InternCareer

Introduction

In the realm of modern web development, Node.js and RESTful APIs have emerged as pivotal technologies. Node.js is an open-source, cross-platform JavaScript runtime environment and library for running web applications outside the clients browser. Others define it as a runtime environment that allows JavaScript to be executed on the server side. Complementing Node.js, RESTful (Representational State Transfer) APIs have become the standard for designing web services, offering a stateless, client-server communication protocol that typically leverages HTTP methods for CRUD (Create, Read, Update, Delete) operations on resources. The most commonly used framework with Node.js when it comes to creating RESTful APIs is Express. Express is a minimalistic and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. It is effectively a tool that simplifies building web applications and APIs using JavaScript on the server side. Express adds helpful utilities to Node HTTP(Hypertext transfer protocol) objects and facilitates the handling of dynamic HTTP objects.

Task 1

In task 1, the main focus was creating a RESTful API for a certain resource such as users, then add controllers to perform CRUD (create, read, update, and delete) operations on it in Node.js. In my case, the resource of choice were tasks, basically making it some sort of task manager API. Before even going even far, the very first thing was to create a server in node, and for us to do that we first have to setup a node project using npm.

We first open the command line then navigate to the folder where you want to setup the Node app. In the terminal write the command below to create the folder holding your Node app one will create and navigate to it.

```
mkdir TaskManger
cd TaskMnager
```

To start the node project we use the command bellow in the command line

```
npm init -y
```

The next step will be to install express, and to do that we use the command bellow in the command line

```
npm install express
```

After this we will then open the folder in an IDE such as sublime or visual studio code and create a file which will name app.js which will act as our main entry point for the project. We will then create the server using the following code

```
const express = ('express');
const app = express();
const port = 3300;

app.listen(port, ()=>{
    console.log('Server is running on port http://localhost${
});
```

For the port number, the most commonly used is 3000 but in my case I used 3300. Then to run the server you just type the command node app.js. The code will work but when you navigate to the site to see what will be displayed, what you will be shown is that the page will say something along the line of can not /GET. This is because we have not added any routes to the page. Since I was working on a task resources, I create the routes that would be used to perform the operations on the tasks. Before performing any testing on the routes, I created an in memory dataset using json for the task. After designing the dataset and adding some dummy data, I tested the routes using postman. Bellow is the JavaScript code I created which contains the in memory dataset, and the routes with the CRUD operations

```
let tasks = [
    {id:1, name:'1st task', description:'The description for
    {id:2, name:'2nd task', description:'The description for
];

//read all tasks
app.get('/api/task', (req, res)=>{
```

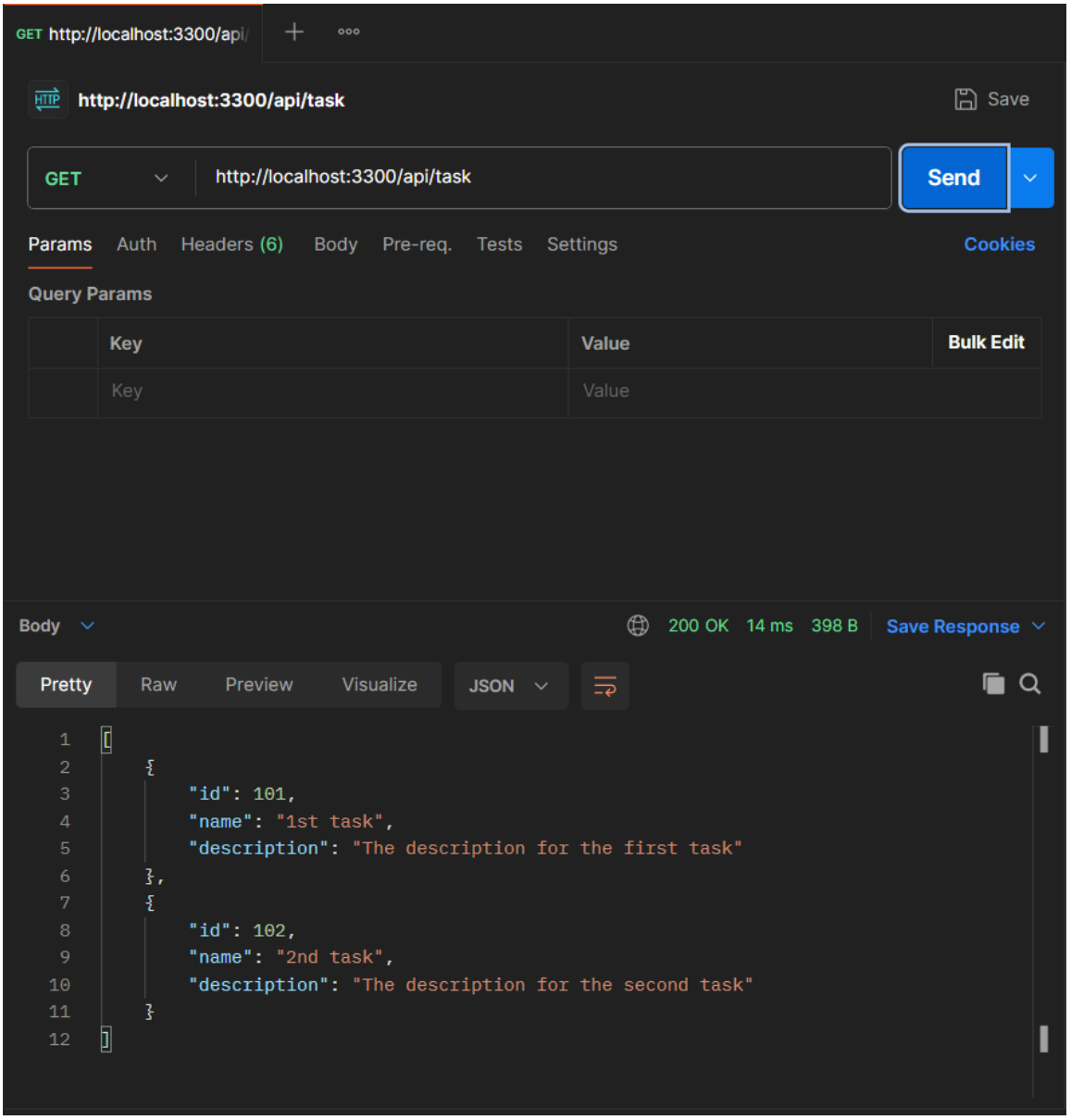
```
    res.json(tasks)
});

//read a single task
app.get('/api/task/:id', (req, res)=>{
    const task = tasks.find(x => x.id === parseInt(req.params
    if (!task) return res.status(404).send('Task can not be f
    res.json(task)
});

//create a task
app.post('/api/task', (req, res)=>{
    const {name, description} = req.body;
    const newTask = {
        id: tasks.lenth + 1,
        name: req.body.name,
        description: req.body.description
    }
    tasks.push(newTask);
    res.status(202).json(newTask);
});

//update a task
app.put('/api/task/:id', (req, res)=>{
    const task = tasks.find(u => u.id === parseInt(req.params
    if (!task) return res.status(404).send('Task can not be f
    task.name = req.body.name;
    task.description = req.body;
    res.json(tasks[task]);
});

//delete a task
app.delete('/api/task:id', (req, res)=>{
    const task = tasks.find(u => u.id === parseInt(req.params
    if (task === -1) return res.status(404).send('Task can no
    tasks.splice(task, 1);
    res.send('Task has been successfully deleted');
});
```

One other thing I did was adding a schema for tasks, task validation middleware and error handling middleware which are shown bellow

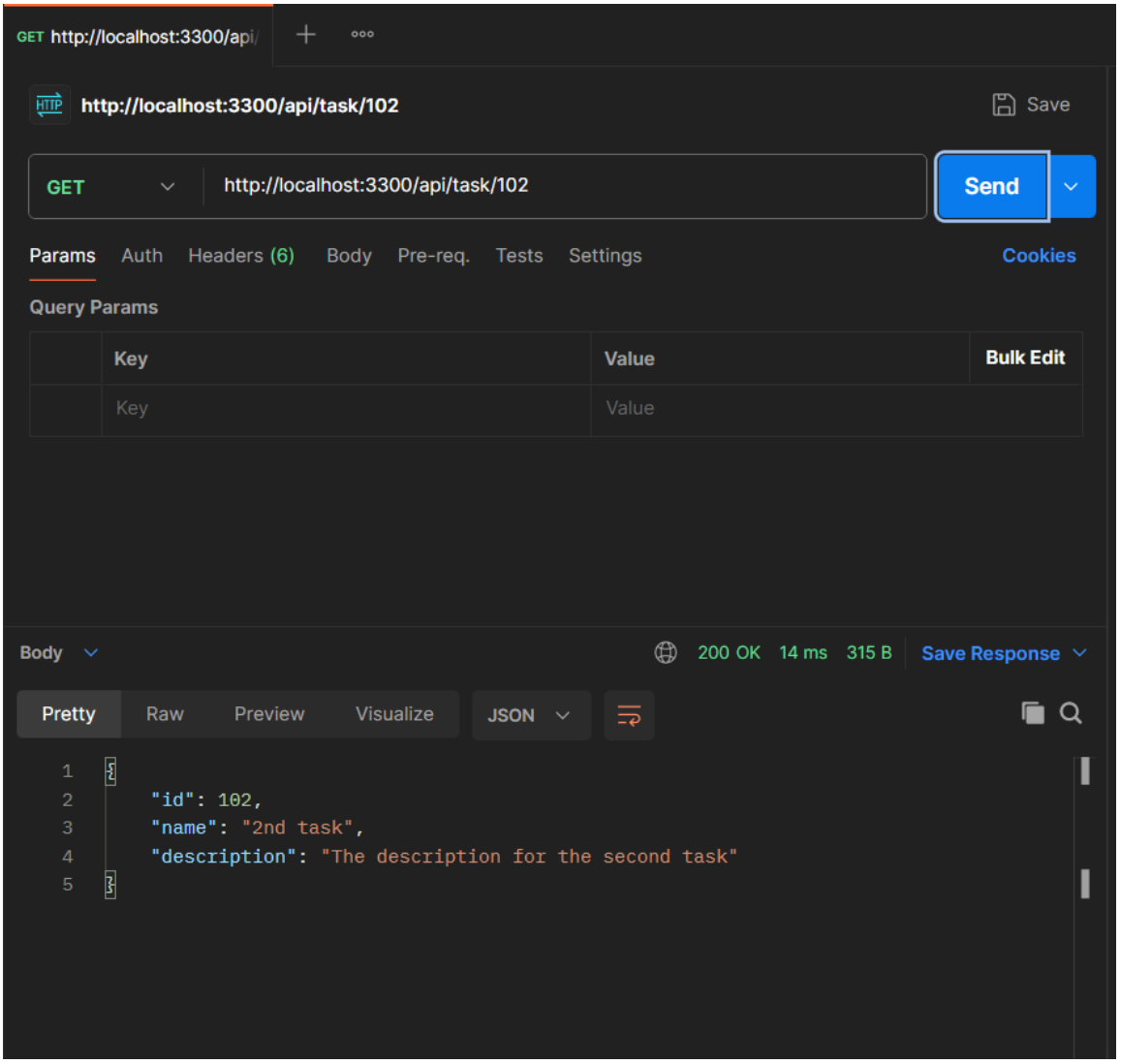
```
const express = ('express');
const joi = ('joi')
const app = express();
const port = 3300;

//task schema
const taskSchema = joi.object({
    name: joi.string().min(3).required(),
    description: joi.string().min(10).required()
});

//task validation
const vaidateTask = (req, res, next)=>{
    const {error} = taskSchema.validate(req.body);
    if (error) return res.status(400).send(error.details[0].m
    next();
}

//code for routes

//error handling middleware
app.use((err, req, res, next)=>{
    console.error(err.stack);
    res.status(500).send('Something went wrong');
});

app.listen(port, ()=>{
    console.log('Server is running on port http://localhost${
});
```

