INFORMATION SYSTEMS 3D
INSY7314
LAB GUIDE 2025
(First Edition: 2021)

The Independent Institute of Education (Pty) Ltd is registered with the Department of Higher Education and Training as a private higher education institution under the Higher Education Act, 1997 (reg. no. 2007/HE07/002). Company registration number: 1987/004754/07.

Table of Contents

1 Introduction

Application Development Security (APDS) introduces the web developer to security concepts and how to incorporate them into development projects. The most efficient way to achieve this is by practical implementation, hence this guide. The concepts are generic and can be applied across any development stack; however, it is essential to understand that application security is not static and can be considered a “moving target”. New threats are constantly surfacing, and constant research is required to keep current. Most importantly, security should be built into the code from inception and not considered an afterthought.

In this document, you will be guided on how to build a basic MERN (Mongo Express React Node) application with an emphasis on security.

The guide assumes you are familiar with the following technologies.

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re- introduction_to_JavaScript [Accessed 10 July 2024].
- https://developer.mozilla.org/docs/Learn/HTML/Introduction_to_HTML [Accessed 10 July 2024.]
- https://developer.mozilla.org/docs/Learn/CSS/First_steps [Accessed 10 July 2024].

Other recommended reading for the course.

- https://developer.mozilla.org/en-US/docs/Web/Security [Accessed 10 July 2024].
- https://react.dev/learn [Accessed 10 July 2024].

It is also assumed that you have the following development environment setup.

- Visual Studio Code https://code.visualstudio.com [Accessed 10 July 2024].
- Node https://nodejs.org (LTS version is fine) [Accessed 10 July 2024].

We will build the application in two parts – a backend and a frontend.

