

DDD - Task Manager

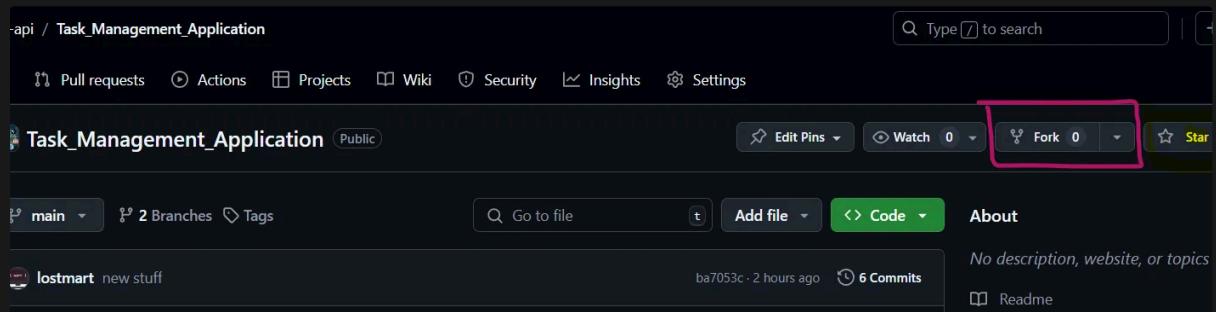
Repo:  GitHub [GitHub - web-rest-api/Task_Management_Application](#)

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

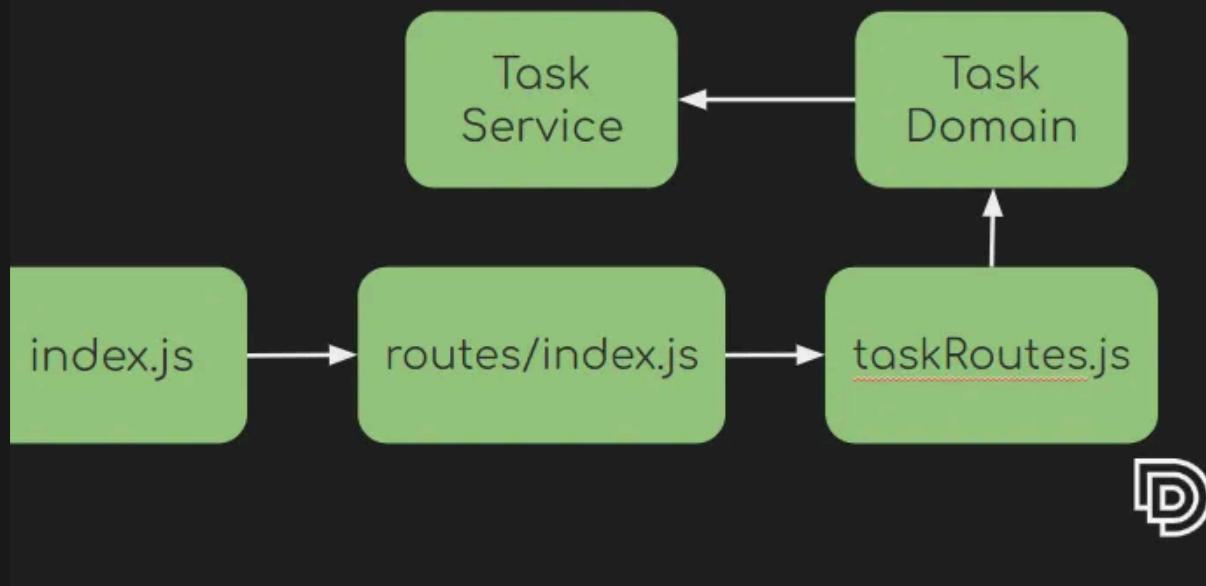
The screenshot shows a code editor with a sidebar displaying the project structure:

- RER
- ND
- domain
- infrastructure
- middleware
- node_modules
- routes
 - index.js
 - taskRoutes.js
- .env
- .gitignore
- db.json
- index.js
- package-lock.json
- package.json
- Readme.md

The right pane shows the content of `index.js`:

```
routes > JS index.js > ...
You, last week | 2 authors (lostmart and one other)
1 const express = require("express")
2 const taskRoutes = require("./taskRoutes.js")
3
4 const router = express.Router()
5
6 // Route grouping
7 router.use("/tasks", taskRoutes)
8
9 module.exports = router
10 |
```

Task route



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

The screenshot shows the POSTMAN application interface. At the top, the URL is set to `http://localhost:3000/api/tasks`. Below the URL, there are tabs for Authorization, Headers (8), Body (selected), Scripts, and Settings. The Body tab contains a JSON object with a single key-value pair: `msg": "hello from task route !"`. Under the Headers section, there are seven entries. At the bottom of the interface, the results are displayed: a green `200 OK` status box, a timing of `40 ms`, a size of `268 B`, and a save button. The JSON response body is shown below the status bar.

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

```

Initialize services
const taskService = new TaskService()

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 !" })
}

```

We reach the `console.log` and see it on the console

then we declare a new variable in this function and pass it in the request object to the next middleware

we can access this in the next function from the request object with its param "message" in this case

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

```

middleware > JS taskValidation.js > taskValidation > taskValidation
You, last week | 2 authors (You and one other)
1
2
3 exports.taskValidation = (req, res, next) => {
4   try {
5     // Destructure request body and validate required fields
6     const { userId, title, description, dueDate, priority } = req.body
7
8     if (!userId || !title || !description || !dueDate || !priority) {
9       // Respond with a 400 Bad Request if any required field is missing
10      return res.status(400).json({
11        error:
12          "All fields are required: userId, title, description, dueDate, priority",
13      })
14
15      // type check
16      // alphanumeric check
17      const isAlphanumeric = (str) => /^[a-zA-Z0-9\s!?\-]+$/ . test(str)
18      // title
19      if (typeof title !== "string" || !isAlphanumeric(title))
20        return res.status(400).json({
21          error: "Title must be a string with no weird characters !",
22        })
23
24      // description (YYYY-MM-DD)
25      if (typeof description !== "string")
26        return res.status(400).json({
27          error: "Description must be a string".

