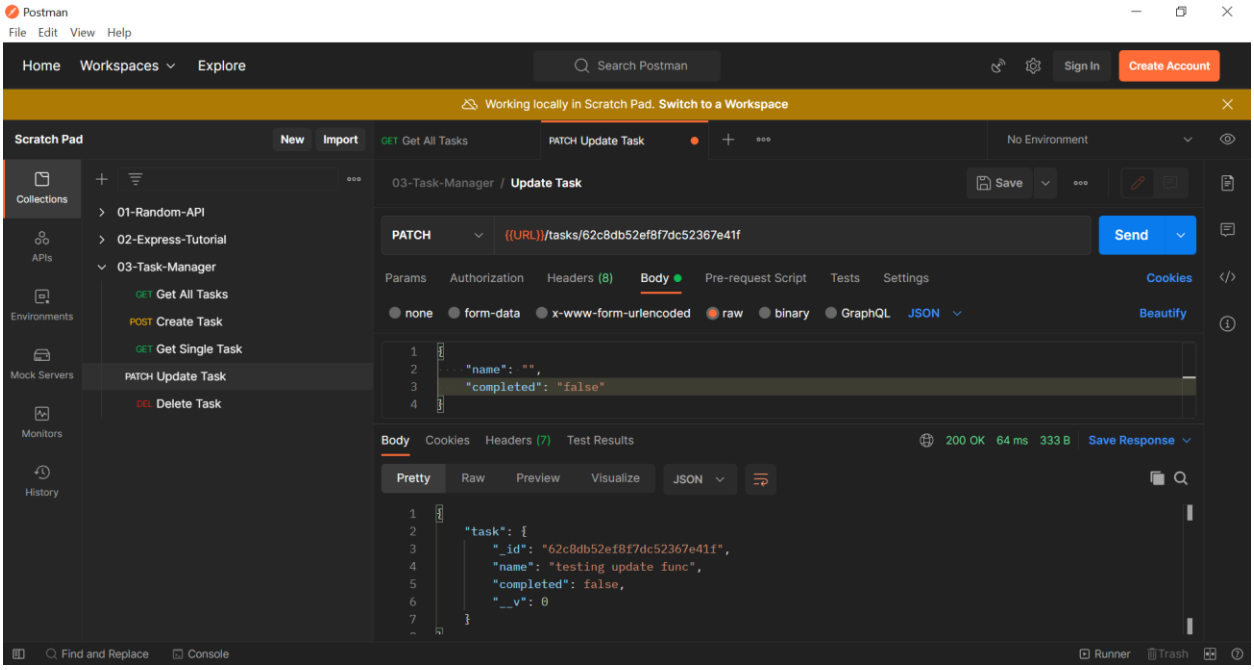
In Postman, updating the first task



Here we are still receiving the last item but not the latest item. Let's check that with the help of the Get All Tasks route.

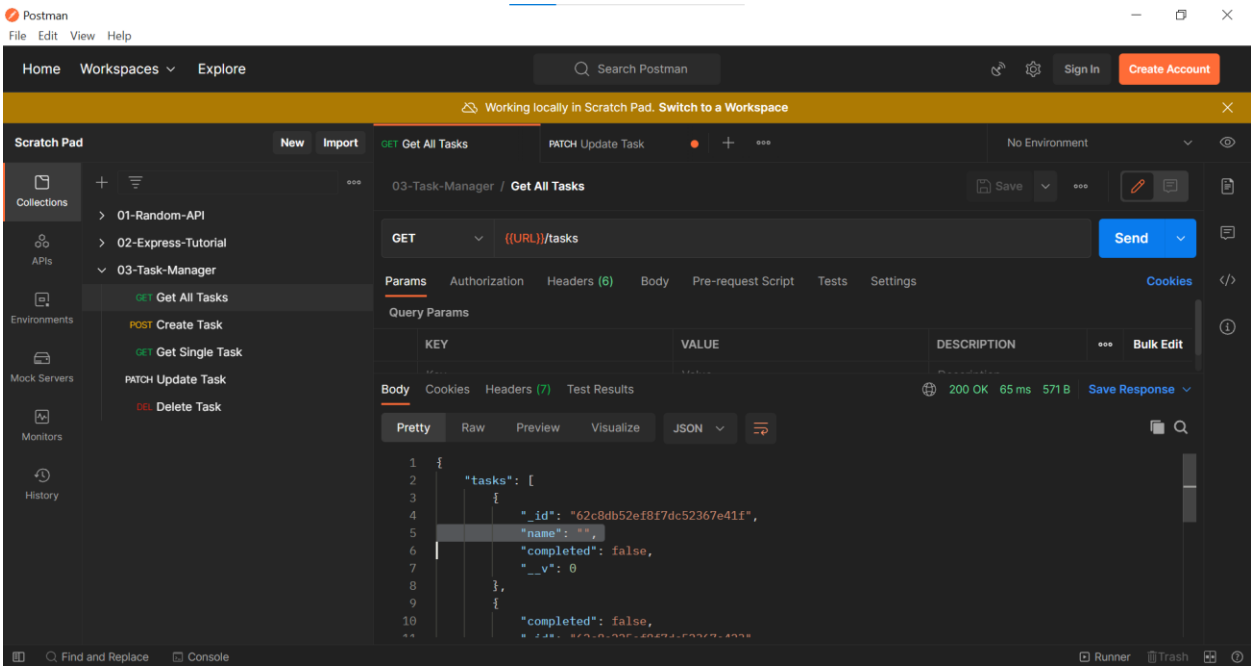


In Postman, sending the empty data through UPDATE



Still, we are receiving the last item instead of the latest item. Also, validators are not working.

In Postman, Get Al Tasks route

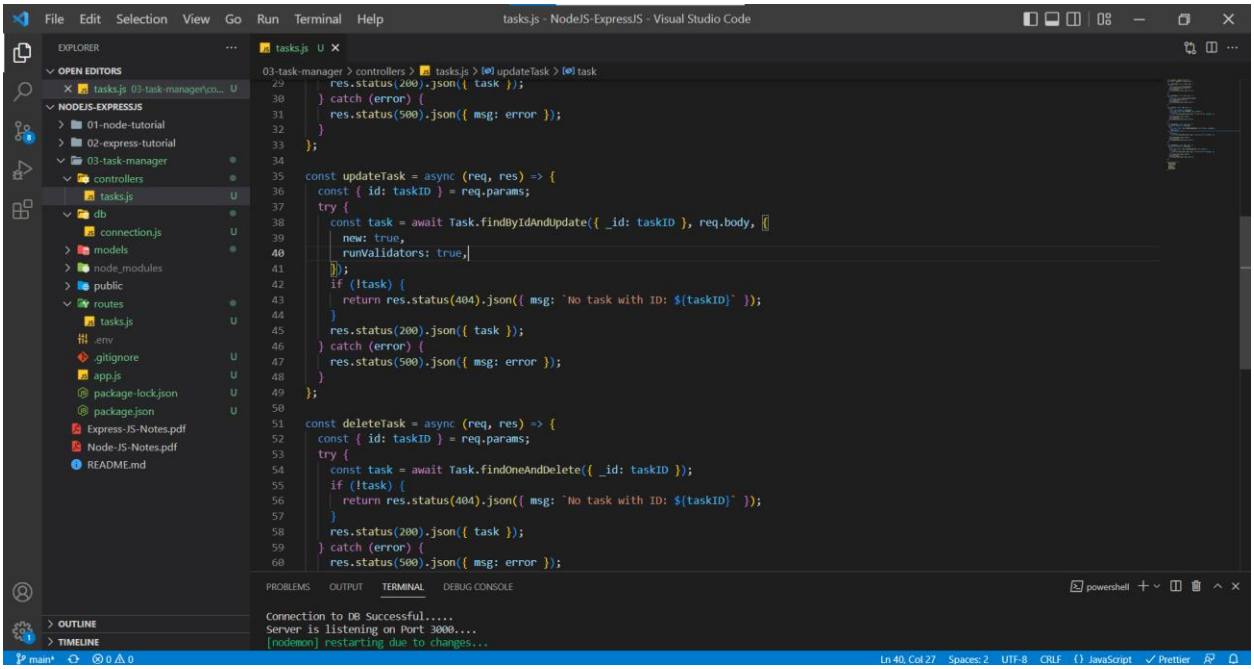


With the current setup, we are facing two issues,

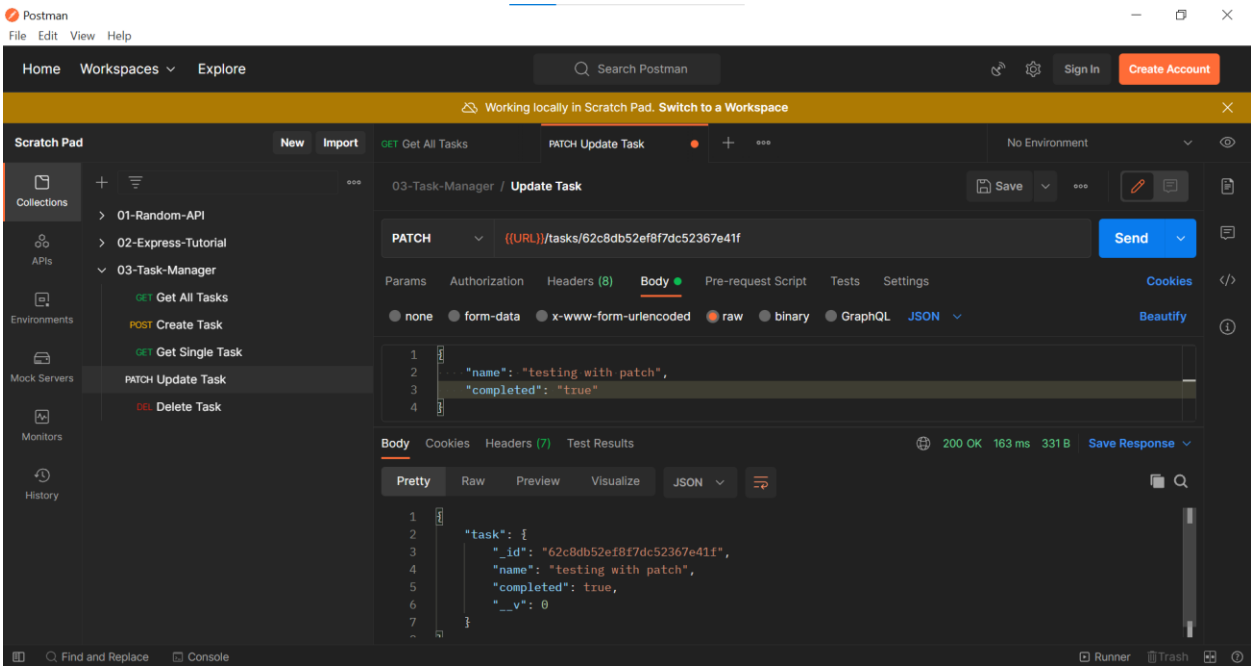
1. Whenever we are updating, we are receiving the old item data instead of the updated item data.
2. Validators are not working.