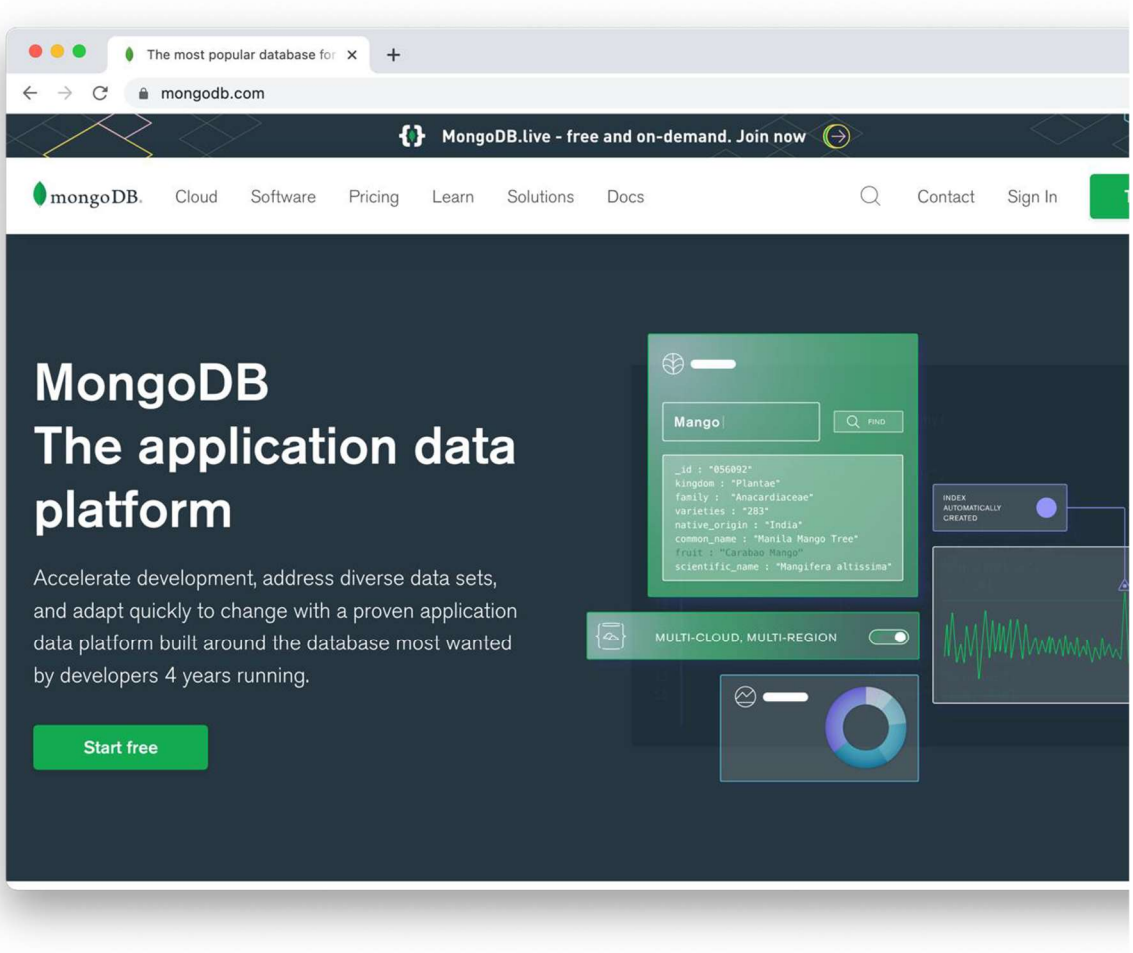
2 Backend

2.1 MongoDB

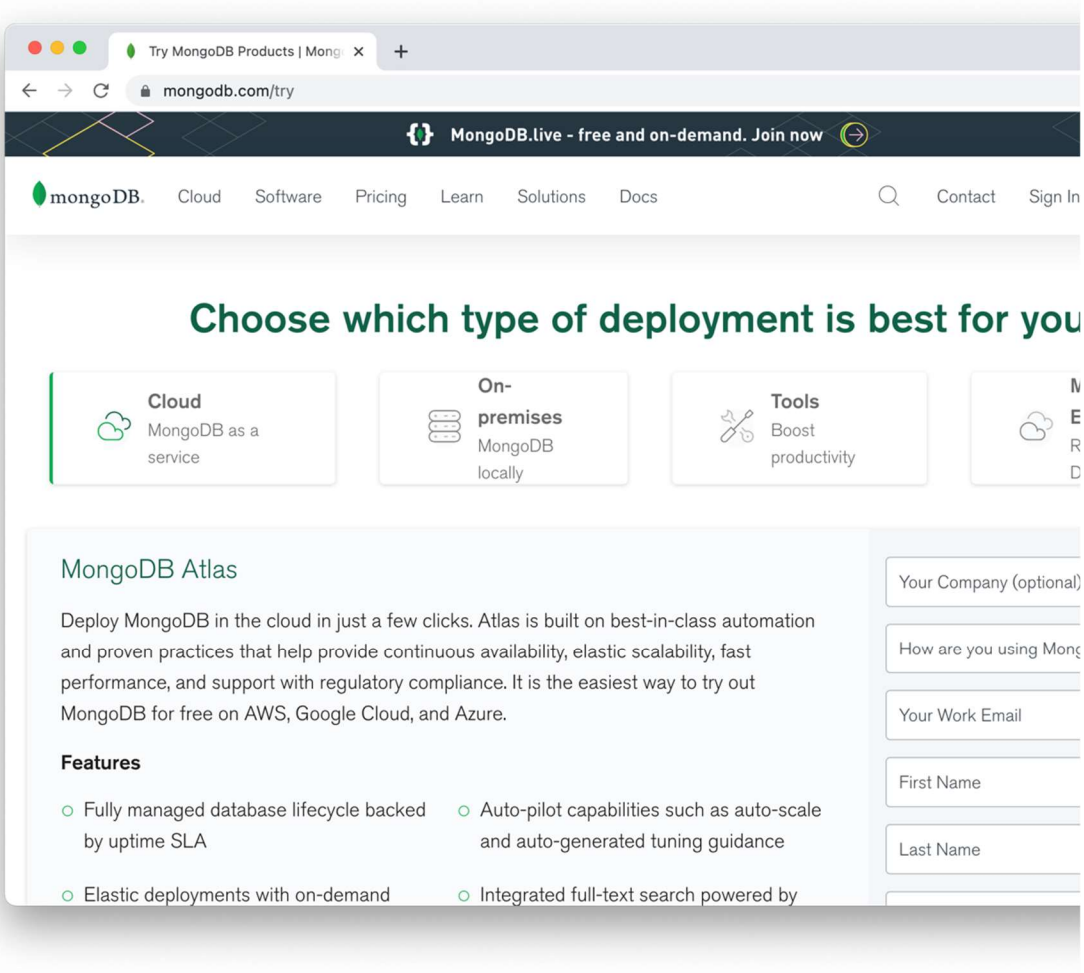
We start by setting up the cloud NOSQL database https://www.mongodb.com [Accessed 10 July 2024].

Remember, websites change, so the screenshots/steps may be different from these. Think in terms of outcomes.

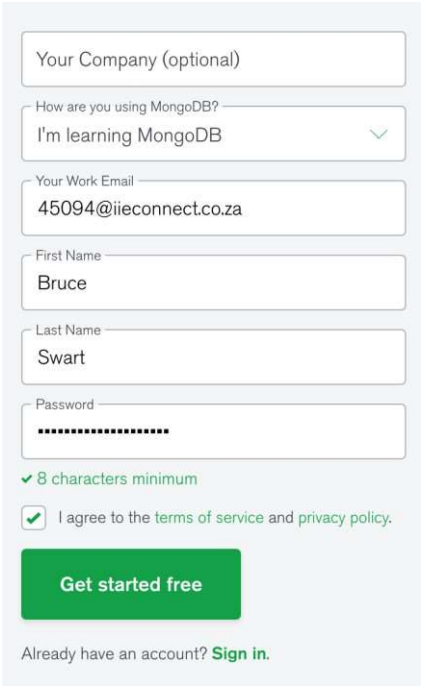
- Click the green “Try Free” button at the top right.



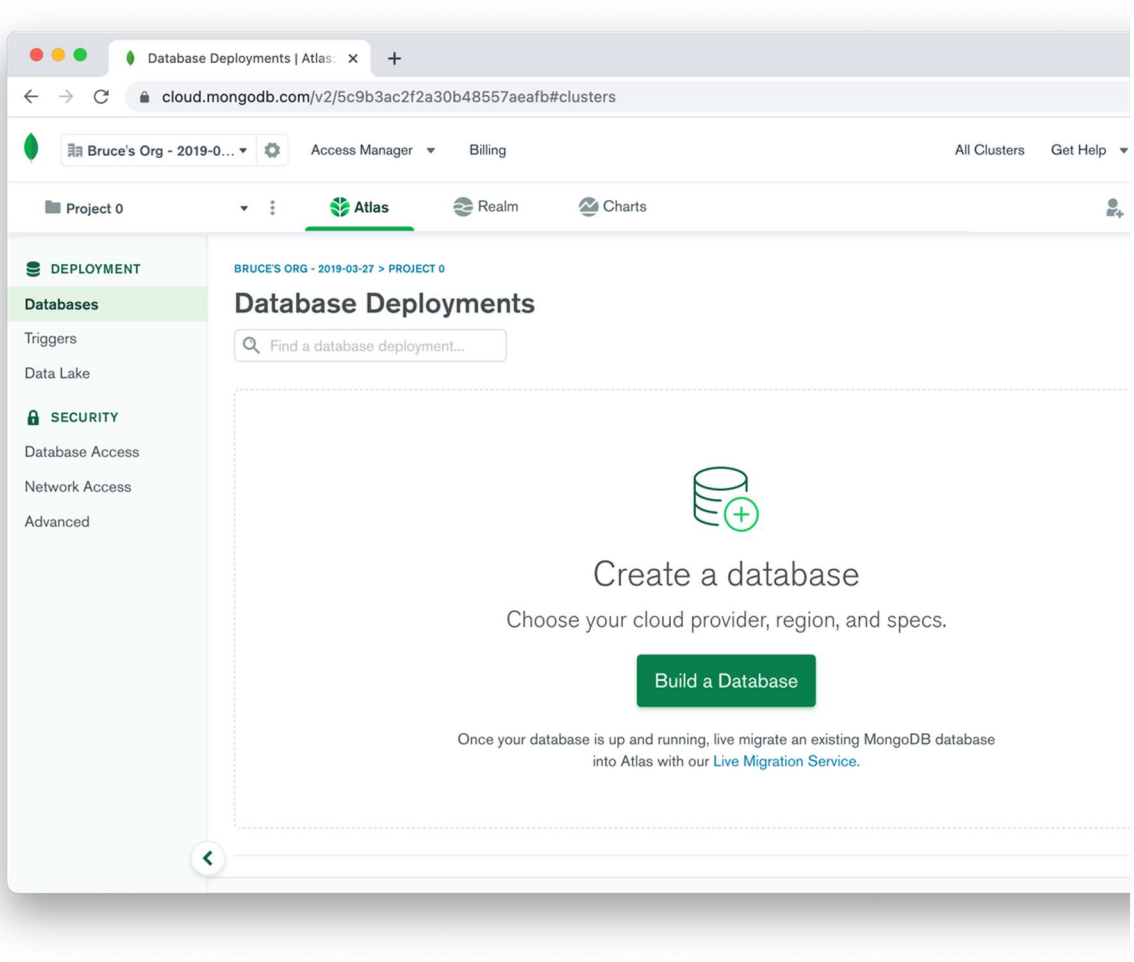
- Click on “Cloud” and fill out your details on the right-hand side – use your student email address.