```

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

The screenshot shows a code editor with a dark theme. On the left is a sidebar with project files: main, infrastructure, middleware, askValidation.js, node_modules, routes, index.js, taskRoutes.js (which is selected and has a yellow background), ignore, json, dex.js, package-lock.json, and package.json. The main area shows the content of taskRoutes.js:

```

JS taskRoutes.js M X
routes > JS taskRoutes.js > ...
You, 16 minutes ago | 1 author (You)
1 const express = require("express")
2 const TaskService = require("../domain/Task/TaskService")
3 const { taskValidation } = require("../middleware/taskValidation")
4 const taskRoutes = express.Router()
5
6 // Initialize services
7 const taskService = new TaskService()
8
9 > taskRoutes.get(... )
10
11 > taskRoutes.post("/", taskValidation, async (req, res) => {
12   }
13
14 module.exports = taskRoutes

```

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

The screenshot shows a POSTMAN interface. The URL is set to `http://localhost:3000/api/tasks`. The request method is POST. The body contains the following JSON:

```

{
  "title": "hello my brother",
  "description": "description"
}

```

The response status is 400 Bad Request. The response body is:

```

{
  "error": "All fields are required: userId, title, description, dueDate, priority"
}

```

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 script to run it on the side: `"json-server": "json-server --watch db.json --port 5000"`

```

{
  "name": "backend",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "start": "nodemon index.js",
    "json-server": "json-server --watch db.json --port 5000"
  },
  "keywords": [
    "task-manager",
    "nodejs",
  ]
}

```

Once installed open a new terminal on the side and run

```

INIS-0A80I2A MINGW64 ~/Documents/EPITA/Advanced-Javascript-Programming/Project01/BackEnd (part-02)
json-server

1.0.0 json-server
--watch db.json --port 5000

# can be omitted, JSON Server 1+ watches for file changes by default
# started on PORT :5000
L-C to stop
db.json...

```

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

Task Service

```

class TaskService {
  constructor() {
    this.apiUrl = "http://localhost:5000/tasks" // 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 are also using the library `axios` to interact with our `json-server`. The method `createTask` is `async` since it's 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

The screenshot shows the VS Code interface with several tabs open:

- `taskRoutes.js`
- `TaskService.js M`
- `{ } db.json M`
- `.env`
- `npm`

The `.env` file contains the following environment variables:

```

1 PORT=3000
2 TASK_SERVICE_URI=http://localhost:5000/tasks
3

```

The `TaskService.js` file imports `axios`, `Task`, and `dotenv`. It defines a `TaskService` class with a constructor that sets the `apiUrl` to the value of `TASK_SERVICE_URI`. The `createTask` method creates a new `Task` object and sends a POST request to the `apiUrl` with the new task data.

```

domain > Task > JS TaskService.js > TaskService > completeTask
You, last week | 1 author (You)
1 const axios = require("axios")
2 const Task = require("./Task")
3 require("dotenv").config() // Load environment variables
4
You, last week | 1 author (You)
5 class TaskService {
6   constructor() {
7     this.apiUrl = process.env.TASK_SERVICE_URI // json-server URL for tasks
8   }
9
10  // Create a new task
11  async createTask(userId, title, description, dueDate, priority) {
12    const newTask = new Task(userId, title, description, dueDate, priority)
13    const response = await axios.post(this.apiUrl, newTask)
14    return response.data
15  }
16

```

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

The POSTMAN interface shows a successful `POST` request to `http://localhost:3000/api/tasks`.

Request Headers:

- Authorization
- Headers (8)
- Body (raw)
- Scripts
- Settings

Request Body (raw JSON):

```

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

```

Response Headers:

- Headers (7)
- Test Results

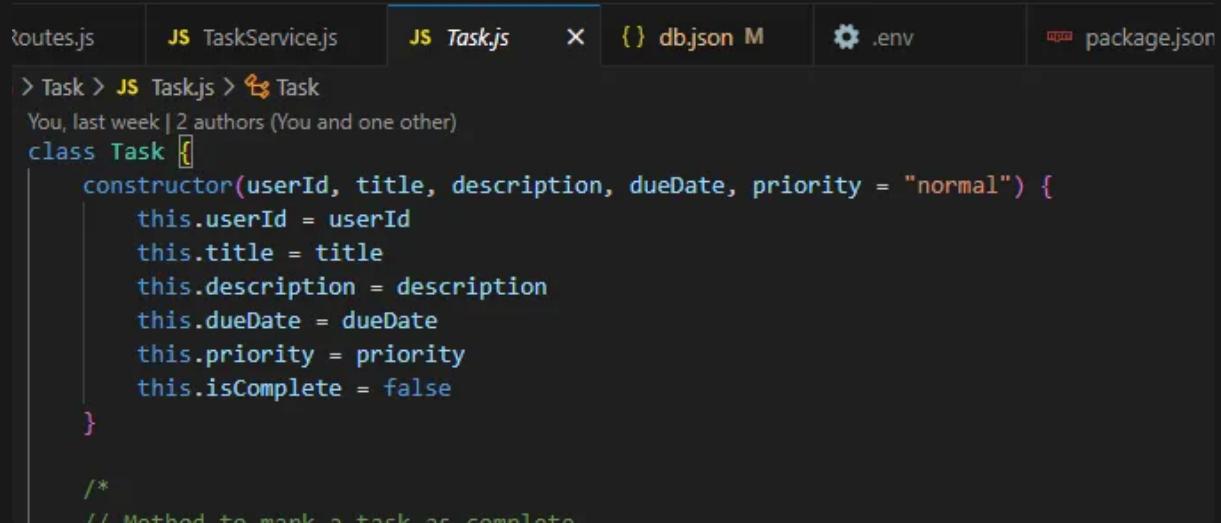
Response Body (JSON):

```

{
  "id": "c064",
  "userId": "5299",
  "title": "title test",
  "description": "description",
  "dueDate": "2024-06-23",
  "priority": "high",
  "isComplete": false
}

```

We can see that the Task class has a default priority of "normal" and a property `isComplete` that defaults to false



```
routes.js | JS TaskService.js | JS Task.js | {} db.json M | .env | package.json
> Task > JS Task.js > Task
You, last week | 2 authors (You and one other)
class Task {
  constructor(userId, title, description, dueDate, priority = "normal") {
    this.userId = userId
    this.title = title
    this.description = description
    this.dueDate = dueDate
    this.priority = priority
    this.isComplete = false
  }

