COMP 1842

Week1 10 - RESTful APIs

Matt Prichard

Introduction

API

REST

Methods/verbs

Coding a simple API

Testing in Postcode

API

What is an API?

- APIs are mechanisms that enable two software components to communicate with each other using a set of definitions and protocols.
- For example, the weather bureau's software system contains daily weather data.
- The weather app on your phone "talks" to this system via APIs and shows you daily weather updates on your phone.

API

What does API stand for?

- API stands for Application Programming Interface. In the context of APIs, the word Application refers to any software with a distinct function.
- Interface can be thought of as a contract of service between two applications. This contract defines how the two communicate with each other using requests and responses.
- Their API documentation contains information on how developers are to structure those requests and responses.

How do APIs work?

- API architecture is usually explained in terms of client and server.
- The application sending the request is called the client, and the application sending the response is called the server.
- So in the weather example, the bureau's weather database is the server, and the mobile app is the client.

How do APIs work?

- A program that has an API implies that some parts of its data is exposed for the client to use. The client could be the frontend of the same program or an external program.
- In order to get this data, a structured request has to be sent to the API. If the request meets the desired requirements, a response which contains the data gets sent back to where the request was made. This response usually comes in the form of JSON or XML data.

API example

- Imagine visiting a new restaurant. You're there to order food, and since you haven't been there before, you don't exactly know what type of food they serve.
- The server then approaches you with a menu so you can pick what you'd like to eat. After making your choice, the server then goes to the kitchen and gets your food.
- In this case, the server is the API who is connecting you to the kitchen. The API's documentation is the menu. The request is made when you pick what you'd like to eat, and the response is the food being served.

What is a web API?

- A Web API or Web Service API is an application processing interface between a web server and web browser. All web services are APIs but not all APIs are web services. REST API is a special type of Web API that uses a standard architectural style.
- The different terms around APIs, like Java API or service APIs, exist because historically, APIs were created before the world wide web. Modern web APIs are REST APIs and the terms can be used interchangeably.

What are REST APIs?

- REST stands for REpresentational State Transfer. REST defines a set of functions like GET, PUT, DELETE, etc. that clients can use to access server data. Clients and servers exchange data using HTTP.
- The main feature of REST API is statelessness. Statelessness means that servers do not save client data between requests.
- Client requests to the server are similar to URLs you type in your browser to visit a website. The response from the server is plain data, without the typical graphical rendering of a web page.

REST – History

REST was defined by Roy Fielding, a computer scientist.

He presented the REST principles in his PhD dissertation in 2000.

Further reading:

https://en.wikipedia.org/wiki/Representational_state_transfer#References

2 key terms - client

Client

- The client is the person or software who uses the API. It can be a developer, for example you, as a developer, can use Twitter API to read and write data from Twitter, create a new tweet and do more actions in a program that you write. Your program will call Twitter's API.
- The client can also be a web browser. When you go to Twitter website, your browser is the client who calls Twitter API and uses the returned data to render information on the screen.

2 key terms - Resource

Resource

- A resource can be any object the API can provide information about.
- In Instagram's API, for example, a resource can be a user, a photo, a hashtag.
- Each resource has a unique identifier.
- The identifier can be a name or a number.

REpresentational State Transfer.

- It means when a RESTful API is called, the server will transfer to the client a *representation* of the *state* of the requested resource.
- For example, when a developer calls Instagram API to fetch a specific user (the resource), the API will return the state of that user, including their name, the number of posts that user posted on Instagram so far, how many followers they have, and more.
- The representation of the state will most likely be be in a JSON format

API call

What the server does when you, the client, call one of its APIs depends on 2 things that you need to provide to the server:

1. An identifier for the resource you are interested in. This is the URL for the resource, also known as the endpoint. In fact, URL stands for Uniform Resource Locator.
2. The operation you want the server to perform on that resource, in the form of an HTTP method, or verb. The common HTTP methods are GET, POST, PUT, and DELETE.

API call example

- For example, fetching a specific Twitter user, using Twitter's RESTful API, will require a URL that identifies that user and the HTTP method GET.
- Another example, this URL: https://twitter.com/OzzyOsbourne has the unique identifier for Ozzy Osbourne's (AKA the Prince of darkness) Twitter page, which is his username, OzzyOsbourne.
- Twitter uses the username as the identifier, and indeed Twitter usernames are unique — there are no 2 Twitter users with the same username.
- The HTTP method GET indicates that we want to get the state (currently sober) of that user.

Methods / Verbs

refresher

GET

- GET is the simplest type of HTTP request method—the one that browsers use each time you click a link or type a URL into the address bar. It instructs the server to transmit the data identified by the URL to the client.
- Data should never be modified on the server side as a result of a GET request.
- In this sense, a GET request is read-only, but of course, once the client receives the data, it is free to do any operation with it on its own side—for instance, format it for display.

POST

- Use POST APIs to create new subordinate resources, e.g., a file is subordinate to a directory containing it or a row is subordinate to a database table.
- When talking strictly about REST, POST methods are used to create a new resource into the collection of resources.

PUT

- Use PUT APIs primarily to update an existing resource (if the resource does not exist, then API may decide to create a new resource or not).
- PUT API Response Codes
  - If a new resource has been created by the PUT API, the origin server MUST inform the user agent via the HTTP response code 201 (Created) response.
  - If an existing resource is modified the 200 (OK) response code SHOULD be sent to indicate successful completion of the request.

DELETE

- As the name applies, DELETE APIs delete the resources (identified by the Request-URI).
- If you DELETE a resource, it's removed from the collection of resources.
- DELETE should perform the contrary of PUT; it should be used when you want to delete the resource identified by the URL of the request.

CRUD

Functionality mapped to HTTP ‘verbs’ or ‘request methods’

CREATE = POST

READ = GET

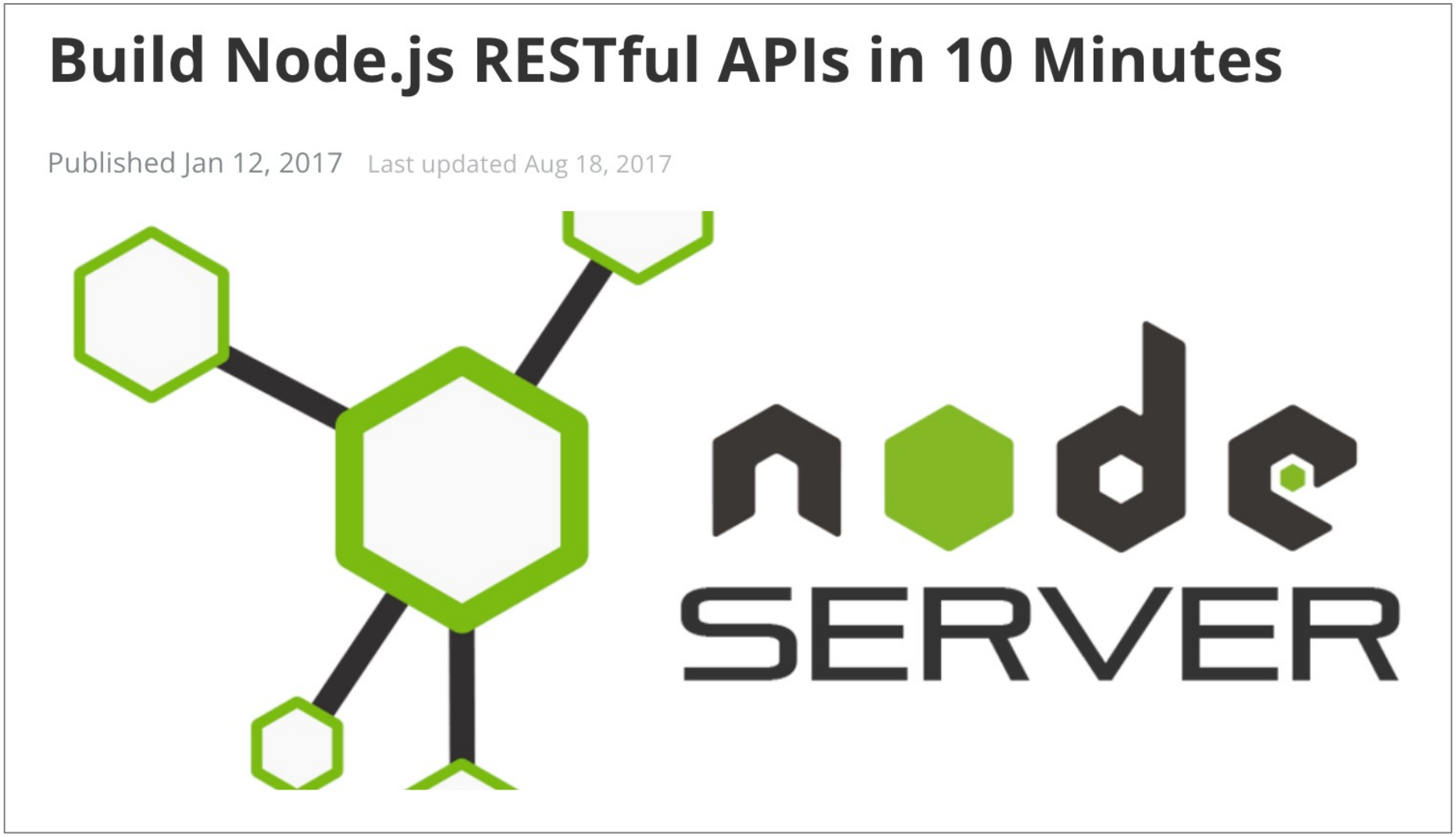
UPDATE = PUT

DELETE = DELETE

HTTP Method	CRUD	Collection Resource (e.g. /users)	Single Resource (e.g. /users/123)
POST	Create	201 (Created), ‘Location’ header with link to /users/{id} containing new ID	Avoid using POST on a single resource
GET	Read	200 (OK), list of users. Use pagination, sorting, and filtering to navigate big lists	200 (OK), single user. 404 (Not Found), if ID not found or invalid
PUT	Update/Replace	405 (Method not allowed), unless you want to update every resource in the entire collection of resource	200 (OK) or 204 (No Content). Use 404 (Not Found), if ID is not found or invalid
DELETE	Delete	405 (Method not allowed), unless you want to delete the whole collection — use with caution	200 (OK). 404 (Not Found), if ID not found or invalid

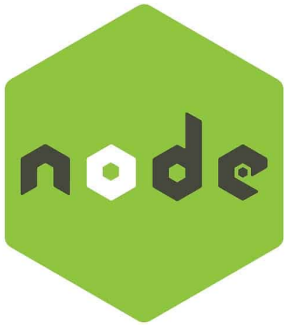
Making a ‘simple’ REST API

Based on the following tutorial: https://www.codementor.io/@olatundegaruba/nodejs-restful-apis-in-10-minutes-q0sgsfhbd