We can fix those issues by using the options object in the findByIdAndUpdate method.

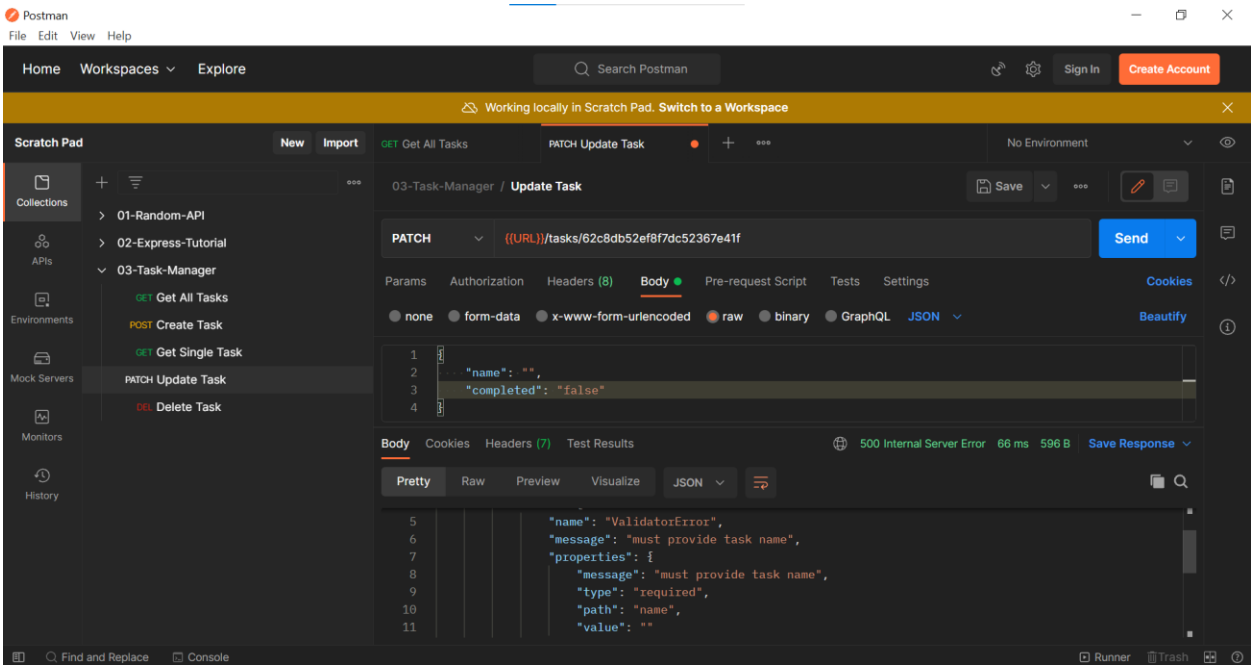
tasks.js in controllers folder



In Postman, the latest item is getting returned as the response.



In Postman, Validation errors are sent back when empty name field is sent in the request.



Instead of using the frontend code provided by the instructor. I have developed the React App which performs the CRUD Operations using the Backend application that we prepared using Node and Express.

Until now, our core functionality is completed but now let's work on improving the backend code.

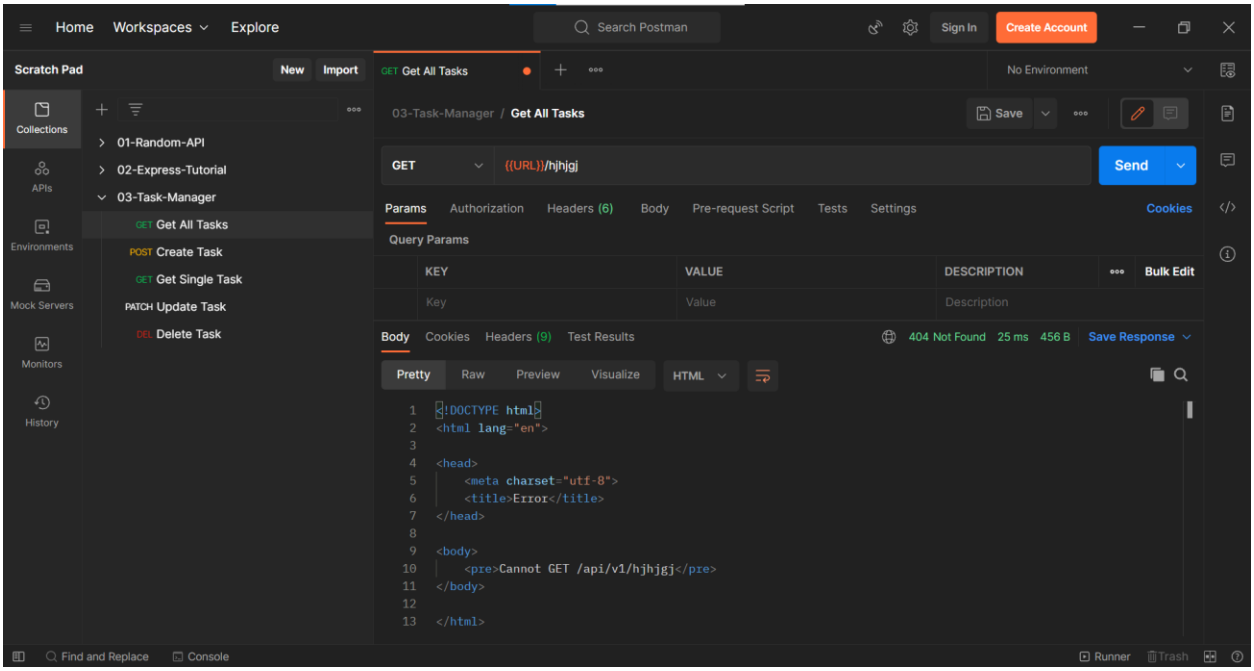
Now let's discuss about **PUT** method vs **PATCH** method

PUT and **PATCH** are both used for updating the resource, but when we are using **PUT** method, we are trying to replace the existing resource (sometimes we may forget to send parameters in request body and those parameters may be removed since we are using **PUT**) and **PATCH** (In PATCH Method, if we don't send the parameter in the request body it won't be replaced and will stay in the data) method is for partial update.

It depends on the requirement to choose between **PUT** and **PATCH**. For this Project we are using **PATCH** method.

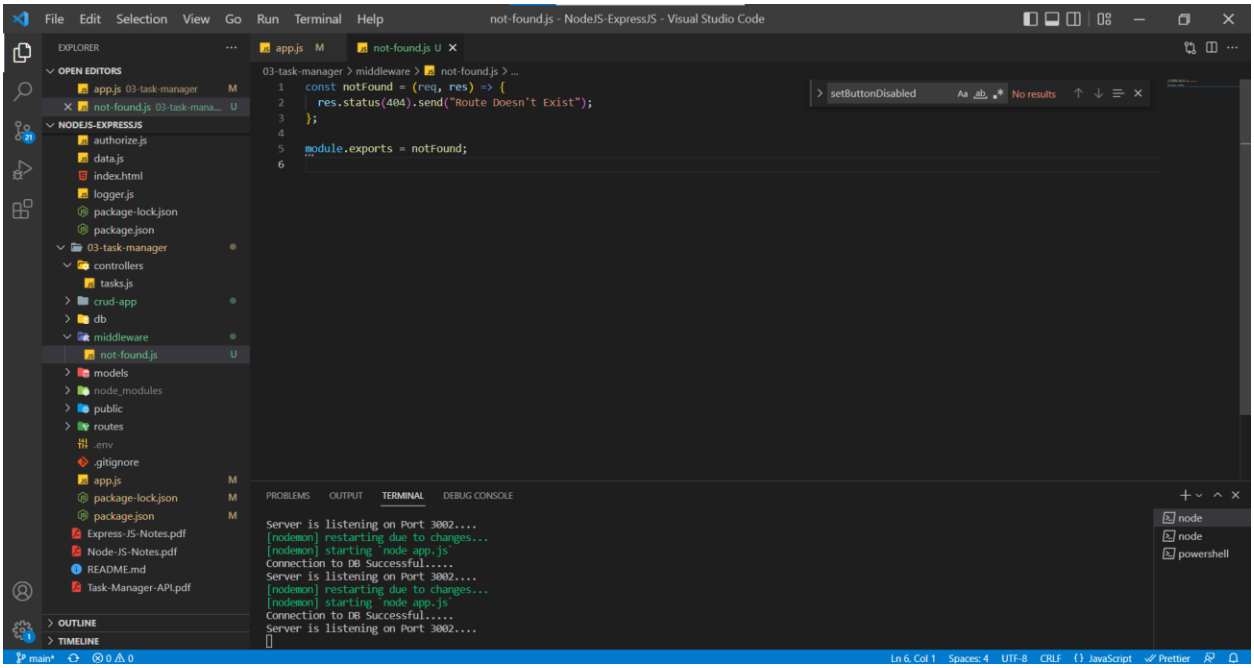
Error Route

When a user enters a route that isn't present in the controllers then we receive an 404 Status Code with an HTML Body.