  /*
   * Method to mark a task as complete
  */
}
```

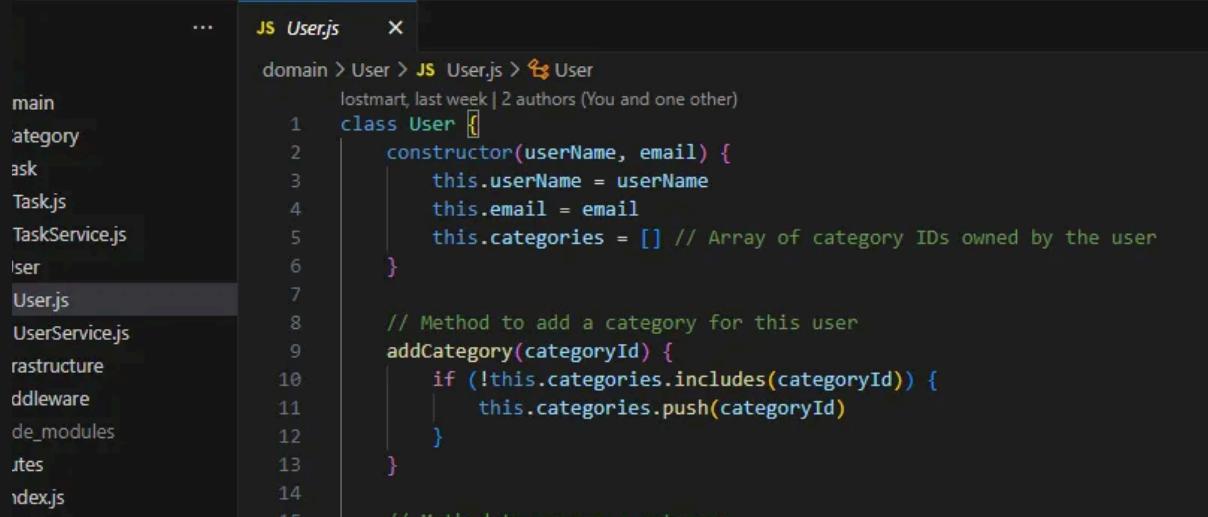
This is the required data to test our Task service

```
{ "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



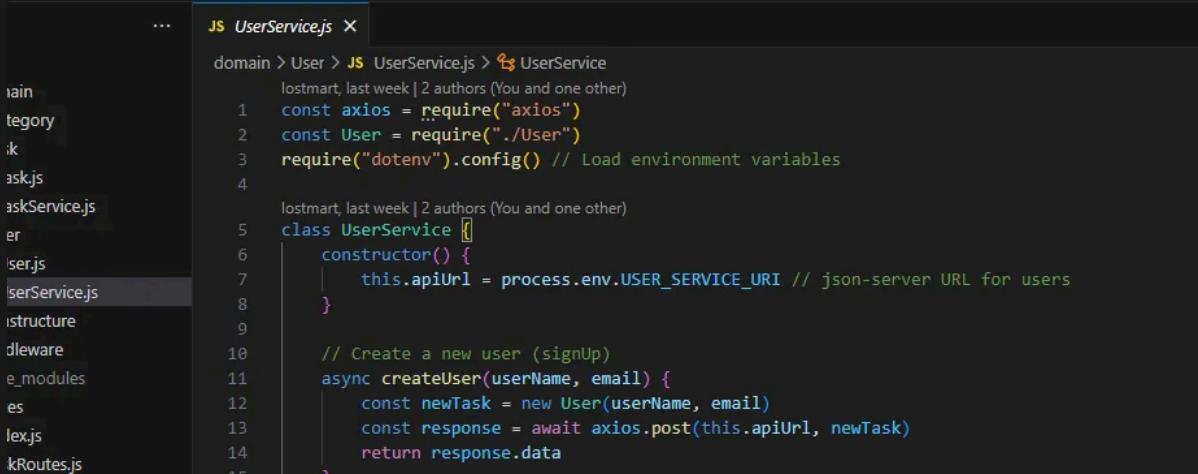
```
main
category
task
Task.js
TaskService.js
User
User.js
UserService.js
rastructure
ddleware
de_modules
jtes
index.js
... | JS User.js | ...
domain > User > JS User.js > User
lostmart, last week | 2 authors (You and one other)
class User {
  constructor(userName, email) {
    this.userName = userName
    this.email = email
    this.categories = [] // Array of category IDs owned by the user
  }

  // Method to add a category for this user
  addCategory(categoryId) {
    if (!this.categories.includes(categoryId)) {
      this.categories.push(categoryId)
    }
  }

  // Method to remove a category
}
```

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

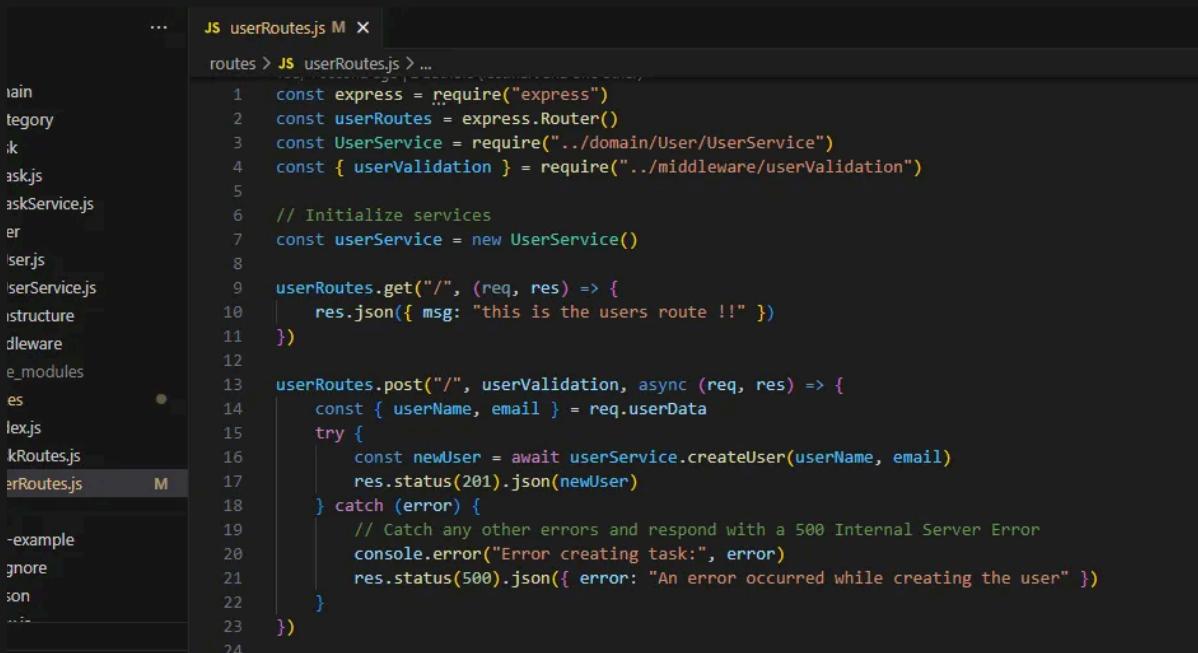
We also need a user service that will create an instance of the User class and store it in the json-server using axios



The screenshot shows a code editor with the file `UserService.js` open. The file imports `axios` and `User` from their respective modules. It also loads environment variables from `dotenv`. The `UserService` class has a constructor that sets the API URL to `process.env.USER_SERVICE_URI`. The `createUser` method uses `axios.post` to create a new user task.

```
... JS UserService.js X
domain > User > JS UserService.js > UserService
lostmart, last week | 2 authors (You and one other)
1 const axios = require("axios")
2 const User = require("./User")
3 require("dotenv").config() // Load environment variables
4
lostmart, last week | 2 authors (You and one other)
5 class UserService {
6   constructor() {
7     this.apiUrl = process.env.USER_SERVICE_URI // json-server URL for users
8   }
9
10  // Create a new user (signUp)
11  async createUser(userName, email) {
12    const newTask = new User(userName, email)
13    const response = await axios.post(this.apiUrl, newTask)
14    return response.data
15  }
16}
```

