DDD - Task Manager

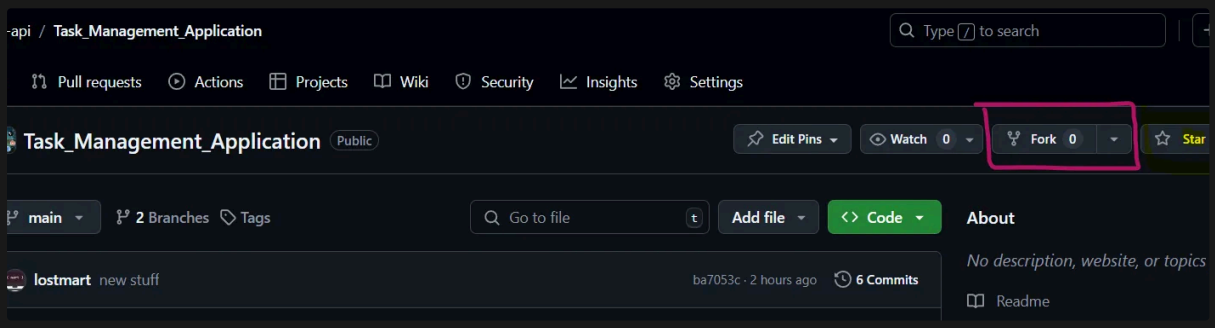
Repo: GitHub **GitHub - web-rest-api/Task_Management_Application**

Prerequisites:

- Node.js (version 14 or higher recommended)
- npm (Node package manager)
- VS Code **code.visualstudio.com**

Getting started

- Fork the repo to your repos or you can use your own



Directory Structure

```
task-manager/ | ├── domain/ | | ├── Task/ | | | ├── Task.js | | | ├──
TaskService.js | | ├── Category/ | | | ├── Category.js | | | ├──
CategoryService.js | | ├── User/ | | | ├── User.js | | | ├──
UserService.js | ├── infrastructure/ | | ├── Database.js | ├── app.js |
├── index.js
```

I'll be working on a Task manager but you can work on whatever topic you feel more suitable or you like best. Cars, Hobbies, Sports, Computer Games, etc. The idea is to make a "something" manager. Each of these collections is going to have Categories and Users. These are in order to sort them later on

Adding Routes

Inside the folder `routes` we'll add a file called `taskRoutes.js` or whatever your domain is (user, car, coffee, etc)

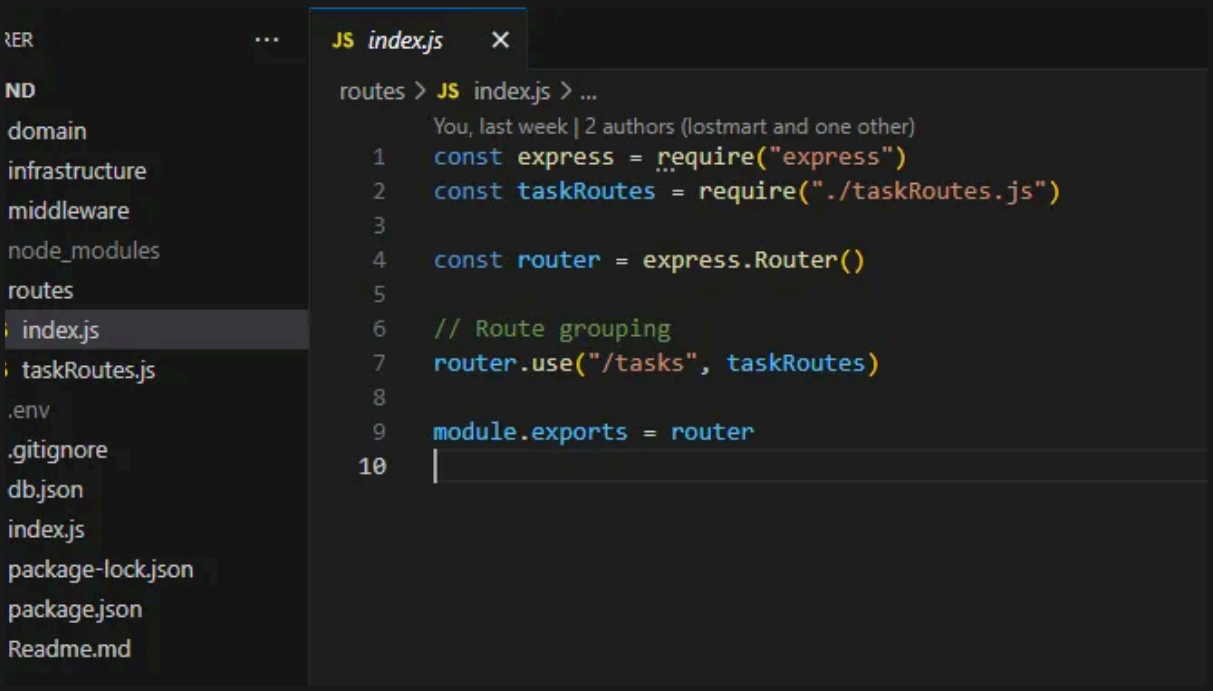
```
const express = require("express") const taskRouter = express.Router()
taskRouter.post("/", (req, res) => { res.json({ msg: "task post reached
...", }) }) module.exports = taskRouter
```

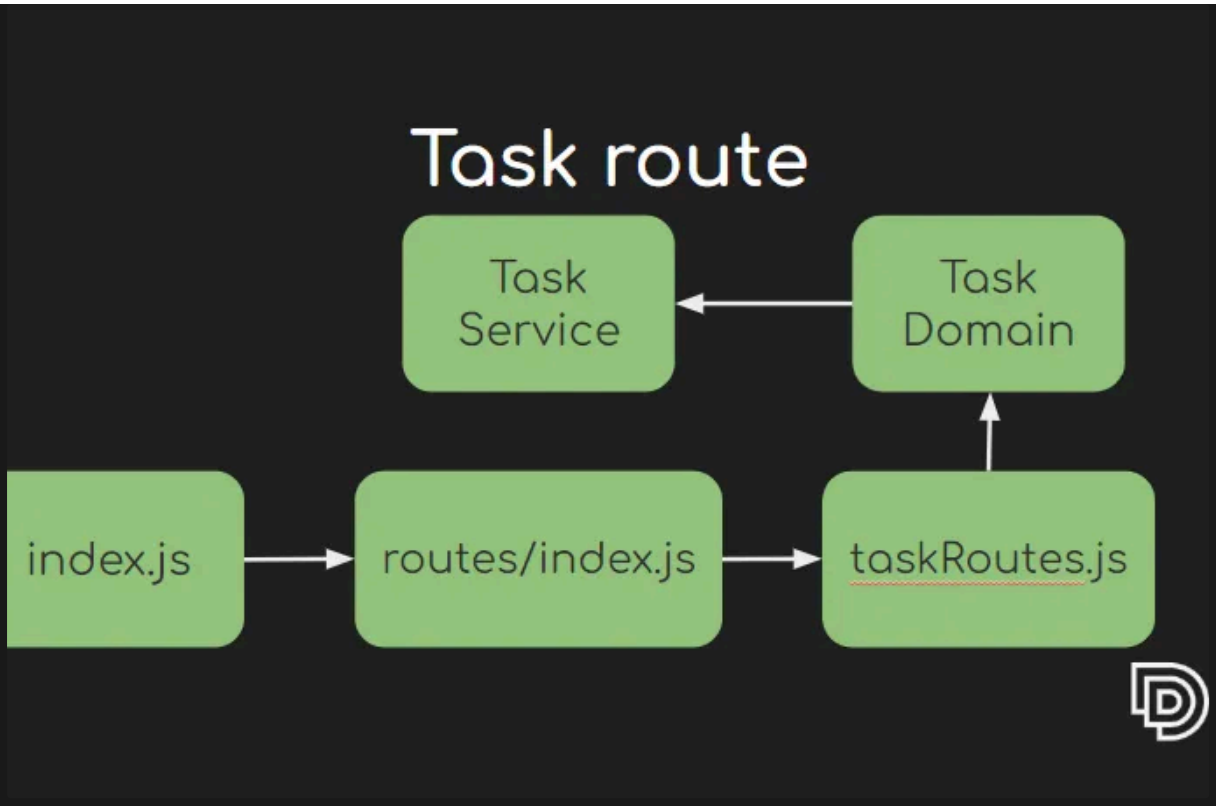
We import the main routes in the router variable

We create a file called `index.js` for the routes. We are going to have more than one route here

```
const express = require("express") const taskRoutes =
require("./taskRoutes.js") const router = express.Router() // Route
grouping router.use("/tasks", taskRoutes) module.exports = router
```

Here's the folder structure





Adding a **TaskService** to your API is a common practice in Domain-Driven Design (DDD) and layered architectures. It separates business logic (how tasks are created, managed, validated, etc.) from the controller logic, which primarily handles request/response processing. This separation has several key benefits:

- 1. Separation of Concerns
- 2. Encapsulation of Business Logic
- 3. Reusability and Consistency
- 4. Scalability and Flexibility

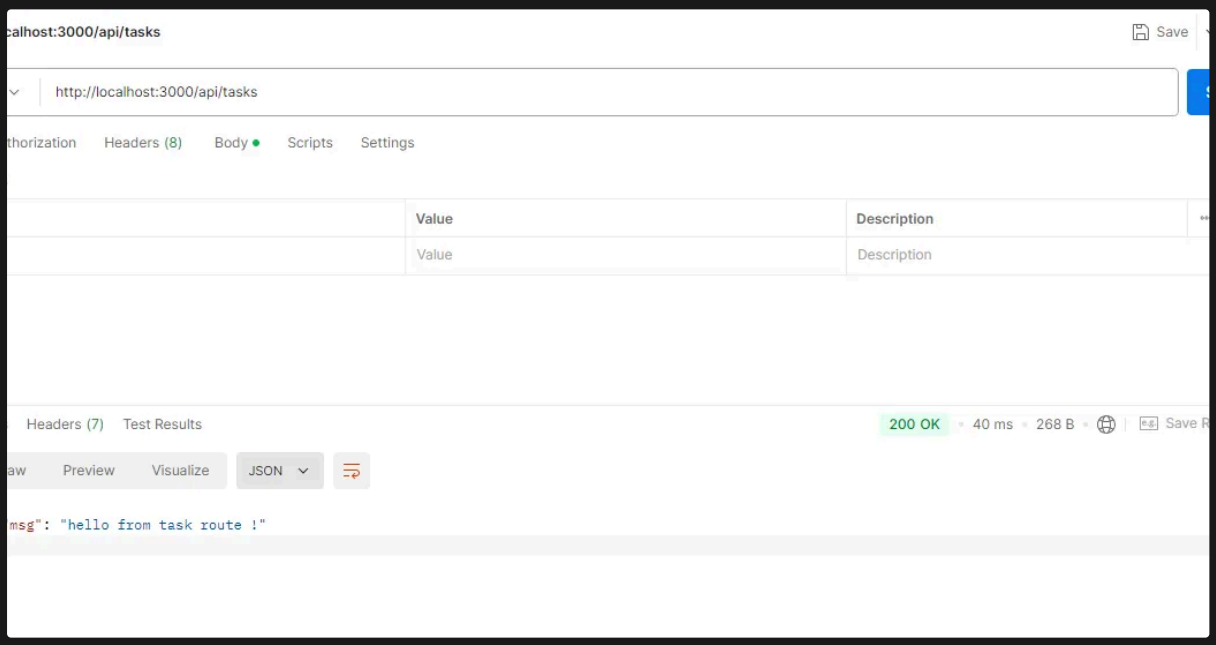
Middleware

In Node.js and Express, *middleware* refers to functions that execute during the request-response cycle. Middleware functions have access to the request and response objects and can modify them or end the response, or pass control to the next middleware function in the stack. They are used for tasks like logging, authentication, data parsing, error handling, and more, making them essential for managing app workflows.

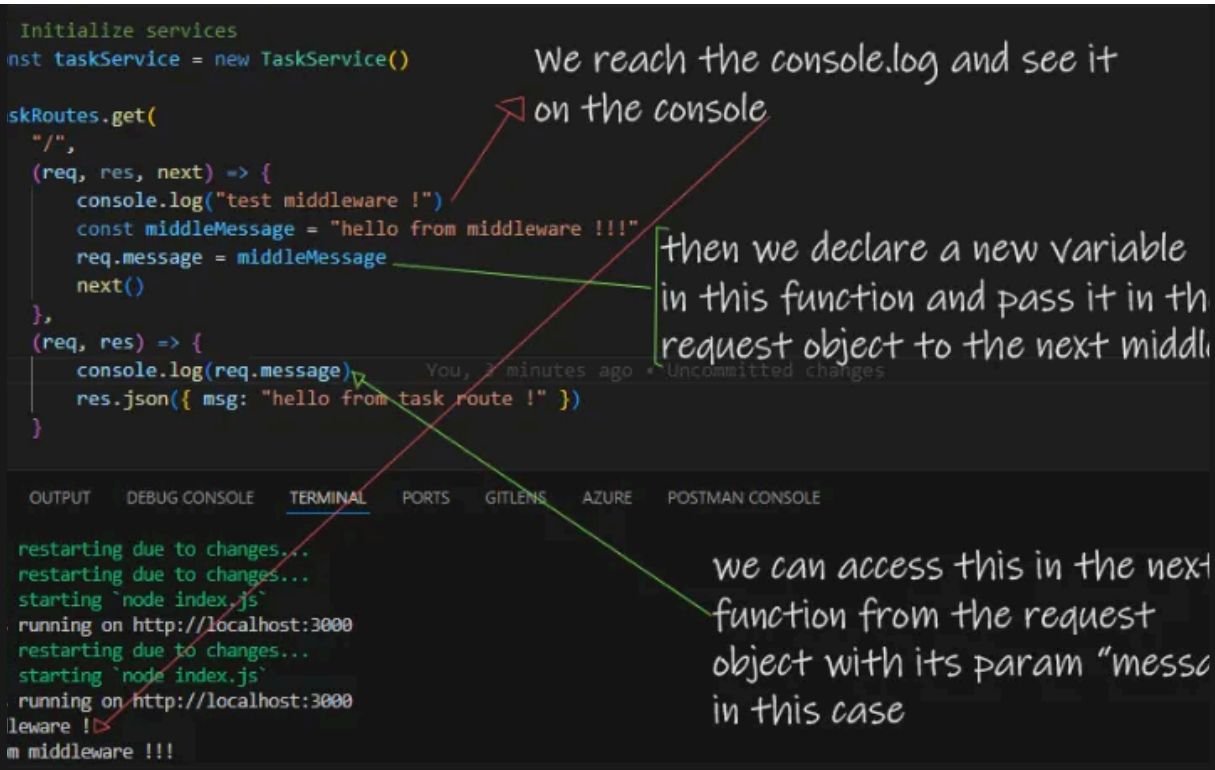
To test a piece of middleware and see it in action we can try the following manipulation. In the `taskRoutes.js` we can create a new endpoint as follows:

```
taskRoutes.get( "/", (req, res, next) => { console.log("test middleware
!") const middleMessage = "hello from middleware !!!" req.message =
middleMessage next() }, (req, res) => { console.log(req.message)
res.json({ msg: "hello from task route !" }) } )
```