Assumptions – Prerequisites

1. You have node and npm installed.
2. You have mongoDB installed locally or are using your University Mongo account.
3. You have nodemon installed globally as we did last week.



nodemon

Setting up the structure and installing the dependencies

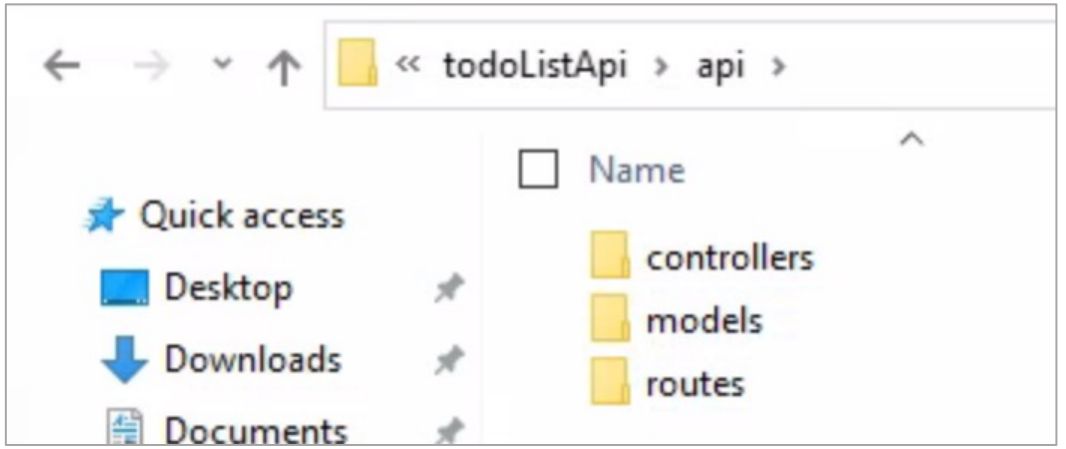
File structure - similar to last week

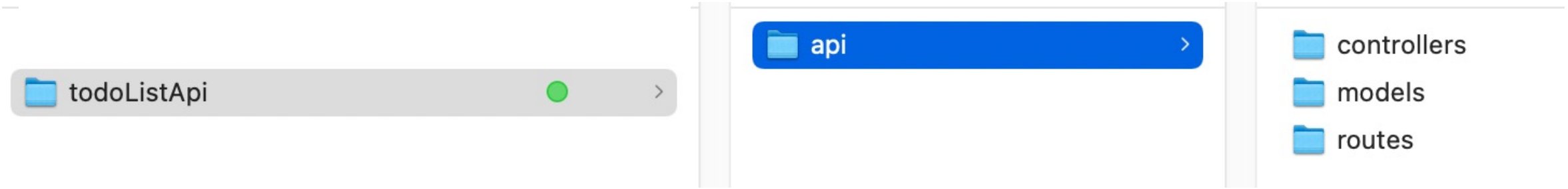
In the root of your G Drive create folders

todoListApi > api > controllers

> models

> routes





Using the node command prompt

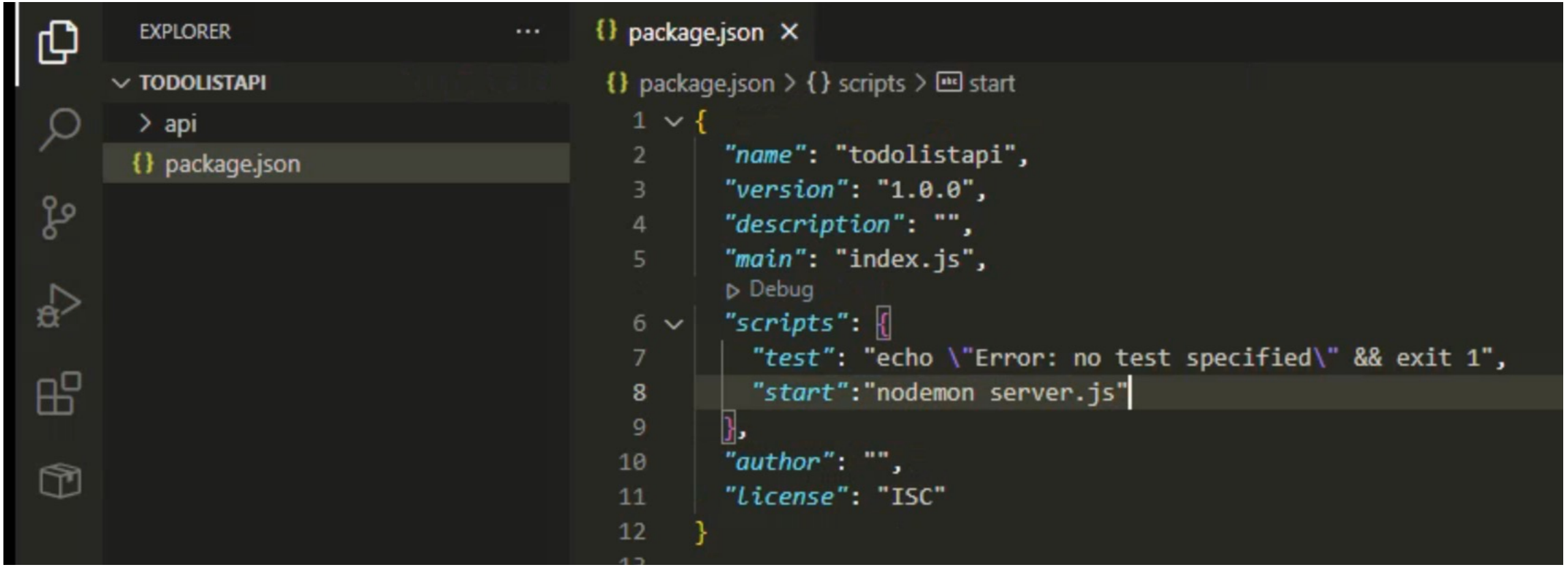
```
cd todoListApi

npm init -y
```

```
C:\Users\pm76\fakeg\todoListApi>npm init -y
Wrote to C:\Users\pm76\fakeg\todoListApi\package.json:

{
  "name": "todolistapi",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "nodemon server.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2",
    "mongoose": "^6.2.4"
  },
  "devDependencies": {},
  "keywords": [],
  "description": ""
}



C:\Users\pm76\fakeg\todoListApi>
```

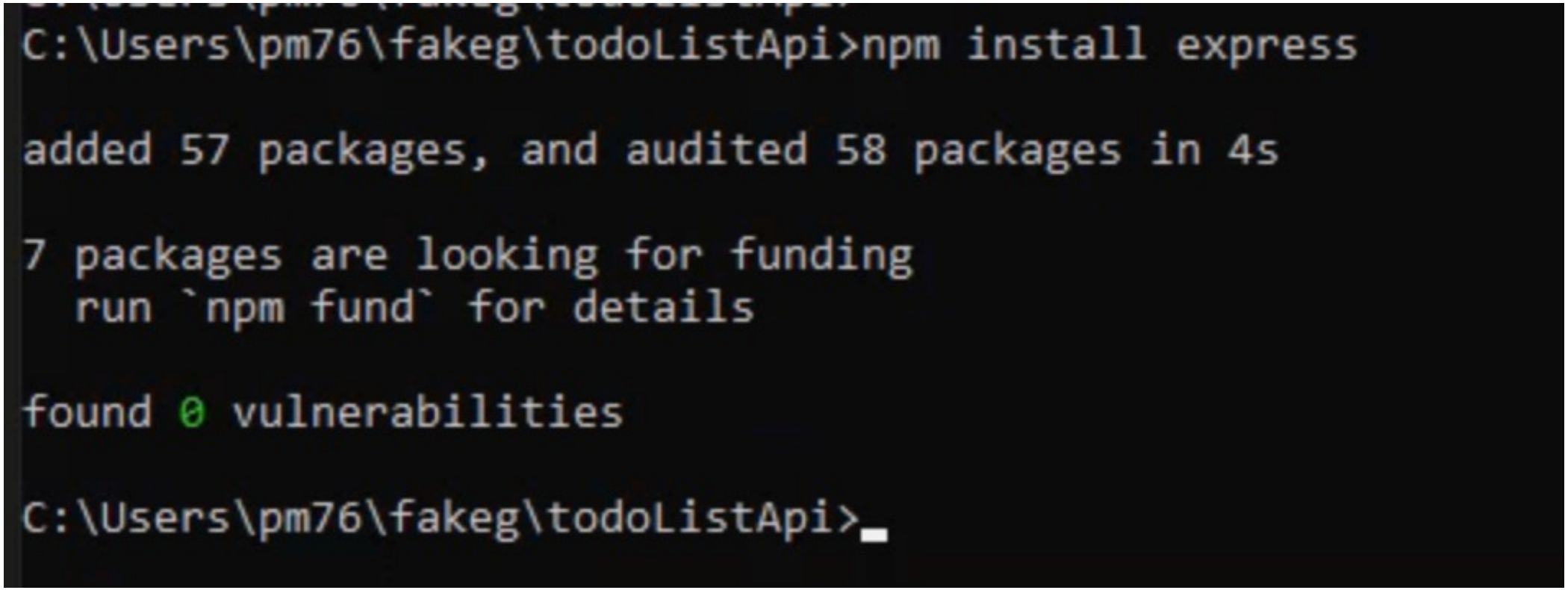
Open todoListApi folder in VS code, In package.json add