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



The screenshot shows the `userRoutes.js` file. It defines an Express router `userRoutes` and initializes it with a `UserService` instance. The `get` method returns a JSON response indicating the user route. The `post` method handles user creation, calling the `createUser` method from the `UserService` and returning a `201` status with the new user data. If an error occurs, it catches the error and returns a `500` status with an error message.

```
... JS userRoutes.js M X
routes > JS userRoutes.js > ...
1 const express = require("express")
2 const userRoutes = express.Router()
3 const UserService = require("../domain/User/UserService")
4 const { userValidation } = require("../middleware/userValidation")
5
6 // Initialize services
7 const userService = new UserService()
8
9 userRoutes.get("/", (req, res) => {
10   res.json({ msg: "this is the users route !!!" })
11 }
12
13 userRoutes.post("/", userValidation, async (req, res) => {
14   const { userName, email } = req.userData
15   try {
16     const newUser = await userService.createUser(userName, email)
17     res.status(201).json(newUser)
18   } catch (error) {
19     // Catch any other errors and respond with a 500 Internal Server Error
20     console.error("Error creating task:", error)
21     res.status(500).json({ error: "An error occurred while creating the user" })
22   }
23 }
```

Notice that we also have a `userValidation` middleware

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

The screenshot shows a code editor with two tabs: "routes.js M" and "JS userValidation.js X". The "userValidation.js" tab is active, displaying the following code:

```
JS userValidation.js > ...
lostmart, last week | 1 author (lostmart)
require("dotenv").config() // Load environment variables      lostmart, last week •

exports.userValidation = async (req, res, next) => {
  try {
    // Destructure request body and validate required fields
    const { userName, email } = req.body

    // Check if required fields are not empty
    if (!userName || !email) {
      return res.status(400).json({
        error: "All fields are required: email, userName",
      })
    }
  }
}
```

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`

The screenshot shows a code editor with the "userValidation.js" file open, containing the following code:

```
' Function to validate email format
const validateEmail = (email) => {
  return email.match(
    `/^(([^<>()[]\\.,;:\\s@"]+(\.[^<>()[]\\.,;:\\s@"]+)*|(.+"))@((\\[[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.)|`)
```

```
' Async function to check if email already exists
async function checkEmailExists(email) {
  try {
    const response = await fetch(
      `${process.env.USER_SERVICE_URI}?email=${email}`
    )
    const data = await response.json()

    if (data.length > 0) {
      throw new Error("Email already exists in the database!")
    }
  } catch (error) {
    console.error("Error:", error)
    throw new Error(error.message || "Failed to check email in the database")
  }
}
```

Than you can make your tests using POSTMAN

No strings

POST ▼ http://localhost:3000/api/users

Params Authorization Headers (9) **Body** ● Scripts Settings

none form-data x-www-form-urlencoded raw binary G

```
1 {  
2   "email": 1234,  
3   "userNmae": "123"  
4 }  
5
```

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON

```
1 {  
2   "error": "Email and userNmae must be strings !"  
3 }
```

or invalid formatting, or empty

POST ▼ http://localhost:3000/api/users

Params Authorization Headers (9) **Body** ● Scripts Settings

none form-data x-www-form-urlencoded raw binary

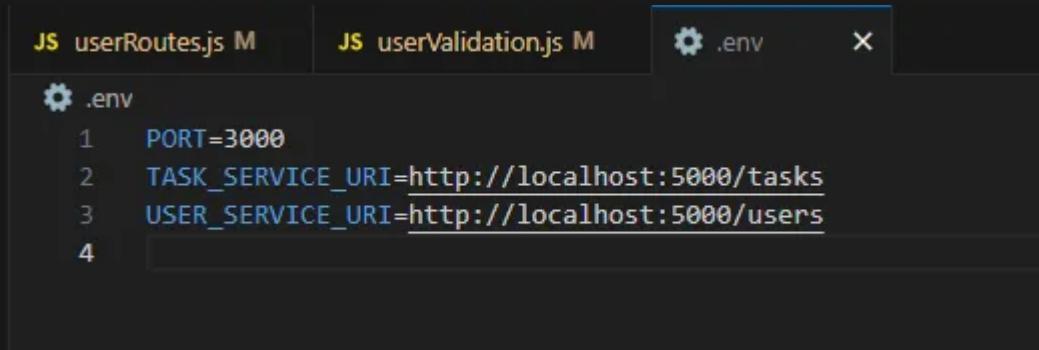
```
1 {  
2   "email": "helloMail!",  
3   "userNmae": "123"  
4 }  
5
```

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON

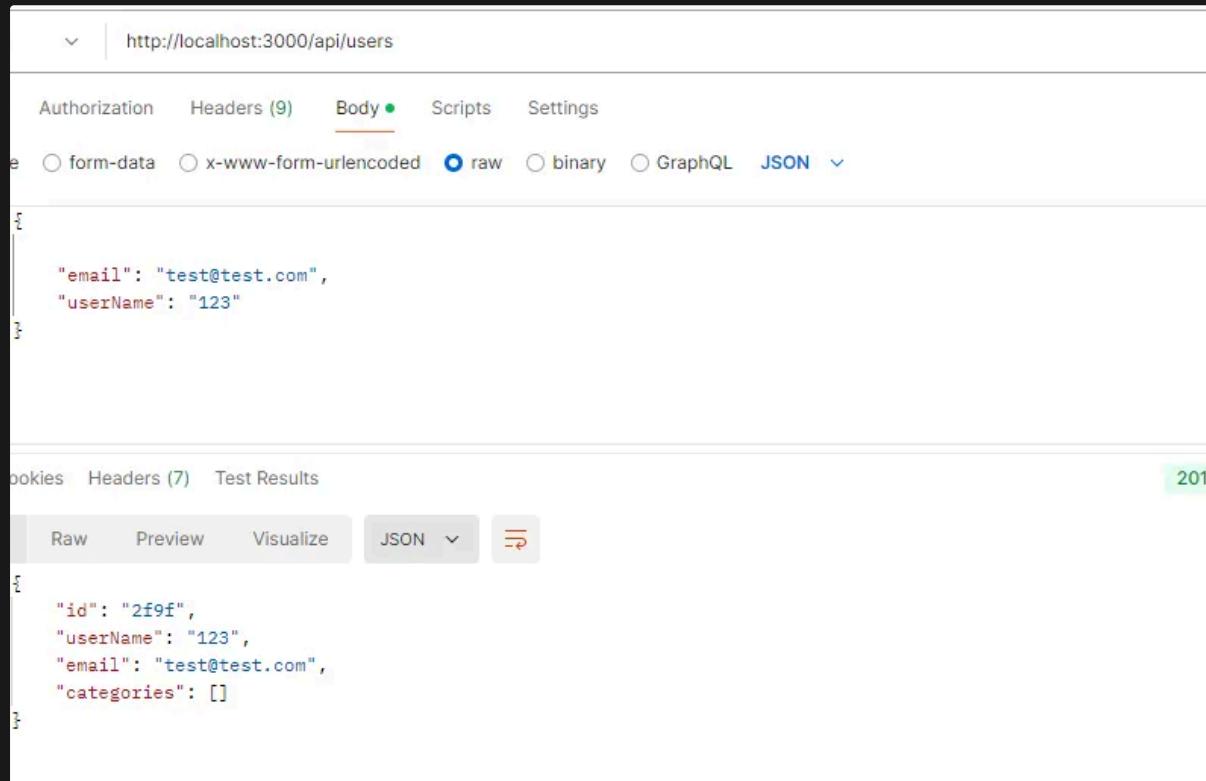
```
1 {  
2   "error": "Invalid email format ..."  
3 }
```