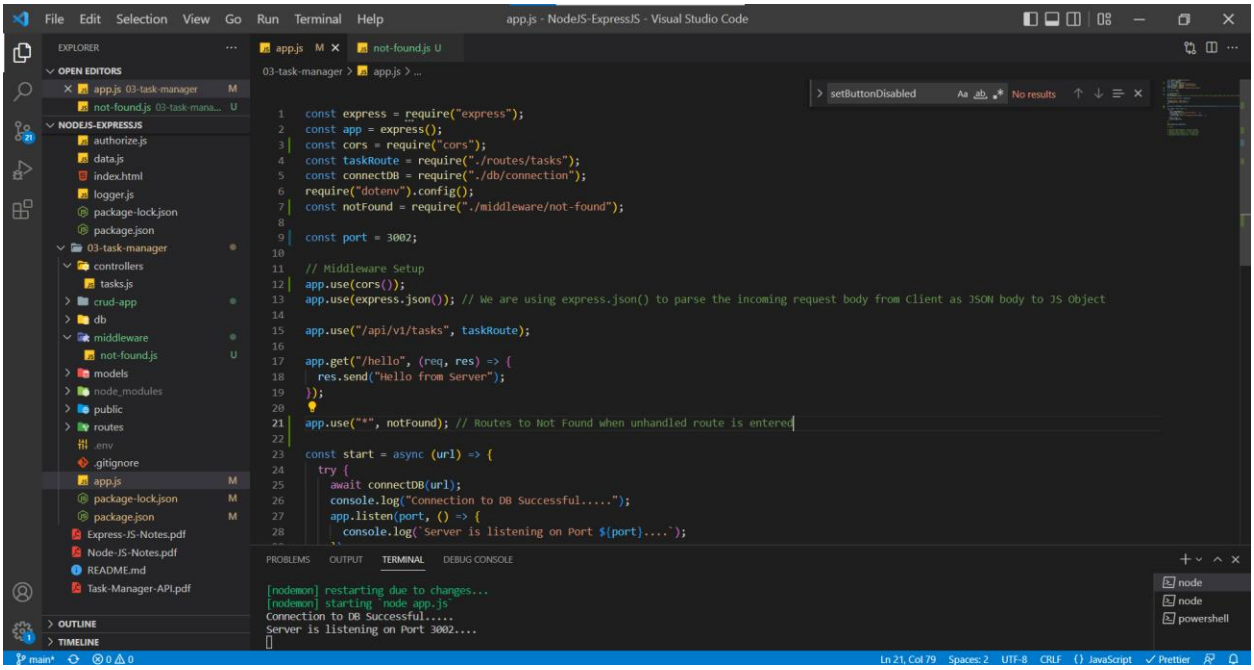


We can solve the above problem by handling the unhandled API routes. We will create a middle function which we will used whenever a user enters with an unhandled route.

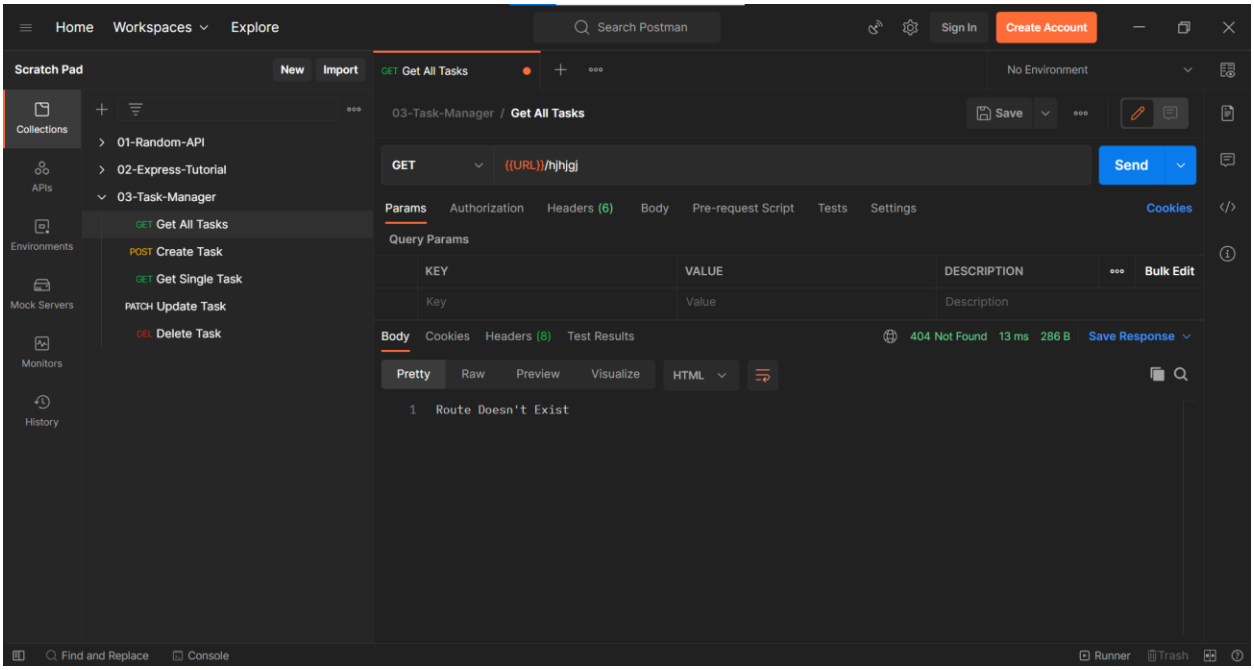
not-found.js in Middleware folder



app.js



In Postman, when an unhandled route is entered we are sending the Response with Status Code 404