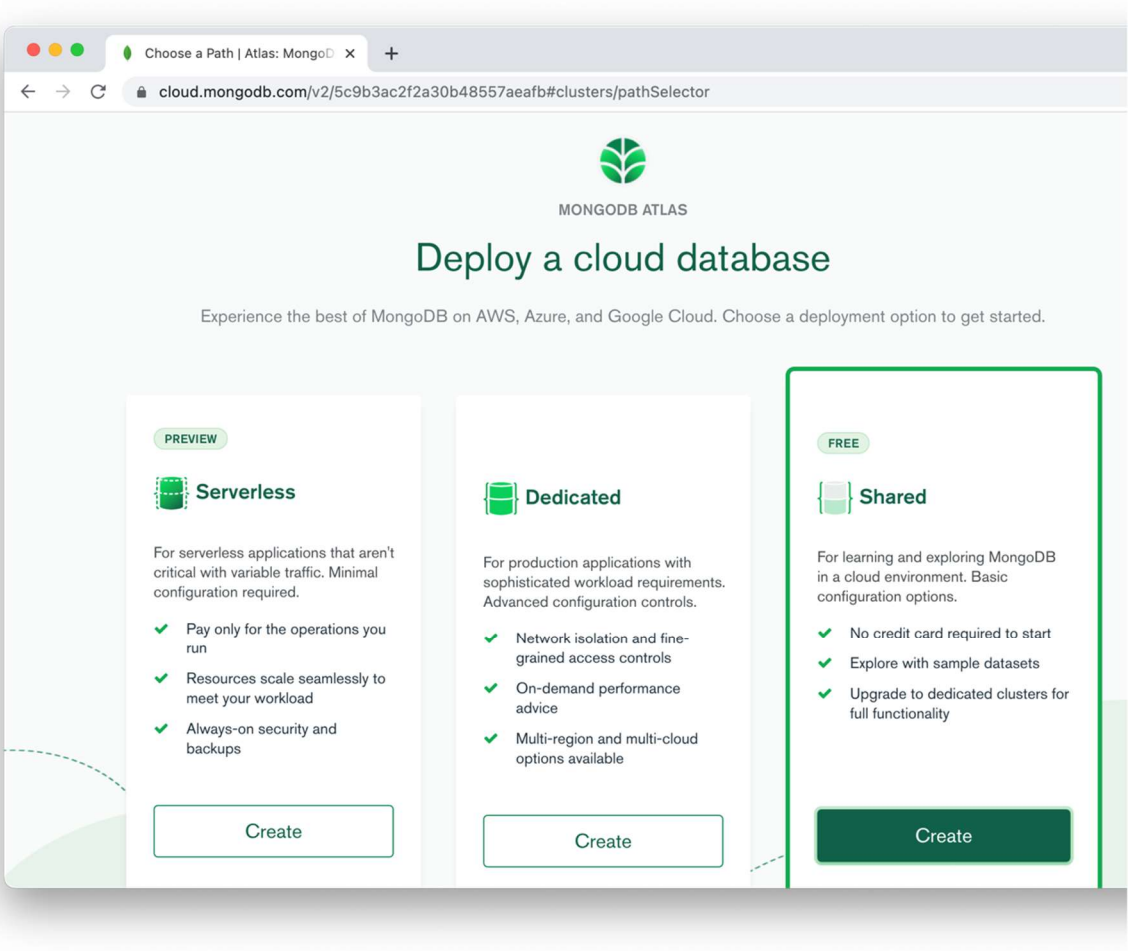
- Click on the “Get started free” button.



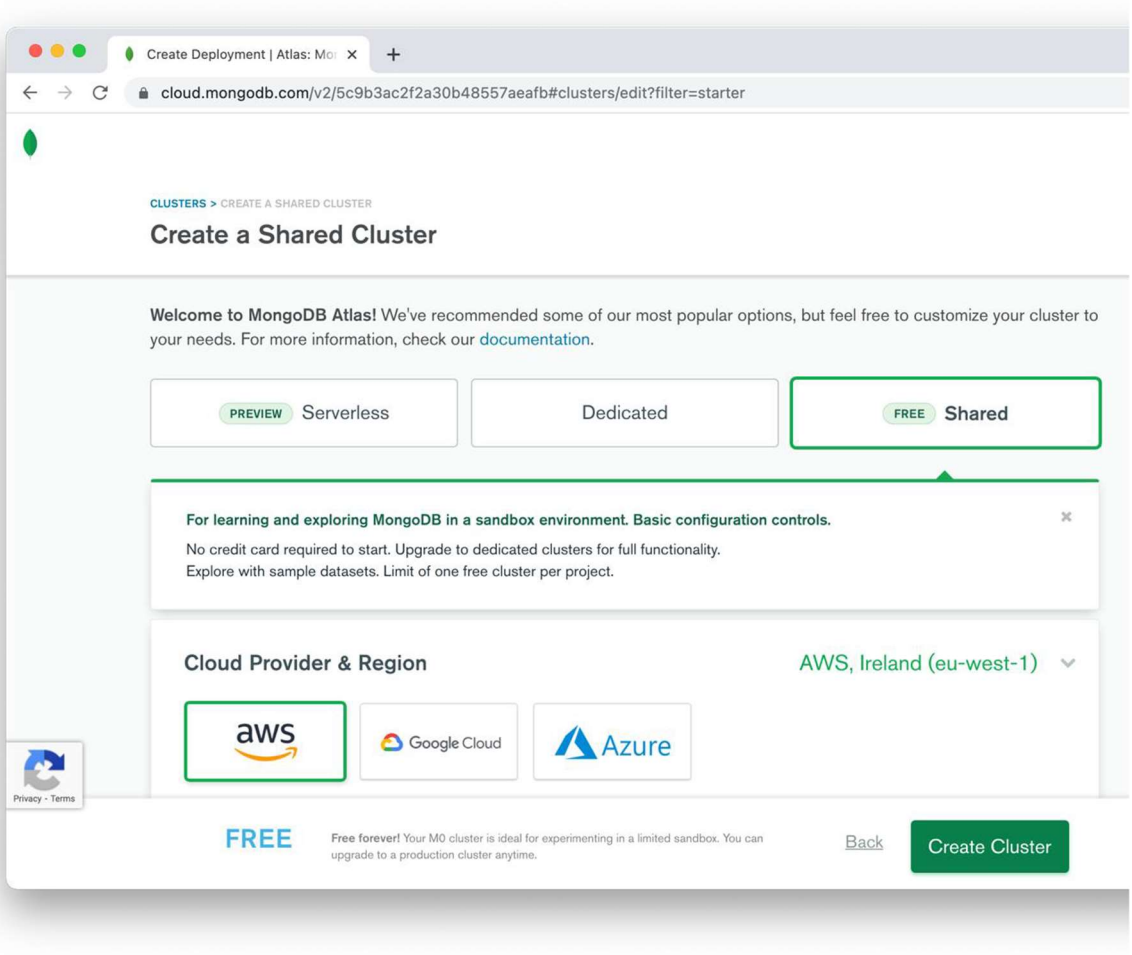
- Click on “Build a database”.