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



```
JS userRoutes.js M JS userValidation.js M .env X
.env
1 PORT=3000
2 TASK_SERVICE_URI=http://localhost:5000/tasks
3 USER_SERVICE_URI=http://localhost:5000/users
4
```

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



http://localhost:3000/api/users

Body (raw JSON)

```
{"email": "test@test.com", "userName": "123"}
```

201

Raw JSON

```
{"id": "2f9f", "userName": "123", "email": "test@test.com", "categories": []}
```

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

The screenshot shows a Postman interface with the following details:

- URL:** `http://localhost:3000/api/users`
- Method:** POST (implied by the Headers tab)
- Headers:** (9 items shown)
- Body:** `raw JSON`
Content:

```
"email": "test@test.com",
"userName": "123"
```
- Status:** 400 Bad Request
- Response:**

```
"error": "Email already exists in the database!"
```

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:

```
JS userRoutes.js      {} db.json      X
{} db.json > [ ]users
1  {
2    "tasks": [
3      {
4        "id": "9ee8",
5        "userId": "7d87",
6        "title": "Complete DDD project",
7        "description": "Finish initial setup",
8        "dueDate": "2024-10-30",
9        "priority": "high",
10       "isComplete": false
11     }
12   ],
13   "categories": [],
14   "users": [
15     {
16       "id": "7d87",
17       "userName": "Mzar",
18       "email": "bob2@example.org",
19       "categories": []
20     },
21     {
22       "id": "2f9f",
23       "userName": "123",
24       "email": "test@test.com",
25       "categories": []
26     }
27   ]
28 }
```

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 <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** ▾

```
1 {  
2 |   "email": "test@test.com",  
3 |   "userName": "123"  
4 }  
5 
```

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON ▾ 

```
1 {  
2 |   "id": "20a8",  
3 |   "userName": "123",  
4 |   "email": "test@test.com",  
5 |   "categories": []  
6 }  
7 
```

same user ID

HTTP <http://localhost:3000/api/tasks>

POST <http://localhost:3000/api/tasks>

Params Authorization Headers (8) **Body** Scripts Settings

none form-data x-www-form-urlencoded raw binary

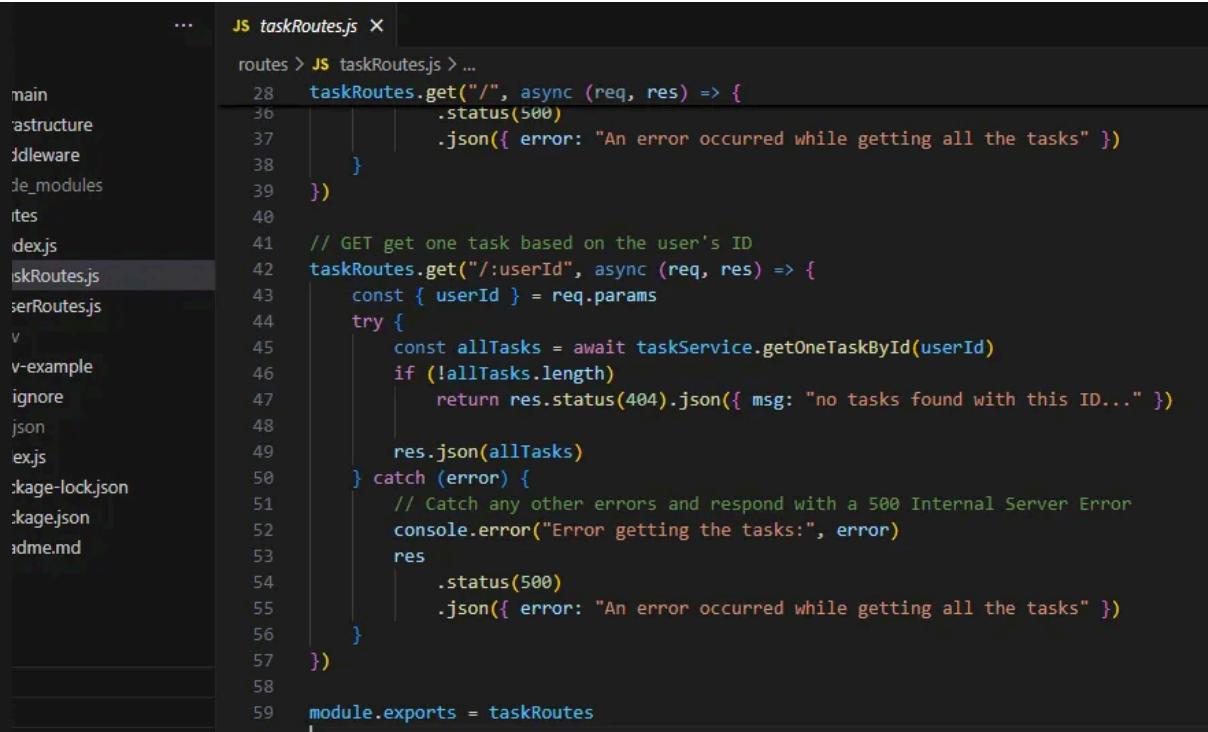
```
1 {  
2   "userId": "20a8",  
3   "title": "title test",  
4   "description": "description",  
5   "dueDate": "2024-06-23",  
6   "priority": "high"  
7 }  
8 }
```

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON [Copy](#)

```
1 {  
2   "id": "1e74",  
3   "userId": "20a8",  
4   "title": "title test",  
5   "description": "description",  
6   "dueDate": "2024-06-23",  
7   "priority": "high",  
8   "isComplete": false  
9 }
```