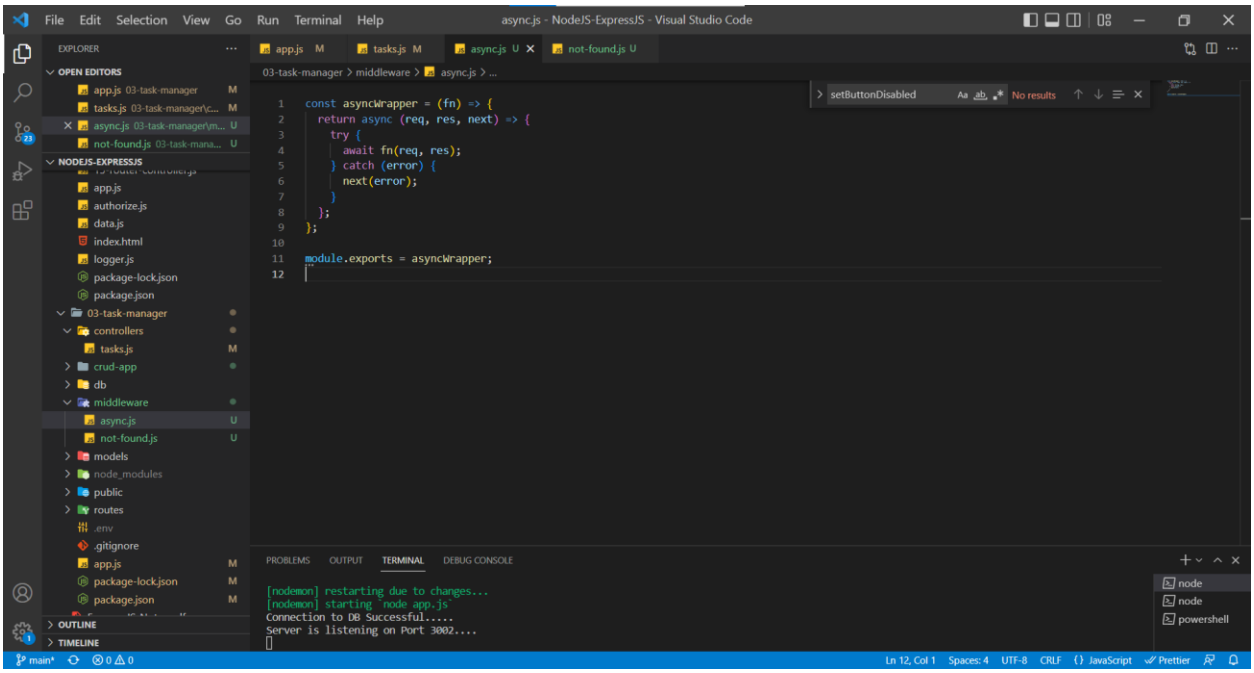


Removing the Redundant Code

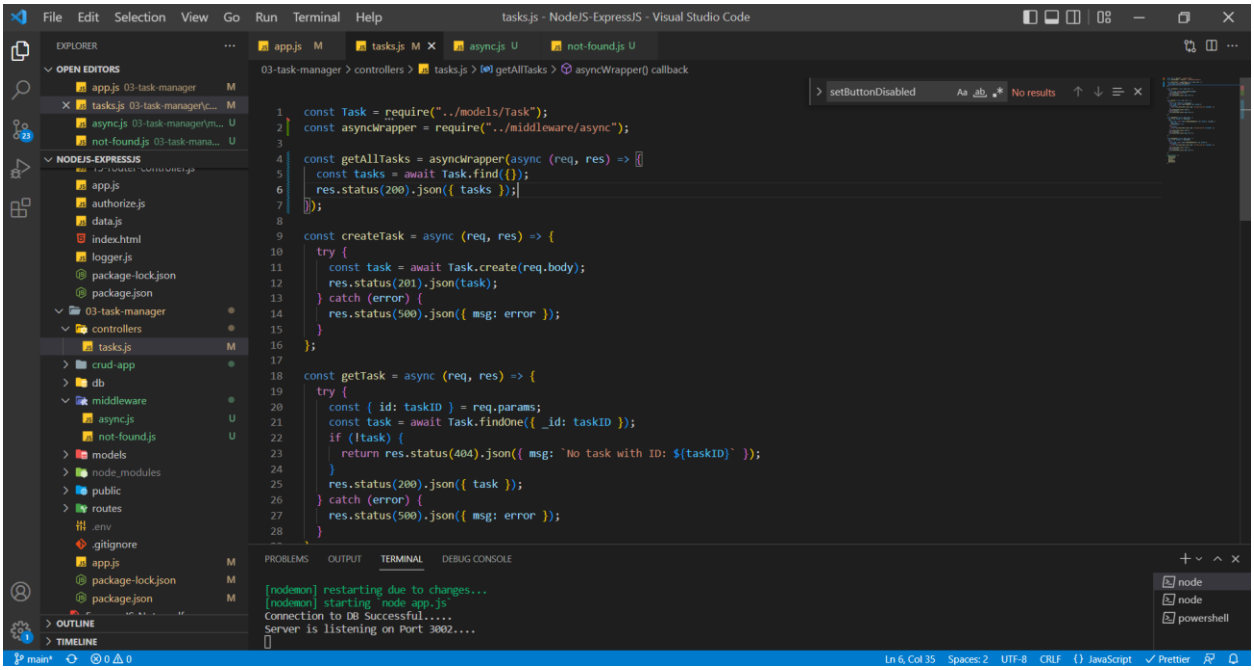
As we can see there a is lot of repetitive code in the Controllers folder where we are using try and catch blocks to handle the asynchronous functions. To remove the redundant code, we can create a middleware function which will take care of handling the asynchronous functions.

Let us create a middleware function named asyncWrapper which is used the wrap the asynchronous functions in our code.