`"start": "nodemon server.js”`

To the scripts around line 8

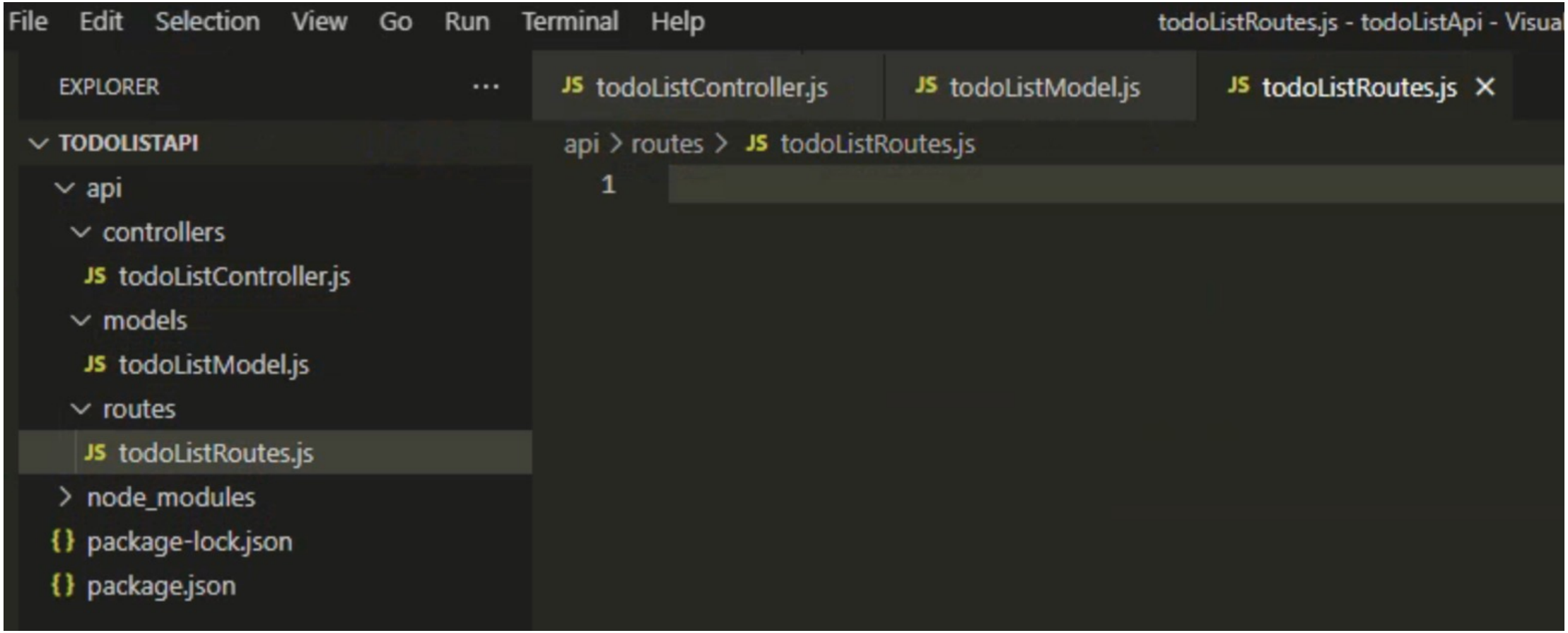


npm install express



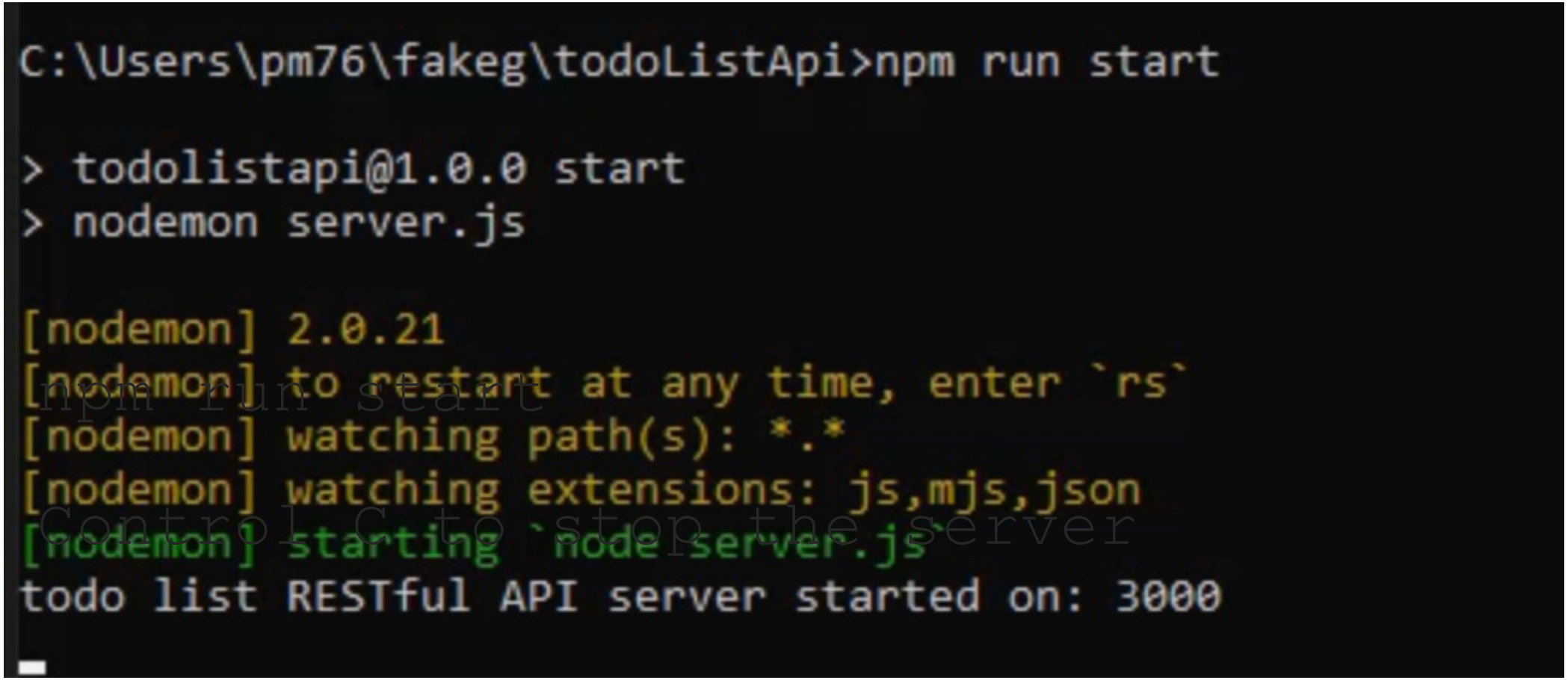
Create the following 3 empty JS files in their relative folders:

todoListController.js, todoListModel.js, todoListRoutes.js



Create server.js in the root todoListApi folder and add the following code

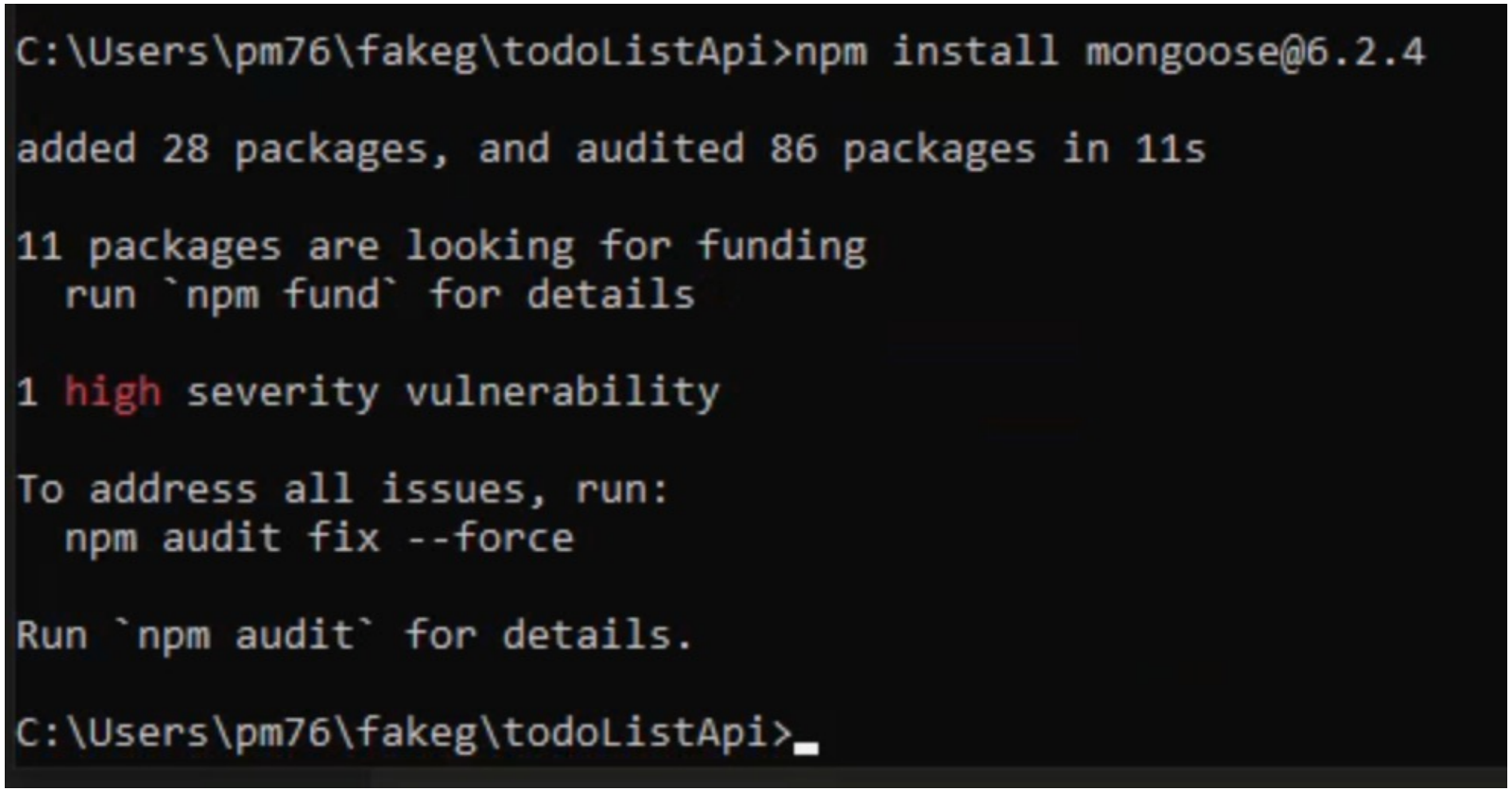
npm run start



Control C to stop the server

Install mongoose

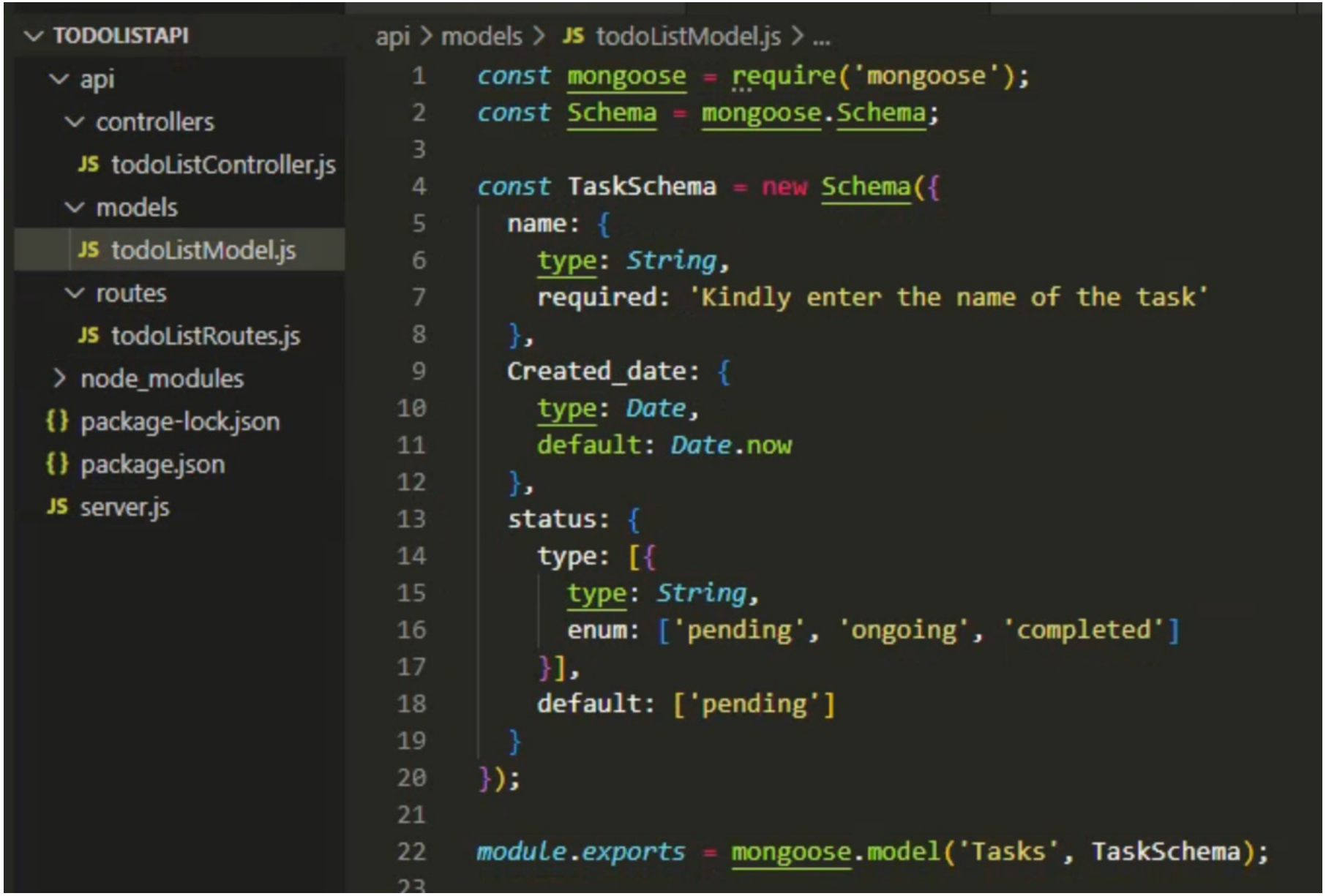
```
npm install mongoose@6.2.4
```



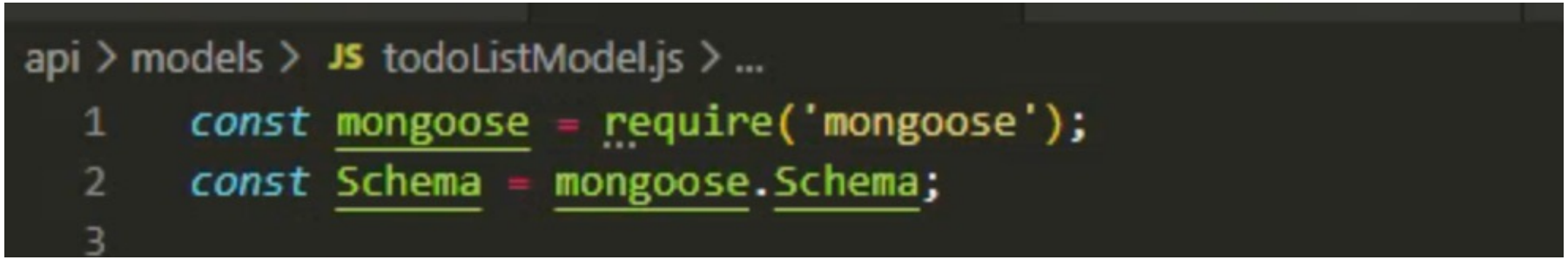
Coding the API

todoListModel.js

todoListModel.js – full code



todoListModel.js – detail ptl



This is where we will define the requirements of the data model that we intend to map to our database in MongoDB.

We are using the Mongoose library as the messenger between our app and our database.

The first thing to do inside of this file is require Mongoose and use the Schema class from the Mongoose library.

todoListModel.js – detail pt 2

```
4   const TaskSchema = new Schema({
5     name: {
6       type: String,
7       required: 'Kindly enter the name of the task'
8     },
```

Now we can begin to write the format for our model ‘TaskSchema’ by creating a new instance of the Schema class, line 4.

Our user model will have 3 attributes: name, date and status

Each attribute will be defined using a key:value pair, with the key being the name of the attribute, and the value being its type, in this case ‘String’. We can also add some basic validation too.

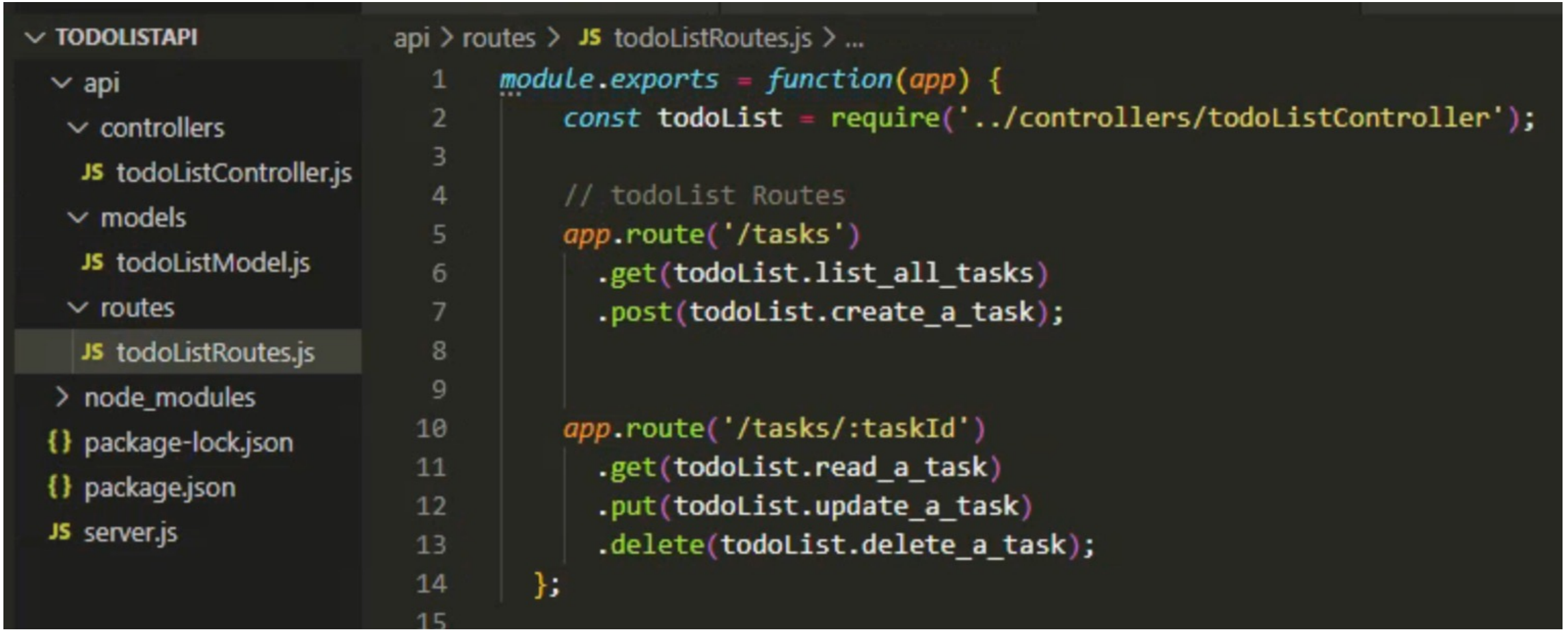
todoListModel.js – detail pt 3

```
20    });
21
22    module.exports = mongoose.model('Tasks', TaskSchema);
23
```

The last step (line 22) is to solidify this Schema as a data model with mongoose and export it from this file for use in other areas of our project.