then visit this with POSTMAN



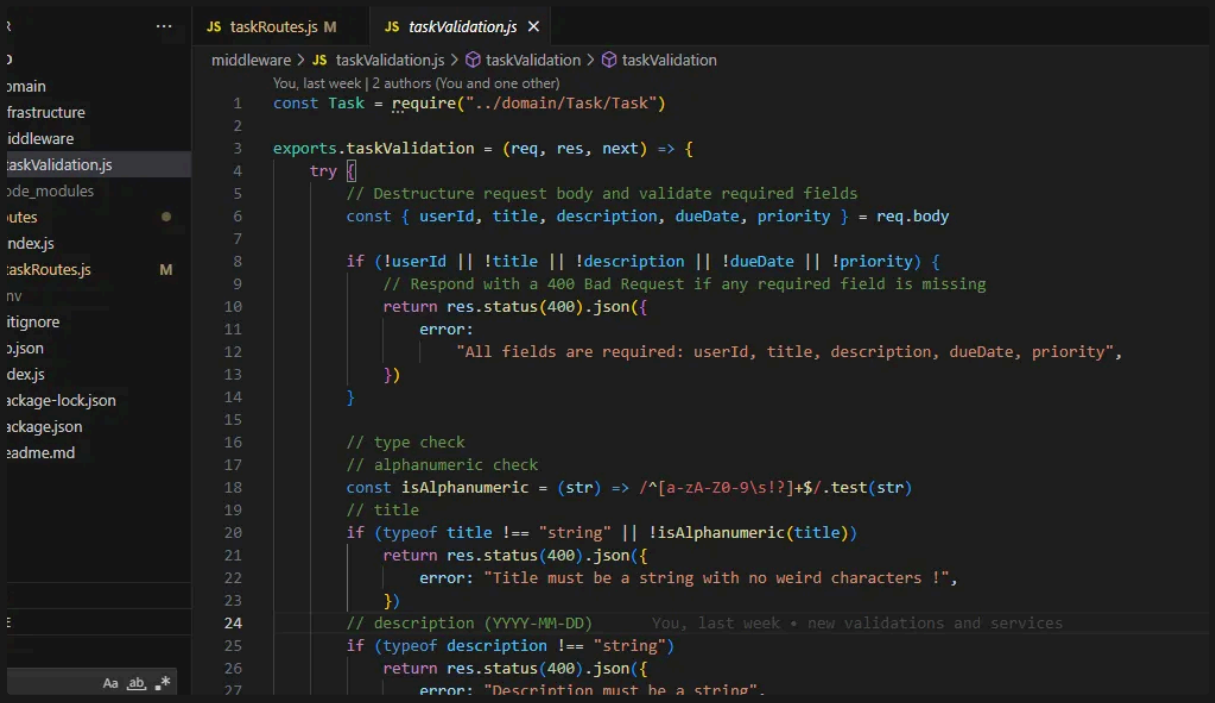
Passing data from one function to the next using the request object



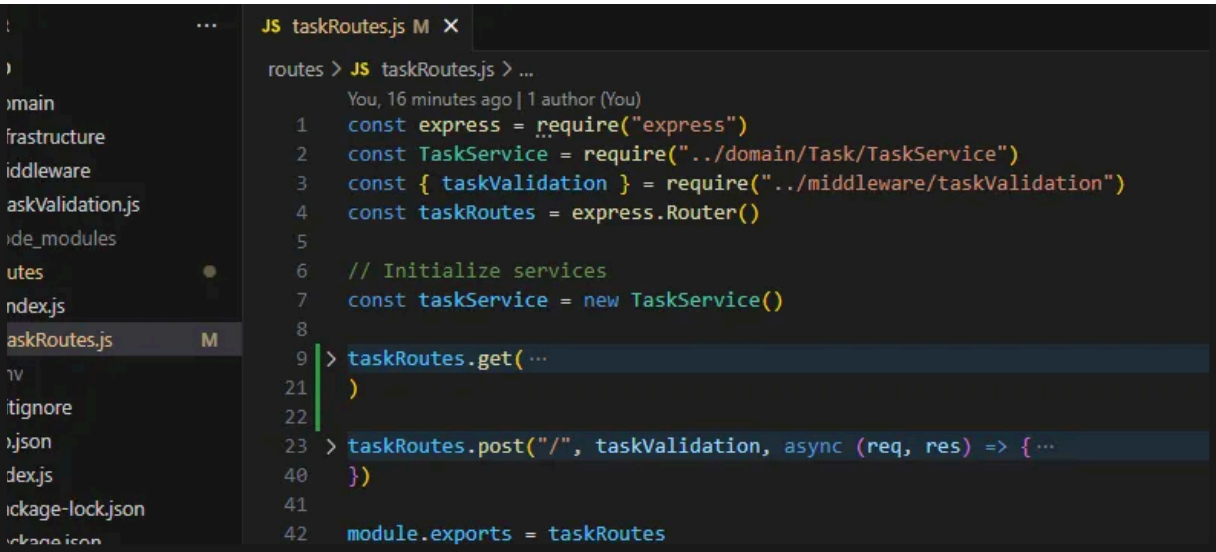
With this in mind, we are going to create a middleware to validate whenever the user sends data to our API

We can create a file called `taskValidation.js` and do all kinds of validation there: all required fields, types (string, boolean, number, etc)

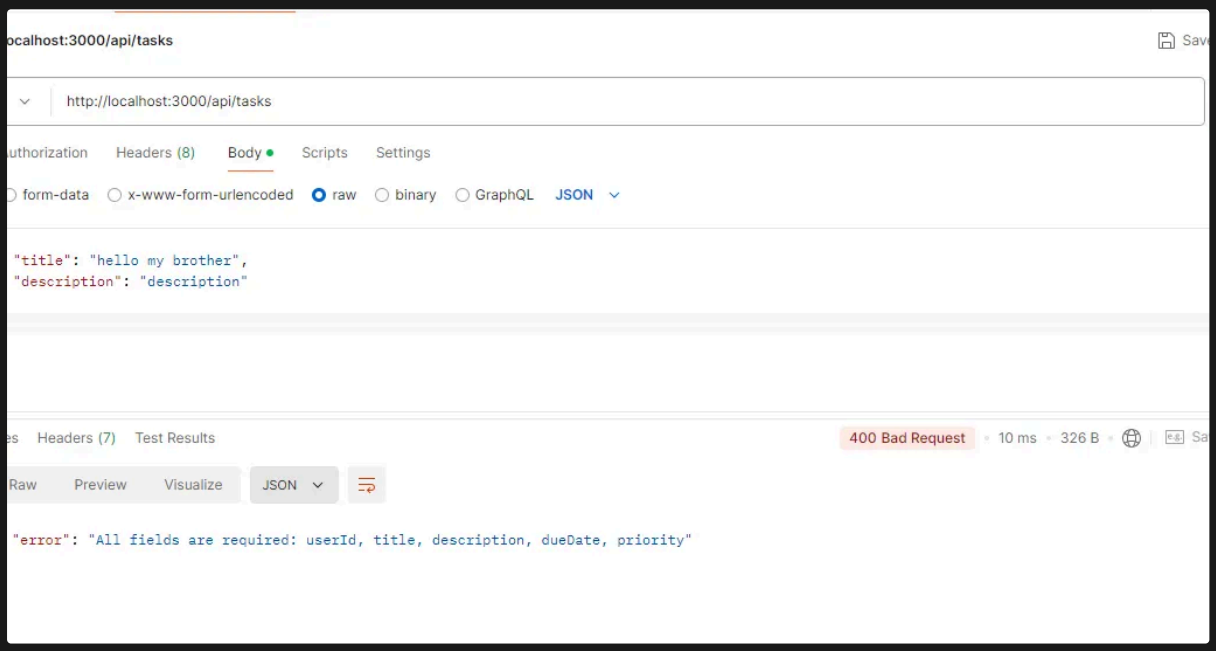
Here's an idea of it



Then we import this to our task router and add it as a middleware. Here's the `taskRoutes.js` whenever the POST endpoint is reached



That means that if we try this with POSTMAN a 400 error will be sent back



Json-server

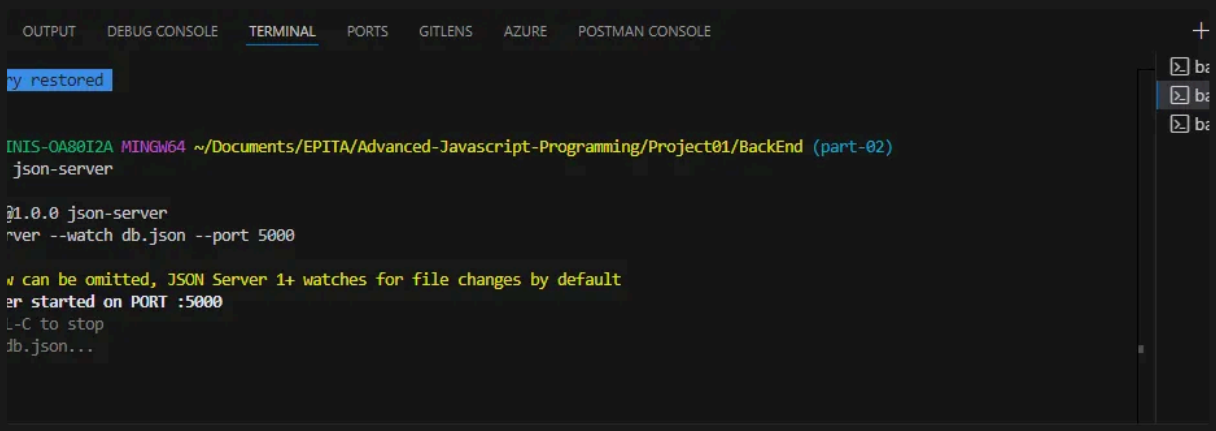
https://www.npmjs.com/package/json-server

```
npm install json-server
```

We are going to use json-server to quickly prototype an sql service we are going to connect to our API.

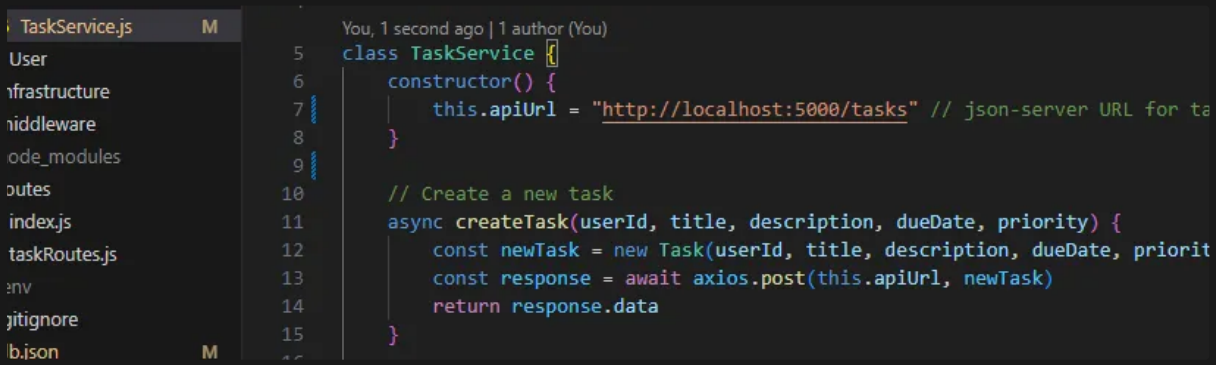
We can add a scrip to run it on the side: `"json-server": "json-server --watch db.json --port 5000"`

Once installed open a new terminal on the side and run



We are going to run this in the `taskService.js`

Task Service



We are also using the library `axios` to interact with our `json-server`. The method `createTask` is async since its connecting to the server in order to create a new task