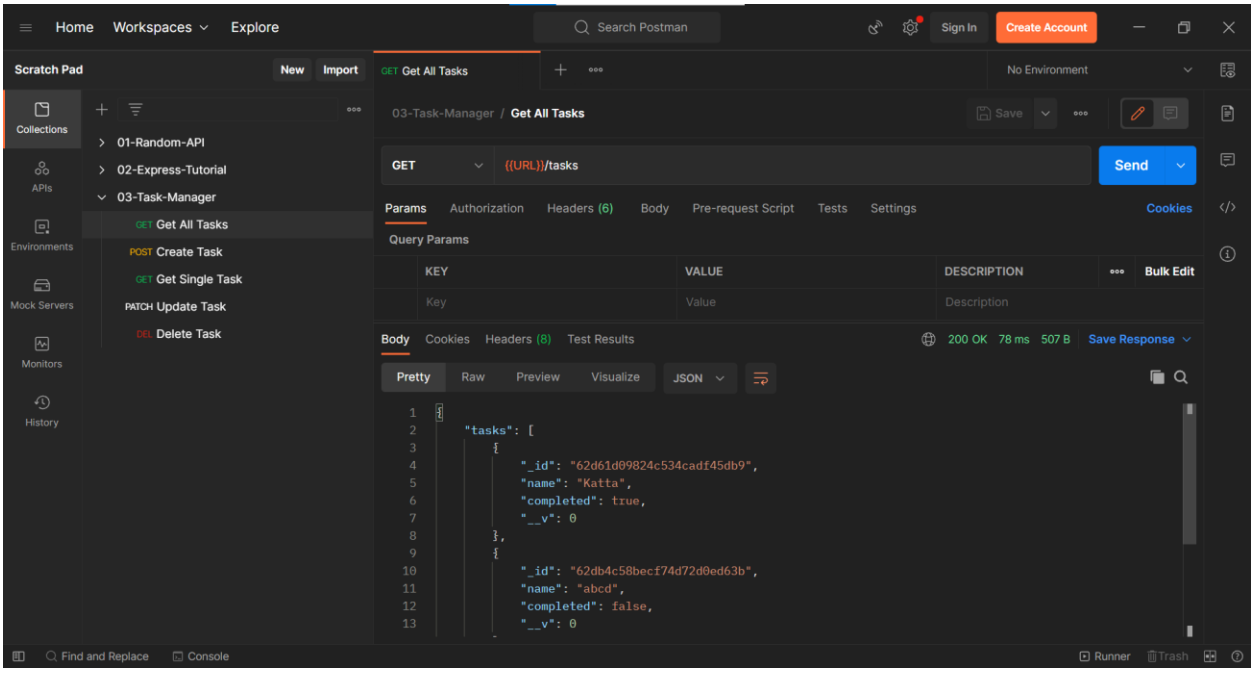
async.js in Middleware folder



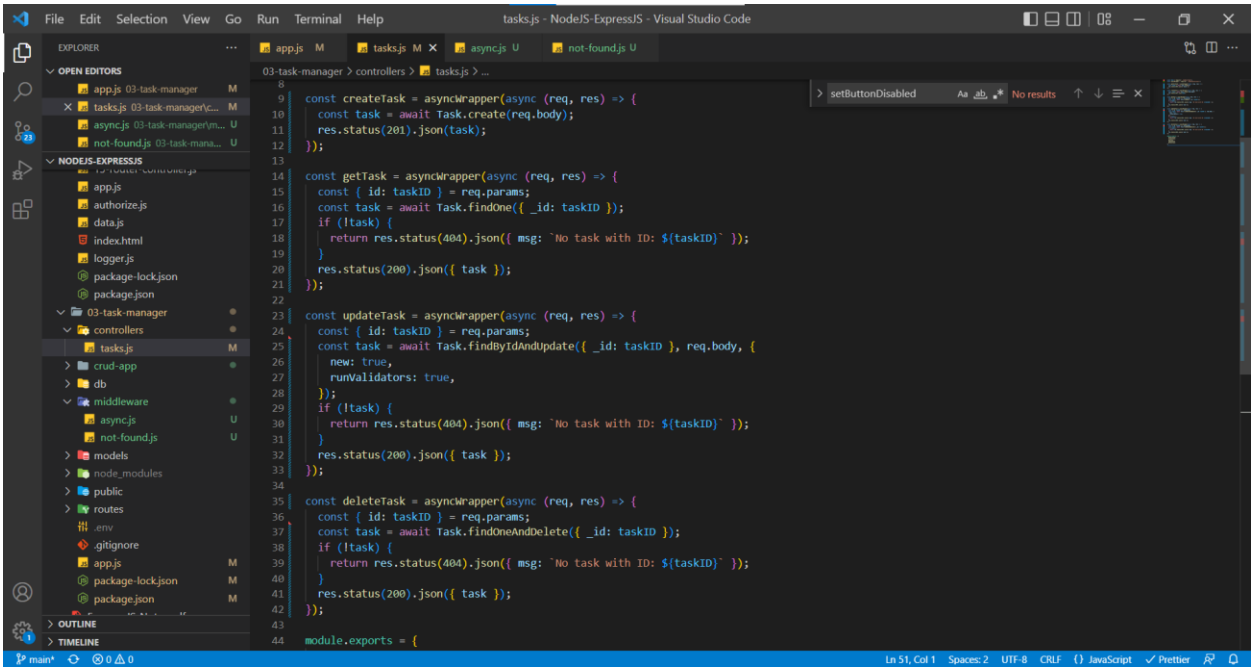
Using AysncWrapper in Tasks.js



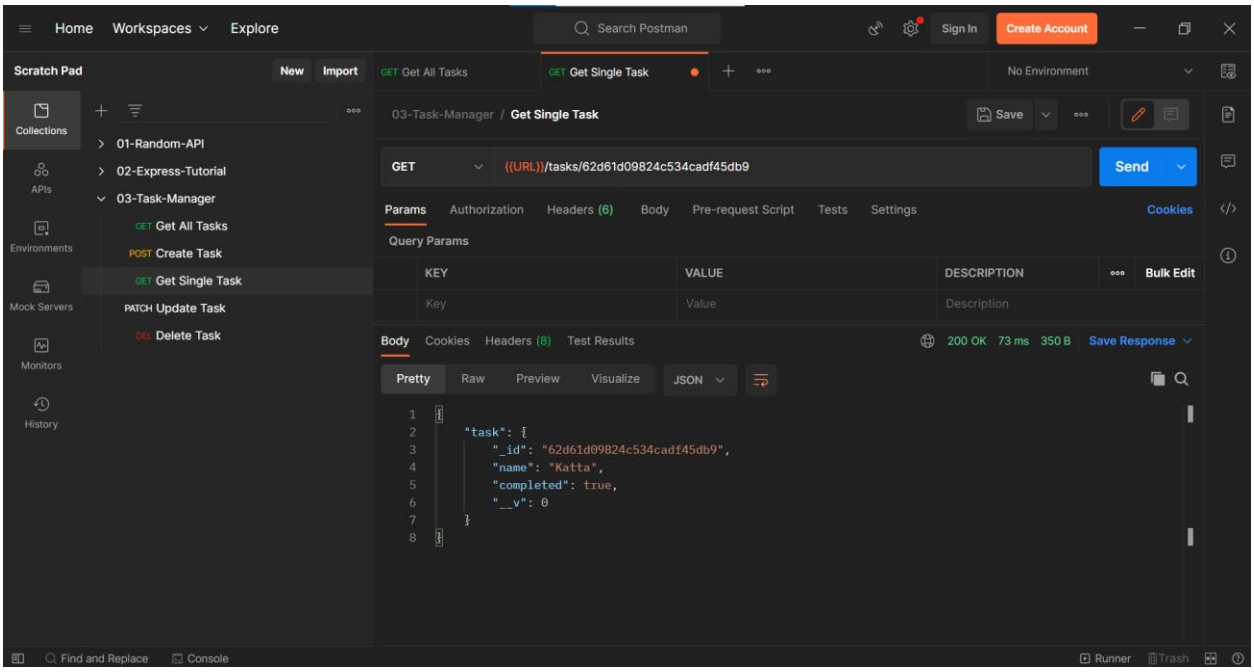
Getting All tasks from Postman



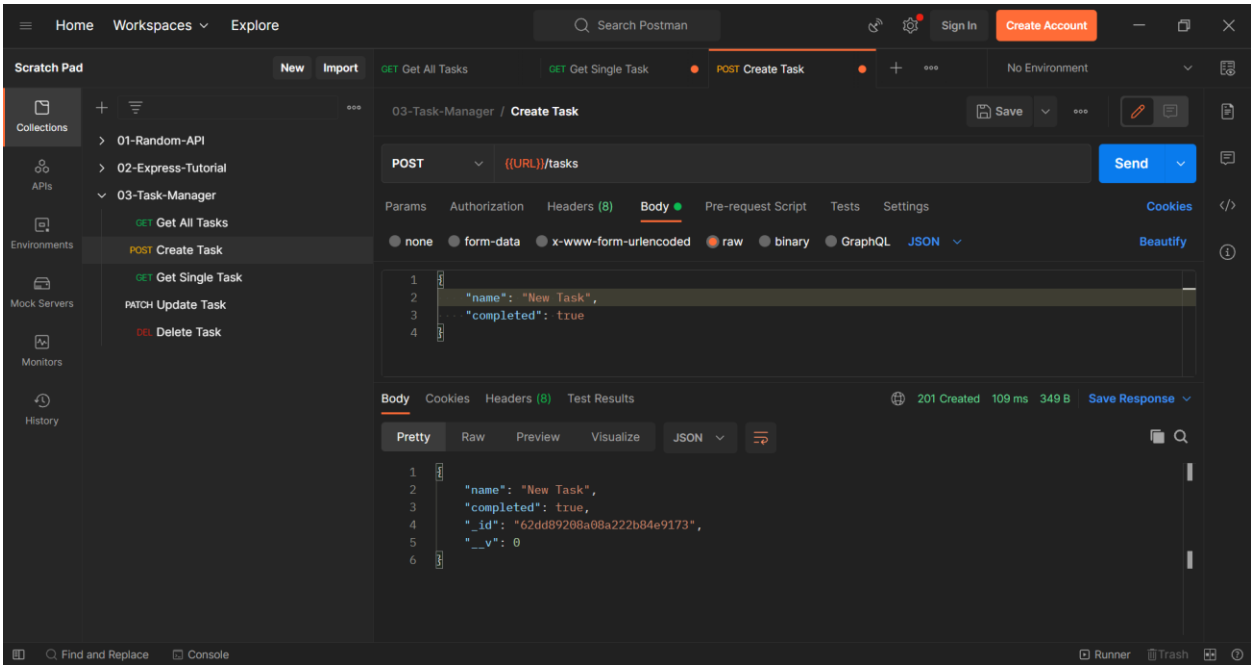
Now let’s apply the AsyncWrapper to all the API Routes