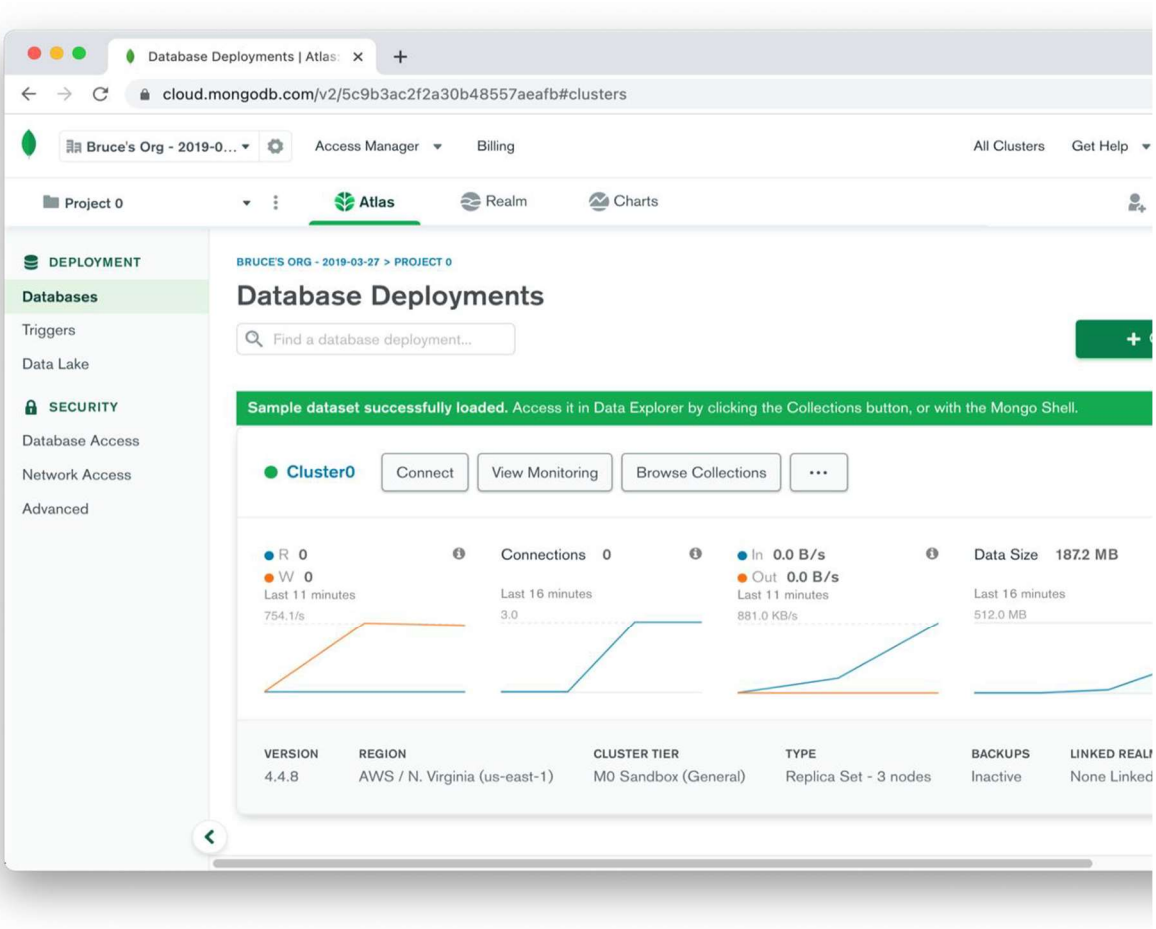
- Select FREE/Shared and click the “Create” button.



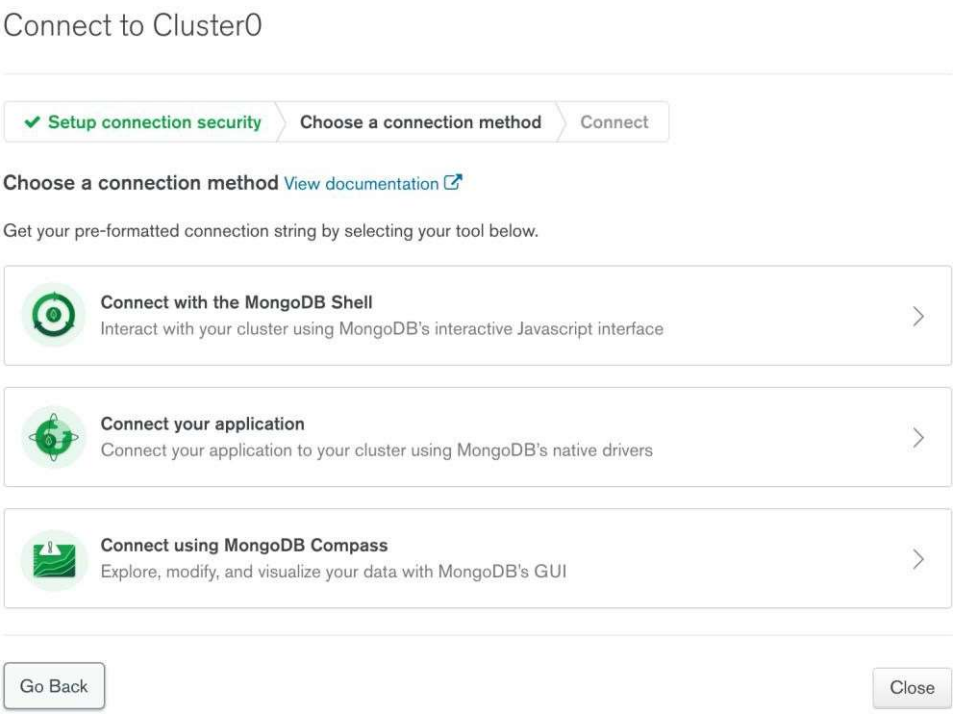
- Click the “Create Cluster” button (all defaults are fine but make sure you are on Shared as illustrated).