In the task route we should have something like this

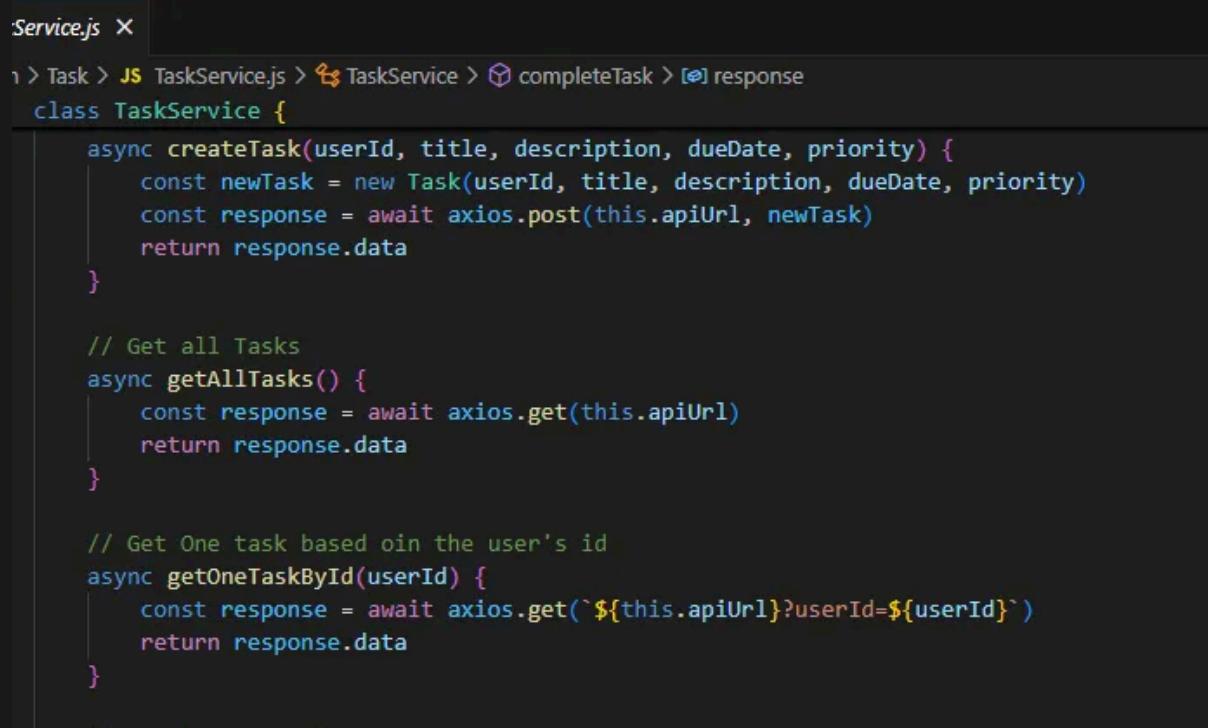


```
JS taskRoutes.js X
...
main
structure
idleware
de_modules
ites
dexjs
taskRoutes.js
serRoutes.js
v
v-example
ignore
json
exjs
package-lock.json
package.json
adme.md

routes > JS taskRoutes.js > ...
28     taskRoutes.get("/", async (req, res) => {
29         .status(500)
30         .json({ error: "An error occurred while getting all the tasks" })
31     })
32
33     // GET get one task based on the user's ID
34     taskRoutes.get("/:userId", async (req, res) => {
35         const { userId } = req.params
36         try {
37             const allTasks = await taskService.getOneTaskById(userId)
38             if (!allTasks.length)
39                 return res.status(404).json({ msg: "no tasks found with this ID..." })
40
41             res.json(allTasks)
42         } catch (error) {
43             // Catch any other errors and respond with a 500 Internal Server Error
44             console.error("Error getting the tasks:", error)
45             res
46                 .status(500)
47                 .json({ error: "An error occurred while getting all the tasks" })
48         }
49     })
50
51     module.exports = taskRoutes
```

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:



```
Service.js X
...
Task > JS TaskService.js > TaskService > completeTask > response
class TaskService {
    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
    }

    // Get all Tasks
    async getAllTasks() {
        const response = await axios.get(this.apiUrl)
        return response.data
    }

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

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

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

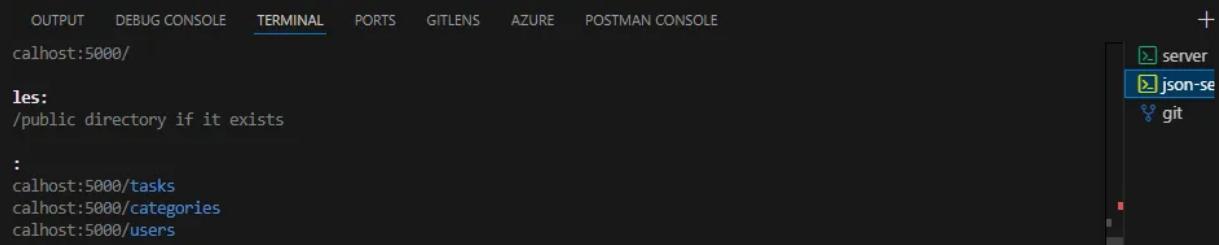
OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS AZURE POSTMAN CONSOLE +

calhost:5000/

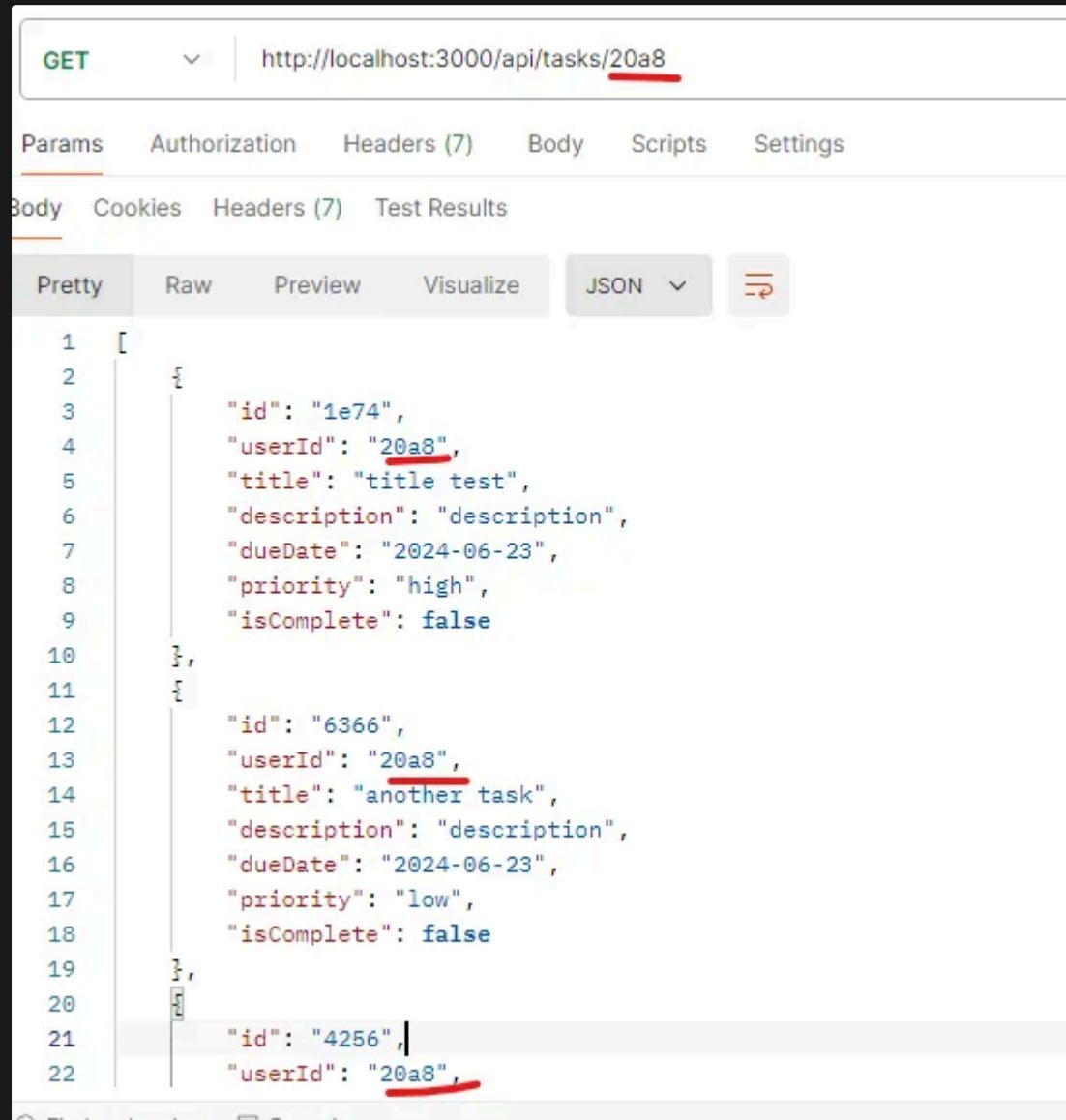
les:
/public directory if it exists

:

calhost:5000/tasks
calhost:5000/categories
calhost:5000/users



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



GET <http://localhost:3000/api/tasks/20a8>

Params Authorization Headers (7) Body Scripts Settings

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON ↻