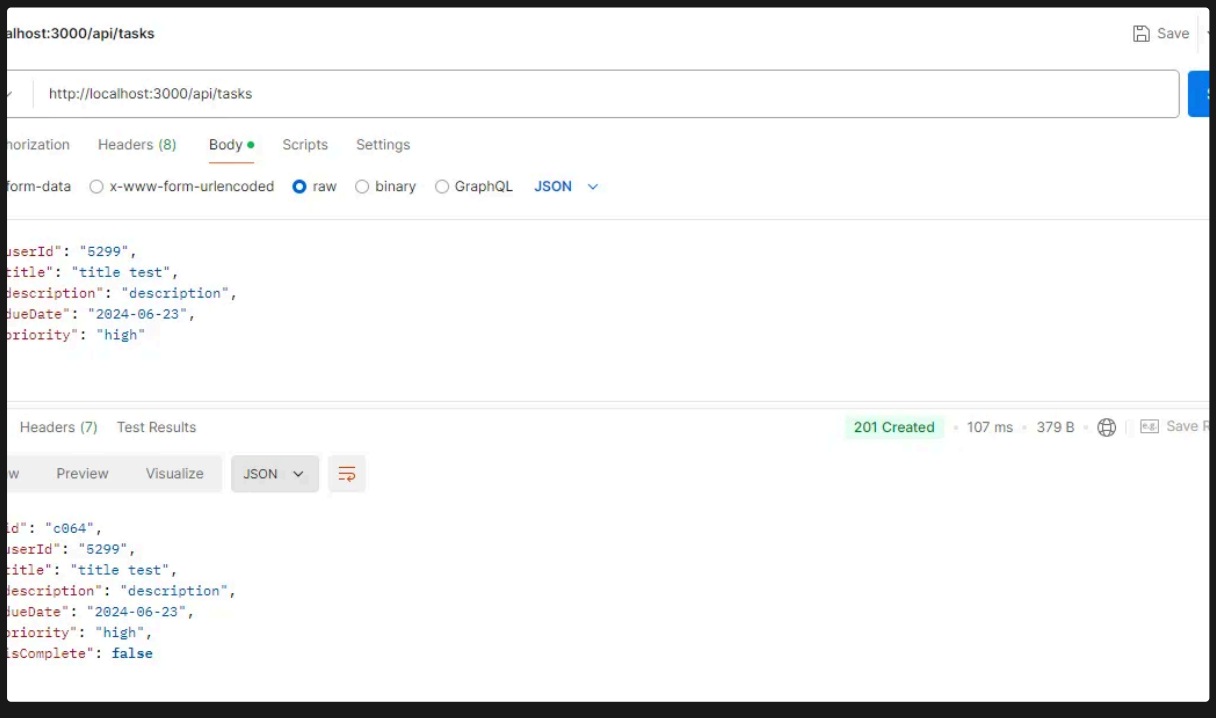
We can also store the apiUrl in an `.env` file so that if we change environments to deployment we only need to change it in one place only

```
.env
PORT=3000
TASK_SERVICE_URI=http://localhost:5000/tasks
```

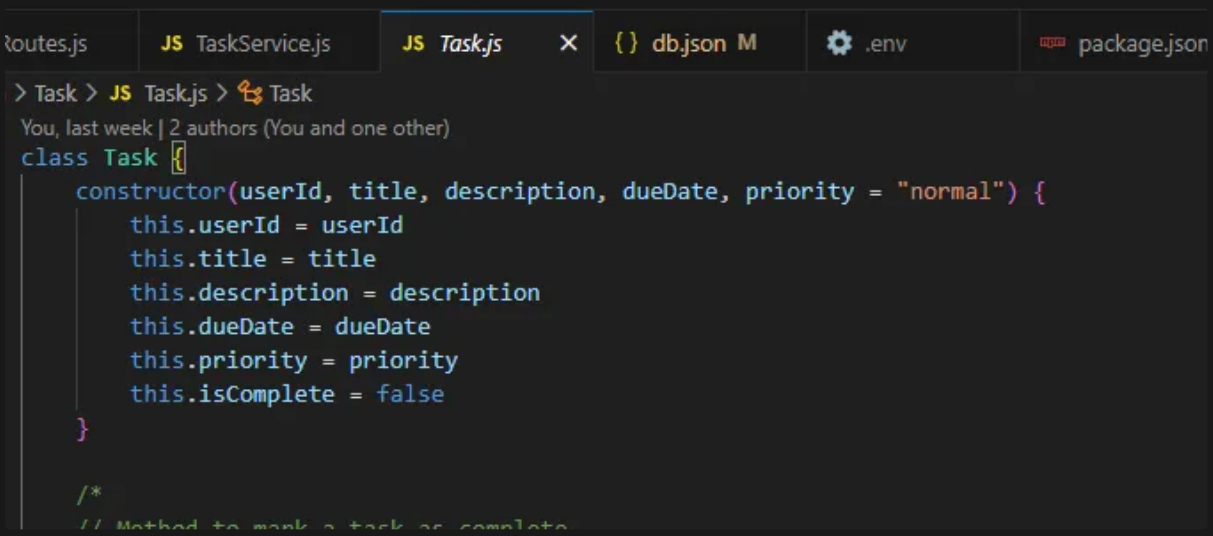
```
domain > Task > TaskService.js > TaskService > completeTask
You, last week | 1 author (You)
const axios = require("axios")
const Task = require("./Task")
require("dotenv").config() // Load environment variables

You, last week | 1 author (You)
class TaskService {
    constructor() {
        this.apiUrl = process.env.TASK_SERVICE_URI // json-server URL for tasks
    }

    // Create a new task
    async createTask(userId, title, description, dueDate, priority) {
        const newTask = new Task(userId, title, description, dueDate, priority)
        const response = await axios.post(this.apiUrl, newTask)
        return response.data
    }
```

We have the `Task` class coming from the domain folder. We can create an instance of it in the `createTask` method

We can test in POSTMAN



We can see that the Task class as a default priority of “normal” and a property `isComplete` that defaults to false



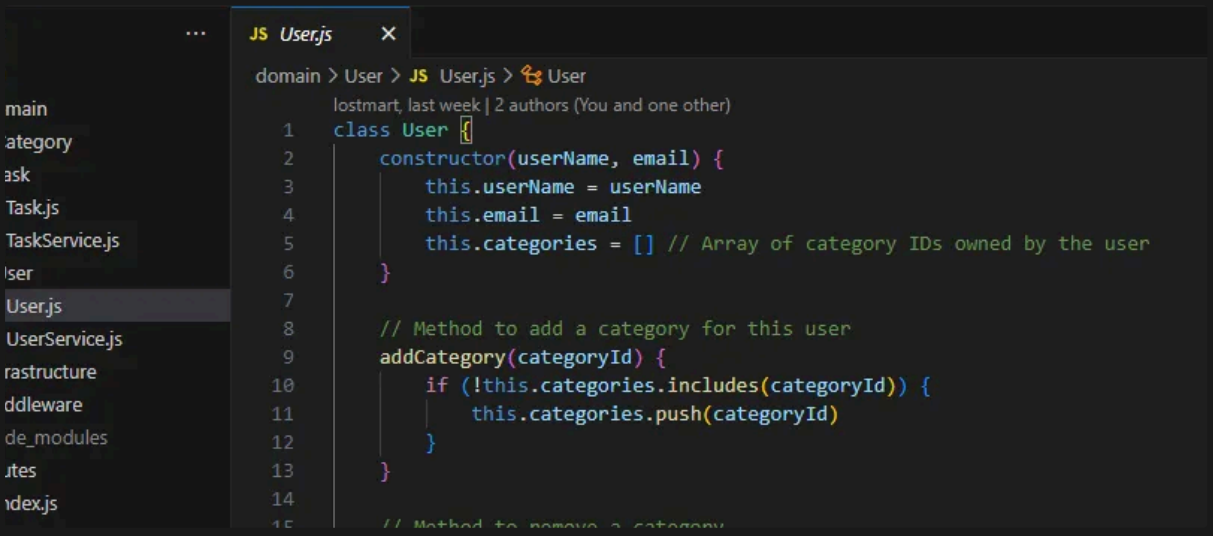
This is the required data to test our Task service

```
{ "userId": "5299", "title": "title test", "description": "description",
"dueDate": "2024-06-23", "priority": "high" }
```

The `userId` is made up, it would be a good idea to take care of this

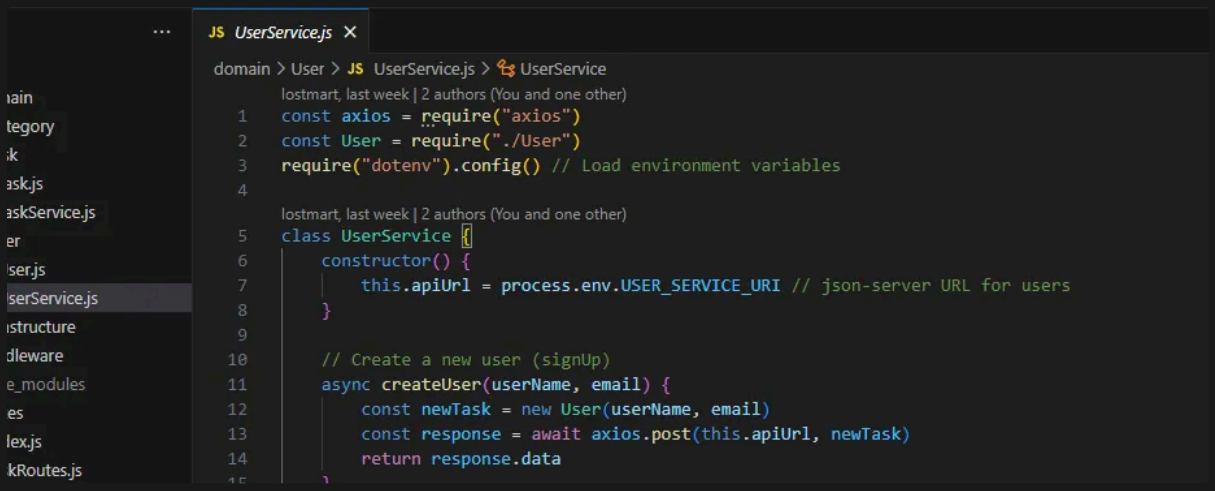
Users Service

We can start in the User domain

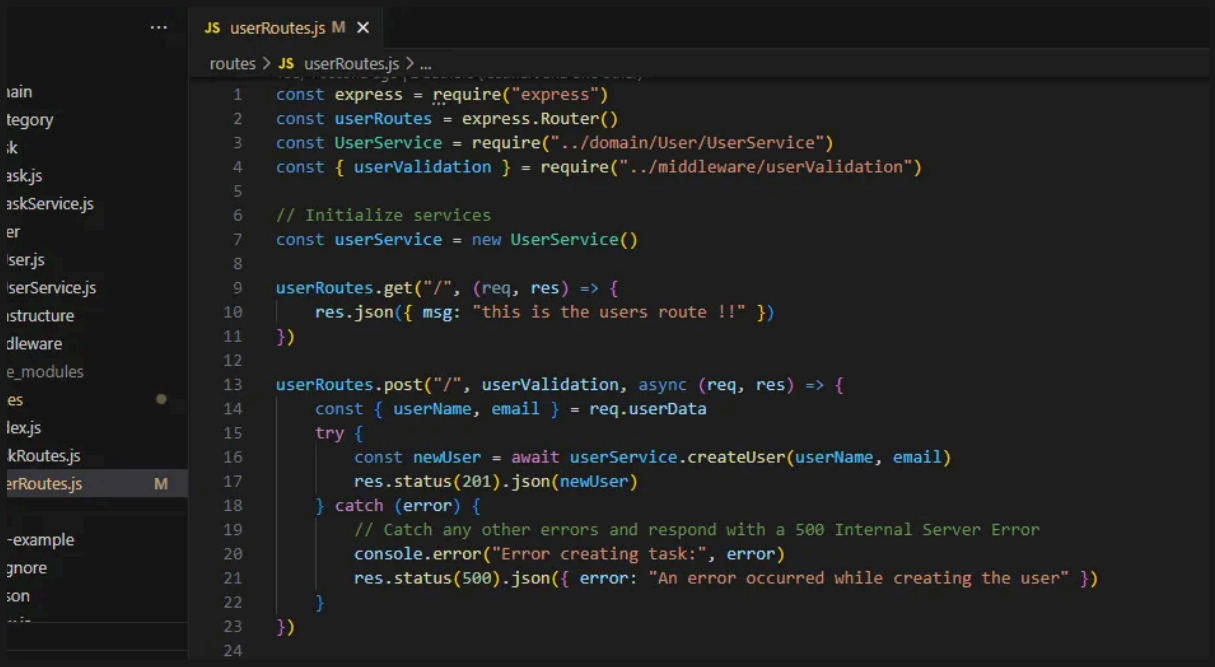


Our user will have a user name and an email for now and an empty array for categories property

We also need a user service that will create an instance of the User class and store it in the json-server using axios

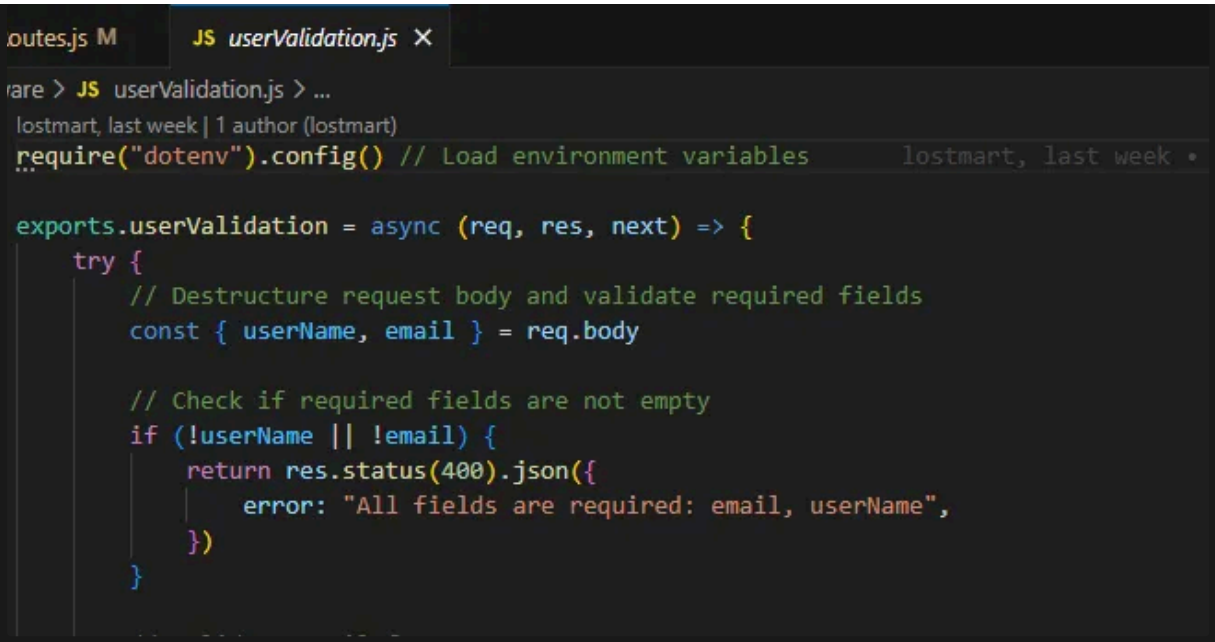


In the user's route we are going to handle this as follows:

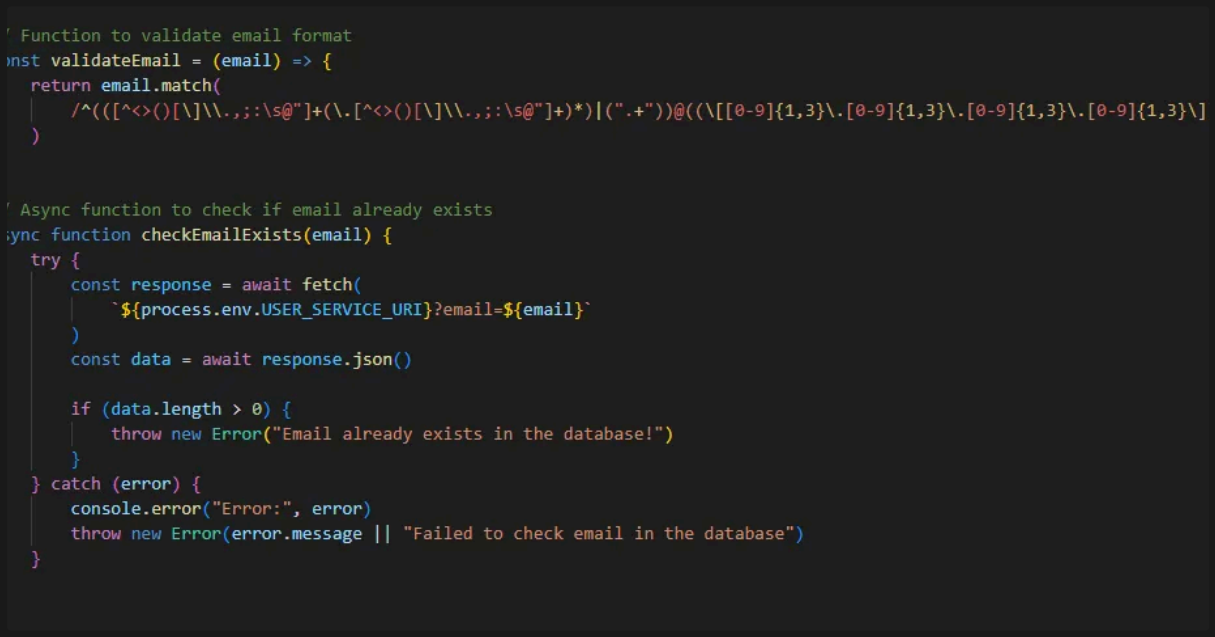


Notice that we also have a `userValidation` middleware

This middleware is a bit more complex, first we check for empty fields ... easy enough

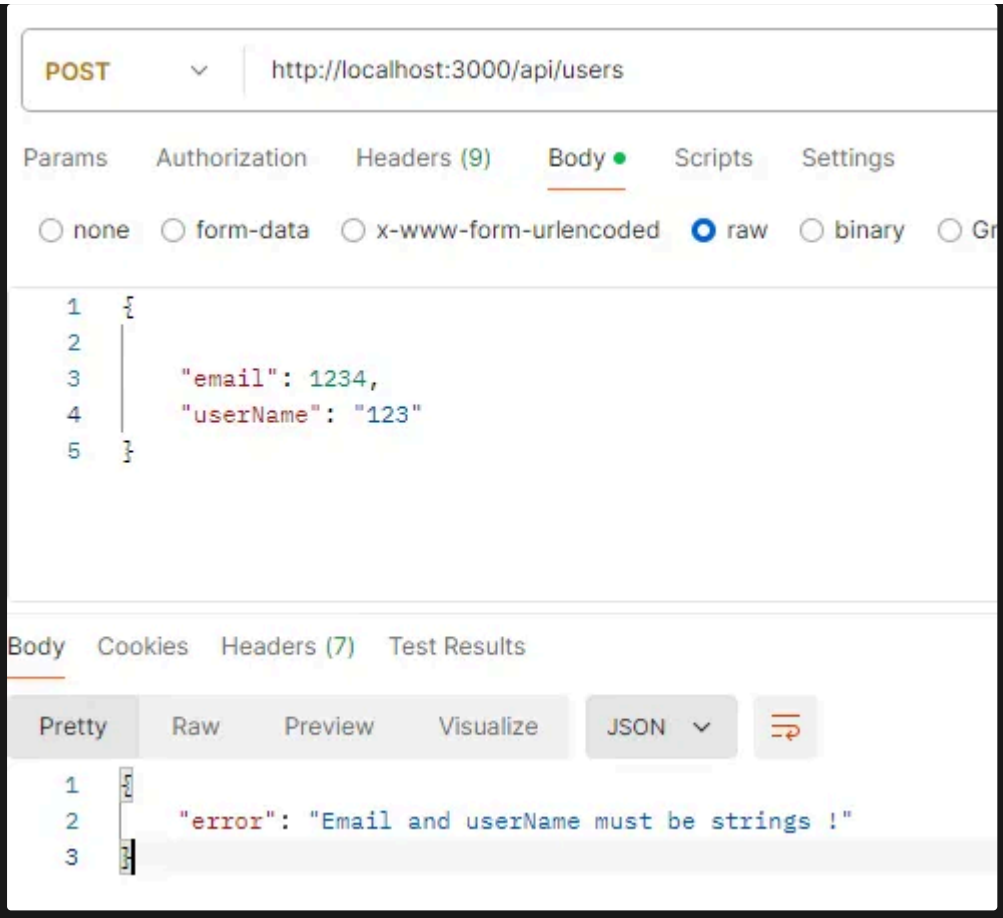


Then we can check types but after that we can also check the email format using JavaScript regular expressions and if the email already exists in our database. The function is called `checkEmailExists`

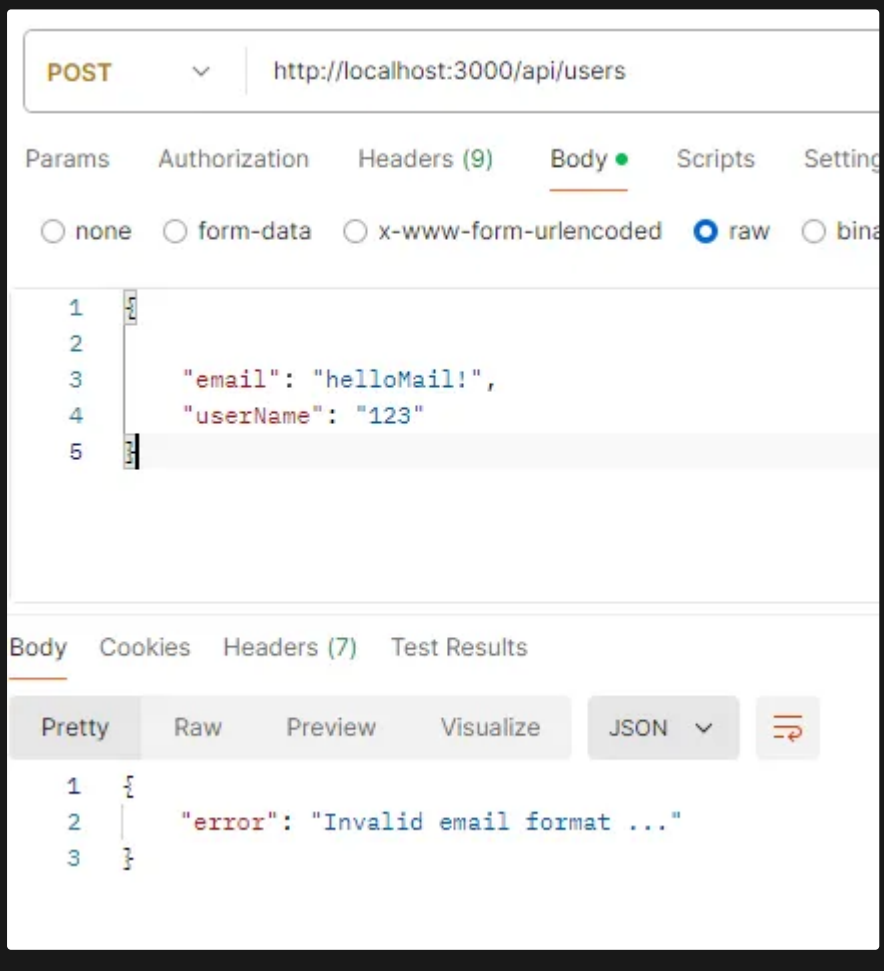


Than you can make your tests using POSTMAN

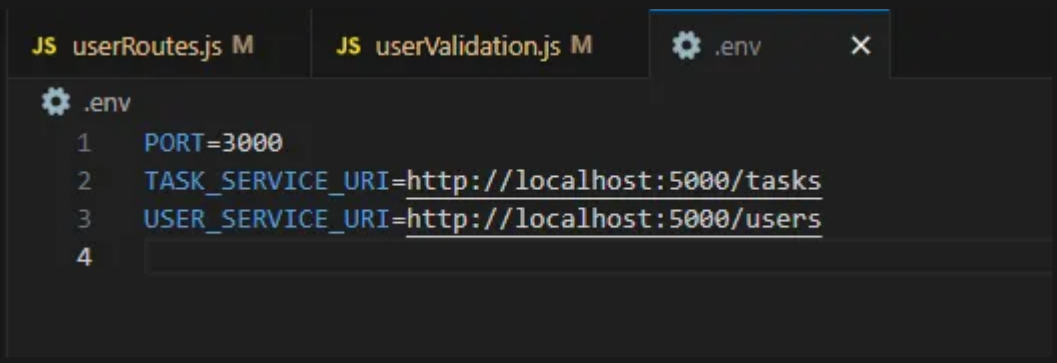
No strings



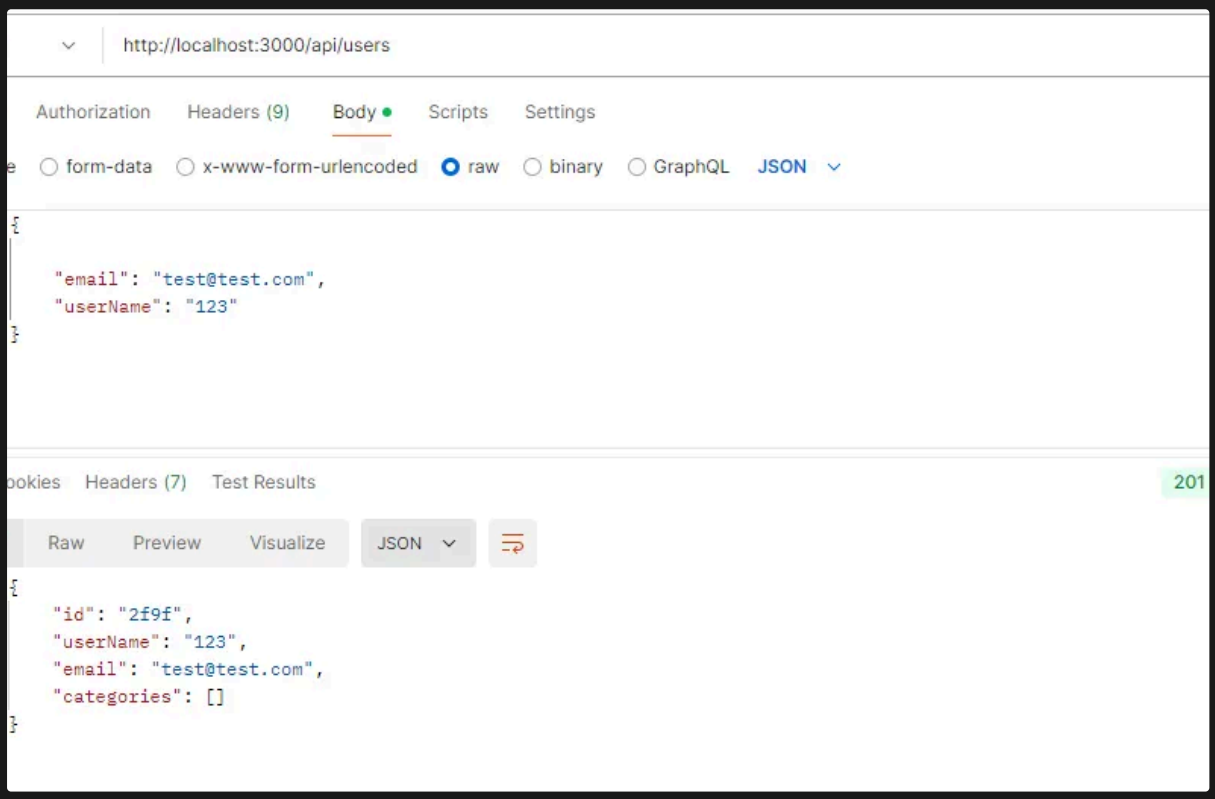
or invalid formatting, or empty



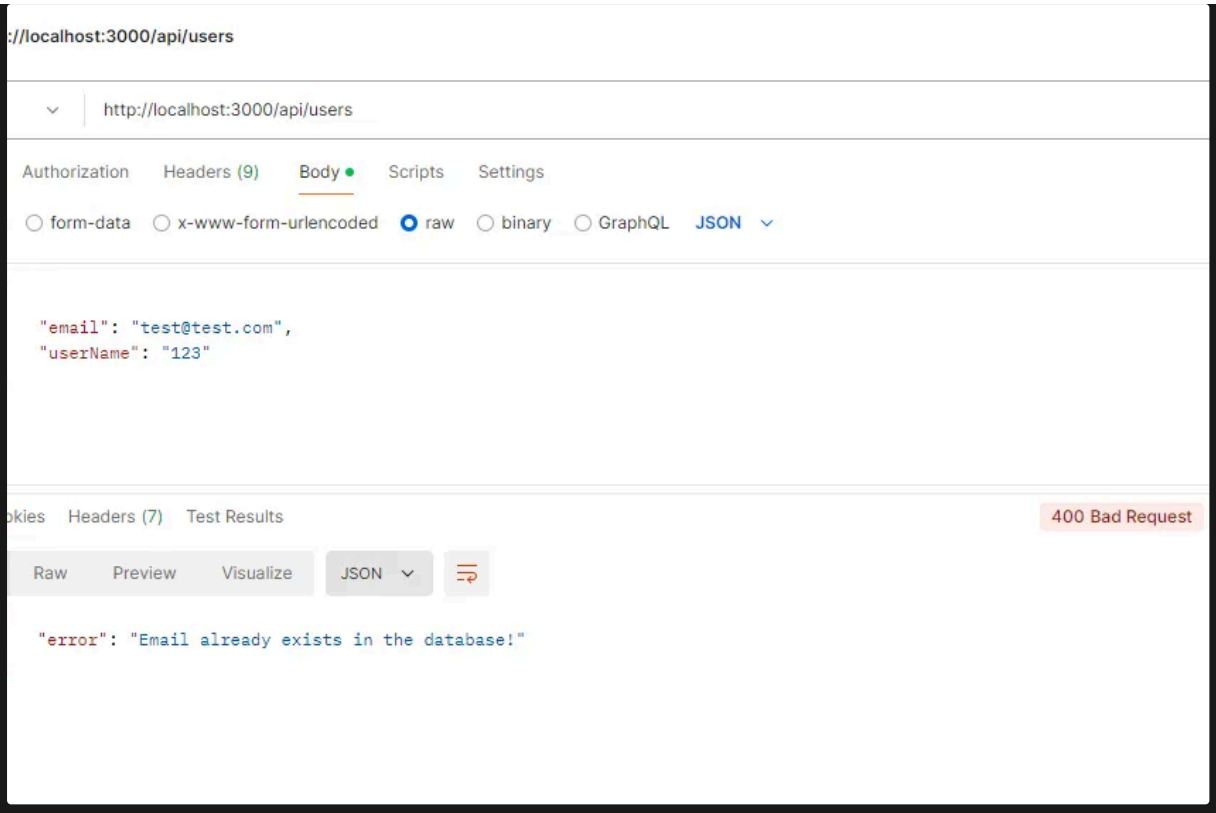
I also added the users' URI for the json-server



We should be able to create a new user if we pass the a correct formatted email and and a user name



and if you send the same request with the exact same email ...