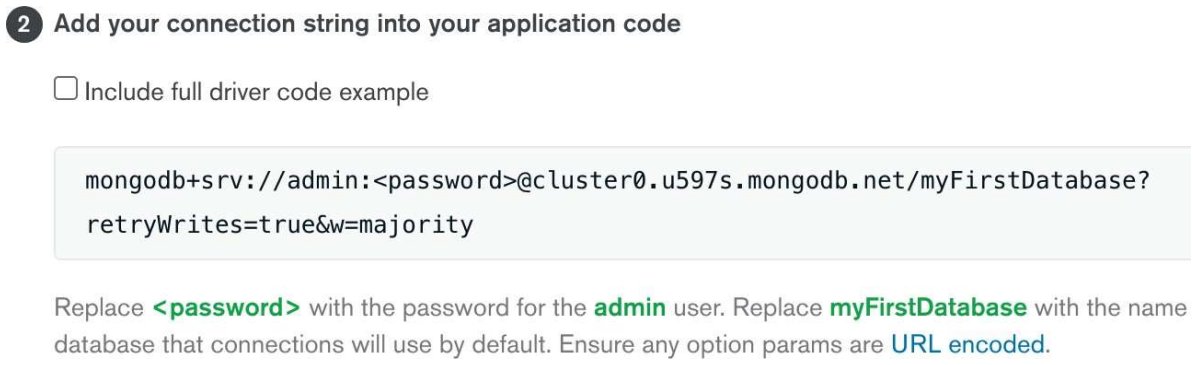
- Now we need to get our connection string. Click on the “Connect” button



- Choose the “Connect your application” option



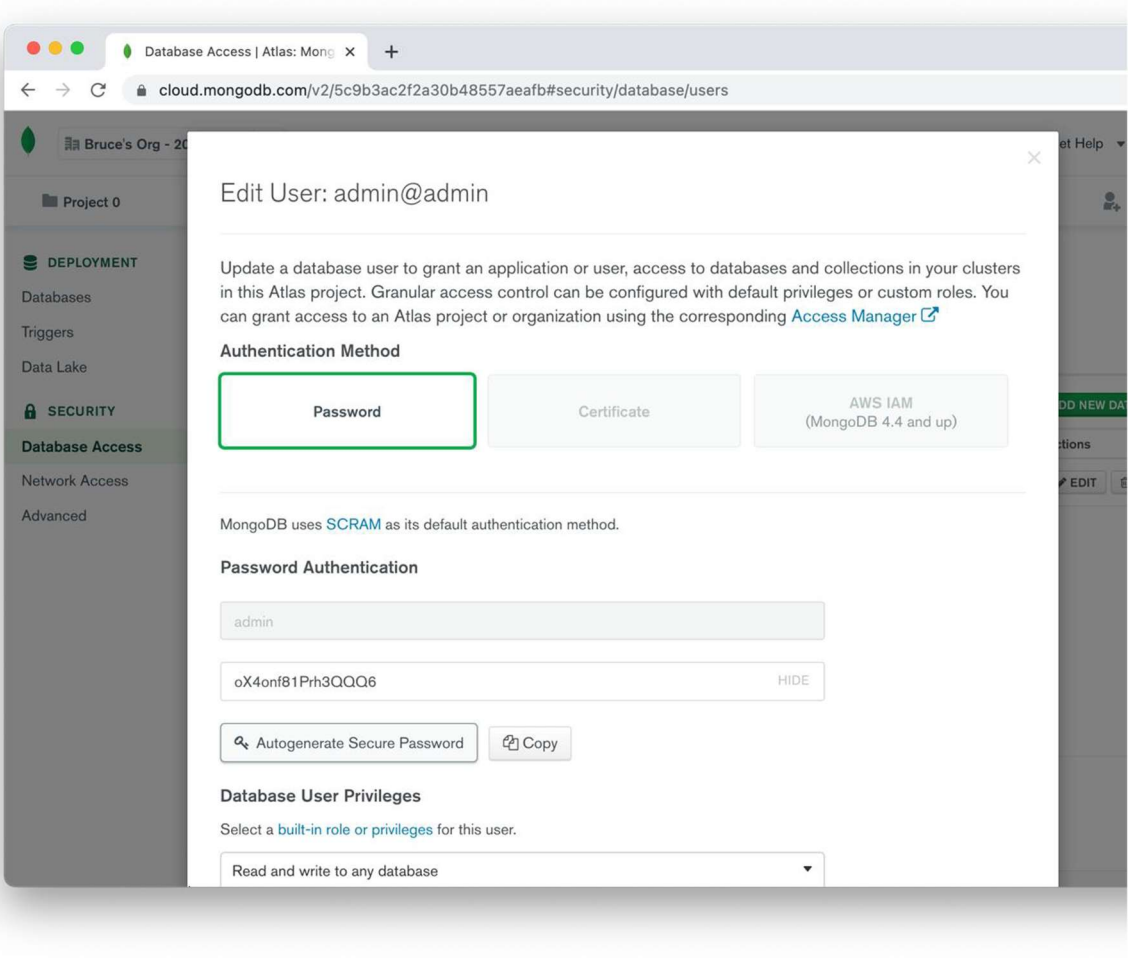
- Copy the connection string somewhere safe.



Lastly – generate a password for your database admin user.

- Using the menu on the left-hand side, choose “Database Access” under “Security” – click the “Autogenerate Secure Password” button and keep the generated password somewhere safe.

- Click on the “Update User” button at the bottom once done.