After adding all the codes above we will now run the server the use postman to test the routes. Bellow are the results of routes after testing them in the same order the are provided

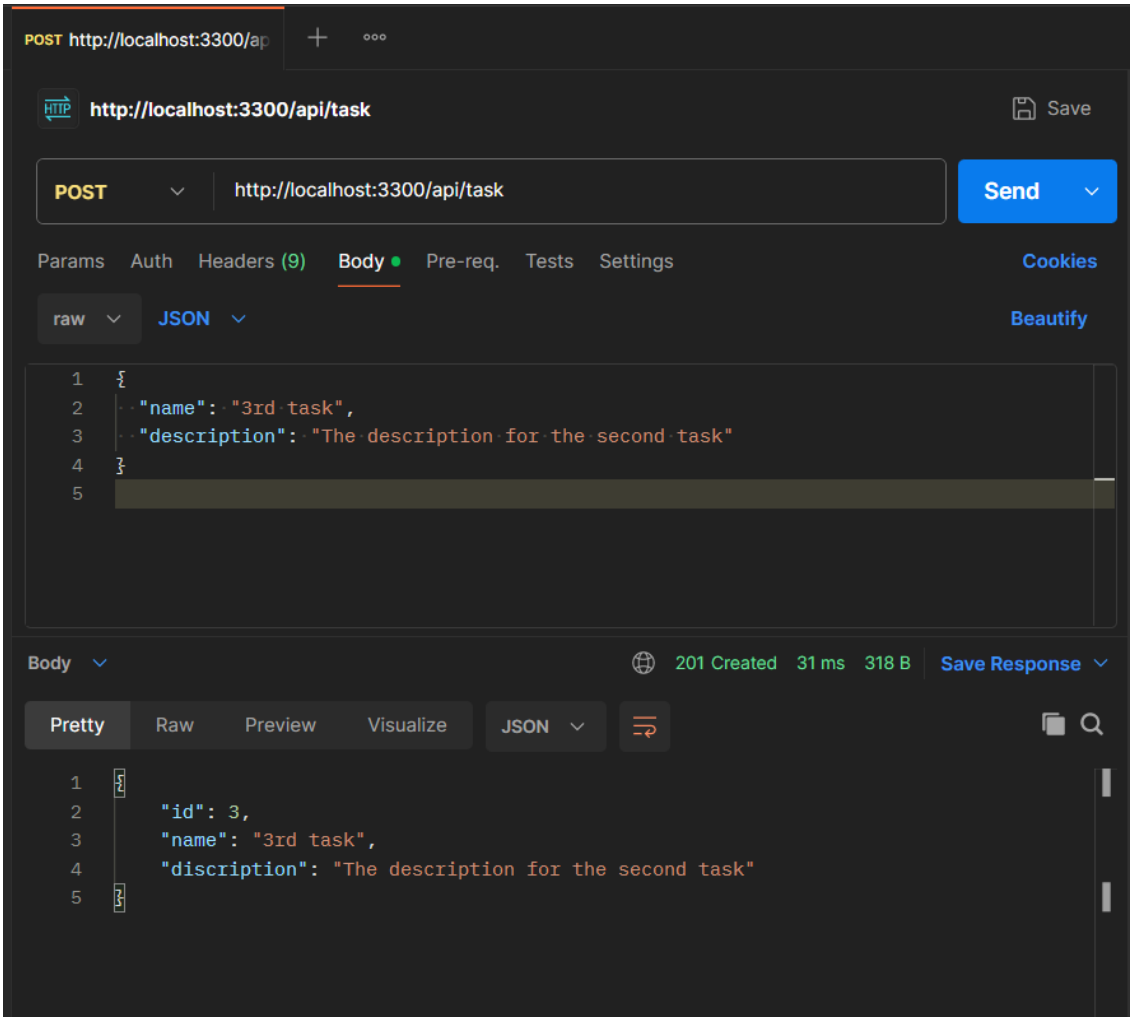
- Read all the tasks



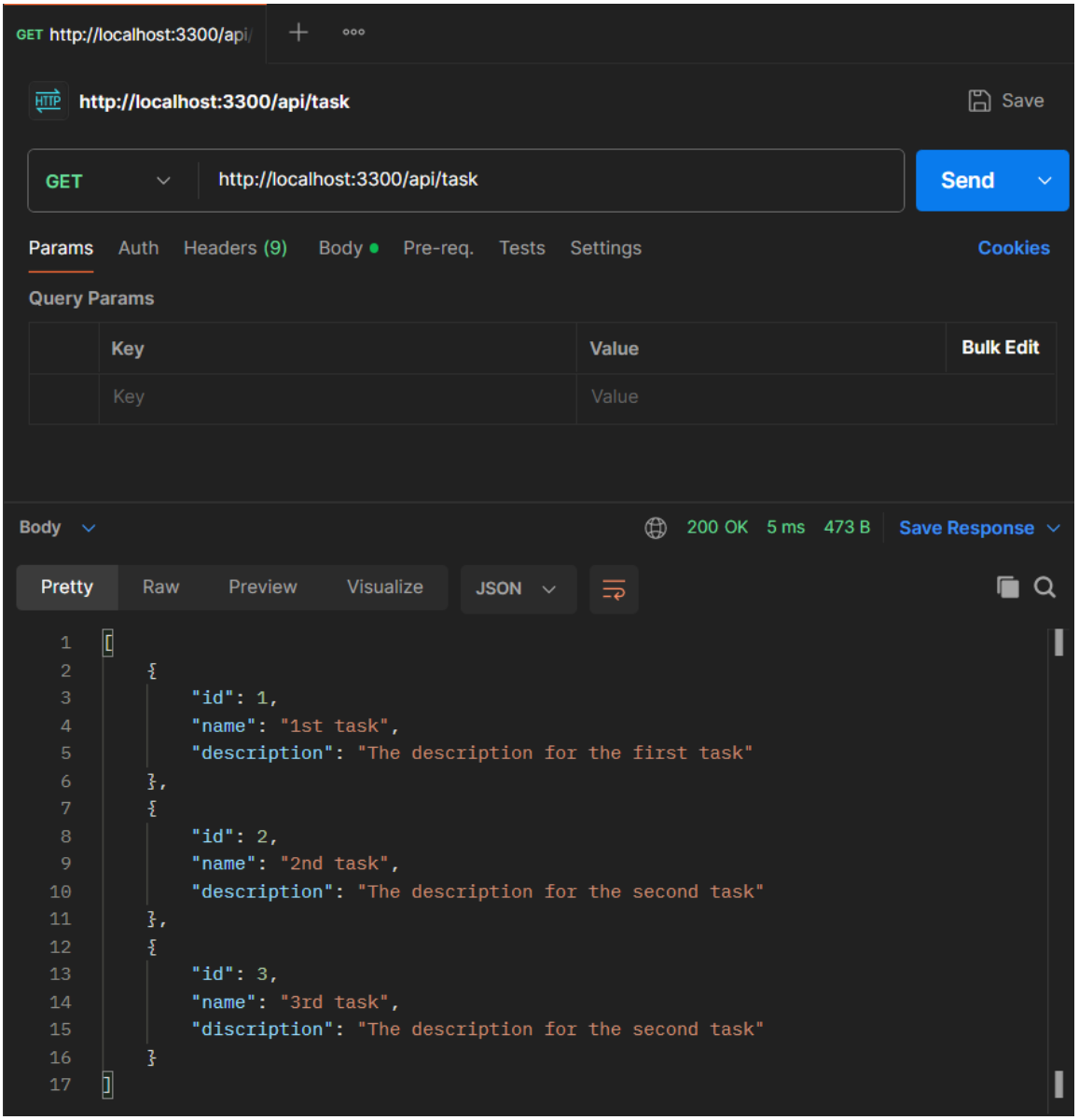
- Read a single task



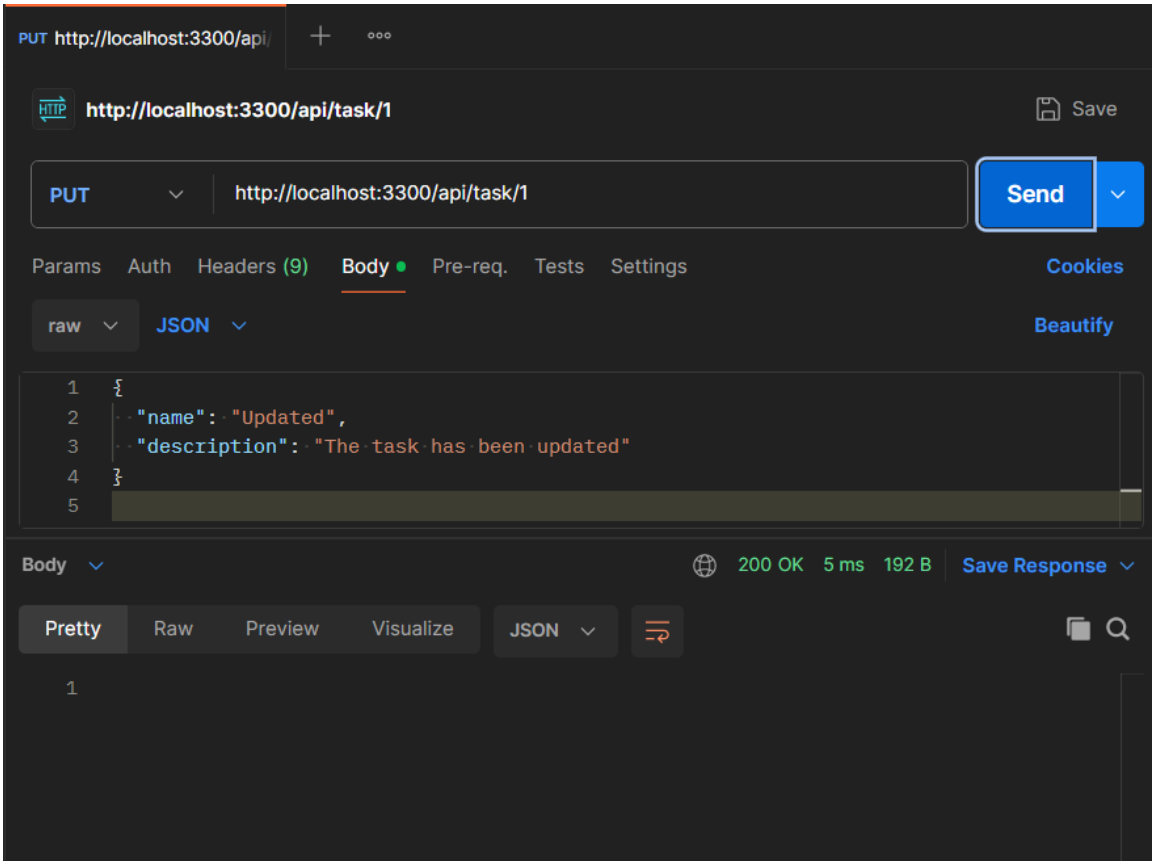
- Create a task



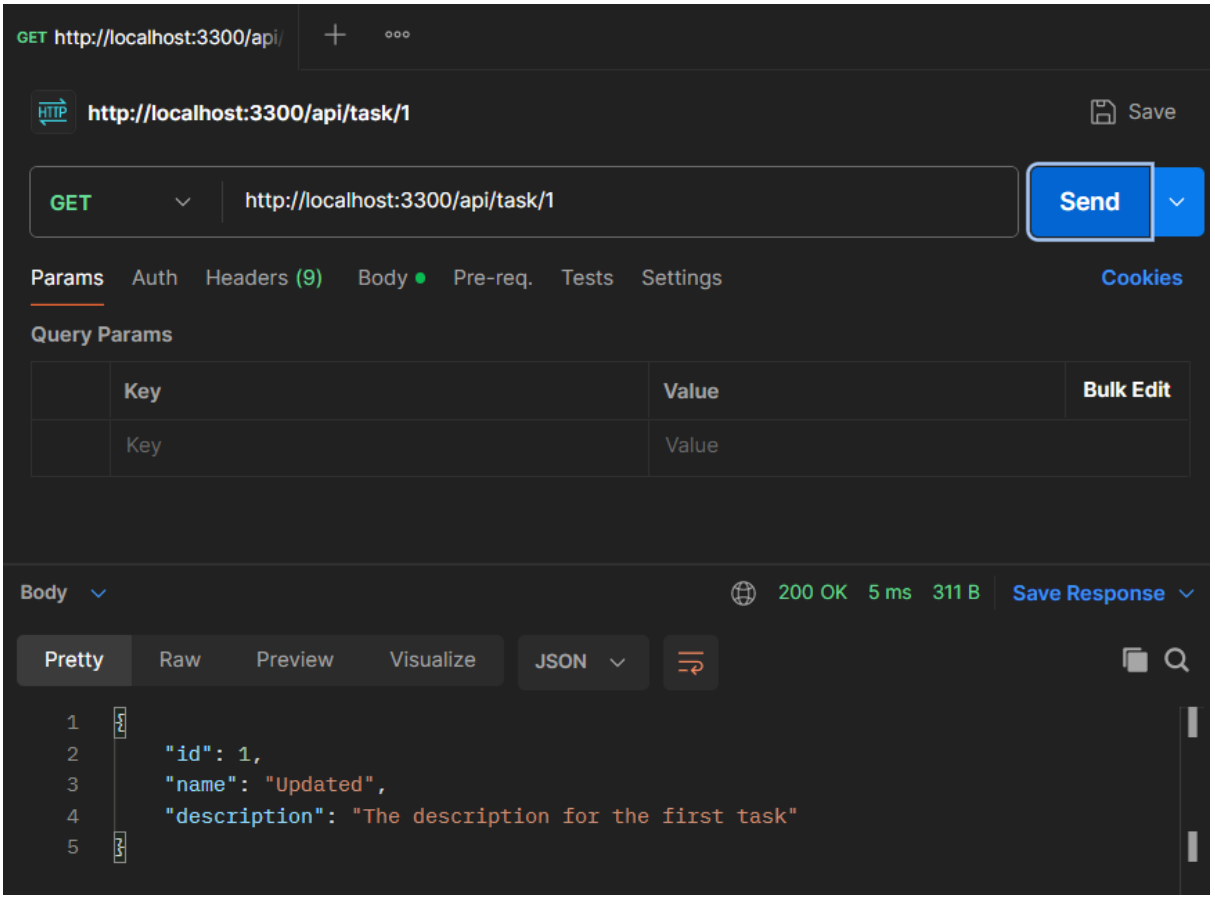
- Read all tasks to see if task has been added



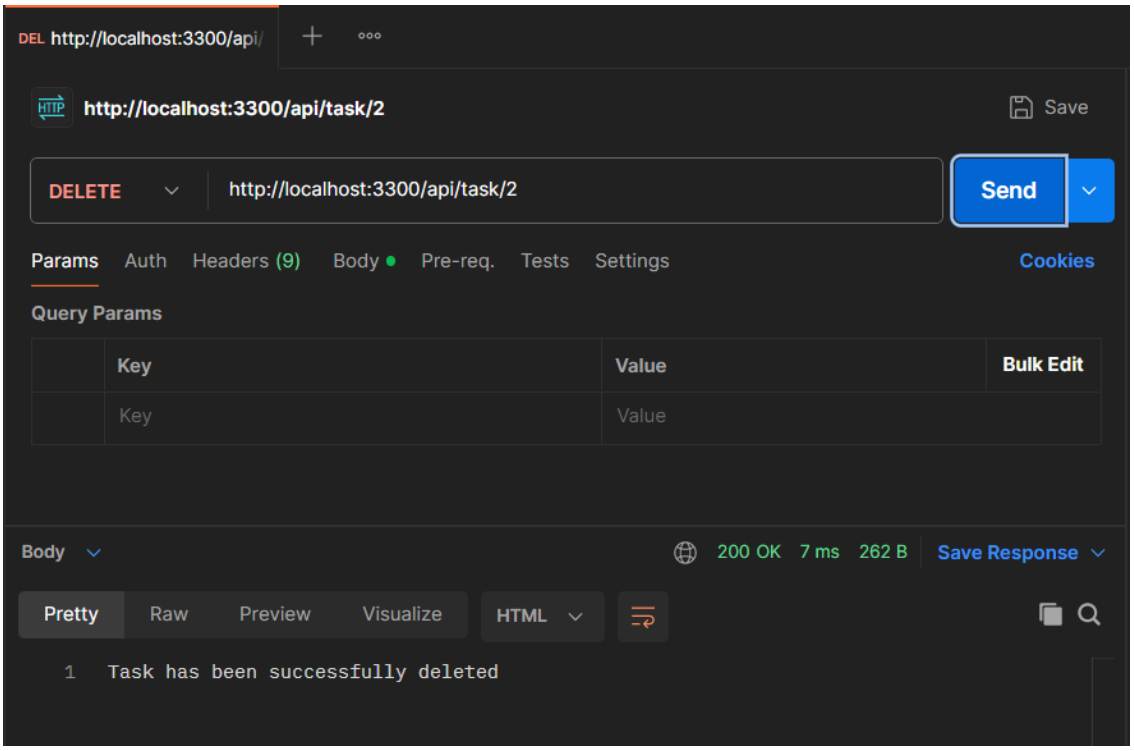
- Update a task



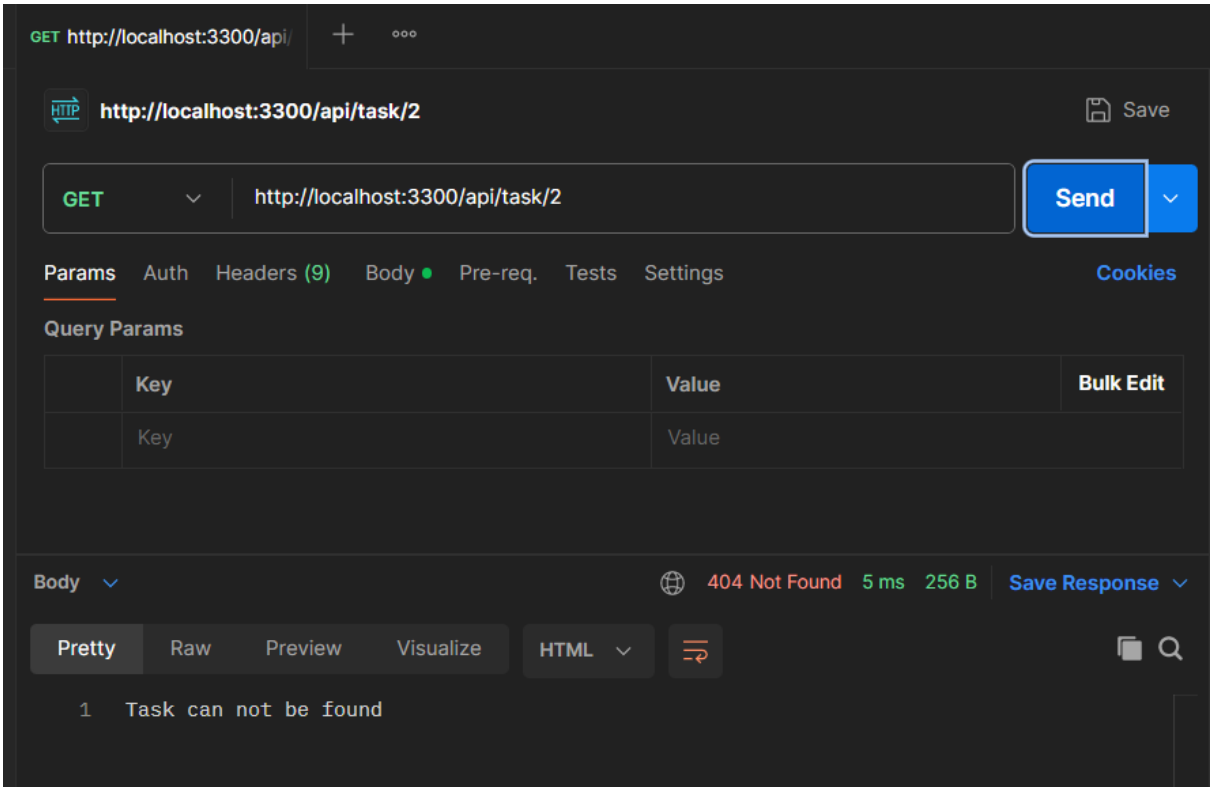
- Read task to see if update has been implemented