```
1 [  
2 {  
3     "id": "1e74",  
4     "userId": "20a8",  
5     "title": "title test",  
6     "description": "description",  
7     "dueDate": "2024-06-23",  
8     "priority": "high",  
9     "isComplete": false  
10 },  
11 {  
12     "id": "6366",  
13     "userId": "20a8",  
14     "title": "another task",  
15     "description": "description",  
16     "dueDate": "2024-06-23",  
17     "priority": "low",  
18     "isComplete": false  
19 },  
20 {  
21     "id": "4256",  
22     "userId": "20a8",
```

Categories Service

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

The screenshot shows the VS Code interface with the Explorer sidebar on the left and the Editor on the right. The Explorer sidebar shows a project structure under 'BACKEND' with folders like 'domain', 'Category', 'Task', 'User', 'infrastructure', 'middleware', 'node_modules', and 'routes'. Inside 'routes', files 'categoriesRoute.js', 'index.js', 'taskRoutes.js', and 'userRoutes.js' are listed. The Editor tab bar has three tabs: 'Category.js', 'categoriesRoute.js', and 'index.js'. The current file is 'categoriesRoute.js', which contains the following code:

```
routes > JS categoriesRoute.js > [?] <unknown>
1 const express = require("express")
2 const categoriesRoutes = express.Router()
3
4
5 module.exports = categoriesRoutes
```

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

The screenshot shows the VS Code interface with the Explorer sidebar on the left and the Editor on the right. The Explorer sidebar shows the same project structure as before. The Editor tab bar has three tabs: 'Category.js', 'index.js', and 'categoriesRoute.js'. The current file is 'index.js', which contains the following code:

```
routes > JS index.js > ...
1 You, 36 seconds ago | 2 authors (lostmart and one other)
2 const express = require("express")
3 const taskRoutes = require("./taskRoutes.js")
4 const userRoutes = require("./userRoutes.js")
5 const categoriesRoutes = require("./categoriesRoute.js")
6
7 const router = express.Router()
8
9 // Route grouping
10 router.use("/tasks", taskRoutes)
11 router.use("/users", userRoutes)
12 router.use("/categories", categoriesRoutes)
13
14 module.exports = router
15
```

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

Here are the `categoriesRoute` and the `CategoryService`

```

1 validation middleware is user and category name exists (no
2   lowercase !
3 routes.post("/", async (req, res) => {
4   [userId, name] = req.body
5
6   const newCategory = await categoriesService.createCategory(
7     userId, name
8   );
9   res.status(201).json(newCategory)
10  } (error) {
11    // Catch any other errors and respond with a 500 Internal Server
12    // Error
13    console.error("Error creating category:", error)
14    res.status(500)
15    .json({ error: "An error occurred while creating the
16      category" })
17
18
19
20
21
22
23

```