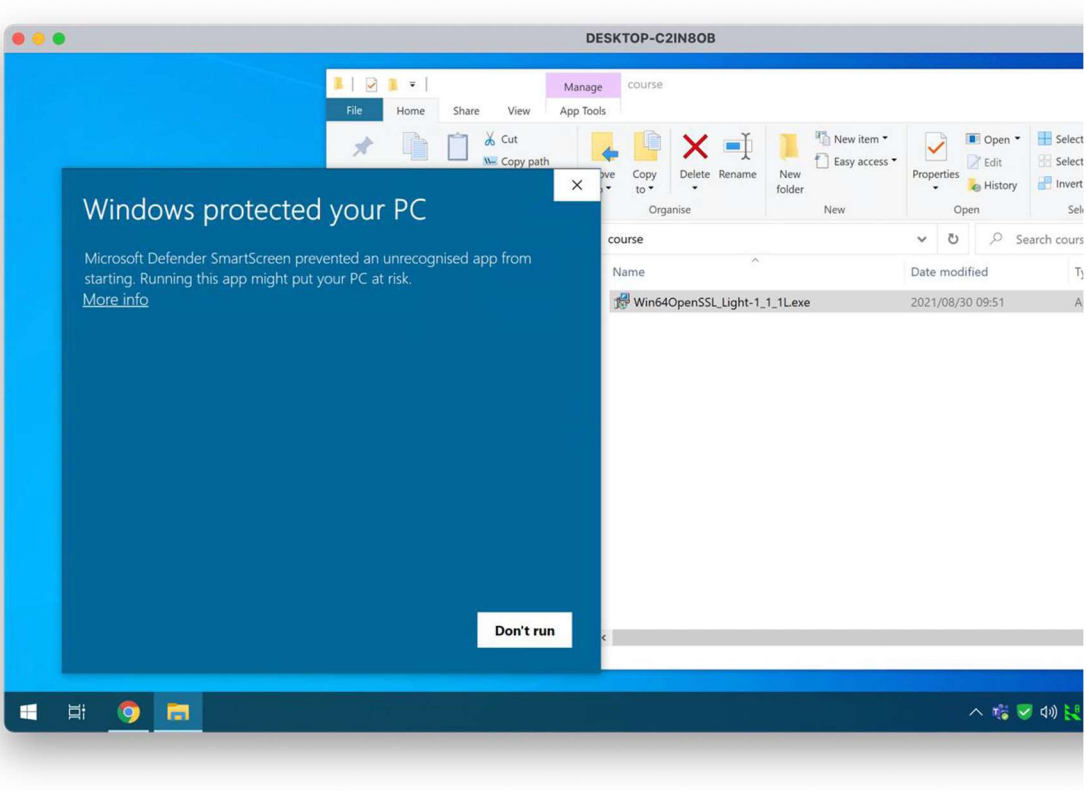
2.2 OpenSSL

Next, we generate security certificates by downloading software from https://slproweb.com/products/Win32OpenSSL.html [Accessed 10 July 2024].

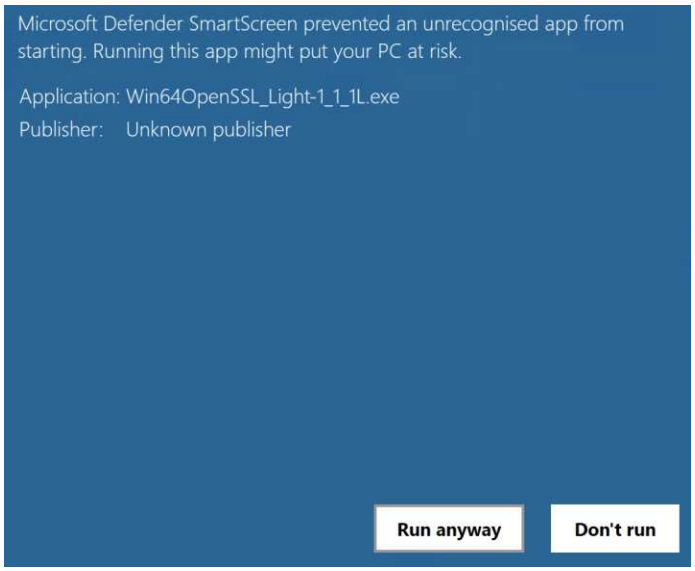
- Choose the Win64 Light version (the first one in this screenshot). Remember – screen layout and/or version number may be different.

File	Type	Description
Win64 OpenSSL v1.1.1L Light EXE | MSI	3MB Installer	Installs the most commonly used essentials of Win64 OpenSSL v1.1.1L (Recommended for use OpenSSL). Only installs on 64-bit versions of Windows. Note that this is a default build of Ope local and state laws. More information can be found in the legal agreement of the installation.
Win64 OpenSSL v1.1.1L EXE | MSI	63MB Installer	Installs Win64 OpenSSL v1.1.1L (Recommended for software developers by the creators of O on 64-bit versions of Windows. Note that this is a default build of OpenSSL and is subject to lo More information can be found in the legal agreement of the installation.
Win32 OpenSSL v1.1.1L Light EXE | MSI	3MB Installer	Installs the most commonly used essentials of Win32 OpenSSL v1.1.1L (Only install this if you for Windows. Note that this is a default build of OpenSSL and is subject to local and state laws can be found in the legal agreement of the installation.
Win32 OpenSSL v1.1.1L EXE | MSI	54MB Installer	Installs Win32 OpenSSL v1.1.1L (Only install this if you need 32-bit OpenSSL for Windows. No build of OpenSSL and is subject to local and state laws. More information can be found in the l installation.

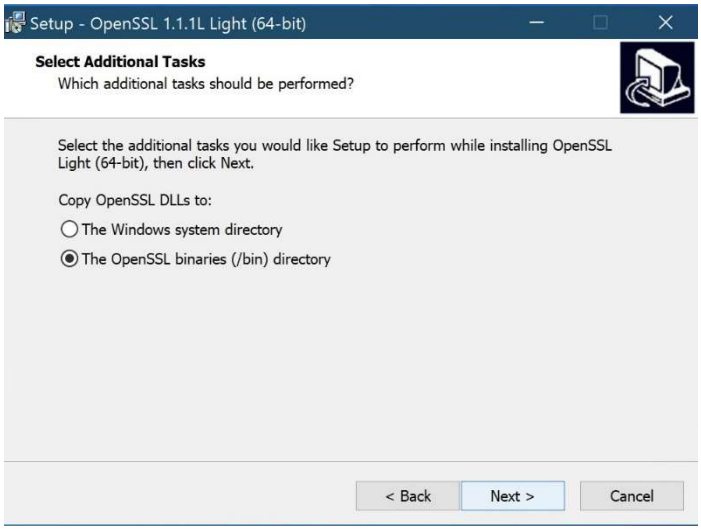
- When installing, Windows 10 or Windows 11 may give you some issues. Click on the "More info" link.



- Click the “Run anyway” button – it is safe!



- Accept most defaults when installing, except for install location – make this somewhere that you can normally access – especially on the campus PC’s. Also, when asked about the DLL location choose the option as below.