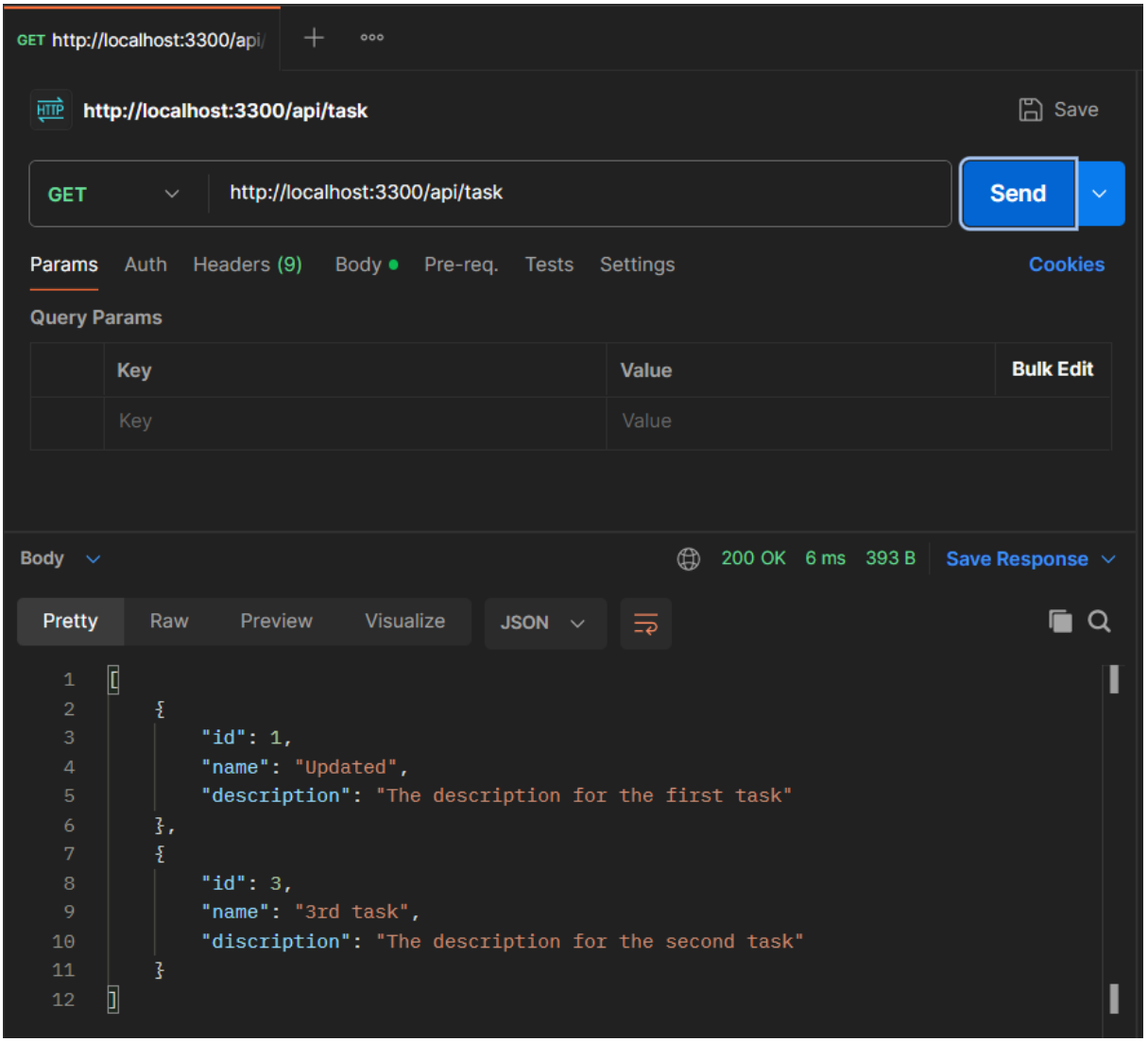
- Delete a task



- Try to read task to sea if it still exists



- Read all tasks to see if task has been deleted



Task 2

In task 2 the main focus is enhancing the previously developed RESTful API with
authentication and authorization using JSON Web Tokens (JWT) for secure access control. For me to perform this task I had to first install Json Web Token. The other thing I did was update the code by starting with designing and adding a user dataset, schema, and user verification, just as I did with with the tasks. The users dataset schema has roles with user been the default role and admin as an additional role. The normal user will have limited access where as the admin will have a universal access. This is show in the following snippets of code bellow.

```
const users = [];
let tasks = [
    {id:1, name:'1st task', description:'The description for
    {id:2, name:'2nd task', description:'The description for
];

//Joi schemas
const taskSchema = joi.object({
    name: joi.string().min(3).required(),
    description: joi.string().min(10).required()
});

const userSchema = joi.object({
    username: joi.string().min(3).required(),
    password: joi.string().min(6).required(),
    role: joi.string().valid('user', 'admin').default('user')
});

//Dat validation middleware
function validateTask(req, res, next) {
    const { error } = taskSchema.validate(req.body);
    if (error) return res.status(400).send(error.details[0].m
    next();
}

function validateUser(req, res, next) {
    const { error } = userSchema.validate(req.body);
    if (error) return res.status(400).send(error.details[0].m
    next();
}
```

I now had to update my code further by implementing user authentication endpoints to handle registration and login. Upon doing this, I updated my routes to make them secure, where by they will require user authentication to access by using tokens which are generated after login is successful. The addition of authentication limits what each user will be able to access after a successful login. The revised code with the updates from task 2 is as follows

```
const express = require('express');
const joi = require('joi');
const jwt = require('jsonwebtoken');
const bcrypt = require('bcrypt');
const app = express();
const port = 3300;

app.use(express.json());

const users = [];
let tasks = [
    {id:1, name:'1st task', description:'The description for
    {id:2, name:'2nd task', description:'The description for
];

// Secret key for JWT
const JWT_SECRET = 'TheSecretKey';

//Joi schemas
const taskSchema = joi.object({
    name: joi.string().min(3).required(),
    description: joi.string().min(10).required()
});

const userSchema = joi.object({
    username: joi.string().min(3).required(),
    password: joi.string().min(6).required(),
    role: joi.string().valid('user', 'admin').default('user')
});

//Dat validation middleware
function validateTask(req, res, next) {
    const { error } = taskSchema.validate(req.body);
    if (error) return res.status(400).send(error.details[0].m
    next();
}

function validateUser(req, res, next) {
```

```
    const { error } = userSchema.validate(req.body);
    if (error) return res.status(400).send(error.details[0].m
    next();
}

// Middleware to authenticate token
function authenticateToken(req, res, next) {
    const authHeader = req.headers['authorization'];
    const token = authHeader && authHeader.split(' ')[1];
    if (token == null) return res.status(401).json({ error: '

    jwt.verify(token, JWT_SECRET, (err, user) => {
        if (err) return res.status(403).json({ error: 'Invali
            req.user = user;
            next();
    });
}

// Middleware for role-based authorization
function authorize(roles = []) {
    return (req, res, next) => {
        if (!roles.includes(req.user.role)) {
        return res.status(403).json({ error: 'Unauthorized' }
        }
        next();
    }
}

// User registration
app.post('/register', validateUser, async (req, res) => {
    try {
        const hashedPassword = await bcrypt.hash(req.body.pas
        const user = {
            id: users.length + 1,
            username: req.body.username,
            password: hashedPassword,
            role: req.body.role || 'user'
        };
```

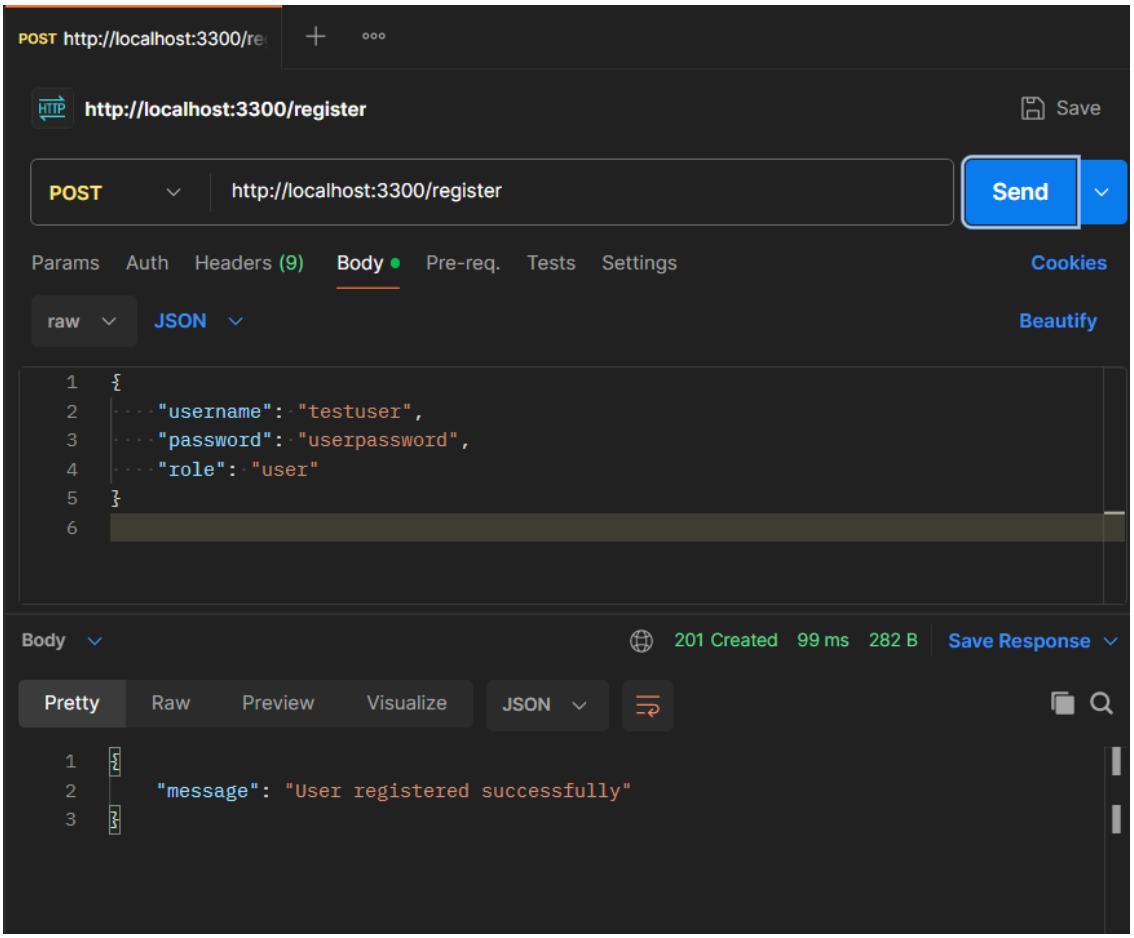
```
        users.push(user);
        res.status(201).json({ message: 'User registered succ
    } catch (error) {
        res.status(500).json({ error: 'Error registering user
    }
});

// User login
app.post('/login', async (req, res) => {
    const user = users.find(user => user.username === req.bod
    if (!user) return res.status(400).json({ error: 'User not

    try {
        if (await bcrypt.compare(req.body.password, user.pass
            const accessToken = jwt.sign({ id: user.id, usern
            res.json({ accessToken: accessToken });
        } else {
            res.status(401).json({ error: 'Invalid credential
        }
    } catch (error) {
        res.status(500).json({ error: 'Error logging in' });
    }
});

//GET all tasks
app.get('/api/task', authenticateToken, (req, res) => {
    const userTasks = tasks.filter(task => task.userId === re
    res.json(userTasks);
});

//GET specific task
app.get('/api/task/:id', authenticateToken, (req, res) => {
    const task = tasks.find(t => t.id === parseInt(req.params
    if (!task) return res.status(404).json({ error: 'Task not
    res.json(task);
});

//Create new task
```

```
app.post('/api/task', authenticateToken, validateTask, (req,
    const task = {
        id: tasks.length + 1,
        name: req.body.name,
        description: req.body.description,
        userId: req.user.id
    };
    tasks.push(task);
    res.status(201).json(task);
});

//Update task
app.put('/api/task/:id', authenticateToken, validateTask, (re
    const task = tasks.find(t => t.id === parseInt(req.params
    if (!task) return res.status(404).json({ error: 'Task not

    task.name = req.body.name;
    task.description = req.body.description;
    res.json(task);
});

//DELETE task
app.delete('/api/task/:id', authenticateToken, (req, res) =>
    const taskIndex = tasks.findIndex(t => t.id === parseInt(
    if (taskIndex === -1) return res.status(404).json({ error

    tasks.splice(taskIndex, 1);
    res.json({ message: 'Task deleted successfully' });
});

//Admin route to get all tasks, including for other users'
app.get('/api/admin/task', authenticateToken, authorize(['adm
    res.json(tasks);
});

//Error handling middleware
app.use((err, req, res, next) => {
    console.error(err.stack);
```