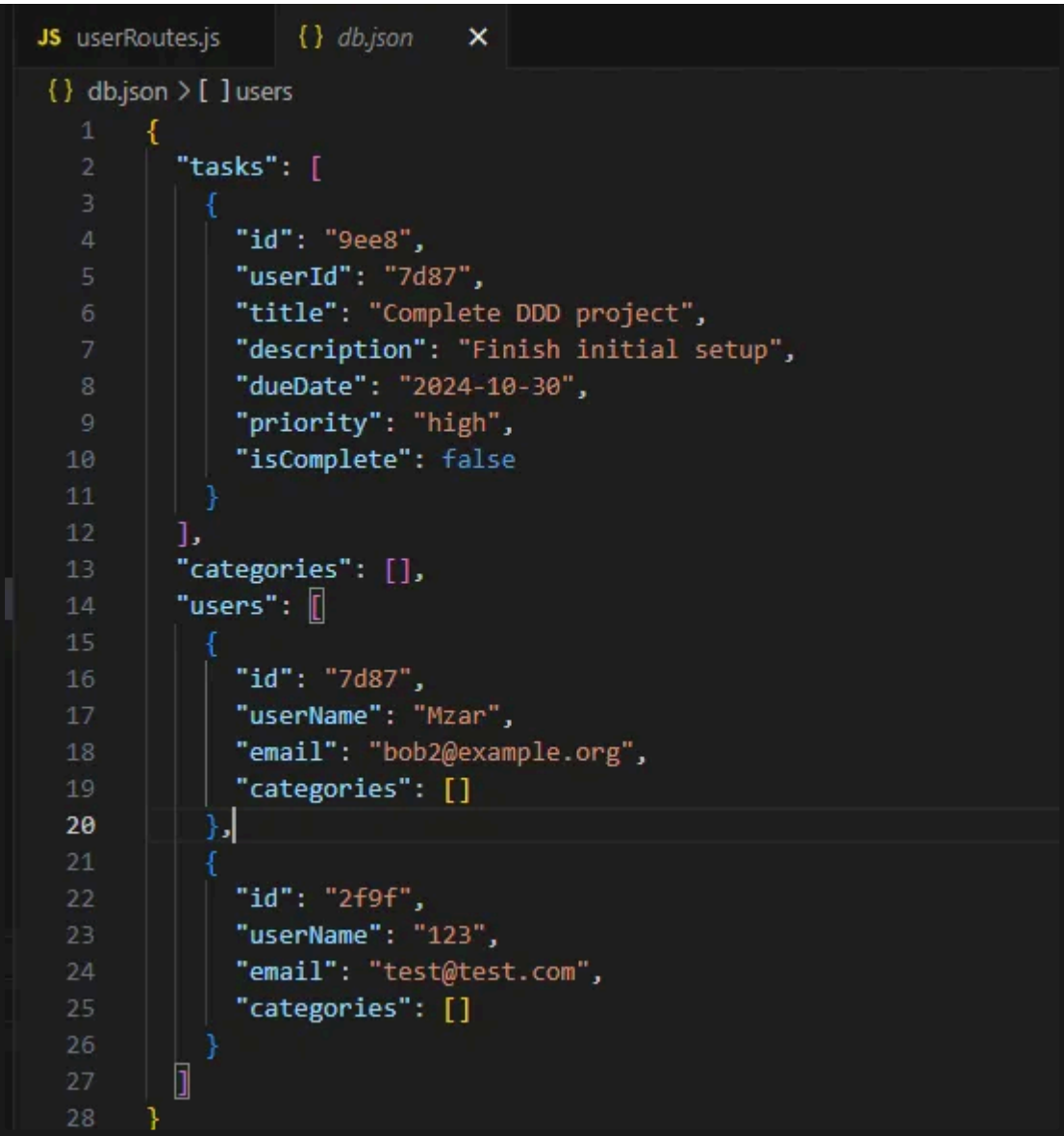


Get task by user ID

Now that we have a user ID, we can create tasks using its ID so that later we can bring only the tasks linked to this user

Maybe we should empty our `db.json` file, so we start fresh

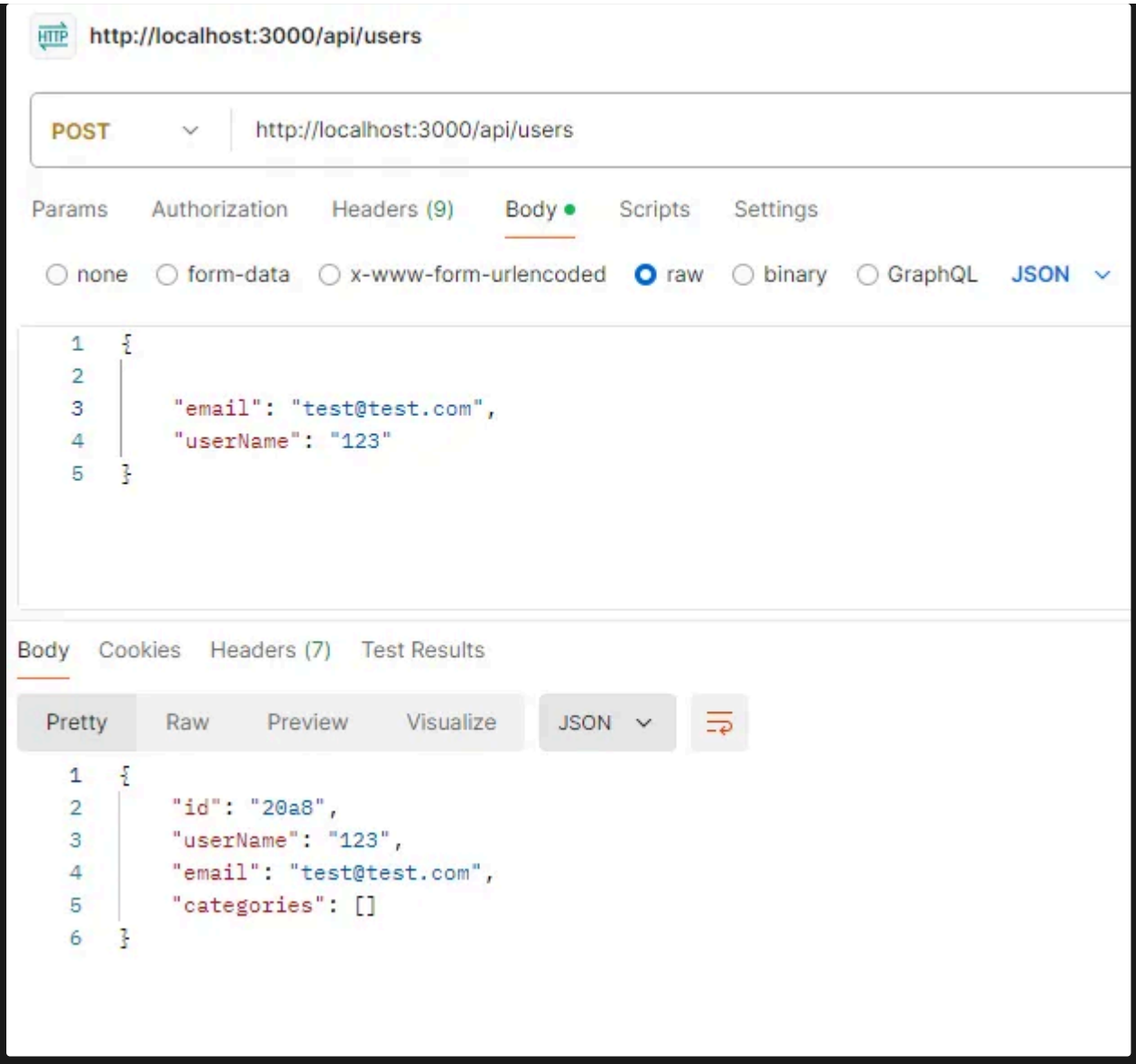
I have this dummy data that I am going to delete. Here is what it looks like:



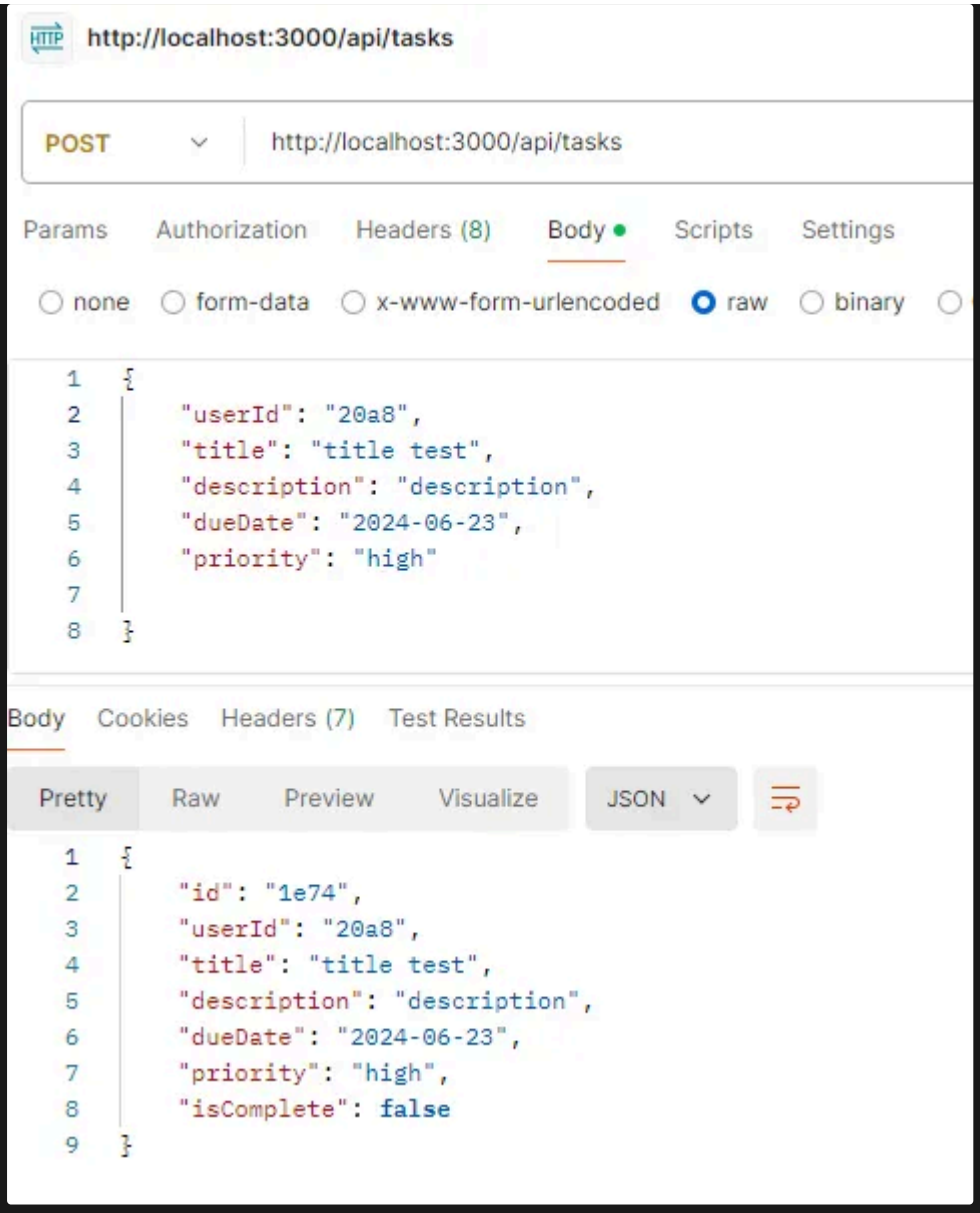
and this is empty, ready for the tasks, categories and users

```
{ "tasks": [], "categories": [], "users": [] }
```