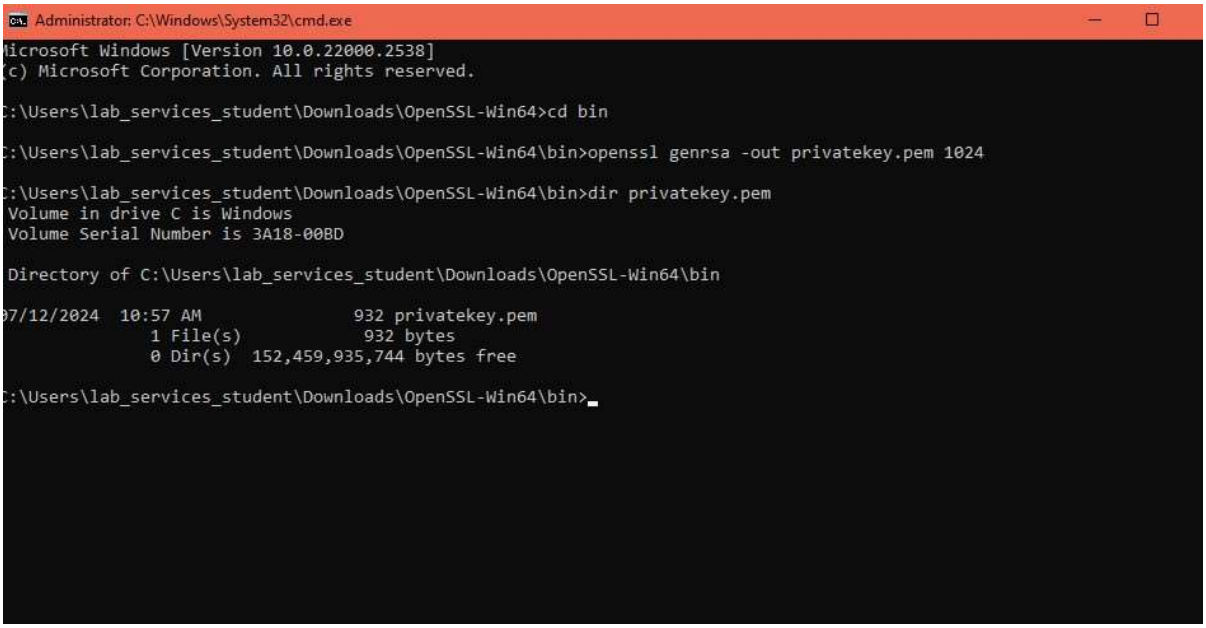


- Startup Visual Studio Code and open a terminal window. Terminal->New Terminal

This is effectively the Windows command line prompt. If this and its commands are new to you – do some research. Windows currently has two command shells: traditional command shell, and PowerShell.

- Navigate to wherever you installed OpenSSL and change to the “bin” folder as shown.

- First generate the private key
  openssl genrsa -out privatekey.pem 1024



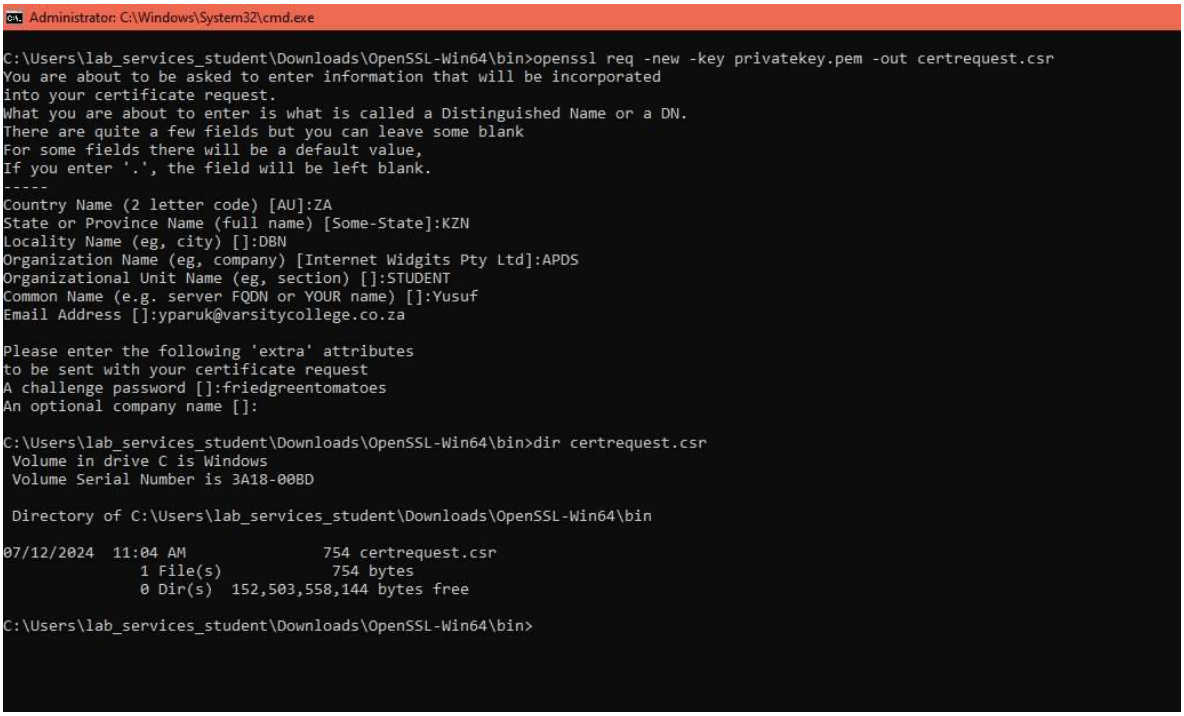
- Before the next step you may have to create an “openssl.cfg” file in the “bin” folder. Do this using your favorite text editor such as Notepad (or Visual Studio Code of course!). The contents of the file come from here https://www.openssl.org/docs/man1.1.1/man1/req.html [Accessed 10 July 2024].

- Scroll down until you find the text as below and paste everything in the black block into your “openssl.cfg” file and save.