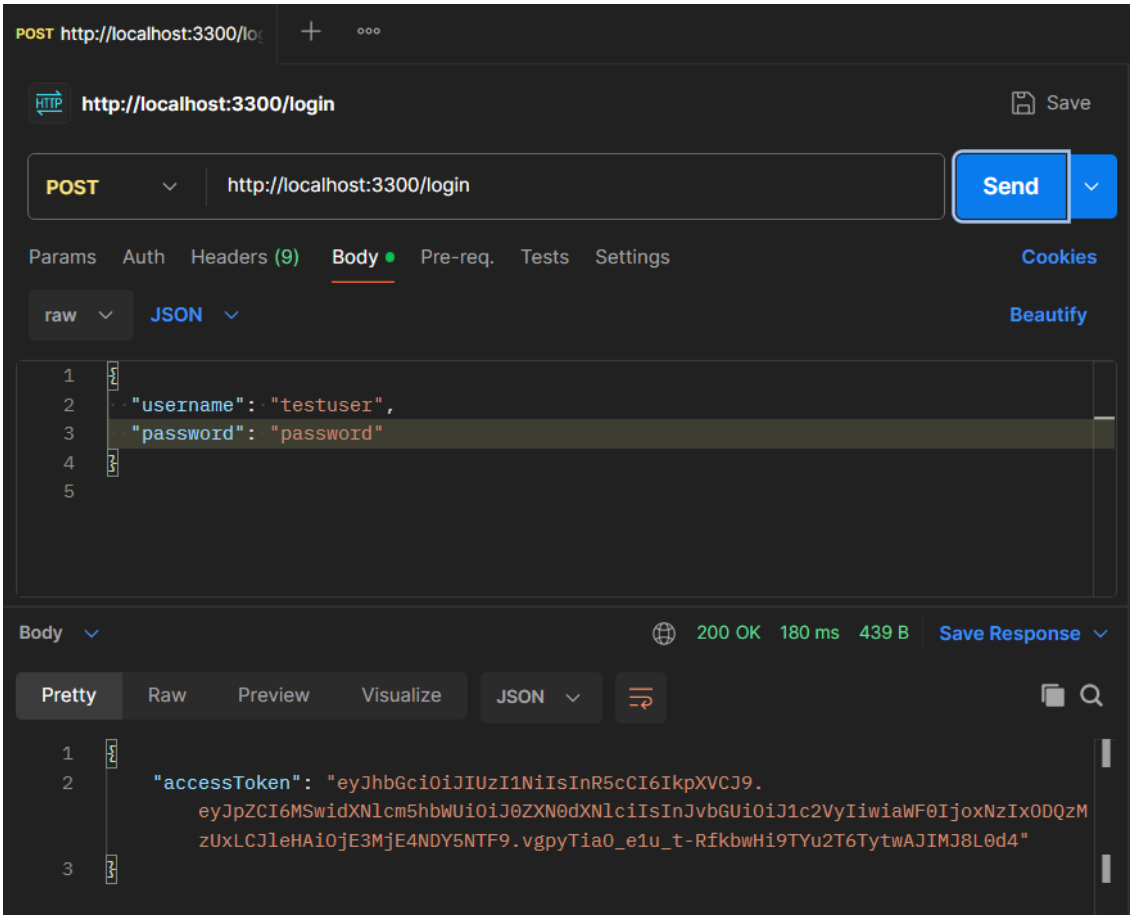
```
    res.status(500).json({ error: 'Something went wrong!' });
});


app.listen(port, ()=>{
    console.log(`Server is running on port http://localhost:$
})
```

After making all these changes or updates, I now had to test the API routes once again to make sure they are working correctly. The test results of the API routes are as follows in the same order they are displayed