todoListRoutes.js

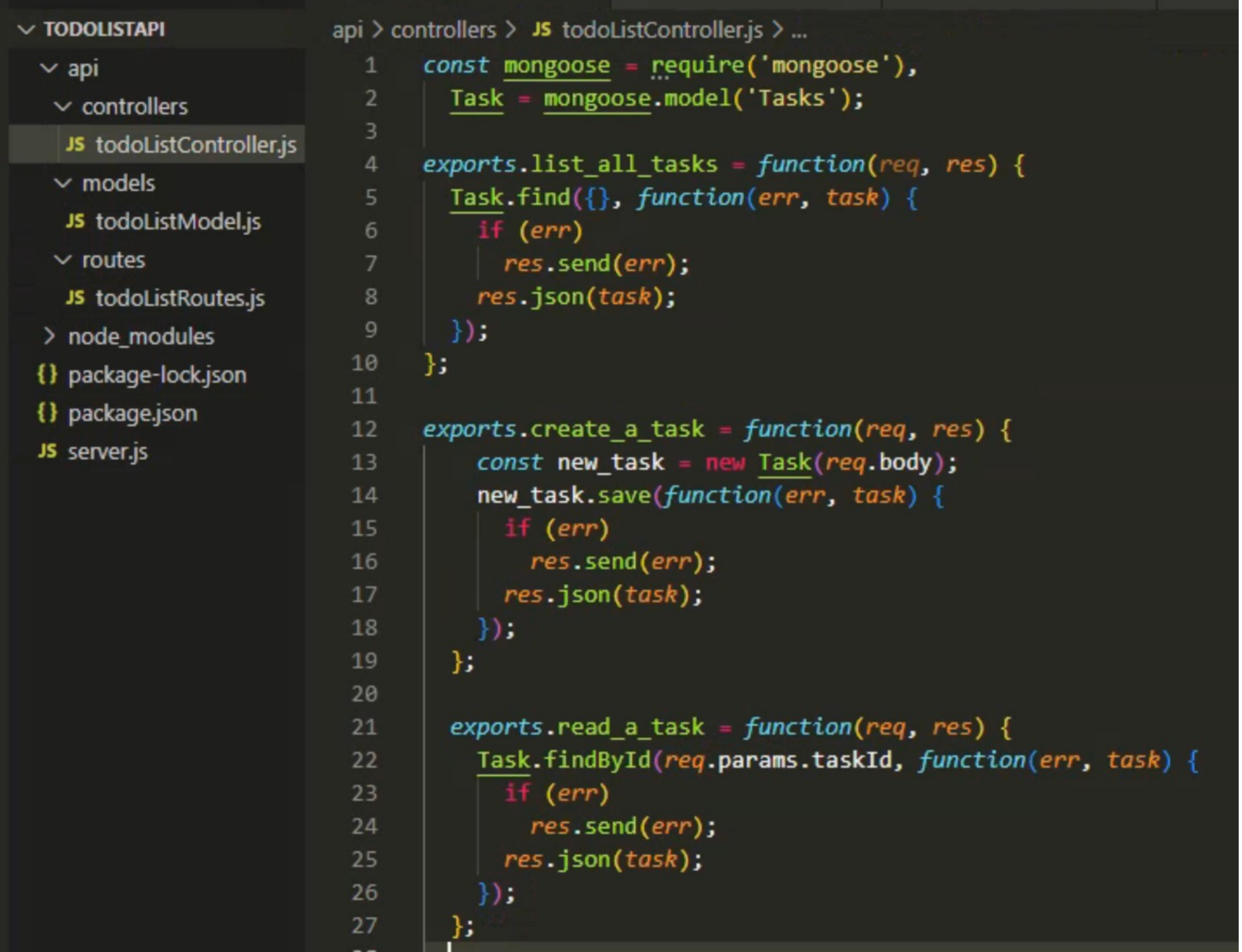
todoListRoutes.js – full code



Setting up routes

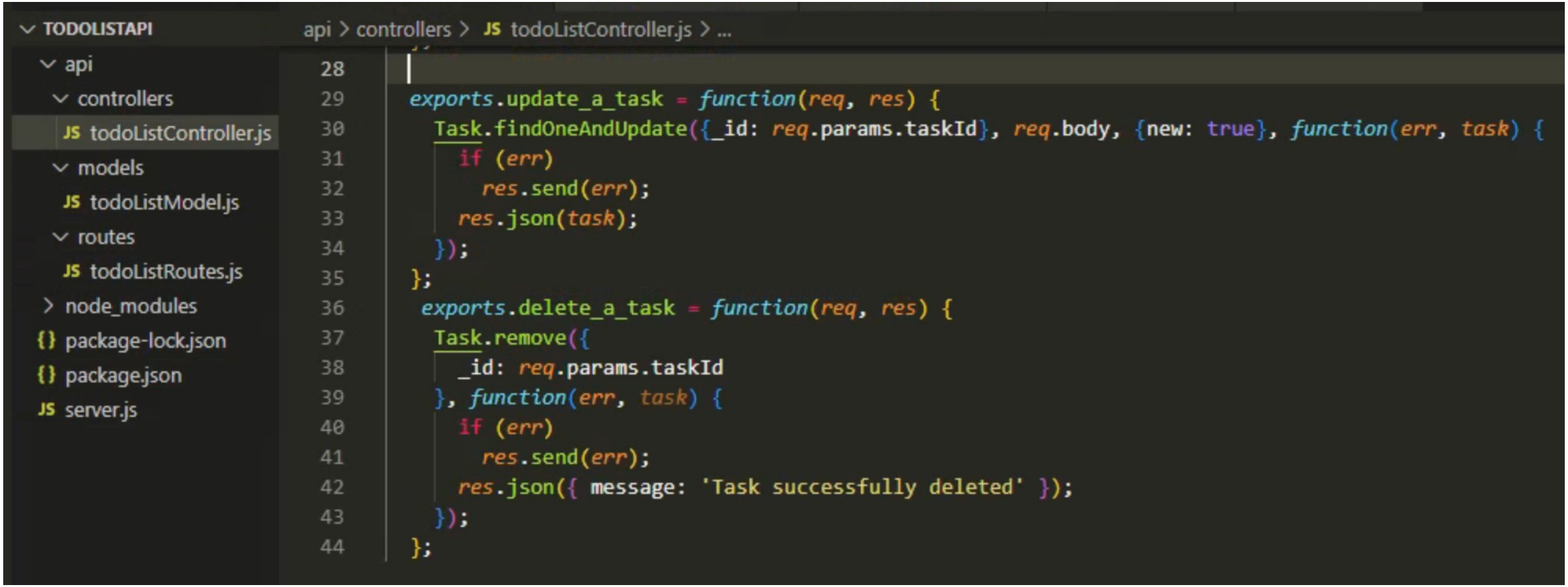
- Routing refers to determining how an application responds to a client request for a specific endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).
- Each of our routes has different route handler functions, which are executed when the route is matched.
- Above we have defined two basic routes(‘/tasks’, and ‘/tasks/taskId’) with different methods
- ‘/tasks’ has to methods(‘GET’ and ‘POST’), while ‘/tasks/taskId’ has GET, PUT and DELETE.
- Also we require the controller so each of the routes methods can call it’s respective handler function.

todoListController.js


TODOLISTAPI
api
controllers
todoListController.js
models
todoListModel.js
routes
todoListRoutes.js
node_modules
package-lock.json
package.json
server.js
api > controllers > todoListController.js > ...
1 const mongoose = require('mongoose'),
2 Task = mongoose.model('Tasks');
3
4 exports.list_all_tasks = function(req, res) {
5 Task.find({}, function(err, task) {
6 if (err)
7 res.send(err);
8 res.json(task);
9 });
10 };
11
12 exports.create_a_task = function(req, res) {
13 const new_task = new Task(req.body);
14 new_task.save(function(err, task) {
15 if (err)
16 res.send(err);
17 res.json(task);
18 });
19 };
20
21 exports.read_a_task = function(req, res) {
22 Task.findById(req.params.taskId, function(err, task) {
23 if (err)
24 res.send(err);
25 res.json(task);
26 });
27 };

Lines 1 -27

Lines 28- 44


TODOLISTAPI
api
controllers
todoListController.js
models
todoListModel.js
routes
todoListRoutes.js
node_modules
package-lock.json
package.json
server.js
api > controllers > todoListController.js > ...
28
29 exports.update_a_task = function(req, res) {
30 Task.findOneAndUpdate({_id: req.params.taskId}, req.body, {new: true}, function(err, task) {
31 if (err)
32 res.send(err);
33 res.json(task);
34 });
35 };
36 exports.delete_a_task = function(req, res) {
37 Task.remove({
38 _id: req.params.taskId
39 }, function(err, task) {
40 if (err)
41 res.send(err);
42 res.json({ message: 'Task successfully deleted' });
43 });
44 };

Setting up the controller

In this controller, we are writing five different functions namely:

- list_all_tasks
- create_a_task
- read_a_task
- update_a_task
- delete_a_task

We will export each of the functions for us to use in our routes.

Each of these functions uses different mongoose methods such as:

find, findById, findOneAndUpdate, save and remove.

Setting up the controller part 2

- Don’t worry too much about the detail of each function but you should be able to get an idea of what is going on each time.
- As you can see all the responses are in the json format
- At the top of the file we are requiring the mongoose library and the model ‘Tasks’ that we made in todoListModel.js

server.js

Wiring it all together

- Earlier on, we had a minimal code for our server to be up and running in the server.js file.
- In this section we will be connecting our handlers(controllers), database, the created models, body parser and the created routes together.
- Open the server.js file created a while ago and follow the following steps to put everything together.
- Essentially, you will be replacing the code in your server.js with the code snippet from this section

server.js – full code

JS server.js > ...

```
const express = require('express'),
  app = express(),
  port = process.env.PORT || 3000,
  mongoose = require('mongoose'),
  Task = require('./api/models/todoListModel'), //created model loading here
  bodyParser = require('body-parser');

// mongoose instance connection url connection
mongoose.Promise = global.Promise;
mongoose.connect('mongodb://pm76:pm76@mongo.cms.gre.ac.uk/pm76');

app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());

const routes = require('./api/routes/todoListRoutes'); //importing route
routes(app); //register the route

app.listen(port);

console.log('todo list RESTful API server started on: ' + port);
```

Important – line 10

If working with your University Mongo account use this line but change pm76 to your username in all 3 places !!

```
9   mongoose.Promise = global.Promise;
10  mongoose.connect('mongodb://pm76:pm76@mongo.cms.gre.ac.uk/pm76');
```

If working at home on your localhost use this connection string

```
9   mongoose.Promise = global.Promise;
10  mongoose.connect('mongodb://localhost/Tododb');
```

Body-parser

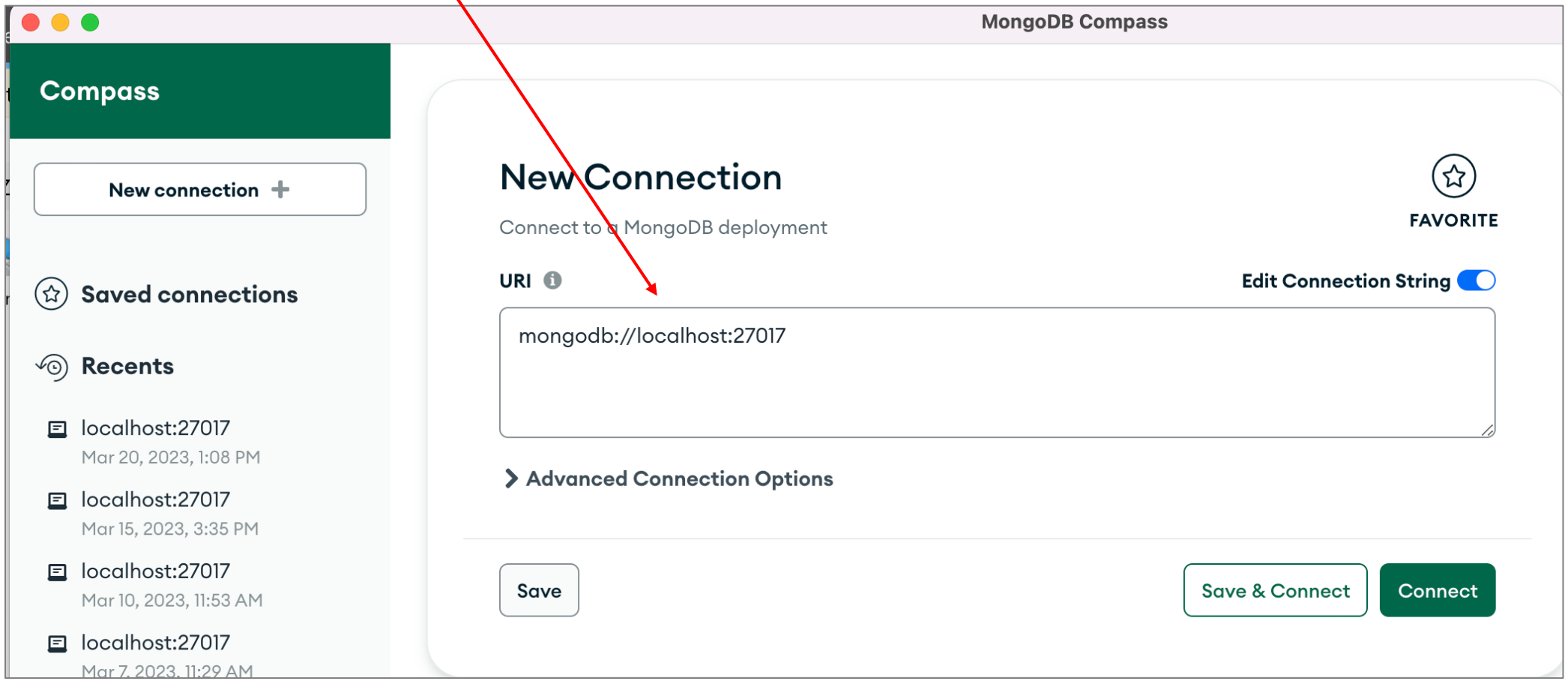
https://www.simplilearn.com/tutorials/nodejs-tutorial/body-parser-in-express-js

- Body Parser is a middleware of Node JS used to handle HTTP POST request.
- Body Parser can parse a string based client request body into JavaScript Object which we can use in our application.
- Specifically in the context of a POST or PUT HTTP requests where the information you want is contained in the body.
- Using body-parser allows you to access req.body from within routes and use that data.
- For example: To create a user in a database.