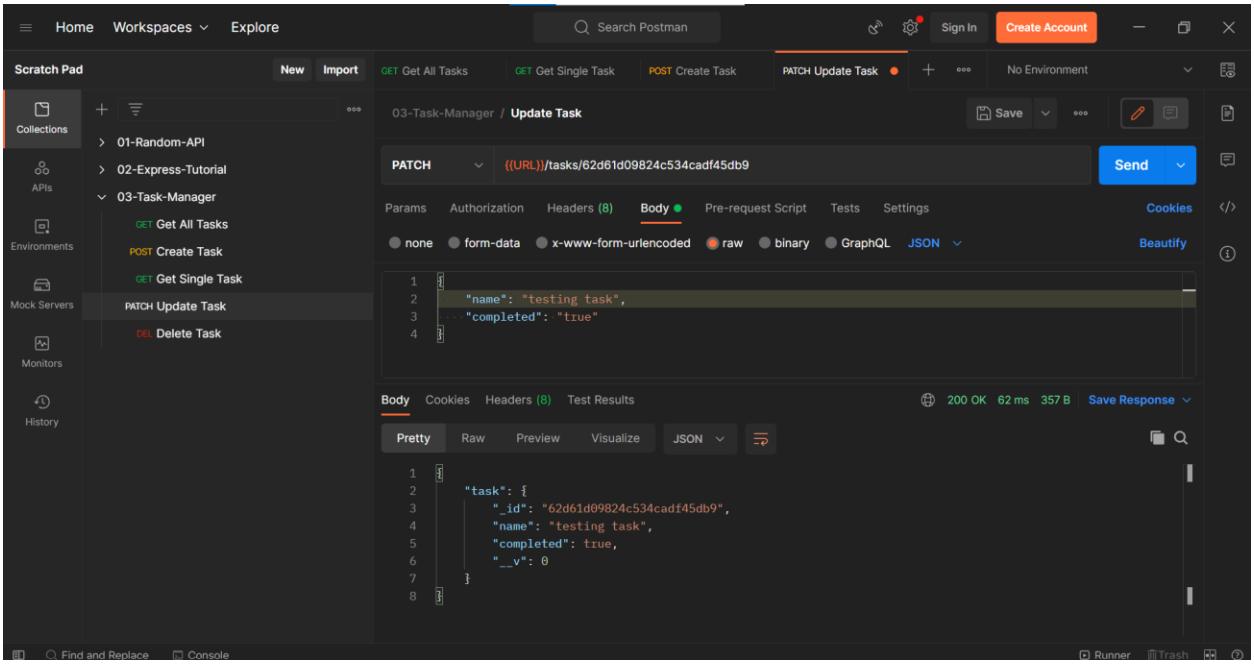
Get Single Task



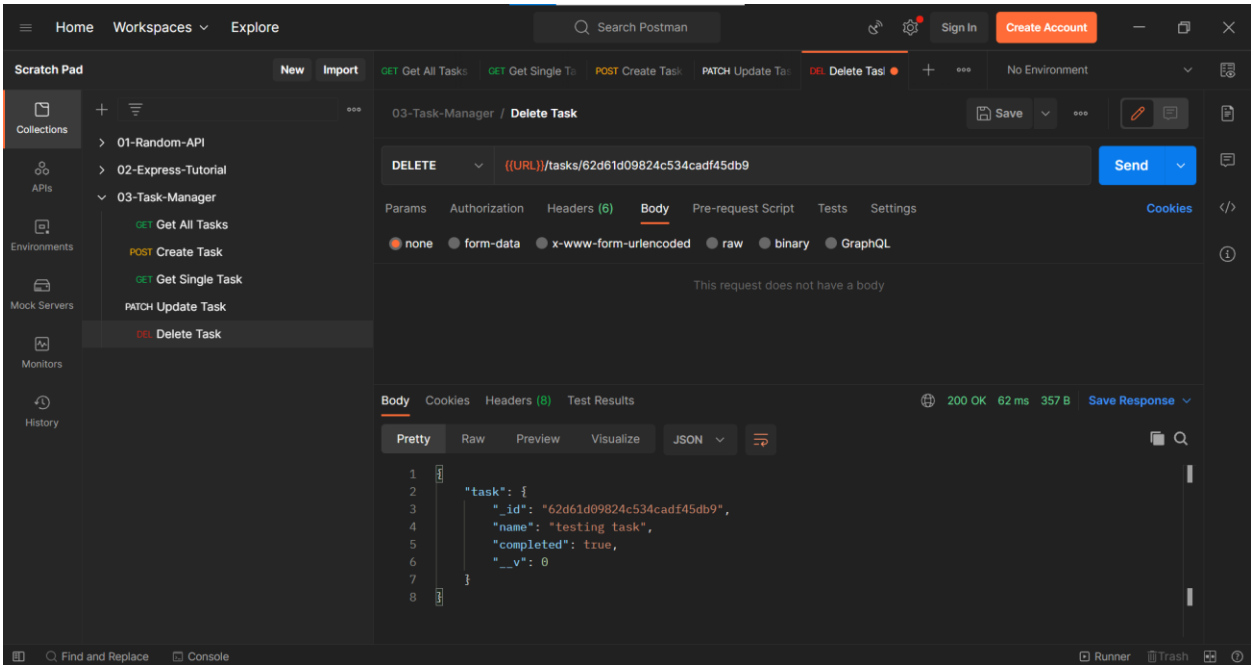
Create Task



Updating Task

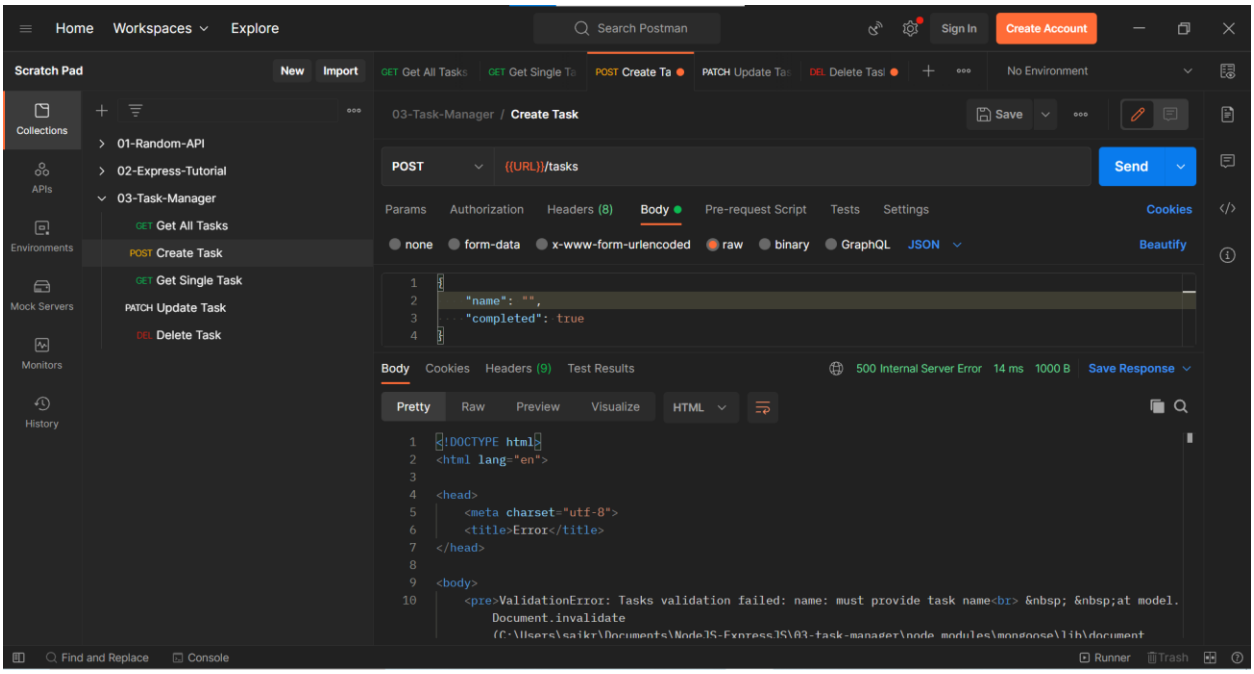


Deleting a Task



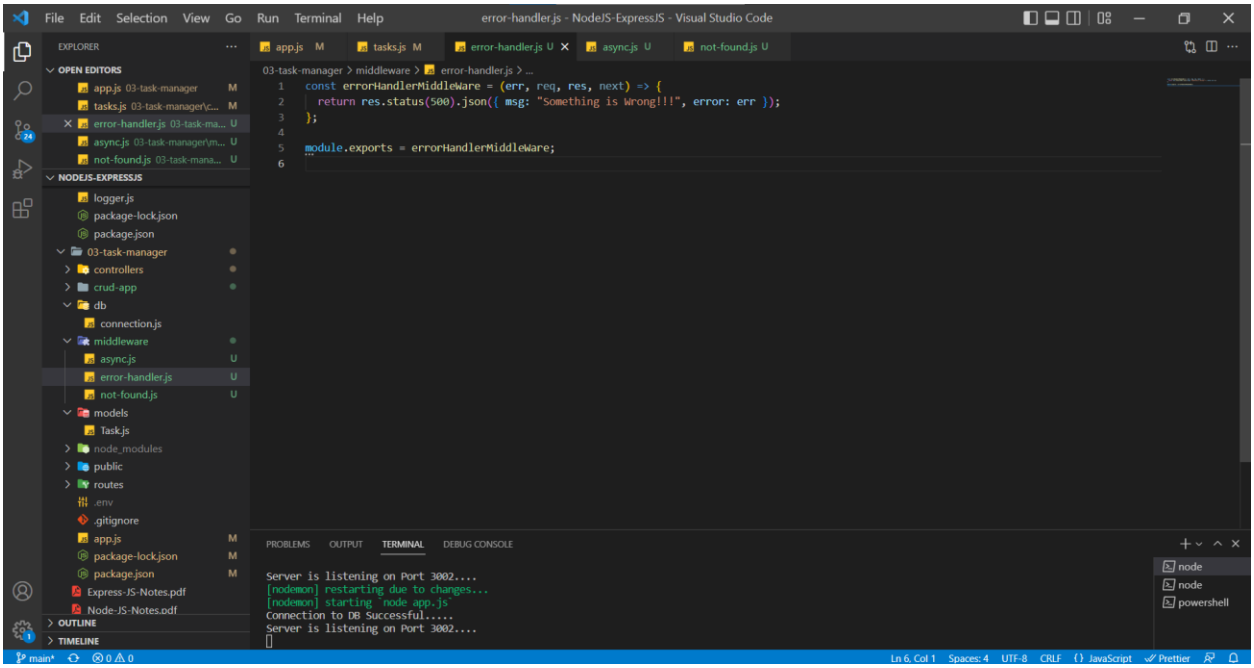
Everything works as expected but here we are not handling the errors.

In the below scenario, we are trying to create a task, but we are sending a value for name property hence we should get an error but here we are receiving and HTML since we are not handling the error.

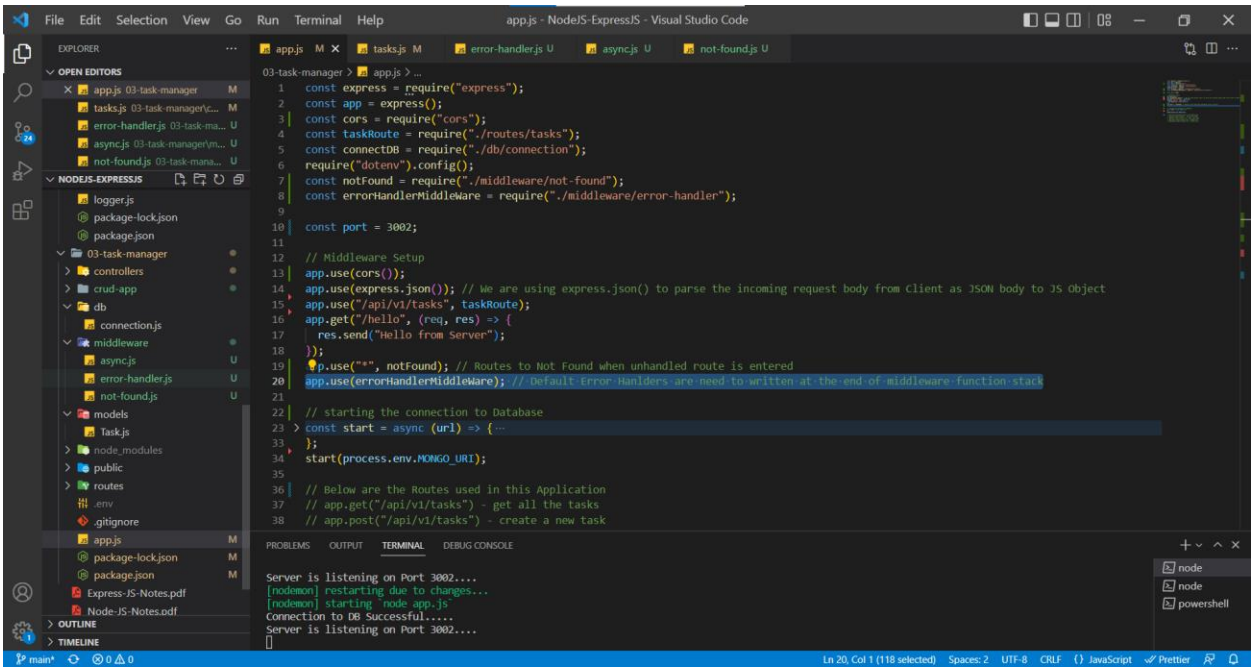


We can handle the above scenario by writing an express default error handling middleware function.

error-handler.js in Middleware folder

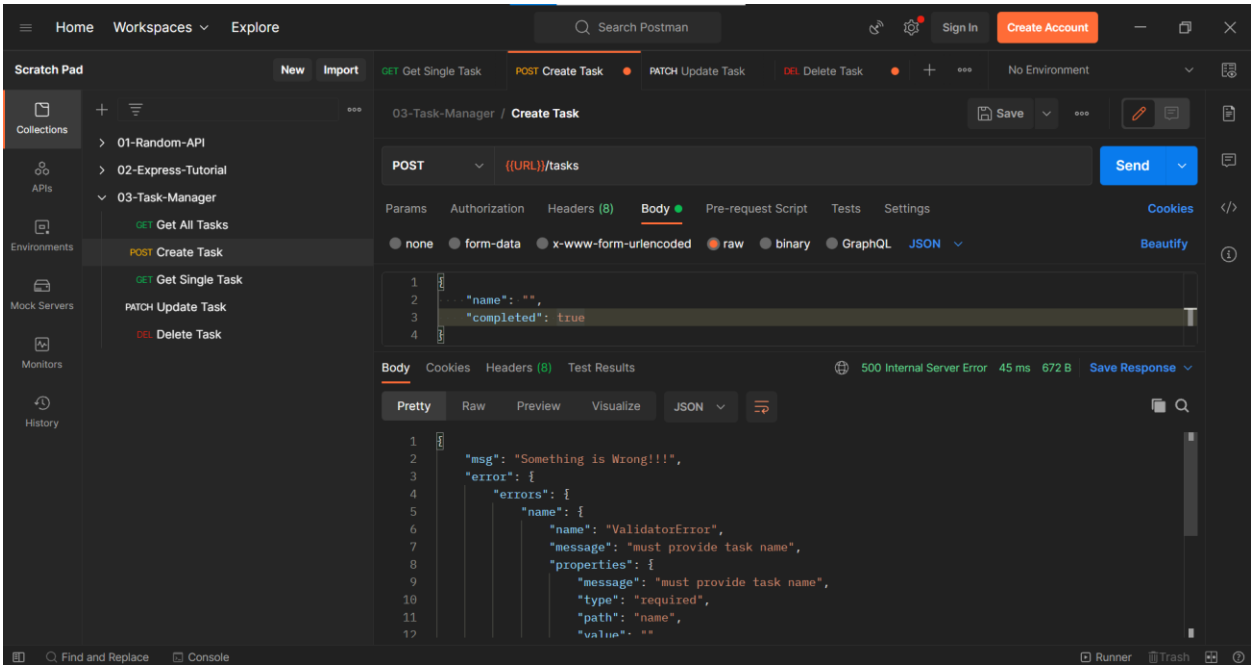


Default Error Handler in express contains 4 parameters (err, req, res, next)