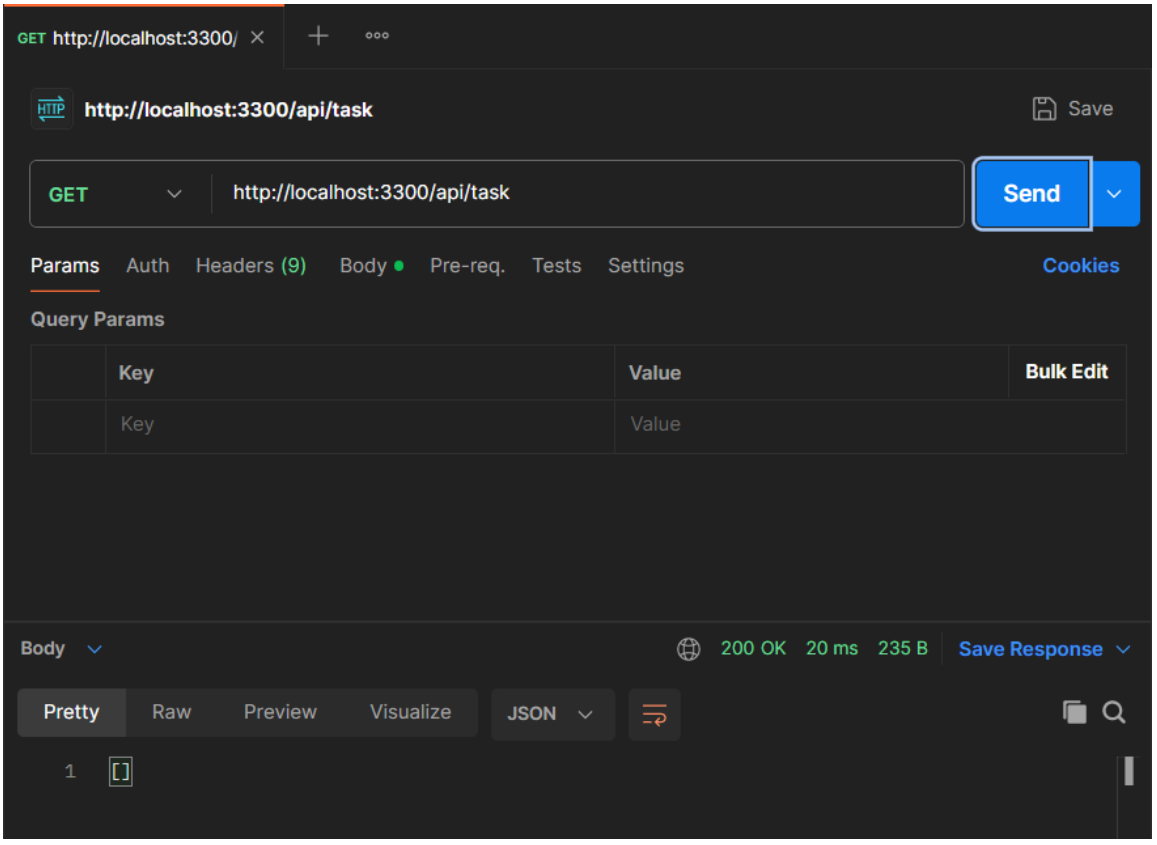
- Normal user creation



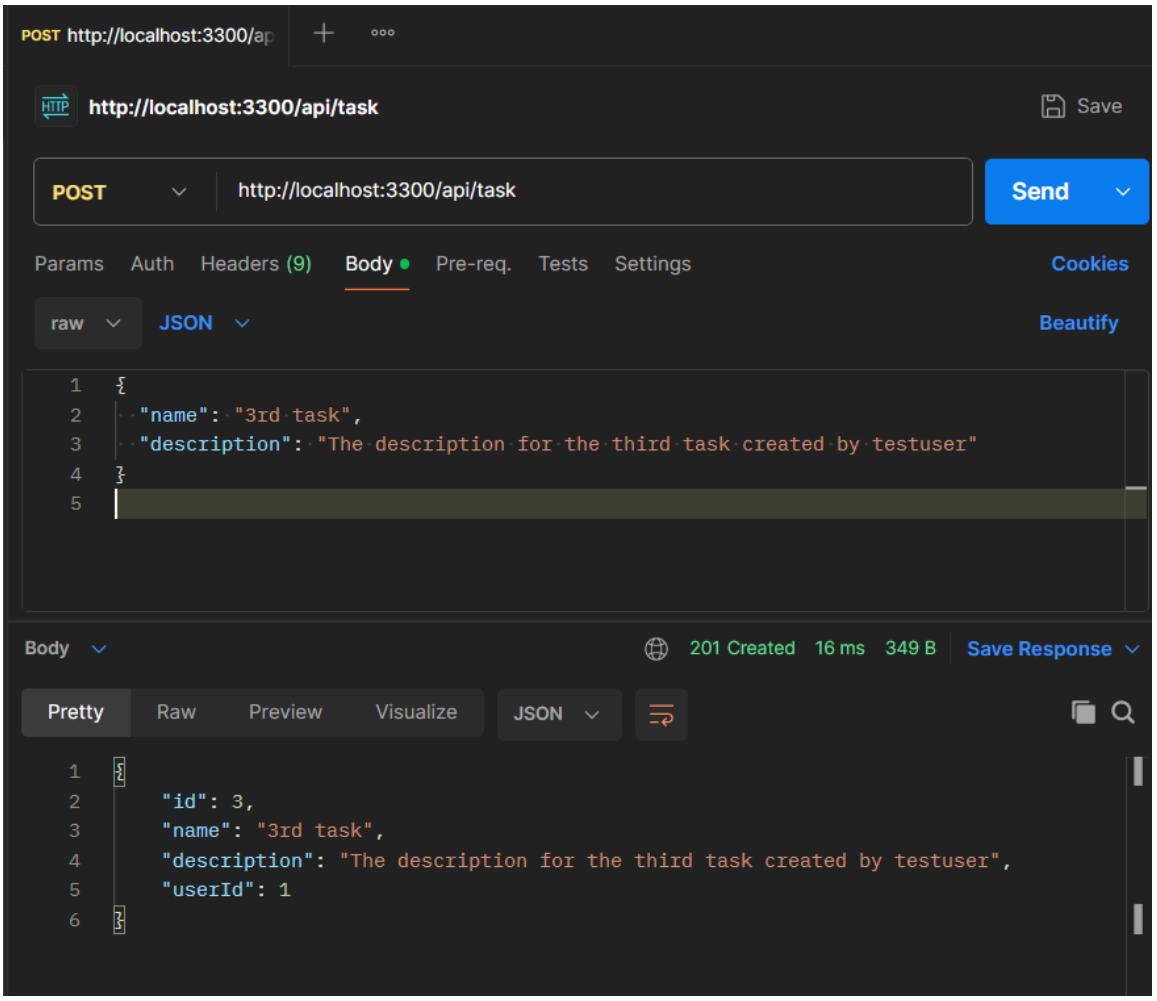
- User login



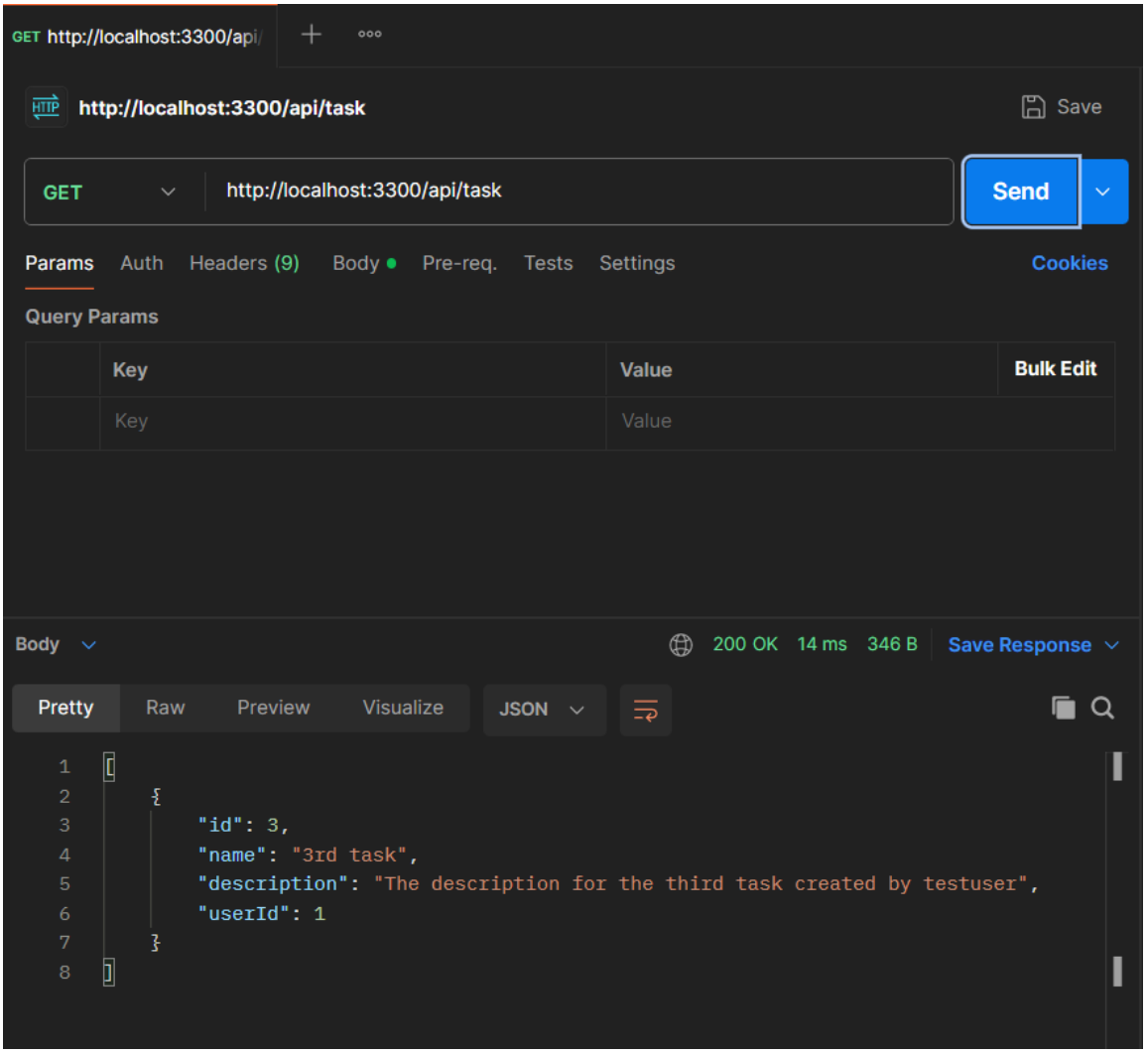
- User trying to read all tasks



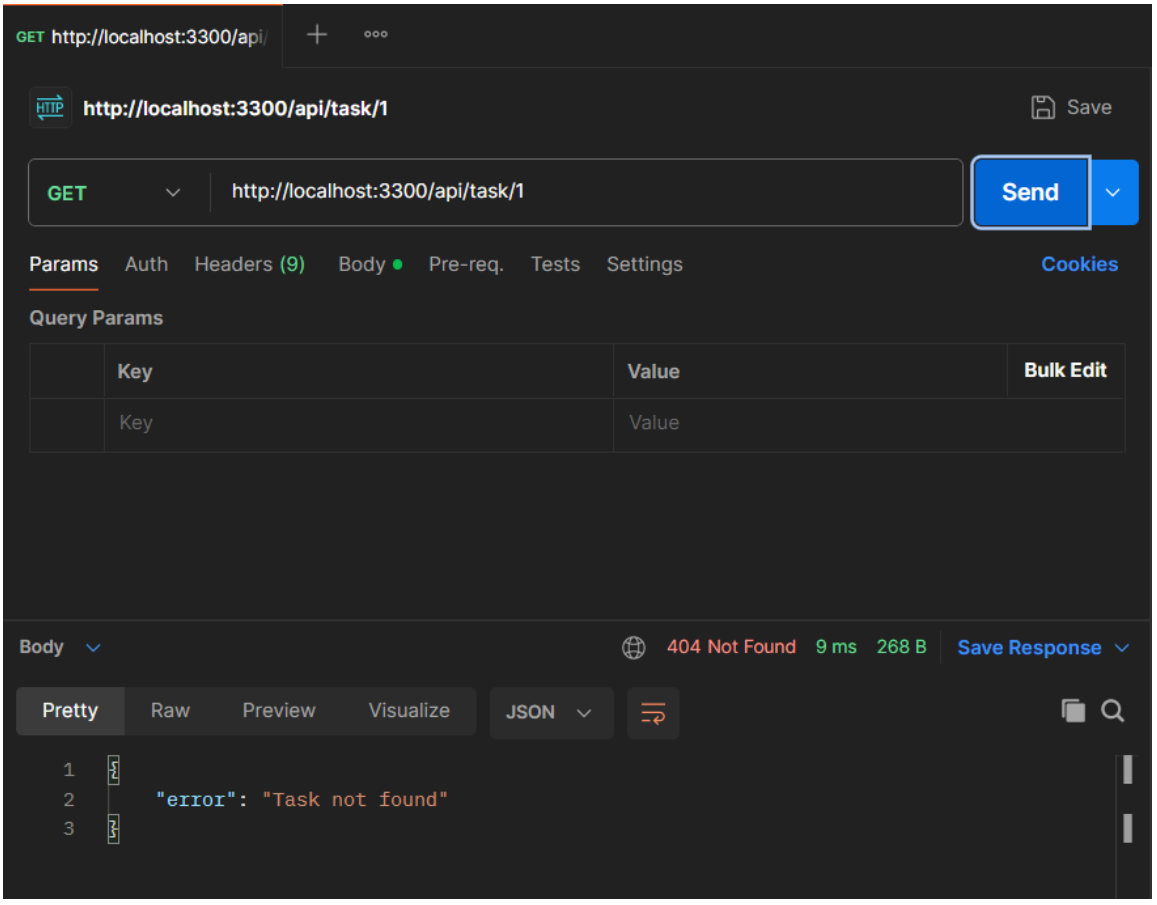