Testing with Postcode

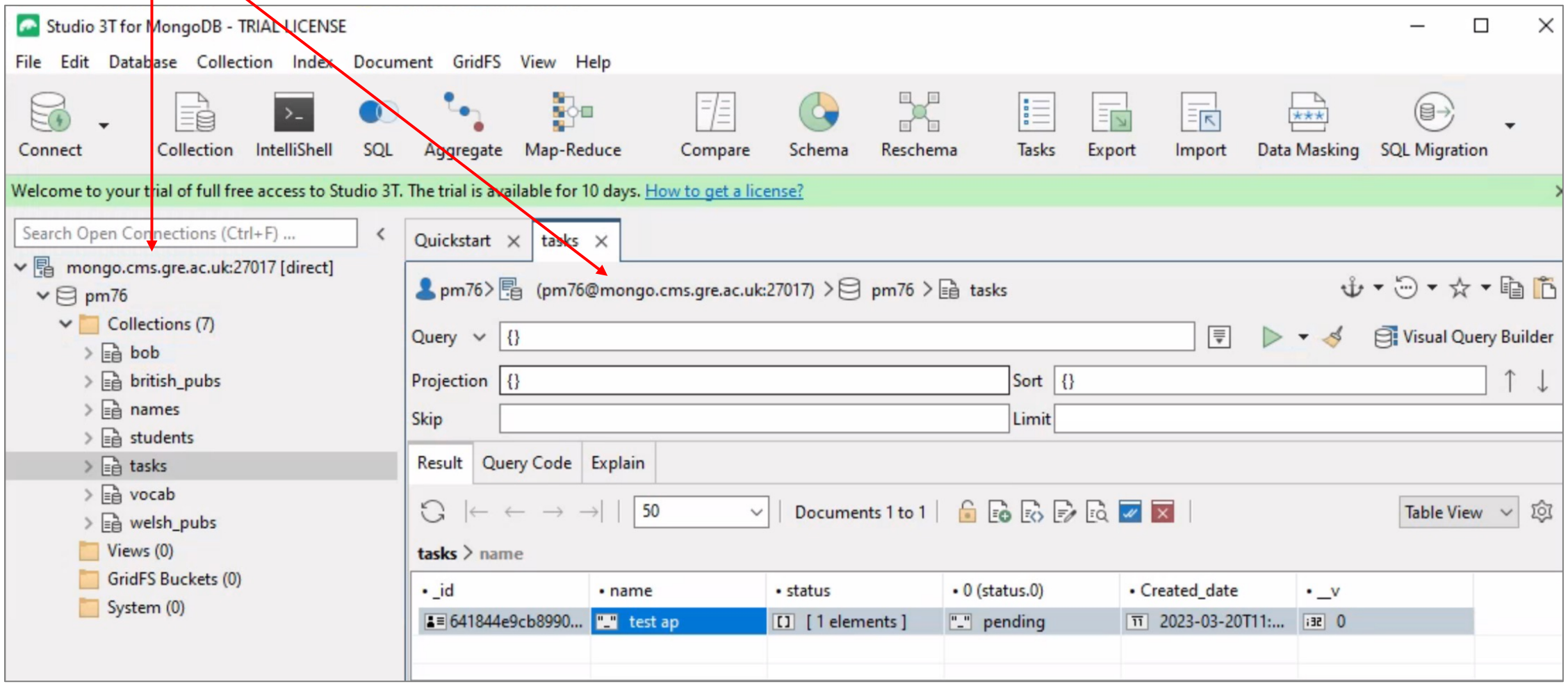
Make sure your Mongodb is connected

This is my localhost version on my Mac



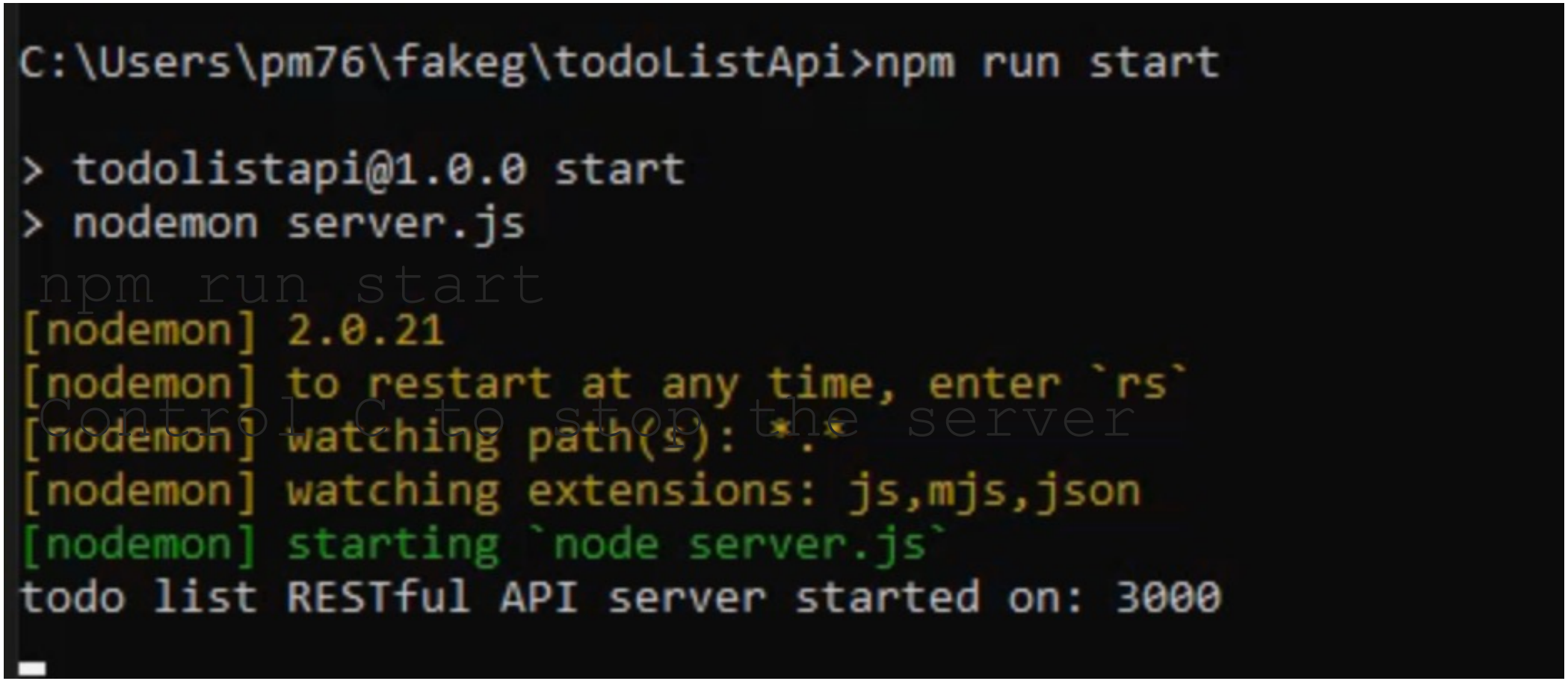
Make sure your Mongodb is connected

This is my University account via Virtual desktop



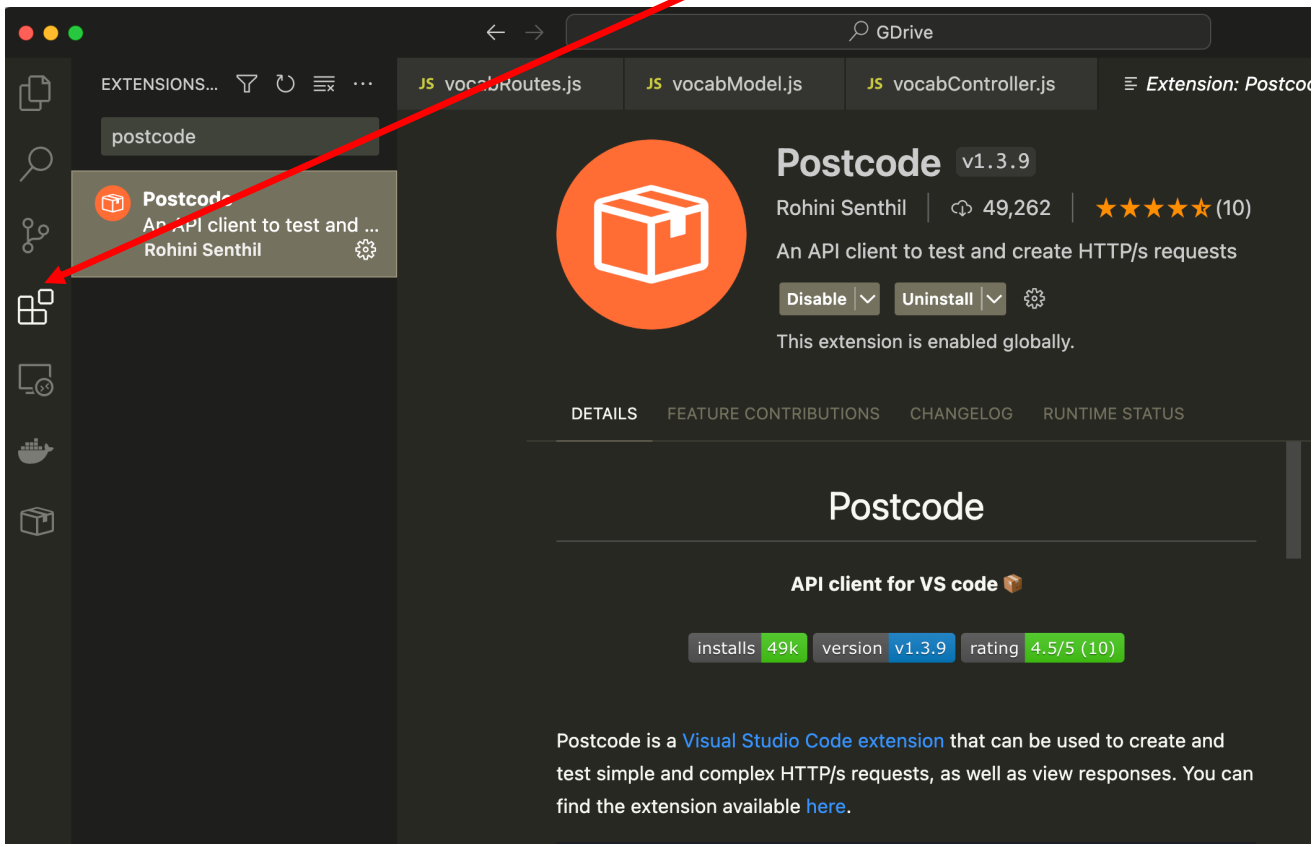
Make sure your API server is running on port 3000

```
npm run start
```