```

100,17 hours ago | Author (100)
5 class CategoryService {
6   constructor() {
7     this.apiUrl = process.env.CATEGORY_SERVICE_URI // json-
8   }
9
10  // Method to create a new category
11  async createCategory(userId, name) {
12    const newCategory = new Category(userId, name)
13
14    const response = await axios.post(this.apiUrl, newCateg
15    return response.data
16  }
17
18  // Get One task based oin the user's id
19  async getCategoryByUserId(userId) {
20    const response = await axios.get(`${this.apiUrl}?userId=
21    return response.data
22  }
23

```

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": []
  },
  ...
]

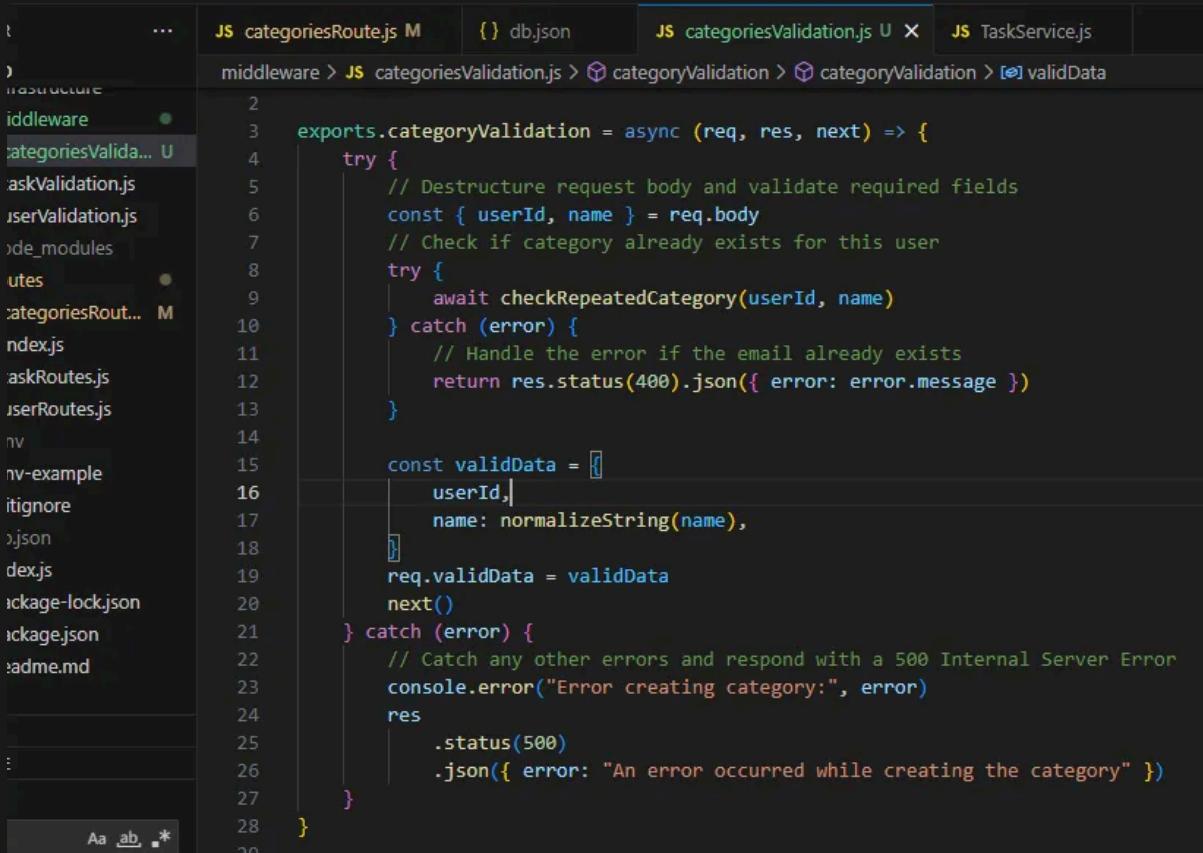
```

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 middleware



```
JS categoriesRoute.js M | {} db.json | JS categoriesValidation.js U X | JS TaskService.js
...
middleware > JS categoriesValidation.js > categoryValidation > categoryValidation > validData
...
2
3   exports.categoryValidation = async (req, res, next) => {
4     try {
5       // Destructure request body and validate required fields
6       const { userId, name } = req.body
7       // Check if category already exists for this user
8       try {
9         await checkRepeatedCategory(userId, name)
10      } catch (error) {
11        // Handle the error if the email already exists
12        return res.status(400).json({ error: error.message })
13      }
14
15      const validData = [
16        userId,
17        name: normalizeString(name),
18      ]
19      req.validData = validData
20      next()
21    } catch (error) {
22      // Catch any other errors and respond with a 500 Internal Server Error
23      console.error("Error creating category:", error)
24      res
25        .status(500)
26        .json({ error: "An error occurred while creating the category" })
27    }
28  }
29 }
```

This is what my category router looks like:

JS categoriesRoute.js M X {} db.json JS categoriesValidation.js U JS TaskService.js

```

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 already
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 category service

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

POST <http://localhost:3000/api/categories>

Params	Authorization	Headers (8)	Body ●	Scripts	Settings
<input type="radio"/> none	<input type="radio"/> form-data	<input type="radio"/> x-www-form-urlencoded	<input checked="" type="radio"/> raw	<input type="radio"/> binary	

```

1  {
2    "userId": "20a8",
3    "name": "pErSonAl"
4
5  }

```

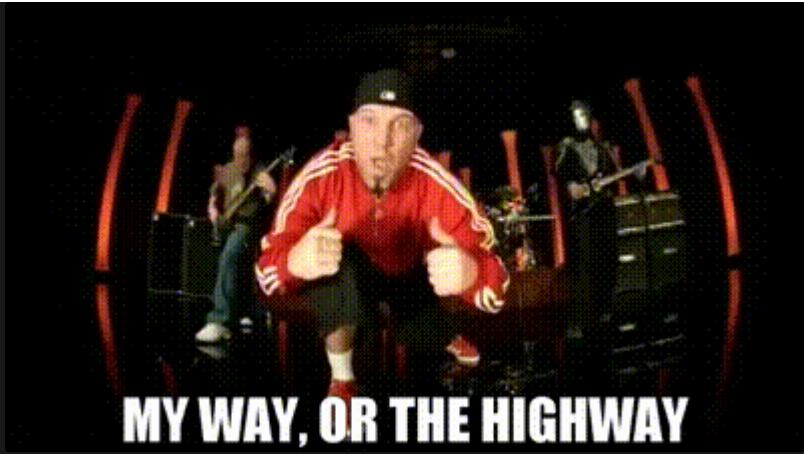

Body	Cookies	Headers (7)	Test Results
------	---------	-------------	--------------

Pretty	Raw	Preview	Visualize	JSON	
--------	-----	---------	-----------	------	--

```

1
2   "id": "baed",
3   "userId": "20a8",
4   "name": "Personal",
5   "tasks": []
6

```



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
const { 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, priority"
})
```

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:

The screenshot shows a POST request to `http://localhost:3000/api/tasks`. The request body is a JSON object:

```
1 {  
2   "userId": "b432",  
3   "title": "Title one",  
4   "dueDate": "11/12/2024",  
5   "priority": "high"  
6 }
```

The response body is a JSON object with an error message:

```
1 {  
2   "error": "All fields are required: description is undefined or null"  
3 }
```

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 empty string

POST | <http://localhost:3000/api/categories>

Params Authorization Headers (9) **Body** Scripts Settings

none form-data x-www-form-urlencoded raw binary

```
1 {  
2   "userId": "b432"  
3 }  
  
Body Cookies Headers (7) Test Results
```

Pretty Raw Preview Visualize JSON 

```
1 {  
2   "id": "f735",  
3   "userId": "b432",  
4   "name": "",  
5   "tasks": []  
6 }
```

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

The screenshot shows a Postman interface. At the top, the URL is `http://localhost:3000/api/categories`. Below it, the **Body** tab is selected, showing a raw JSON payload: `"userId": "b432"`. The response section shows a status of 400 with the error message: `"error": "You already have this category ! - Category: "`.

We can inject our `validateRequiredFields` utility function here

```
5u, 16 seconds ago | 2 authors (lostmart and one other)
require("dotenv").config() // Load environment variables
const { validateRequiredFields } = require("../utils/validation.js")

exports.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 for developers to write their own documentation while writing their code. This practice also bridges 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

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