- User creating a task



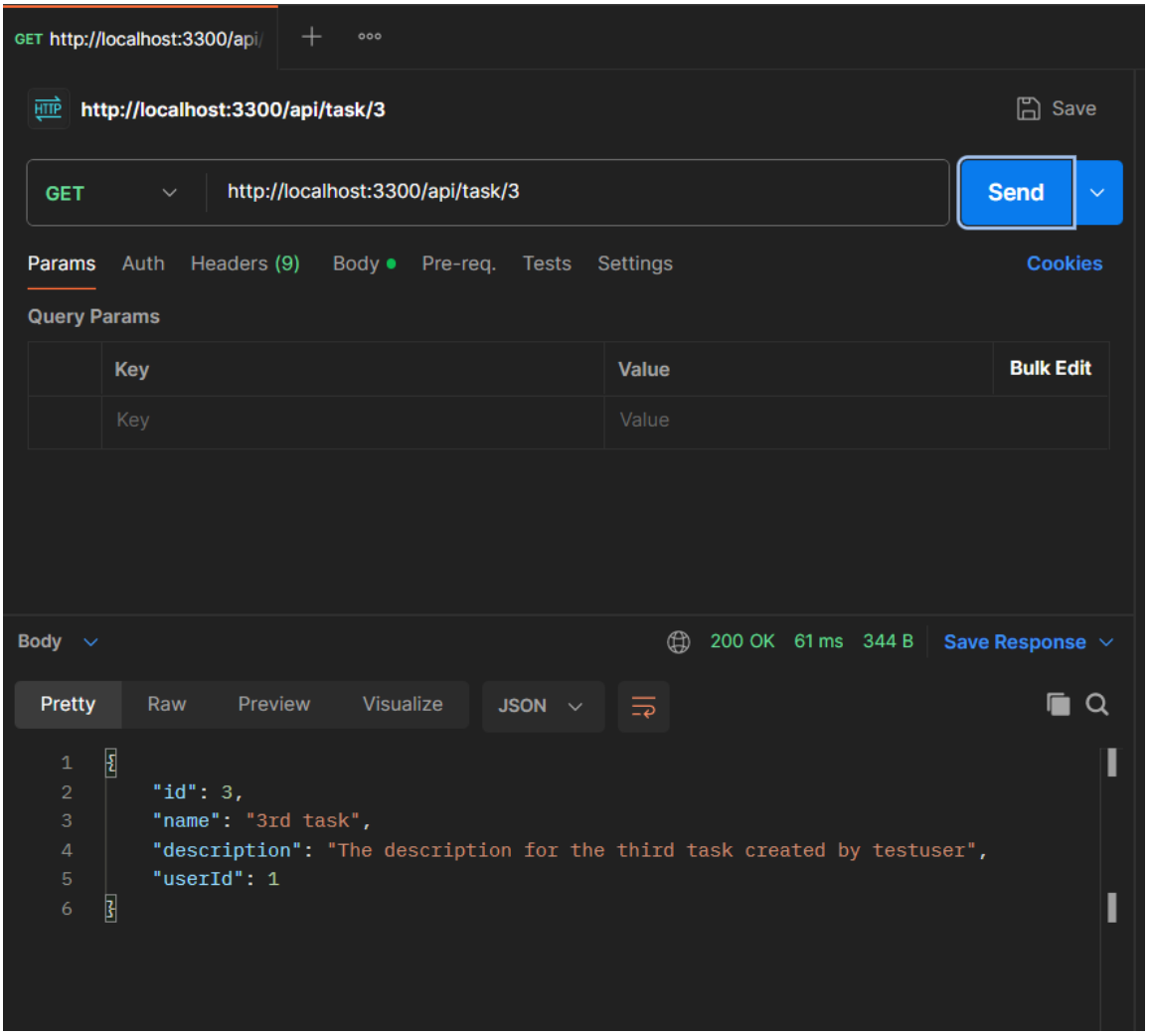
- User reading tasks after creating one



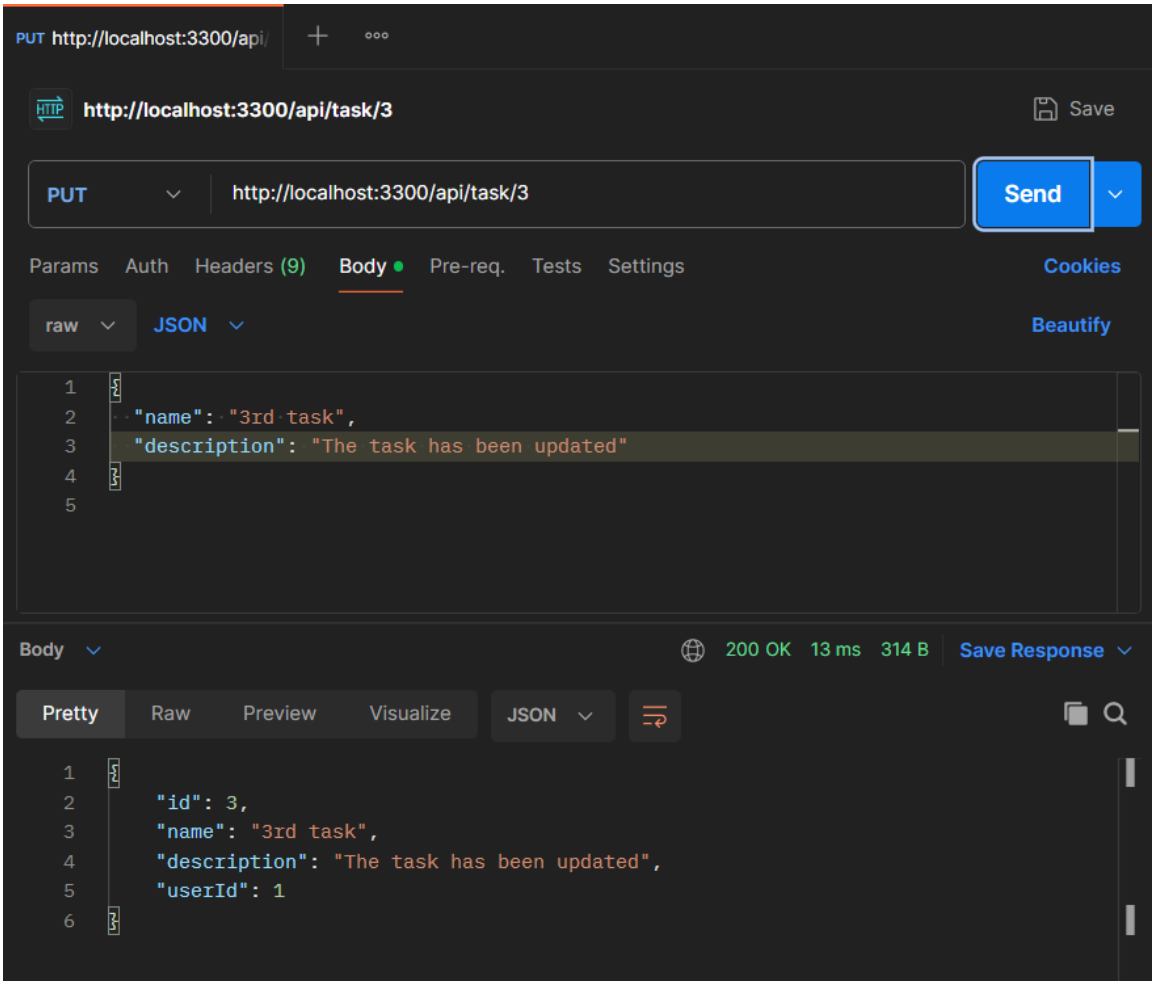
- User trying to access tasks they did not create