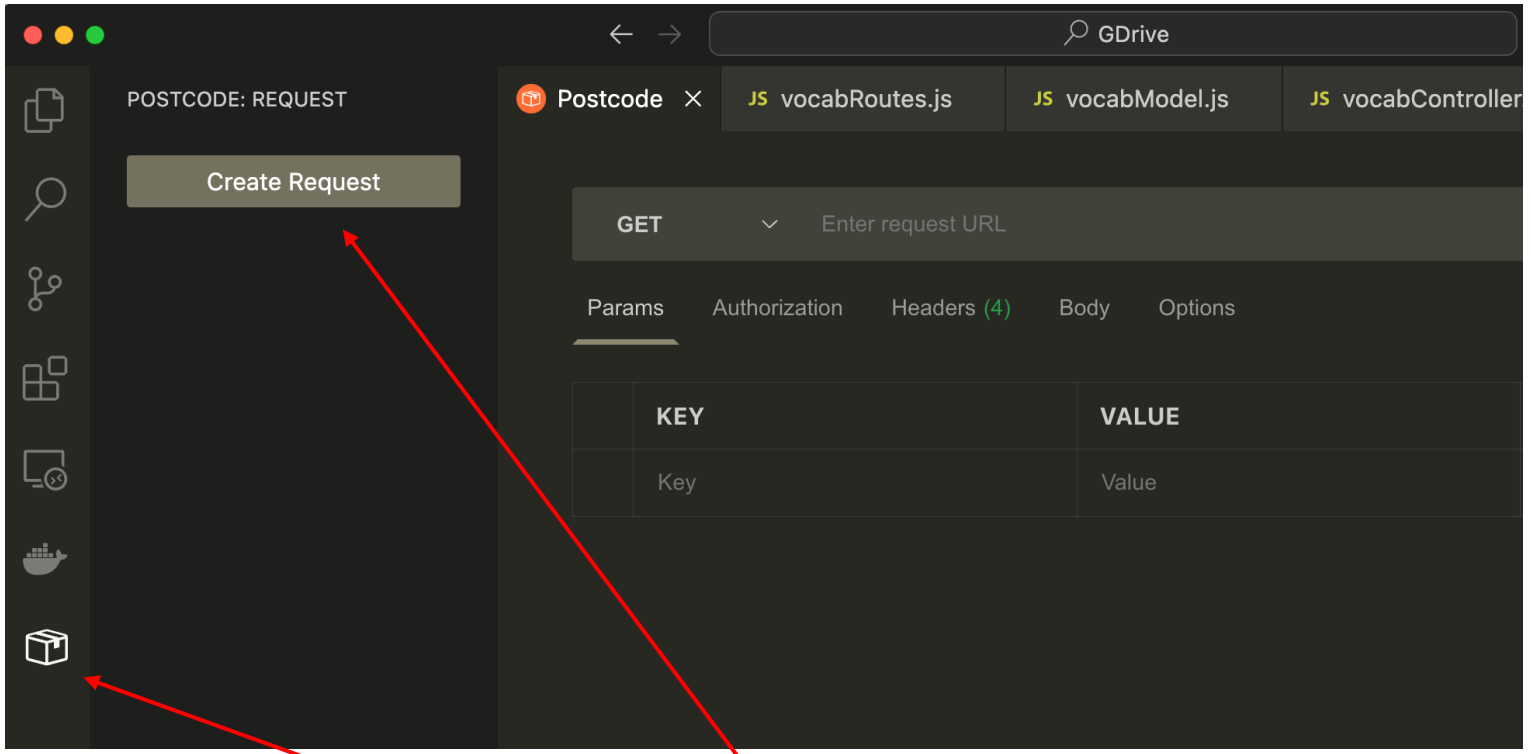


Install Postcode

Search for it in the VS code extension library

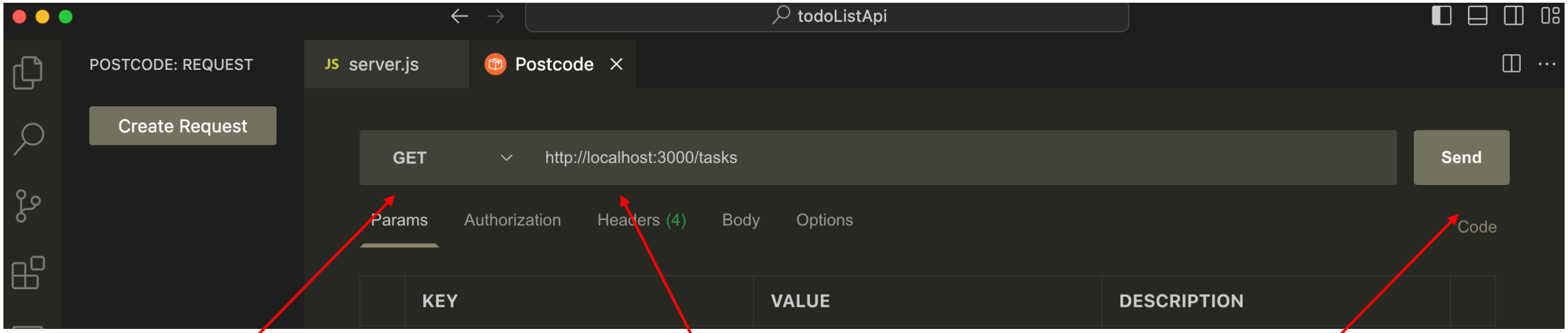


Postcode



Once installed you should see it’s icon on the left hand panel, click the icon and you will see the ‘Create Request’ button. Click this button

Test GET



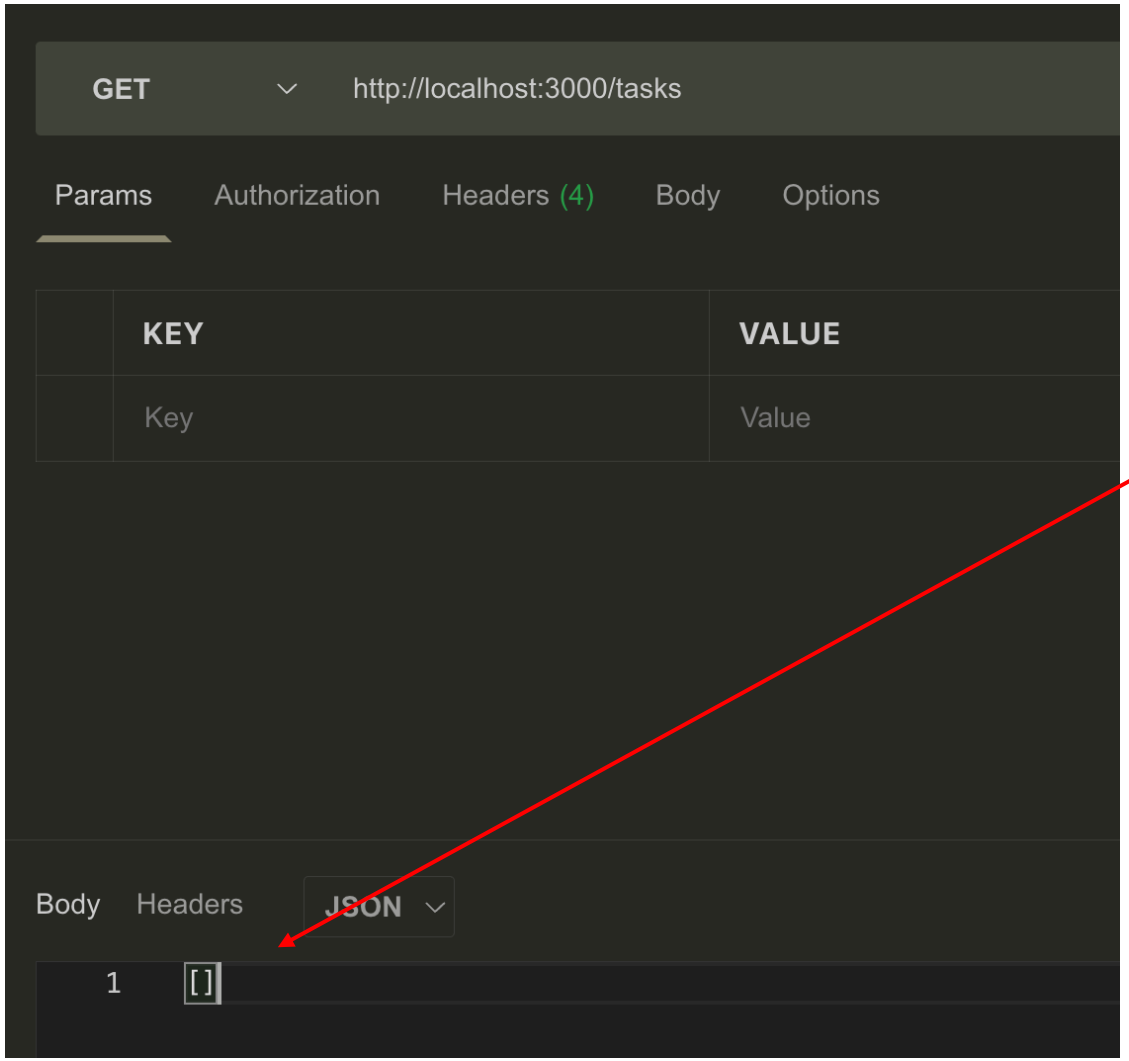
Select GET from the dropdown menu. Enter:

http://localhost:3000/tasks into the address bar and click Send.