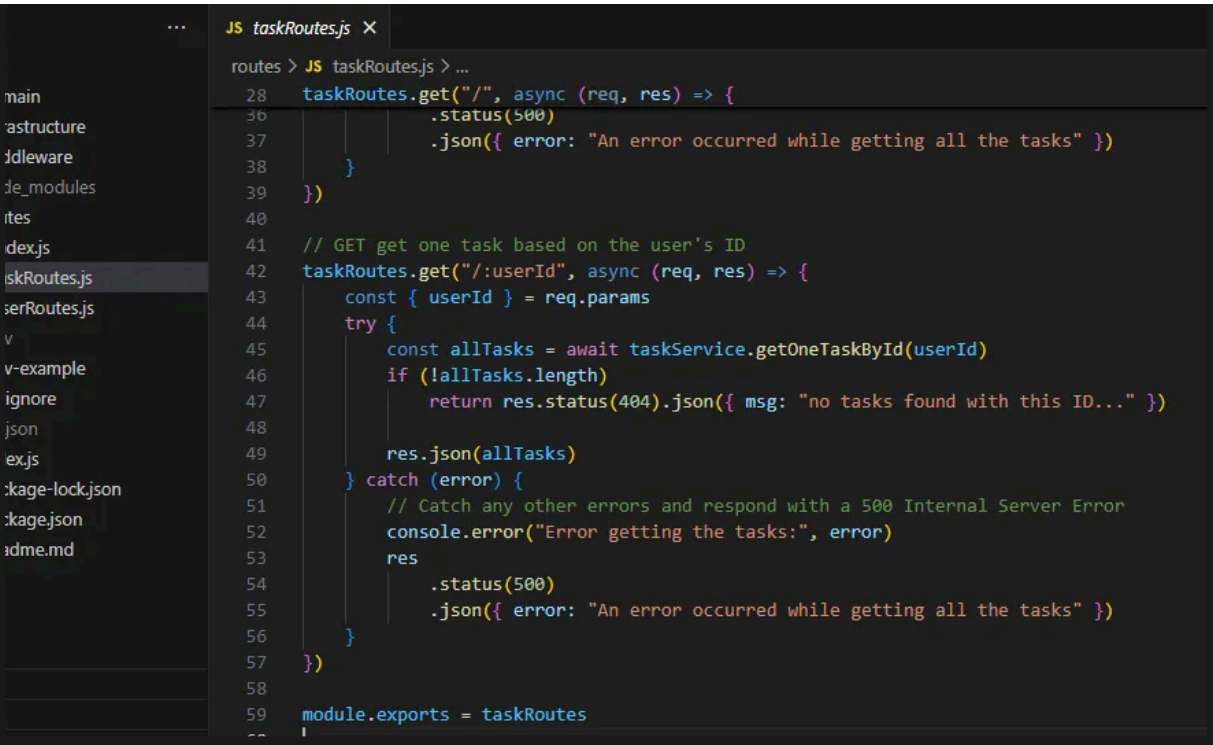
We create a new user and use its ID to create a new task


http://localhost:3000/api/users
POST
http://localhost:3000/api/users
Params
Authorization
Headers (9)
Body
Scripts
Settings
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
{
"email": "test@test.com",
"userName": "123"
}
Body
Cookies
Headers (7)
Test Results
Pretty
Raw
Preview
Visualize
JSON
{
"id": "20a8",
"userName": "123",
"email": "test@test.com",
"categories": []
}

same user ID

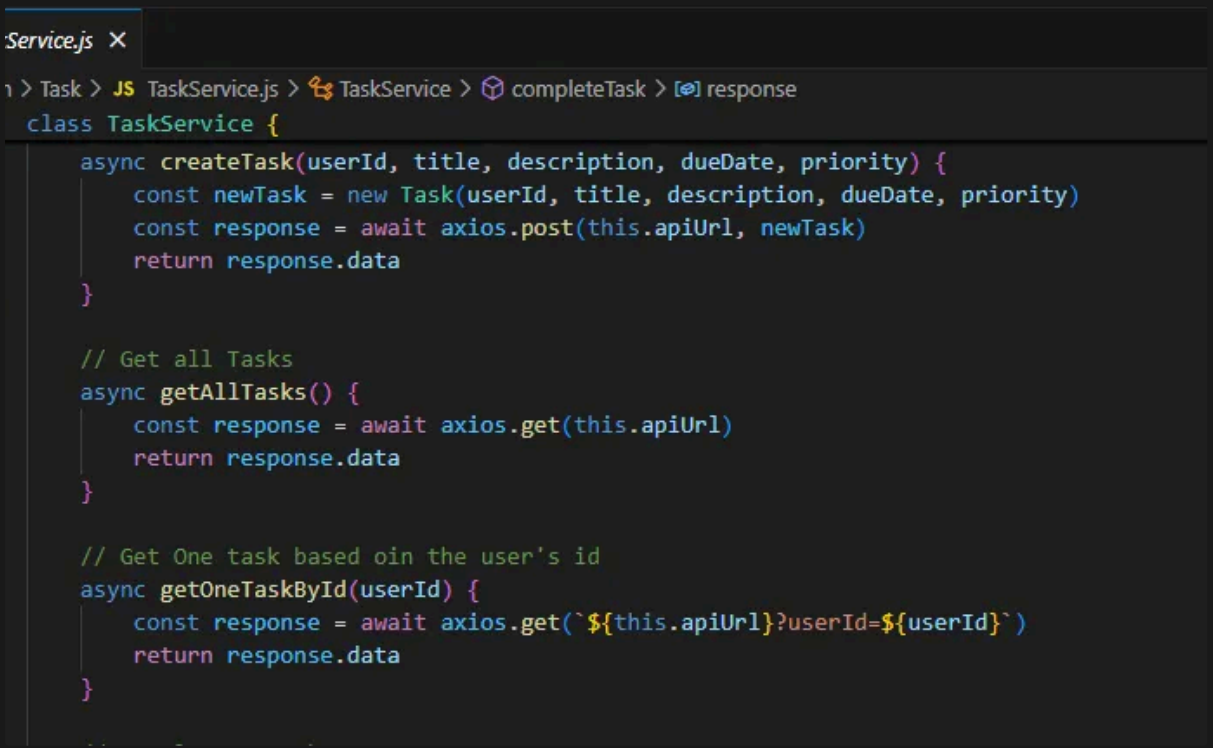


In the task route we should have something like this



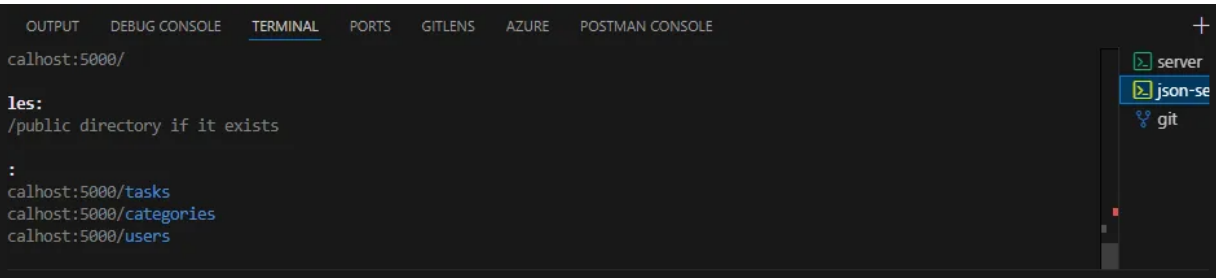
We accept a param called `userId` and we pass it to our task service class. This class has a method called `getOneTaskById` We call this in a asynchronous function

The task service will take care of this as follows:

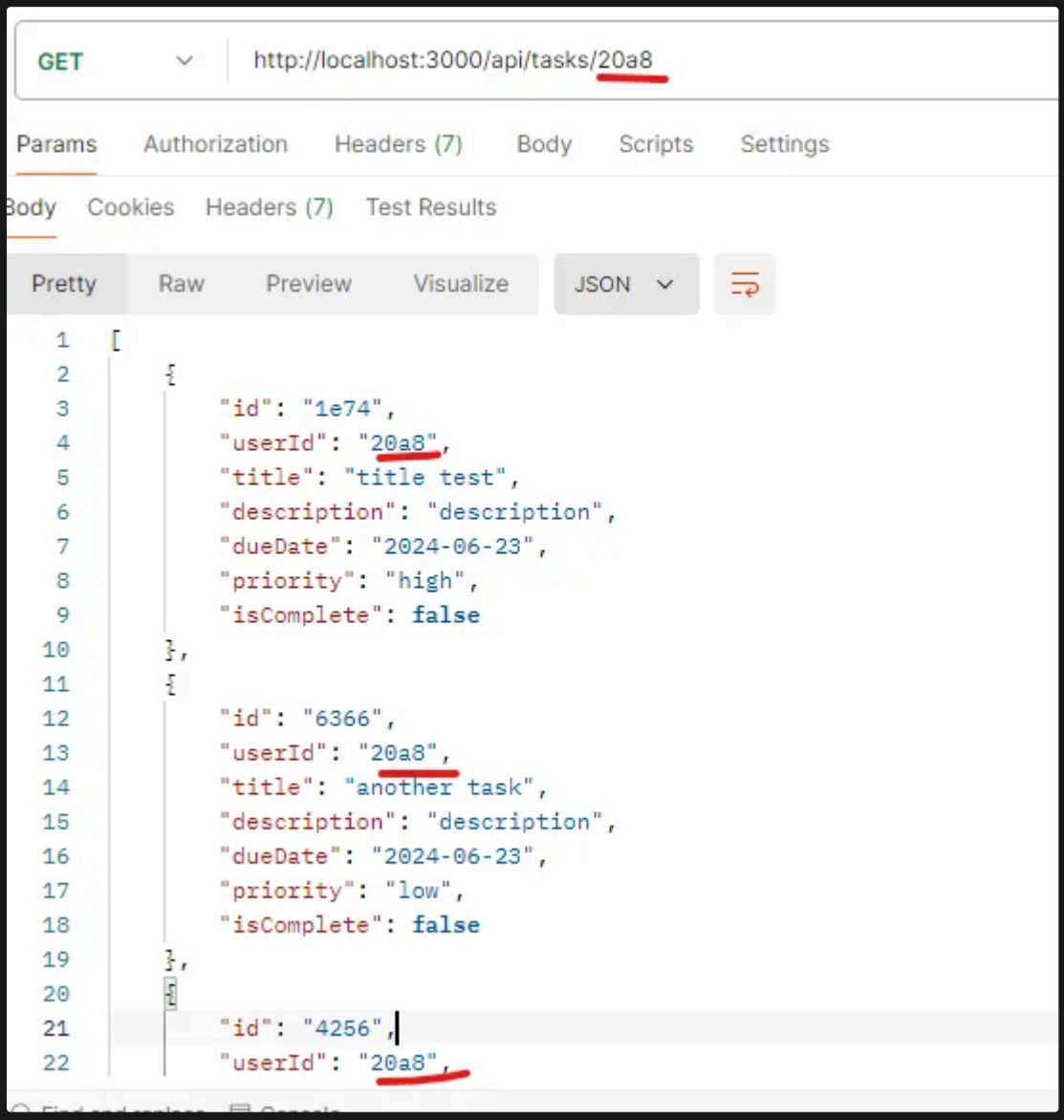


We use `axios` to query our `json-server`

BTW I like to use VS code terminal as follows, maybe this tip can help you

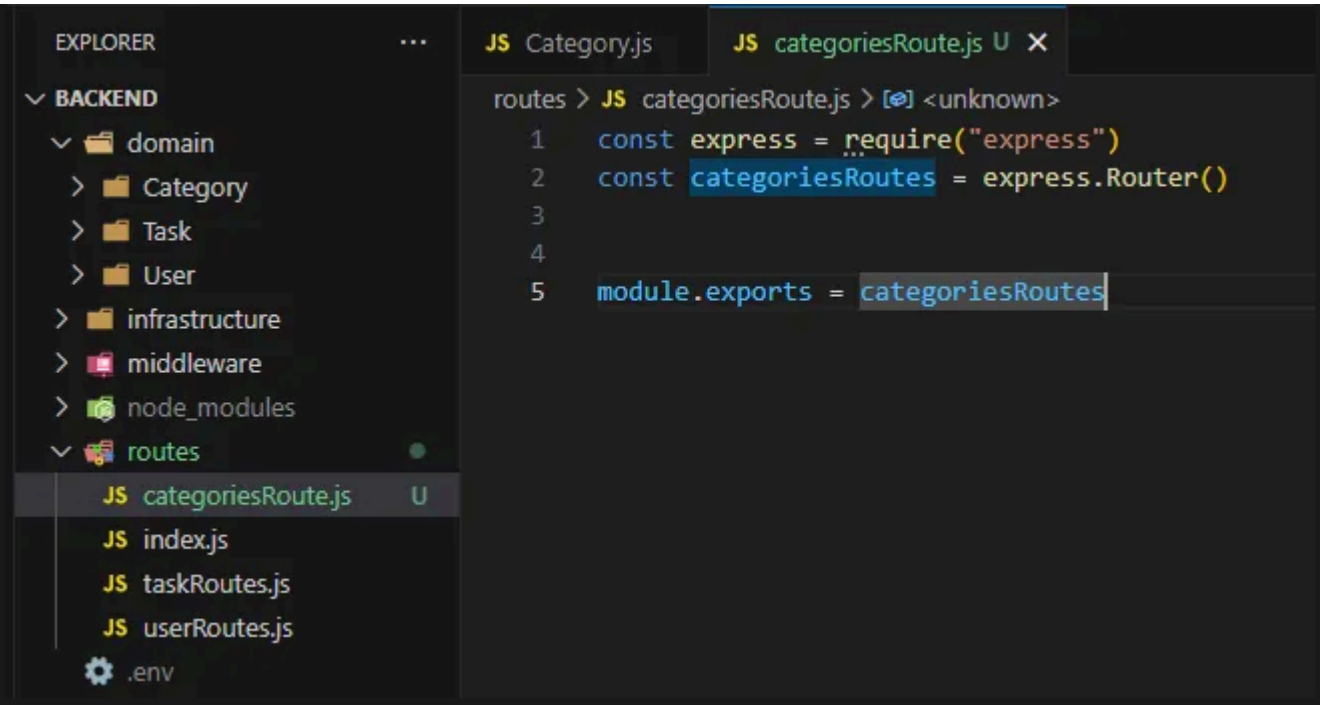


I created three tasks with this user and then we should be able to retrieve it from this endpoint