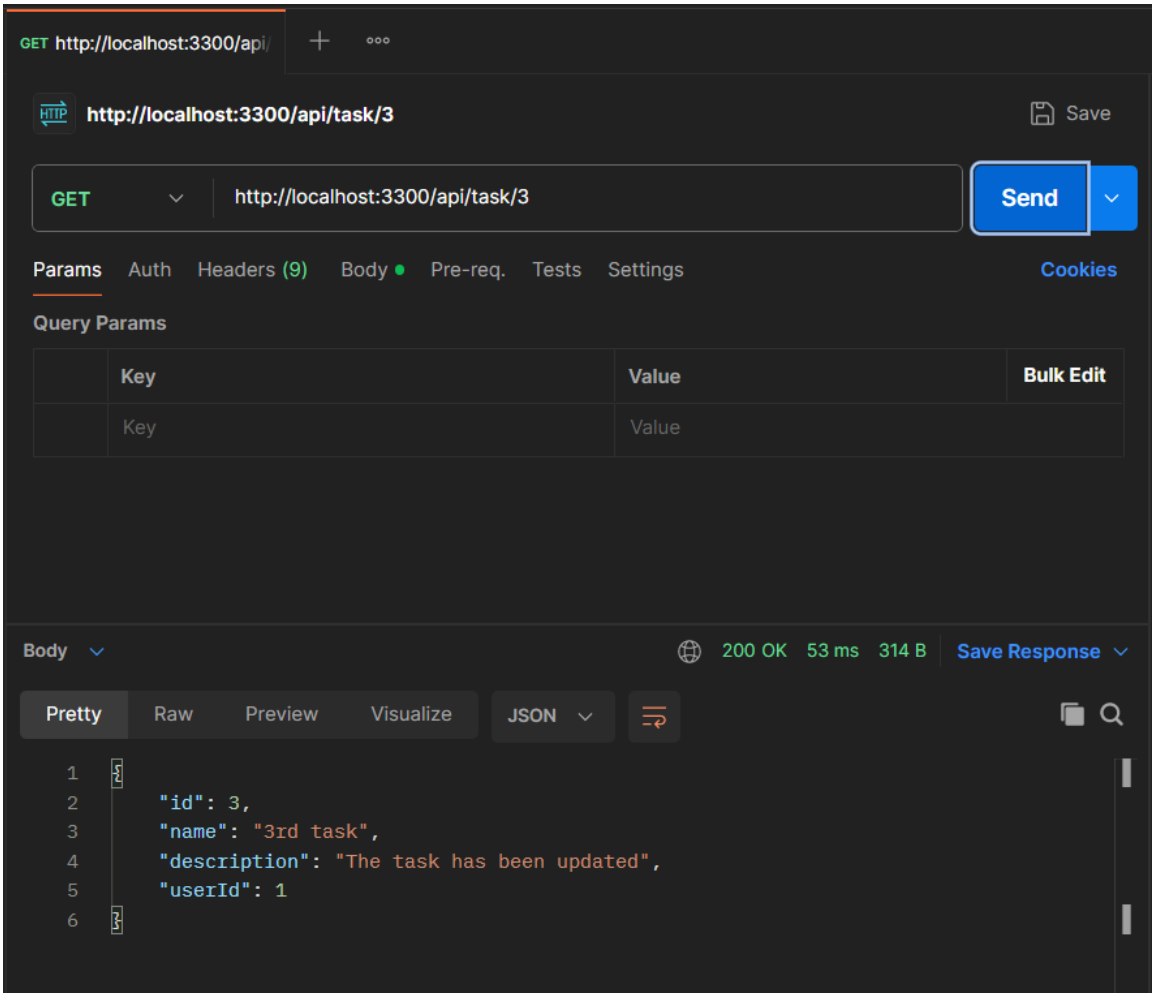
- User trying to read the task they created