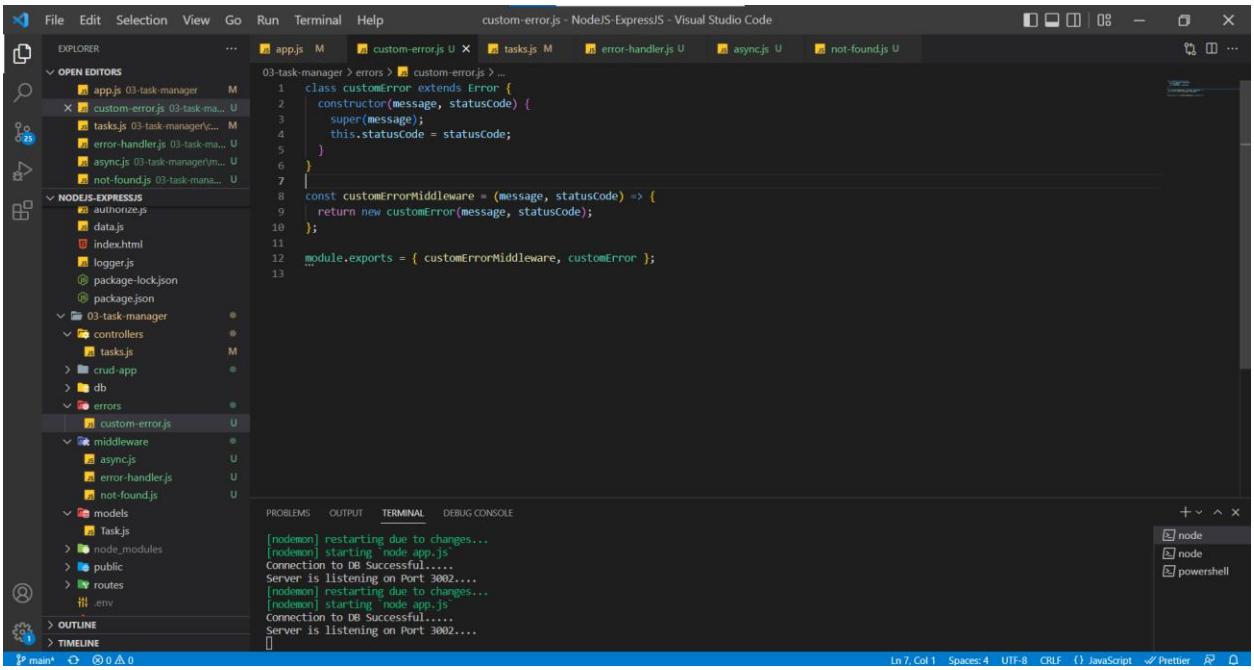
In app.js we need to place error-handler middleware function at the bottom of the middleware functions stack.

In Postman, For Creating the Task

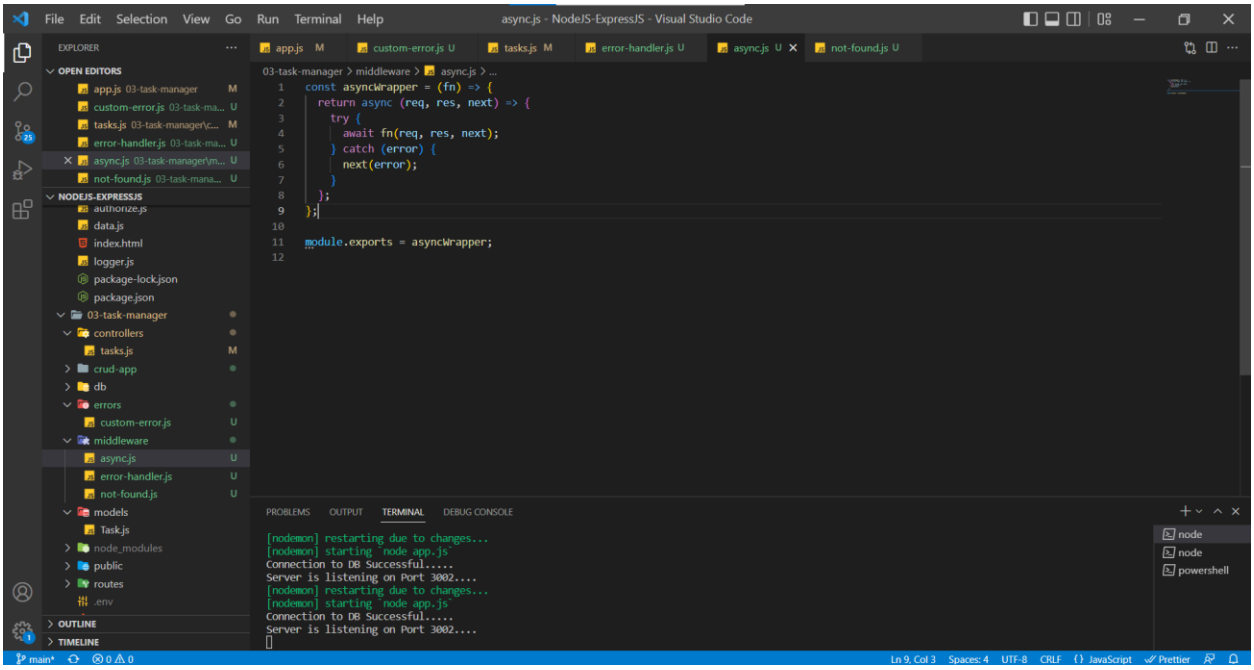


Since we handled the Default Errors in express, now it's the time to remove the redundant code used to handle Custom Errors in the project.

Creating a custom error file in errors folder

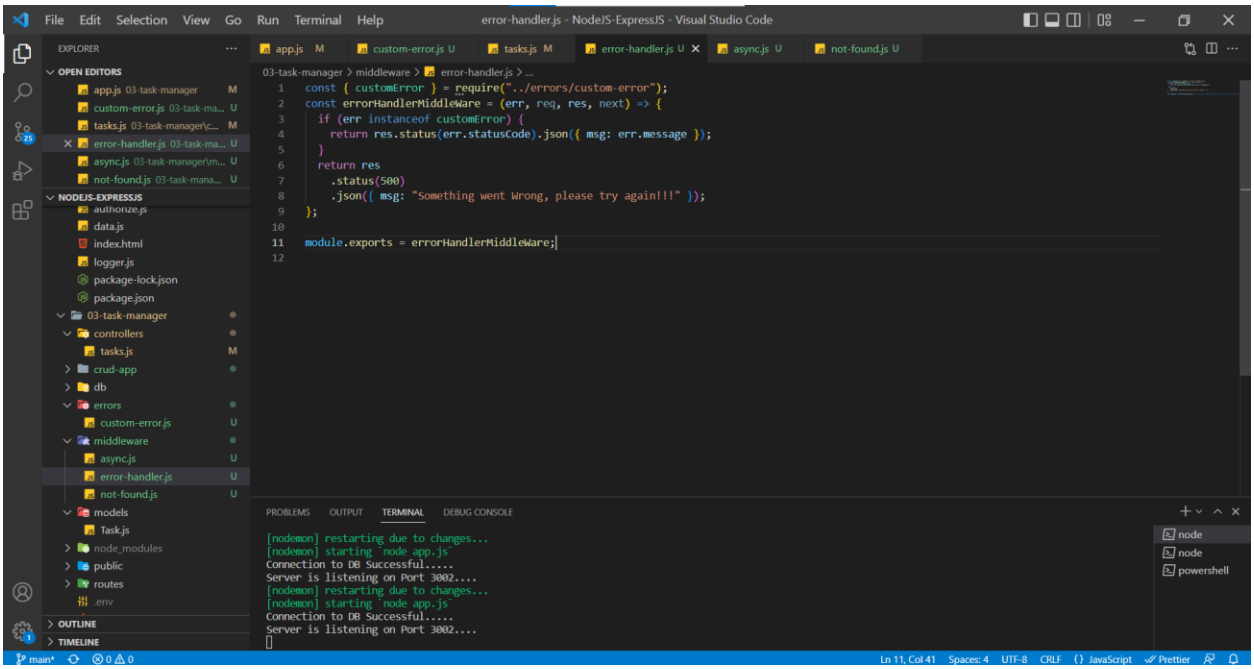


Need to use the next() method to transfer the error the custom error file or object