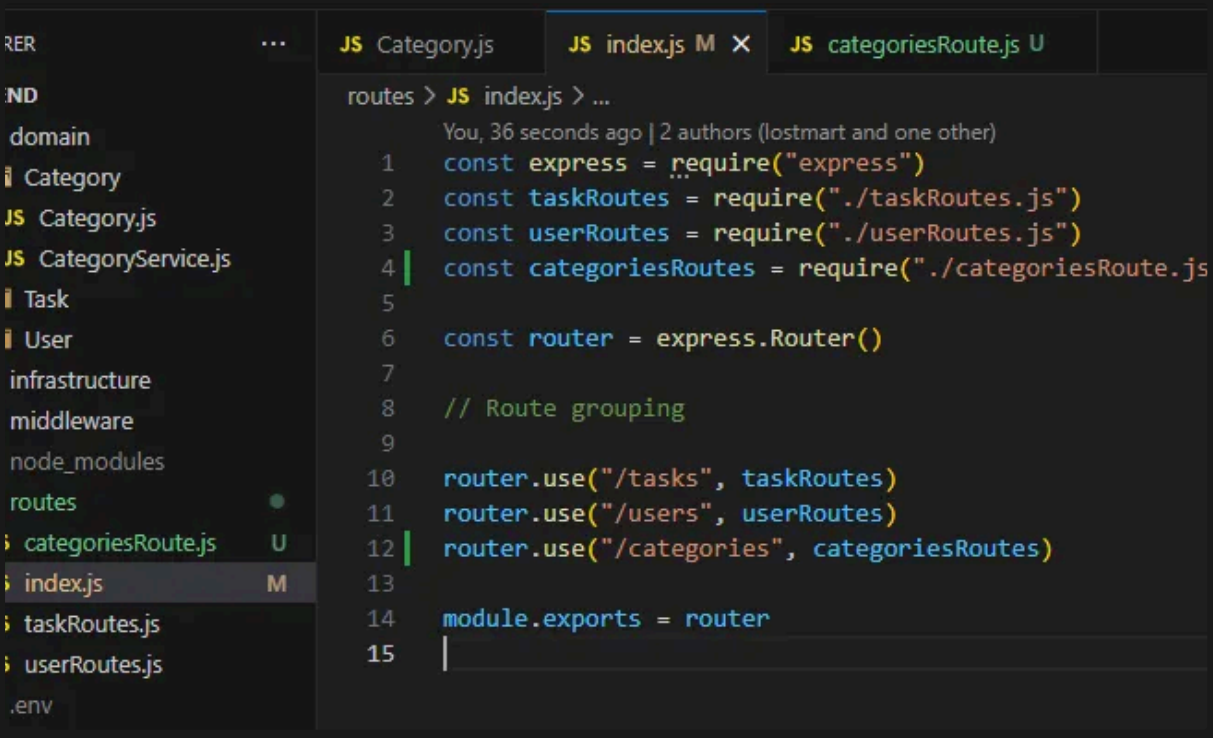


Categories Service

We are going to create a new file called `categoriesRoute.js` in the `routes` folder

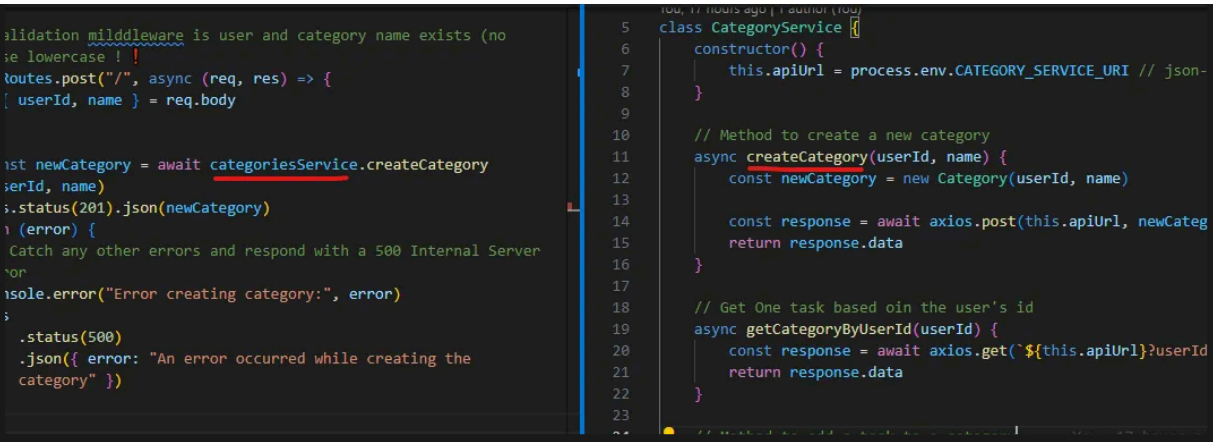


We are going to import and use it in the index route



In the `categoriesRoute` we are going to import the categories service

Here are the `categoriesRoute` and the `CategoryService`



```
alidation milddleware is user and category name exists (no
se lowercase ! !
outes.post("/", async (req, res) => {
 userId, name } = req.body


st newCategory = await categoriesService.createCategory
serId, name)
.status(201).json(newCategory)
 (error) {
 Catch any other errors and respond with a 500 Internal Server
or
nsole.error("Error creating category:", error)
  .status(500)
  .json({ error: "An error occurred while creating the
  category" })
```

```
class CategoryService {
    constructor() {
        this.apiUrl = process.env.CATEGORY_SERVICE_URI // json-
    }

    // Method to create a new category
    async createCategory(userId, name) {
        const newCategory = new Category(userId, name)

        const response = await axios.post(this.apiUrl, newCateg
        return response.data
    }

    // Get One task based oin the user's id
    async getCategoryByUserId(userId) {
        const response = await axios.get(`${this.apiUrl}?userId
        return response.data
    }
```

Maybe you see the comment in the category Route. A user can create the exact same category so would end up with a user creating a "conflict" by having the same category repeated as many times to his heart's content

Look at how many "Personal" categories this user can create!

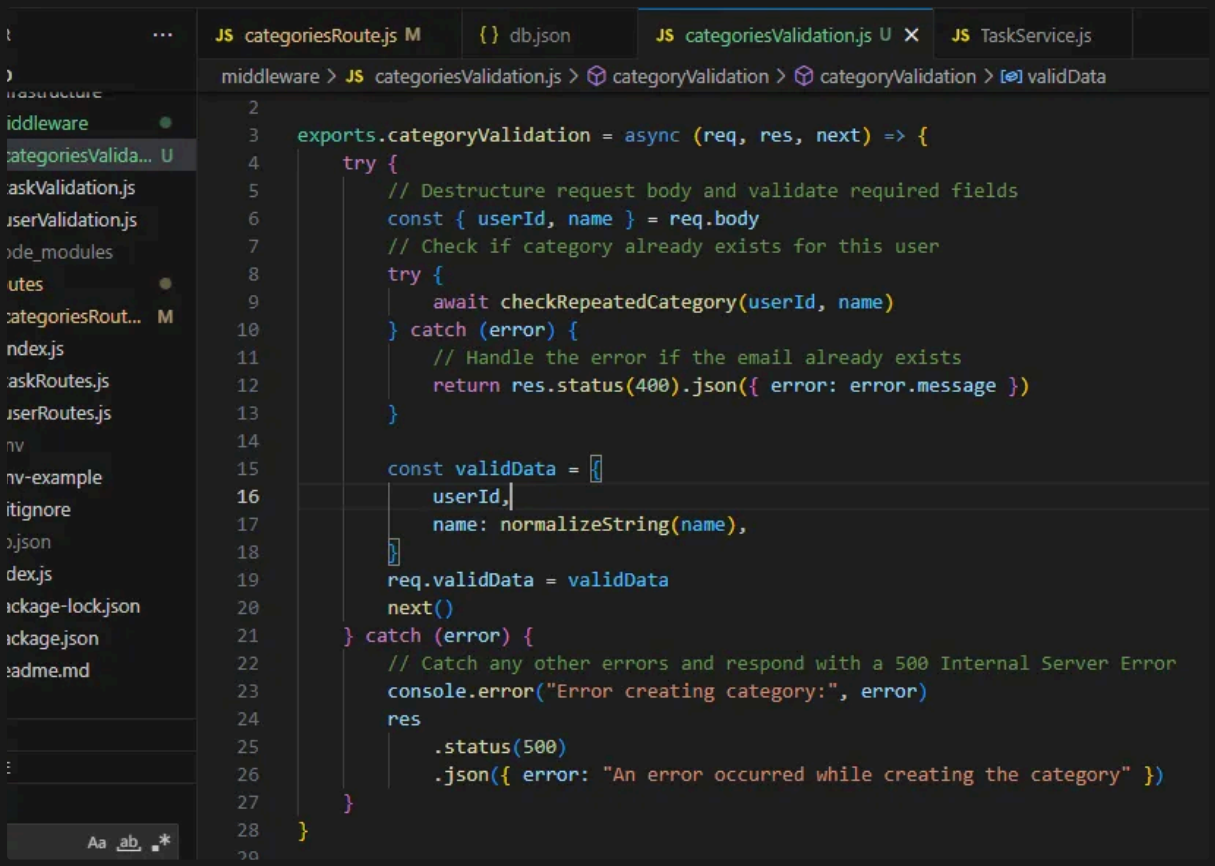

```
"categories": [
  {
    "id": "81ff",
    "userId": "20a8",
    "name": "Work",
    "tasks": []
  },
  {
    "id": "5f2e",
    "userId": "20a8",
    "name": "Personal",
    "tasks": []
  },
  {
    "id": "da83",
    "userId": "20a8",
    "name": "Personal",
    "tasks": []
  },
  {
    "id": "bd67",
    "userId": "20a8",
    "name": "PersonAl",
    "tasks": []
  },
  {
    "id": "3890",
    "userId": "20a8",
    "name": "PersonAl",
    "tasks": []
  },
```

This will be a headache later ! So we need to cater for this. Maybe you can think of a solution?

I'll create mine and hide it in a toggle heading bellow. BTW, we should do this in the categories middleware

▸ Show solution (same user duplicate categories)

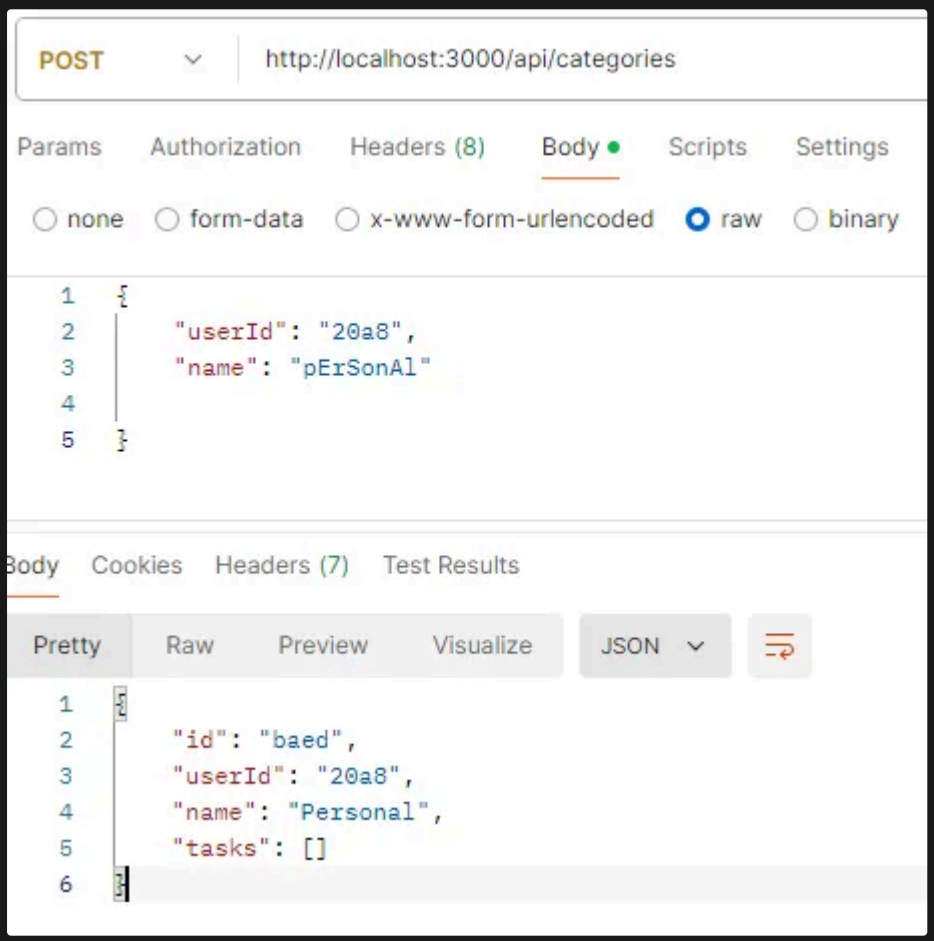
Finally, in order to stay consistent with the category names we can pass the normalized value to the router from the category validation middeware

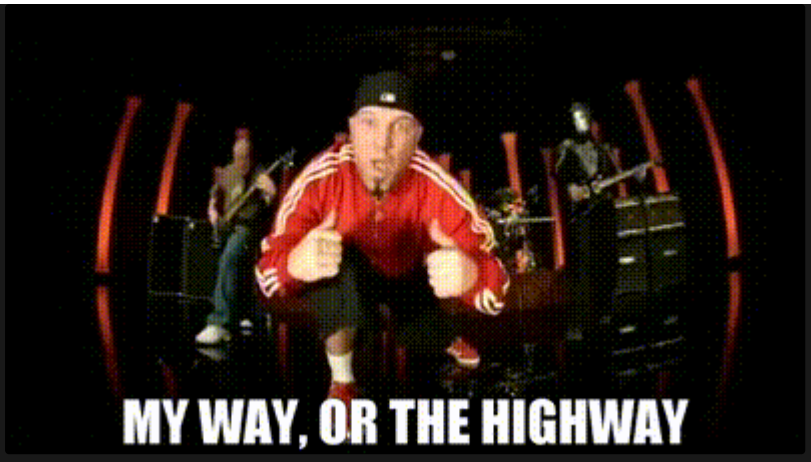


This is what my category router looks like:

```
routes > JS categoriesRoute.js > ...
4   const categoriesRoutes = express.Router()
5
6   // Initialize services
7   const categoriesService = new CategoryService()
8
9   // create a new category with normalized values and making sure the user doesn't have this category al
10  categoriesRoutes.post("/", categoryValidation, async (req, res) => {
11      const { userId, name } = req.validData
12
13      try {
14          const newCategory = await categoriesService.createCategory(userId, name)
15          res.status(201).json(newCategory)
16      } catch (error) {
17          // Catch any other errors and respond with a 500 Internal Server Error
18          console.error("Error creating category:", error)
19          res
20              .status(500)
21              .json({ error: "An error occurred while creating the category" })
22      }
23  })      You, 20 hours ago • new categry service
```

If the user gets a little funky with his category names, we will ignore it and make it our way





Refactor validation and help functions

Some validation function are repetitive, like type, empty, alphanumeric characters, etc

Maybe we can create a folder to put these together and re-use them

We can create a folder called `utils`

Inside here we create a file called `validation.js` Here we are going to make a more powerful function to check if the provided field of fields are empty. So far we've been doing this by checking one field at a time and sometimes more than one by the use of an `if` statement

User validation for required fields:

```
// Destructure request body and validate required fields
const { userName, email } = req.body

// Check empty for required fields
if (!userName || !email) {
    return res.status(400).json({
        error: "All fields are required: email, userName",
    })
}
```

then again, we de-structure from the body and check for an `undefined` value.