This is targeting the route '/tasks' from todoListRoutes.js

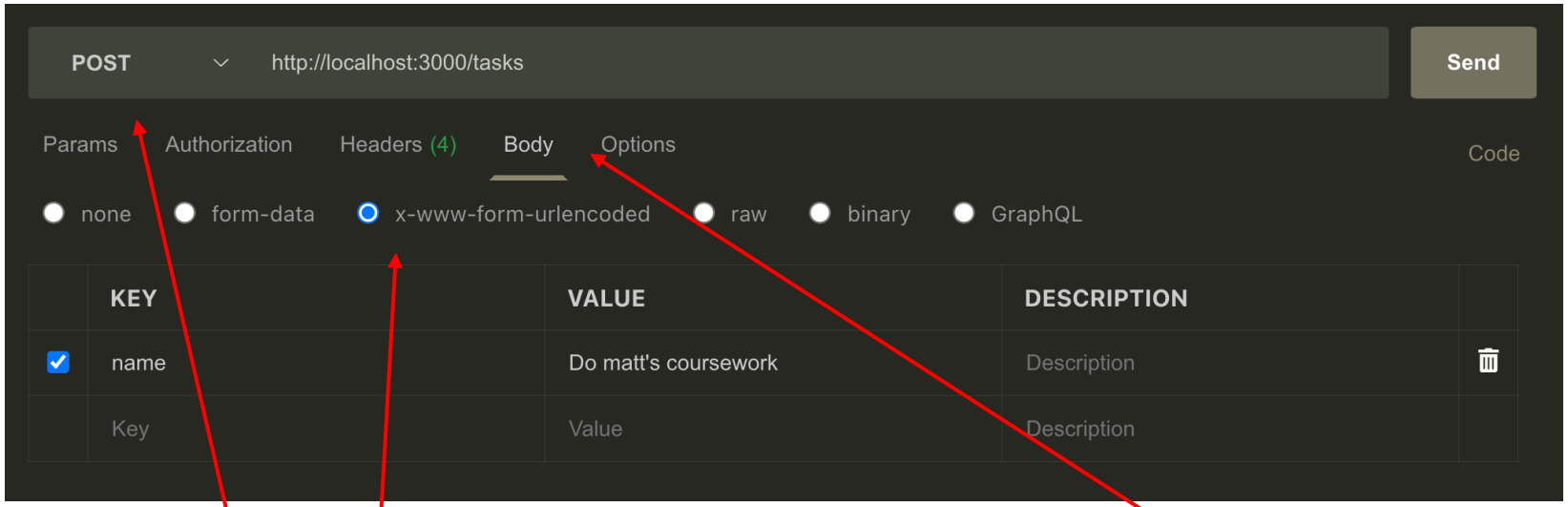
```
5   // todoList Routes
6   app.route('/tasks')
7     .get(todoList.list_all_tasks)
8     .post(todoList.create_a_task);
9
```

Test GET



The first time you run this test you will get [ ] because there is nothing in the database yet

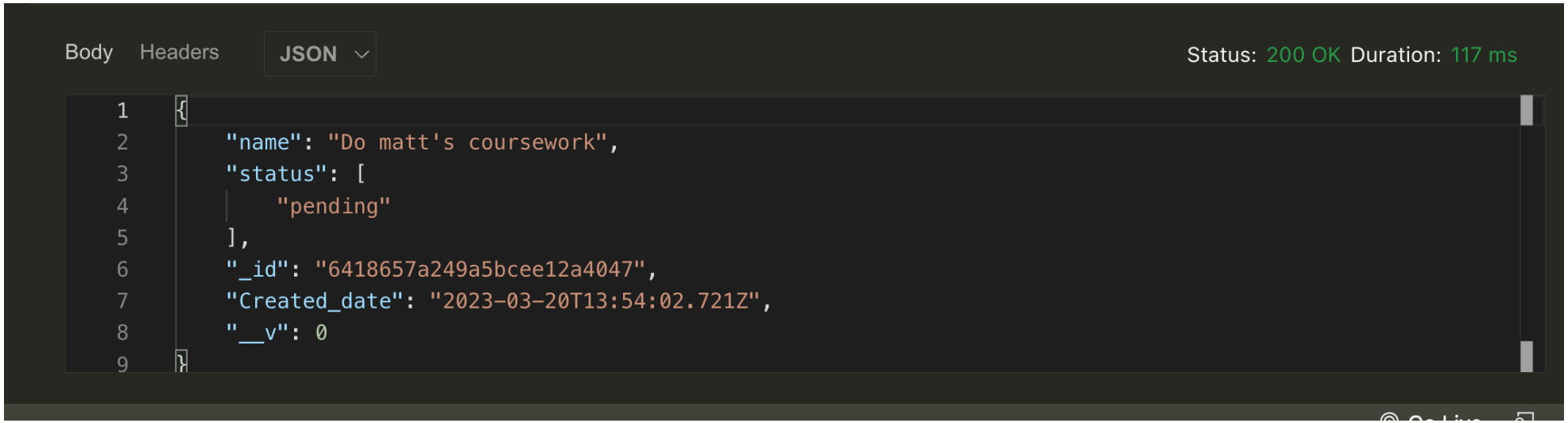
Test POST



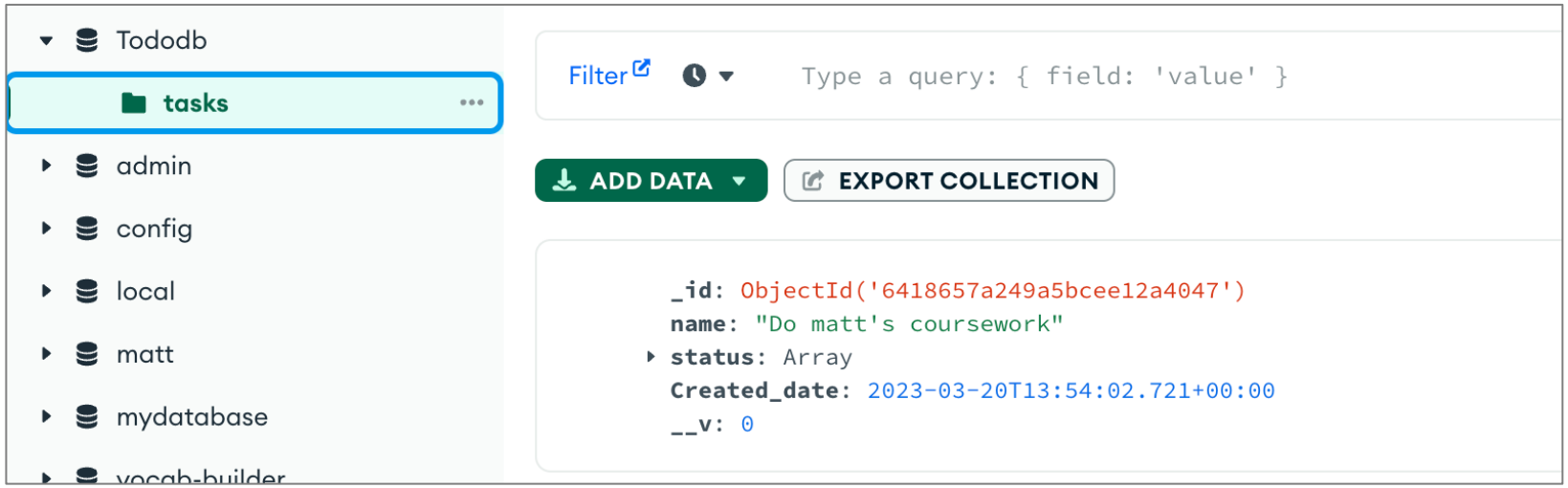
Select POST from the dropdown menu. Select the body tab then check the ‘x-www-form-urlencoded’ radio button.

Under KEY enter ’name’ which is we defined in our Schema in todoListModel.js

Then under VALUE enter some data for your todo list. Click Send

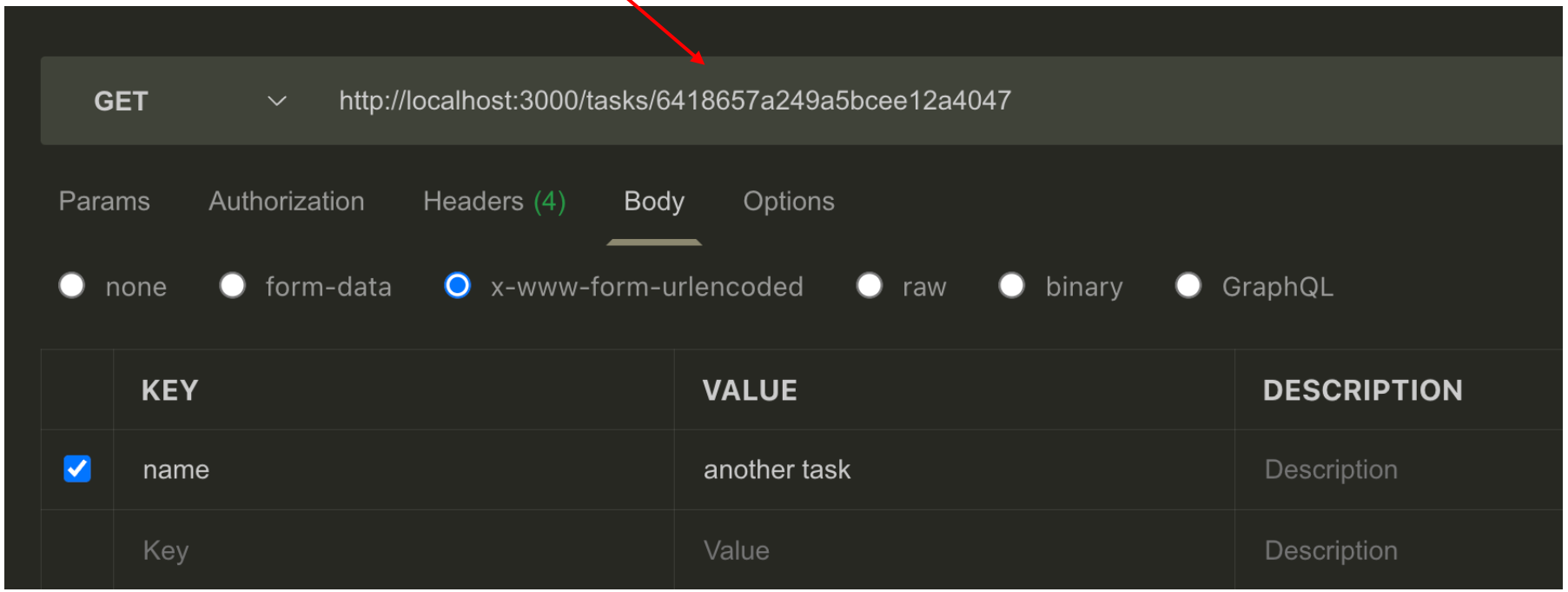


You should now see your data in the output area in Postcode and you can check your database to confirm the collection and document exist.



Further tests

You can test your other routes with PUT and DELETE and GET and individual task etc… by appending an id that you copy from the database and add to the URL



Final lab task

Using the slides (23 onwards) get the basic RESTful API working

Sources

https://aws.amazon.com/what-is/api

https://code.tutsplus.com/tutorials/a-beginners-guide-to-http-and-rest--net-16340

https://medium.com/extend/what-is-rest-a-simple-explanation-for-beginners-part-1-introduction-b4a072f8740f

https://www.freecodecamp.org/news/what-is-rest-rest-api-definition-for-beginners

https://www.codementor.io/@olatundegaruba/nodejs-restful-apis-in-10-minutes-q0sgsfhbd

Enough for today.