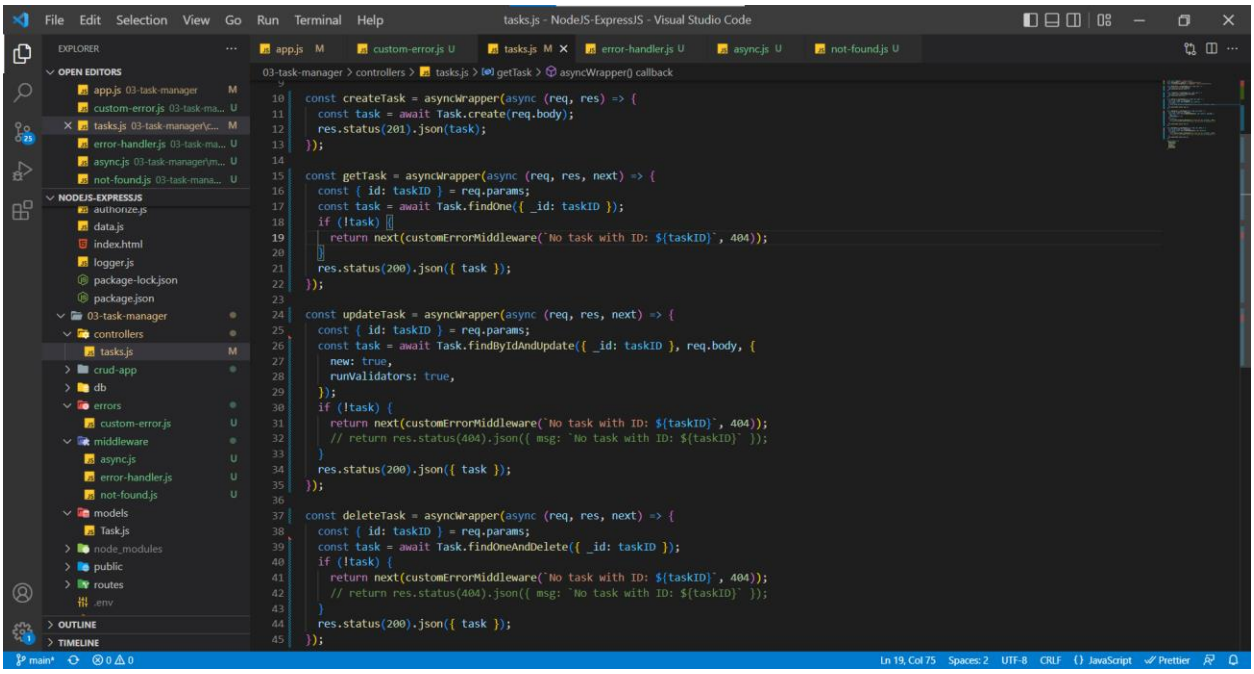


Adding extra lines of code to handle the custom errors in the application

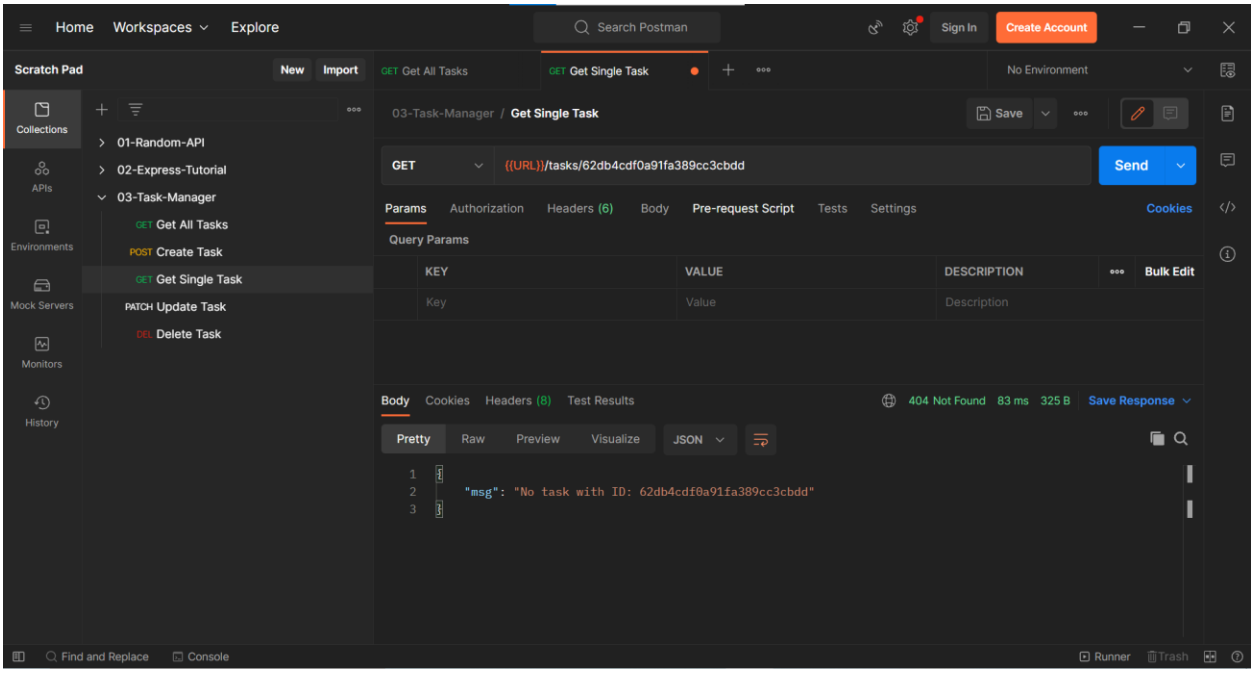
error-hanlder.js in middleware folder



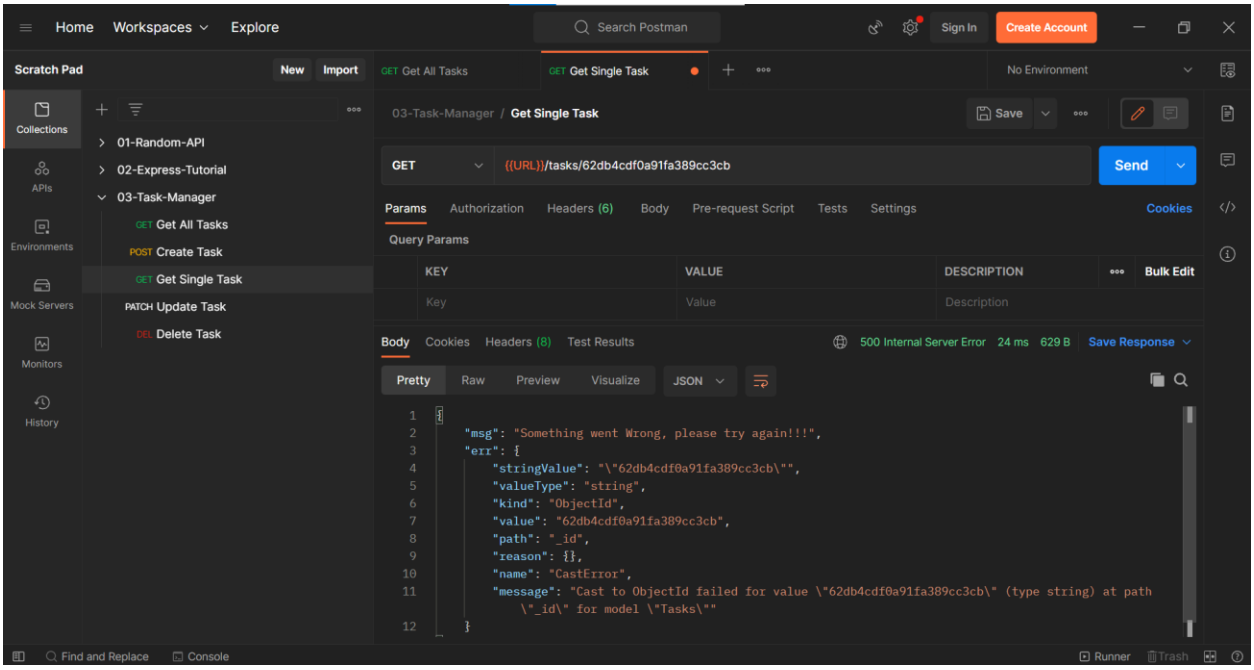
Using the newly created custom error function in the app.js with help of next() method to instantiate the Error object and send it to the default handler function in error-handler.js file



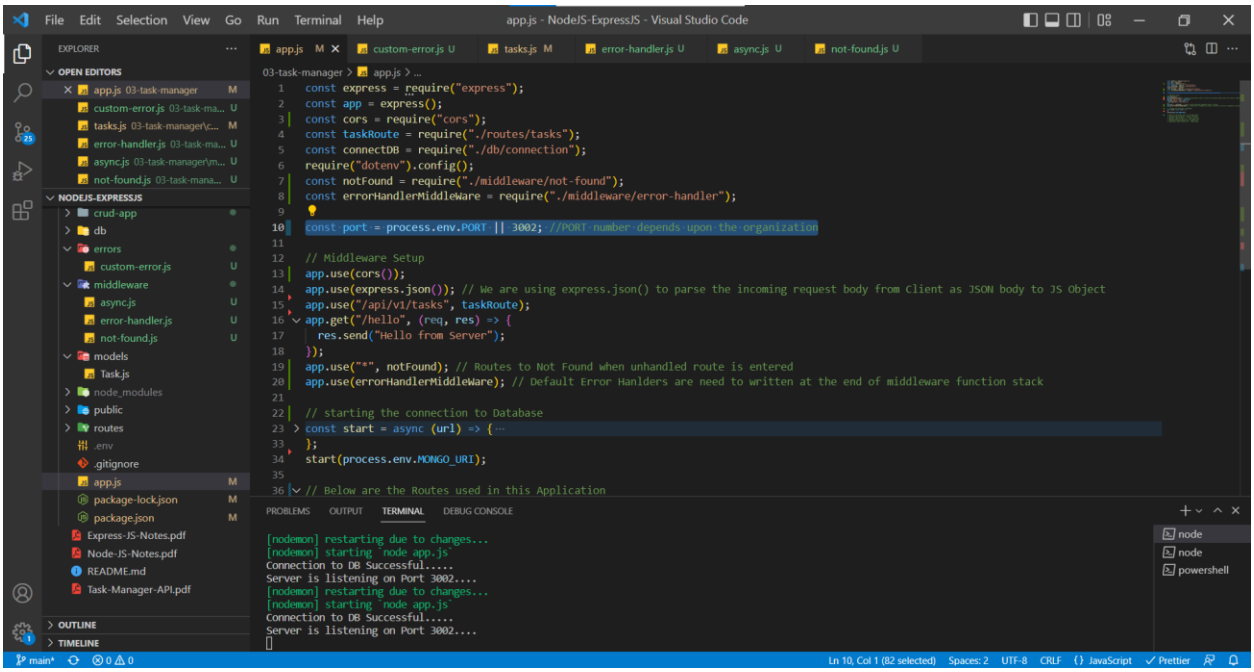
In Postman, Get Single Task – when there is no task with ID



In Postman, Get Single Task – when Cast Error or any other generic error (Not a custom Error)



Sometimes, applications require different Port Numbers, hence hardcoding the PORT number is not the right way. We need to follow the below format.



Let’s use a different PORT number to run our application