Task validation

```
 Destructure request body and validate required fields
nst { userId, title, description, dueDate, priority } = req.body

 not empty
 (!userId || !title || !description || !dueDate || !priority) {
  // Respond with a 400 Bad Request if any required field is missing
  return res.status(400).json({
      error:
          "All fields are required: userId, title, description, dueDate, prio
  })
```

It's fine, but a little basic and not very specific. The error simply give a list of things that are required but doesn't explicitly explain which ones are missing and so forth.

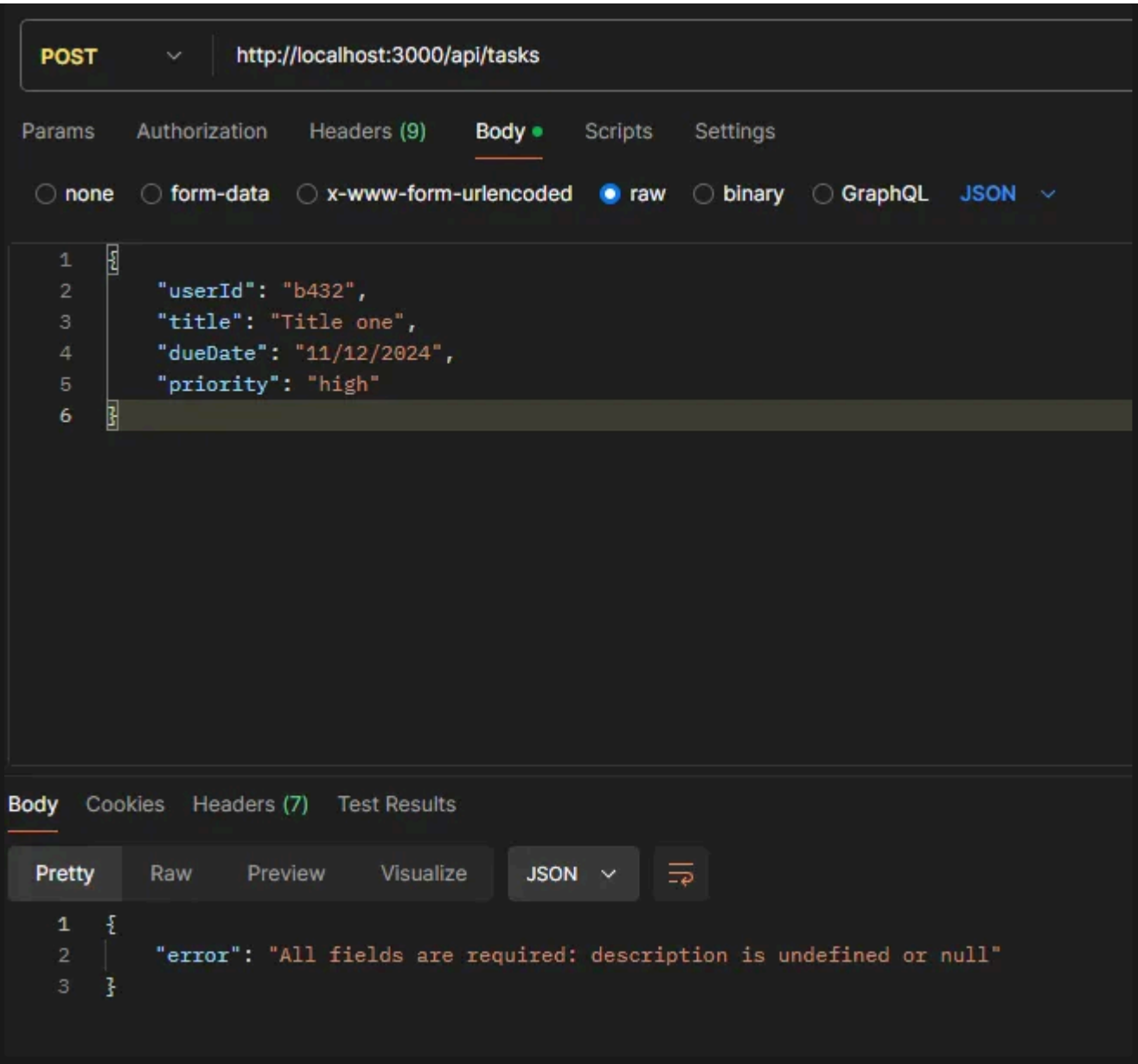
In the validation function we can try something like this: `validation.js`

```
exports.validateRequiredFields = ({ fieldName, value } = data) => { if
(!value) { throw new Error( `All fields are required: ${fieldName} is
undefined or null` ) } return true }
```

and then in the task validation for example we could call it like this

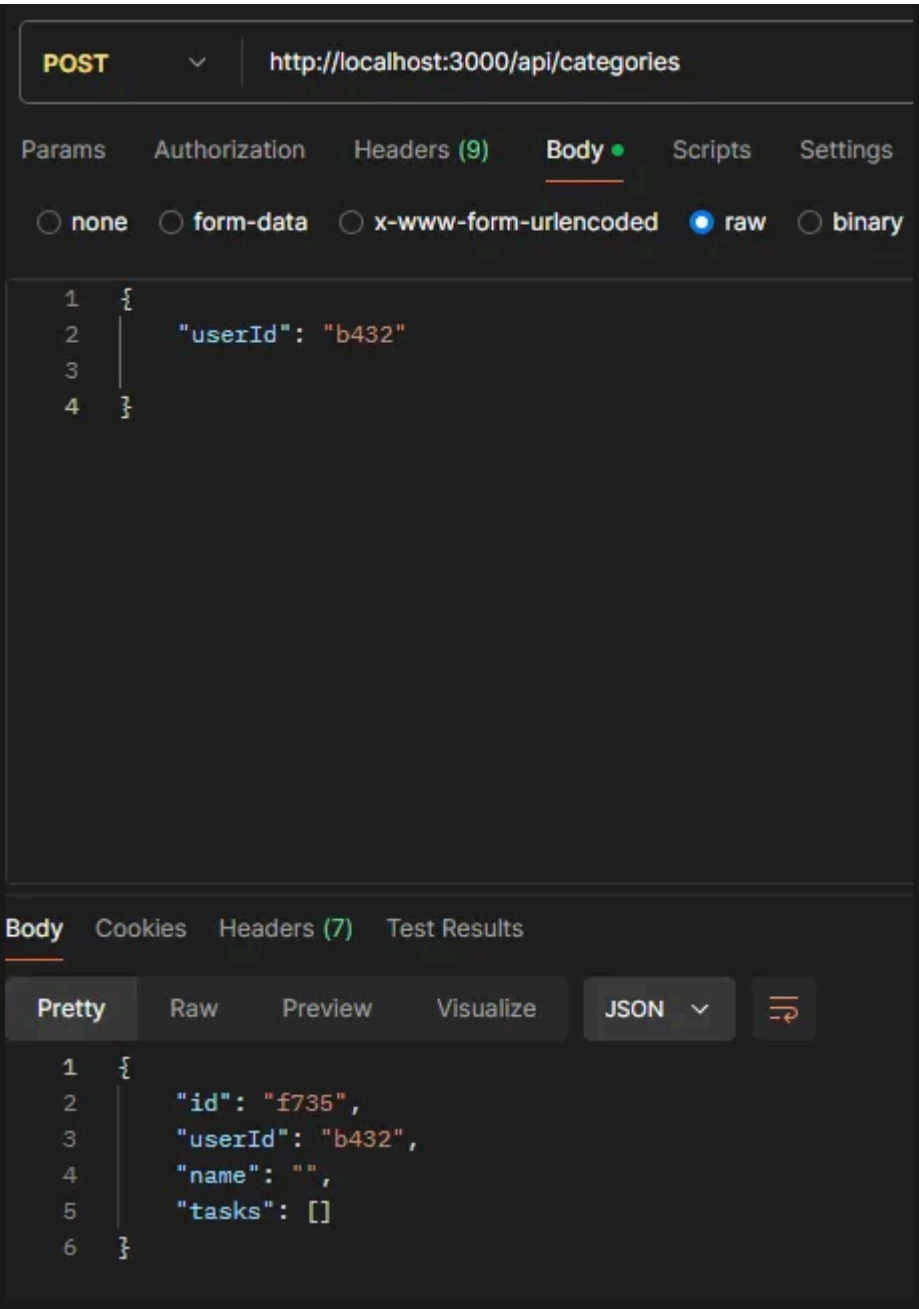
```
const { validateRequiredFields } = require("../utils/validation") //
check empty for every try { validateRequiredFields({ fieldName:
"description", value: description }) } catch (error) { // Handle the
error response here console.error("Validation error:", error.message)
res.status(400).json({ error: error.message }) }
```

Here's the result:

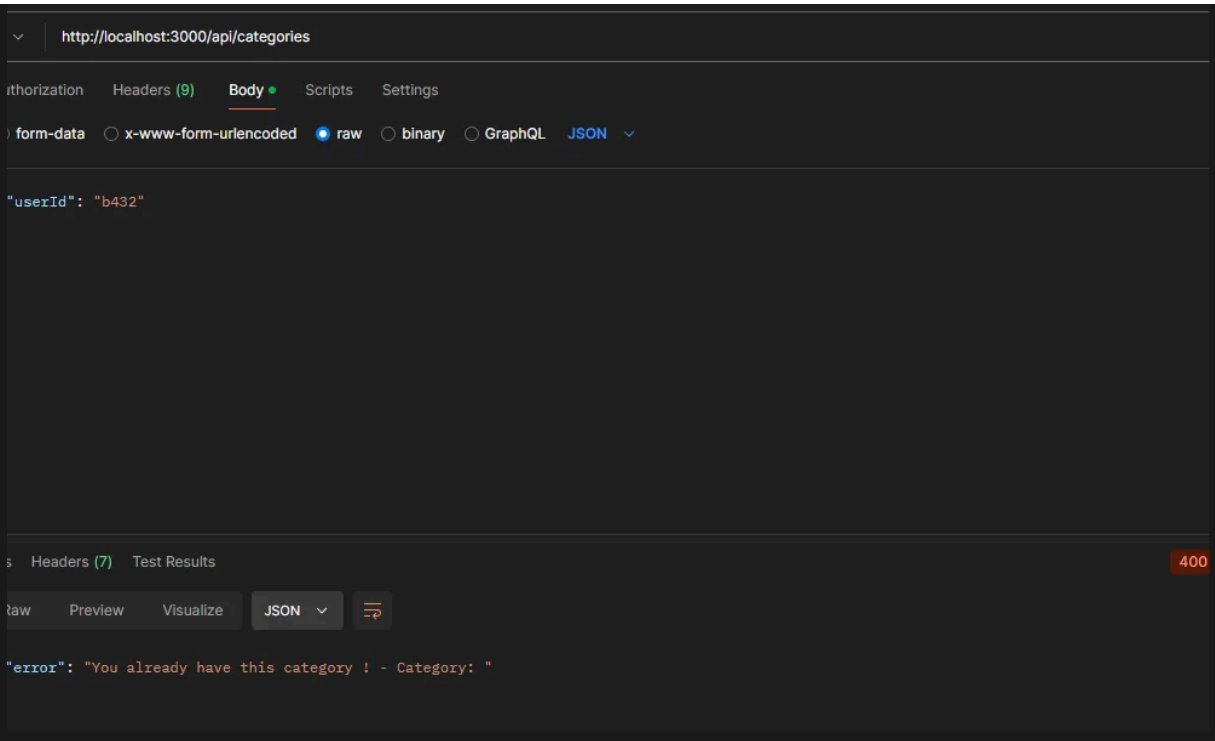


It works and now we have more personalized error messages, also we can re-use this function for other domains like, categories and users

We can import this new function called `validateRequiredFields` and use it whenever we need it. For example, the categories can be empty right now. This is a category for this user that is an empry string



and if we try to recreate it, it says you already have this category (empty string)



We can inject our `validateRequiredFields` utility function here

```
ou, 16 seconds ago | 2 authors (lostmart and one other)
equire("dotenv").config() // Load environment variables
onst { validateRequiredFields } = require("../utils/validation.js")

xports.categoryValidation = async (req, res, next) => {
    try {
        // Destructure request body and validate required fields
        const { userId, name } = req.body
        // check empty
        try {
            validateRequiredFields({ fieldName: "name", value: name })
        } catch (error) {
            return res.status(400).json({ error: error.message })
        }
```

With this implementation we separate the checking from sending the response. The `validateRequiredFields` is a pure function that doesn't know about response, user input, or anything else

JSDocs

🔗 Use JSDoc: Index

JSDocs is a great practice foe developers to write their own documentation while writing their code. This practice also brides the gap between JavaScript and TypeScript. On your `validation.js` file you can start typing /** and VS code will hint you with the JsDocs boiler plate