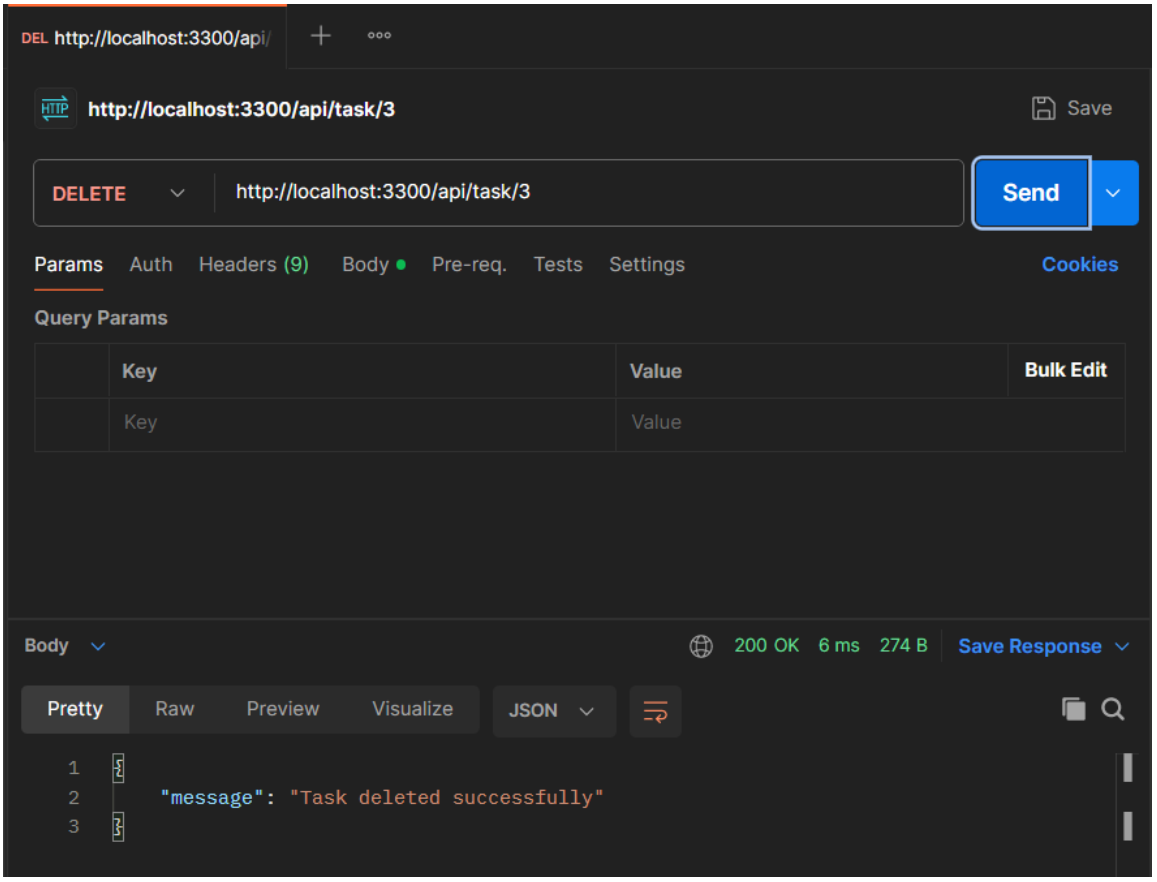
- User updating task



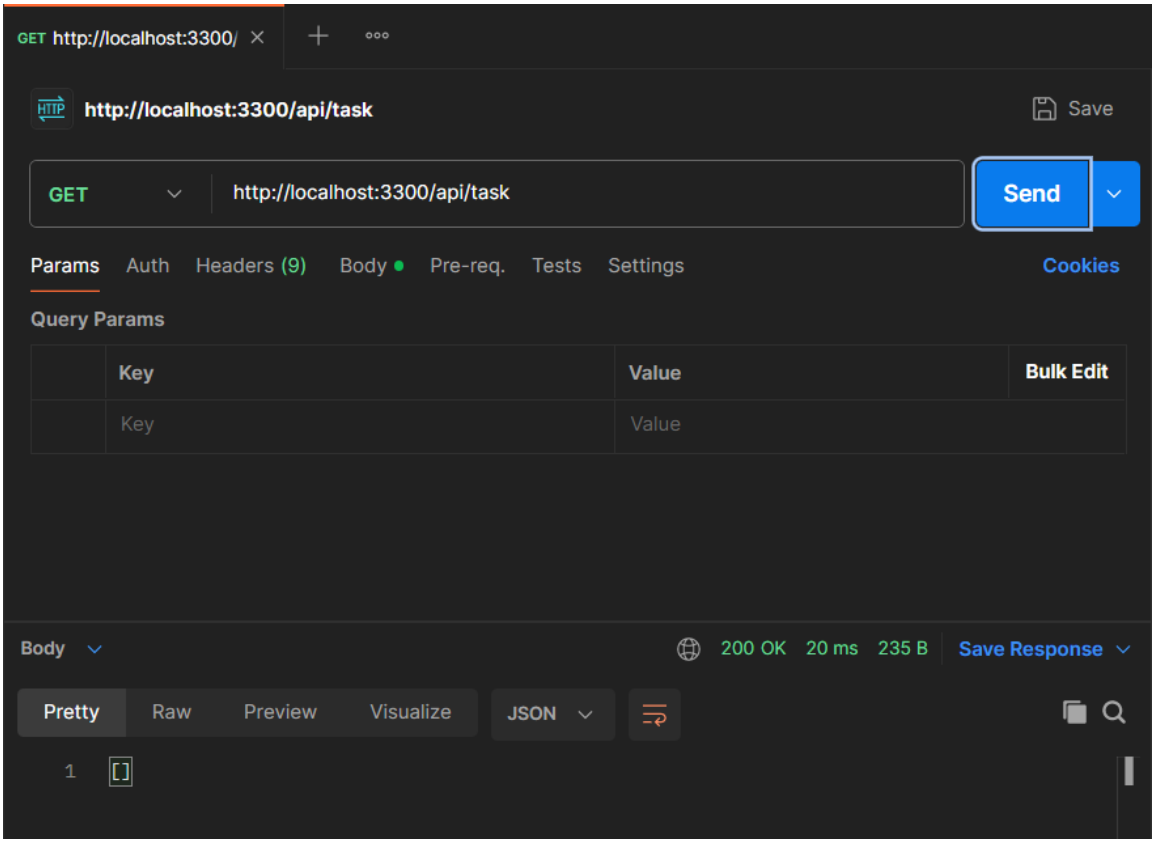
- User reading new updated task



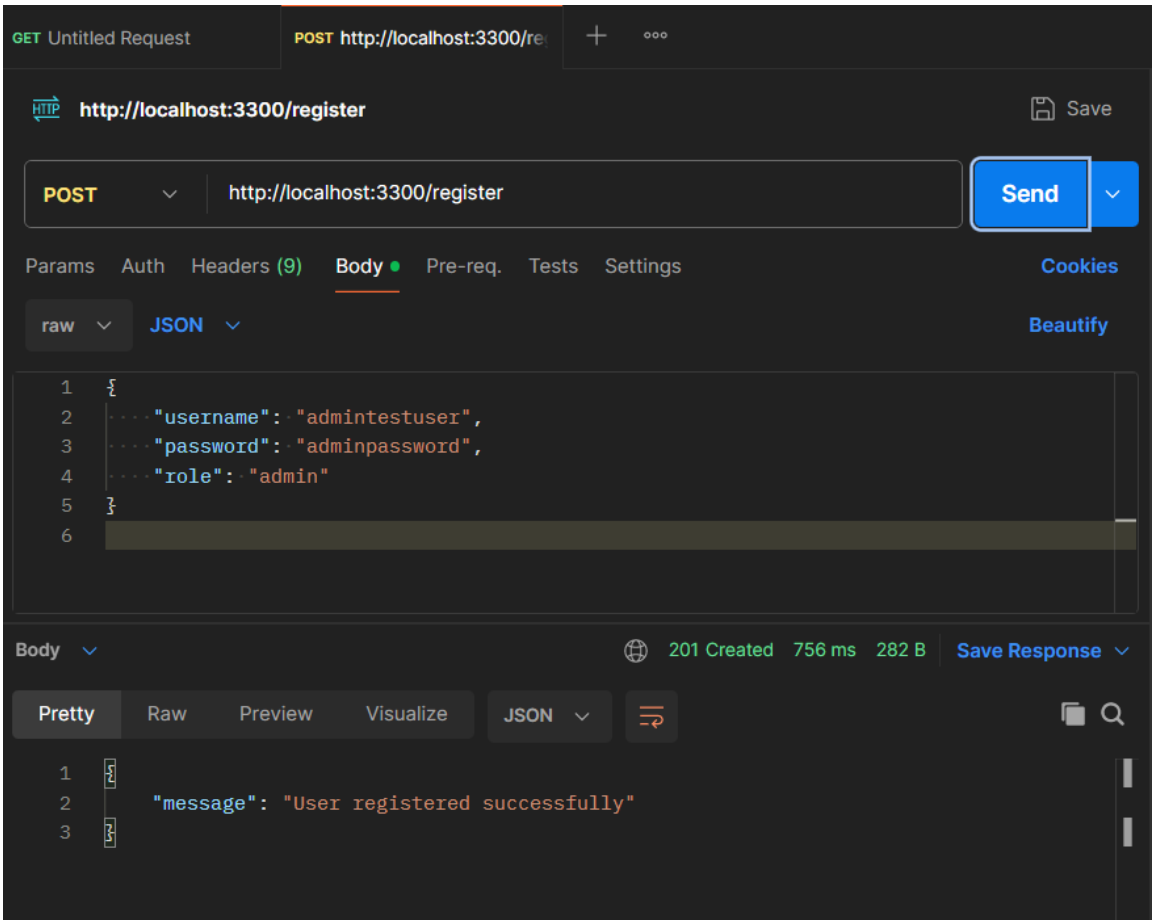
- User deleting task



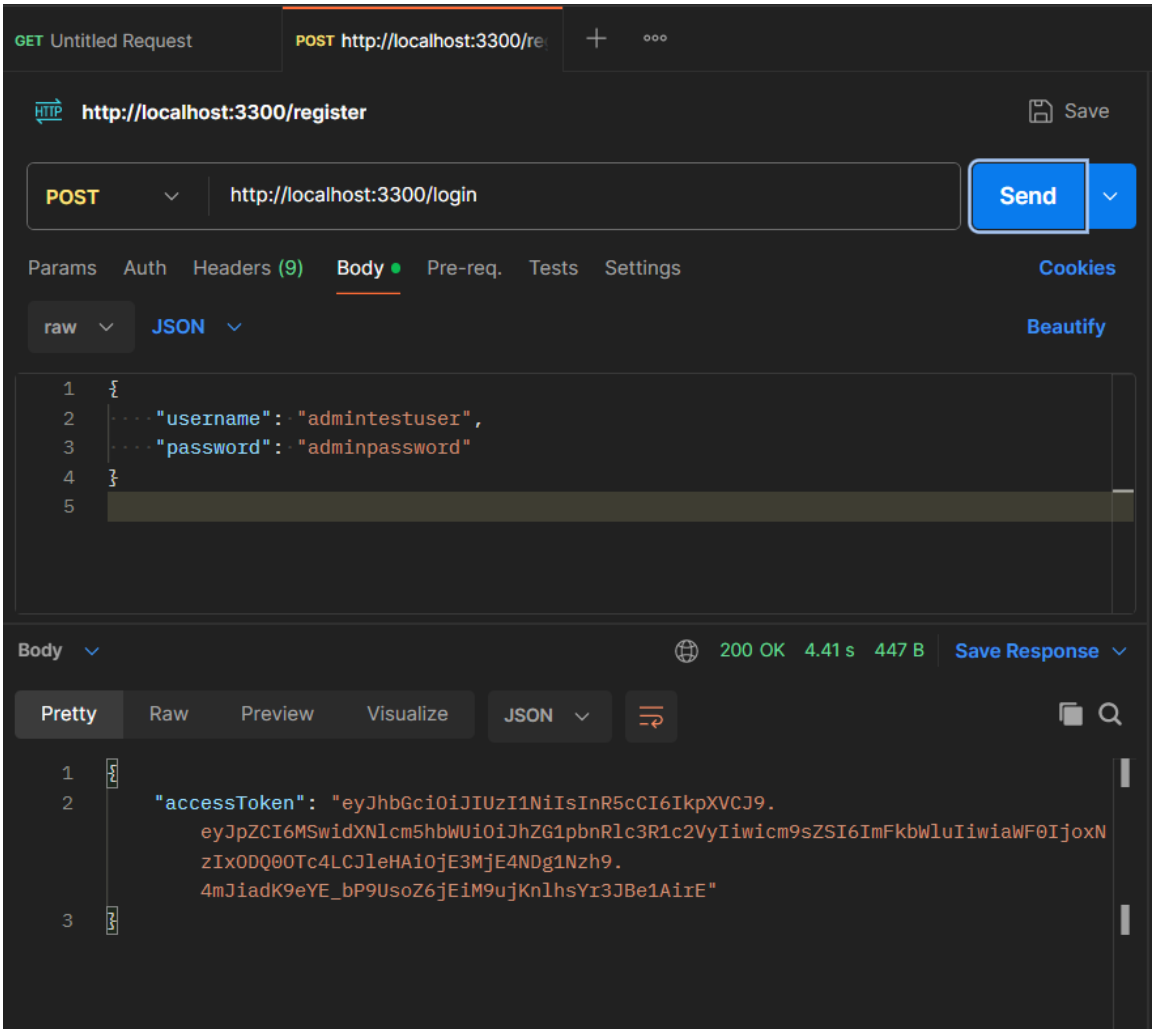
- User trying to read tasks after deleting one



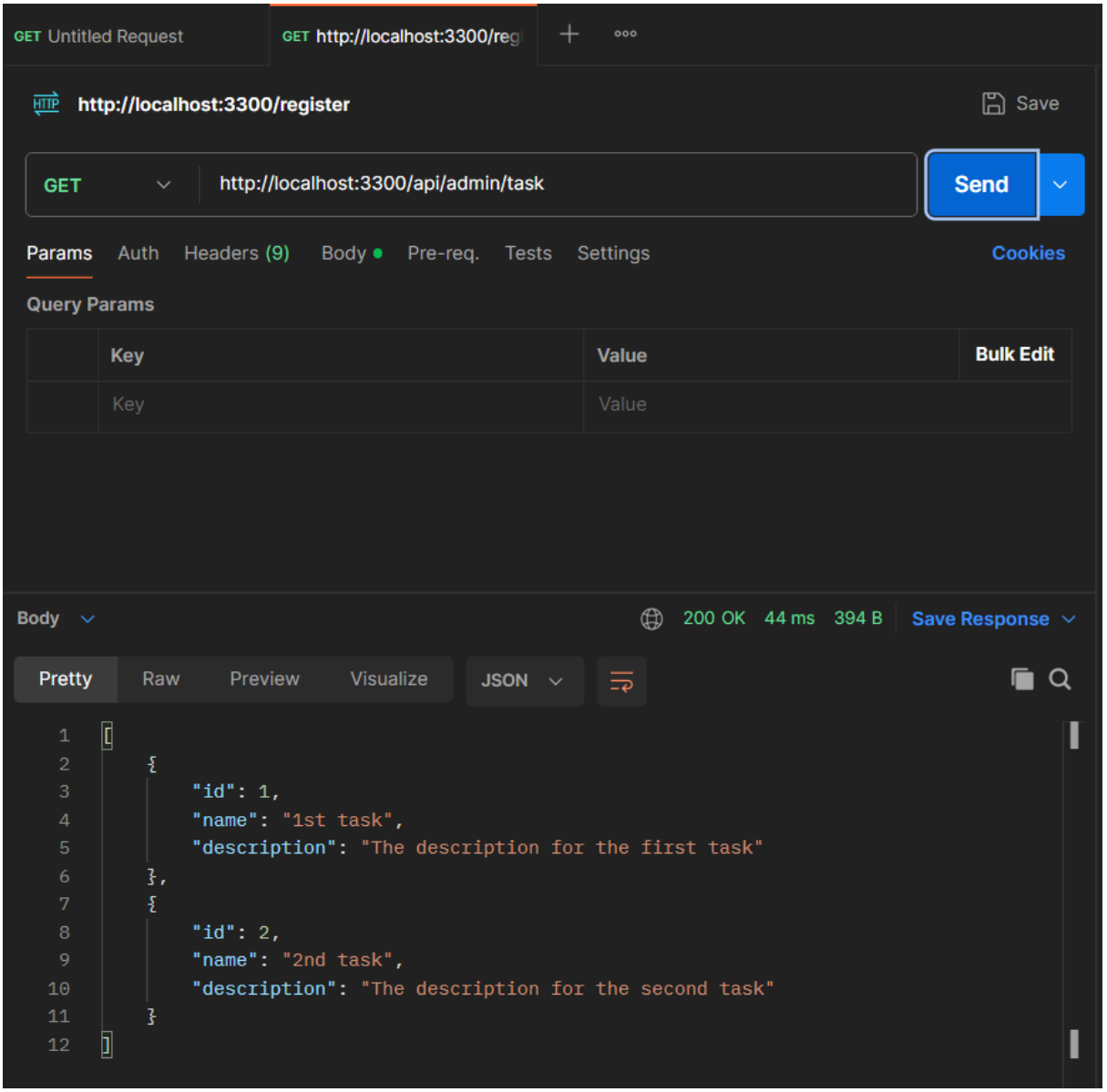
- Registering an admin user



- Admin user trying to login



- Admin user reading all tasks


GET Untitled Request
GET http://localhost:3300/reg
http://localhost:3300/register
Save
GET
http://localhost:3300/api/admin/task
Send
Params
Auth
Headers (9)
Body
Pre-req.
Tests
Settings
Cookies
Query Params
Key
Value
Bulk Edit
Key
Value
Body
200 OK
44 ms
394 B
Save Response
Pretty
Raw
Preview
Visualize
JSON
[
{
"id": 1,
"name": "1st task",
"description": "The description for the first task"
},
{
"id": 2,
"name": "2nd task",
"description": "The description for the second task"
}
]