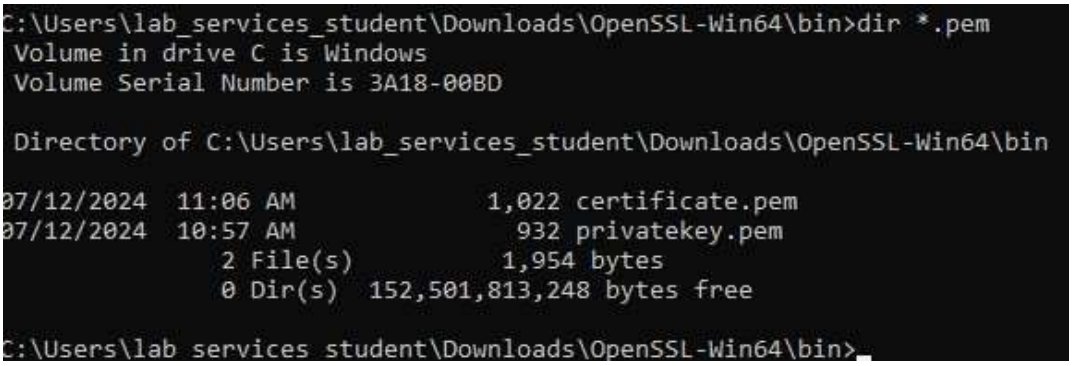
Sample configuration file prompting for field values:

```
[ req ]
default_bits            = 2048
default_keyfile         = privkey.pem
distinguished_name      = req_distinguished_name
attributes              = req_attributes
req_extensions          = v3_ca

dirstring_type = nobmp

[ req_distinguished_name ]
countryName                     = Country Name (2 letter code)
countryName_default             = AU
countryName_min                 = 2
countryName_max                 = 2

localityName                    = Locality Name (eg, city)

organizationalUnitName          = Organizational Unit Name (eg, section)

commonName                      = Common Name (eg, YOUR name)
commonName_max                  = 64

emailAddress                    = Email Address
emailAddress_max                = 40

[ req_attributes ]
challengePassword               = A challenge password
challengePassword_min           = 4
challengePassword_max           = 20

[ v3_ca ]

subjectKeyIdentifier=hash
authorityKeyIdentifier=keyid:always,issuer:always
basicConstraints = critical, CA:true
```

- Now you can run the next command – this creates the certificate signing request.
  openssl req -new -key privatekey.pem -out certrequest.csr
-



- Now the final command – signing the certificate using your private key
  openssl x509 -req -in certrequest.csr -signkey privatekey.pem -out certificate.pem

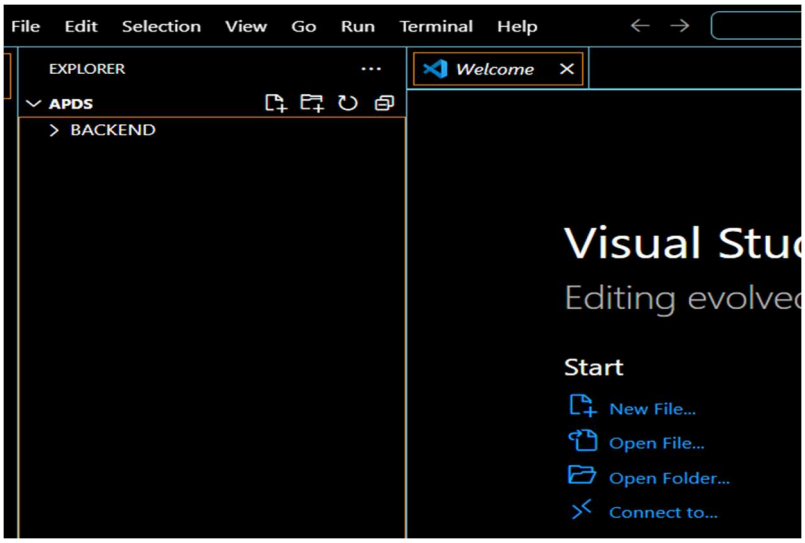


- Keep these two files certificate.pem and privatekey.pem somewhere safe.

2.3 Backend API

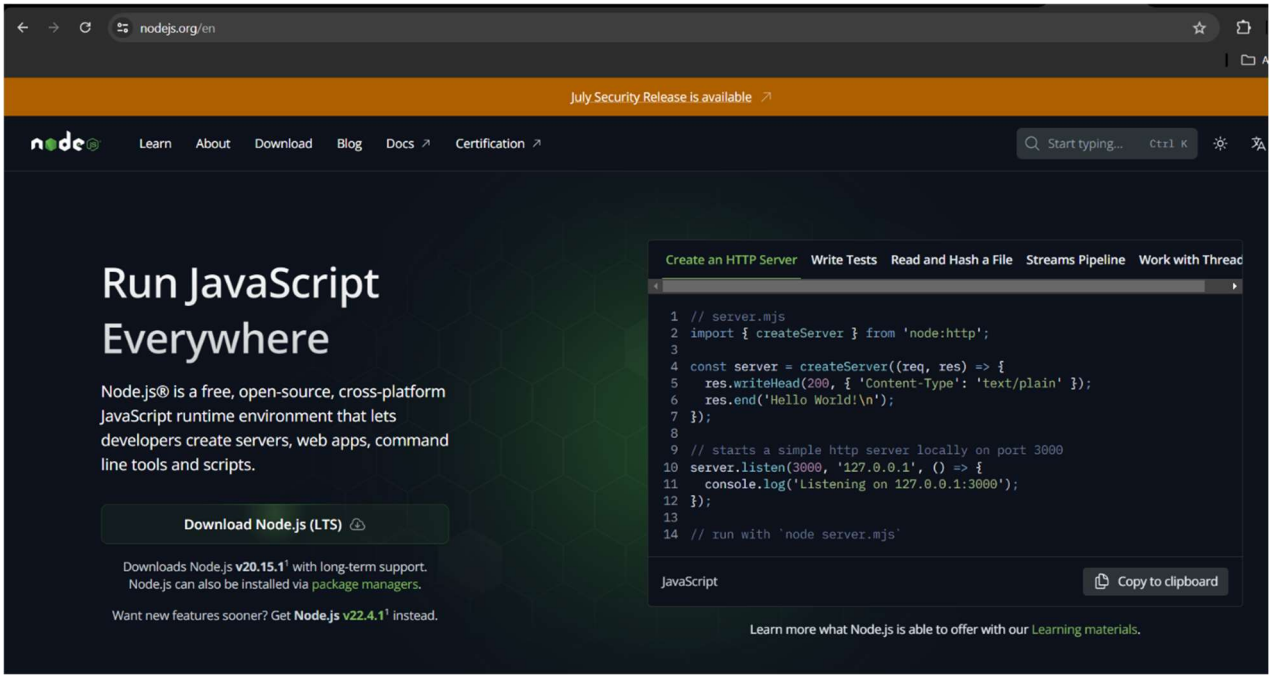
Next, we will start building out the standalone backend API that interacts with the MongoDB database in the cloud and provides data to the frontend.

- Create a folder in Visual Studio Code called “BACKEND” –where all your project files will be stored.



-
-
-
- Install node.js which we use to write server-side applications.

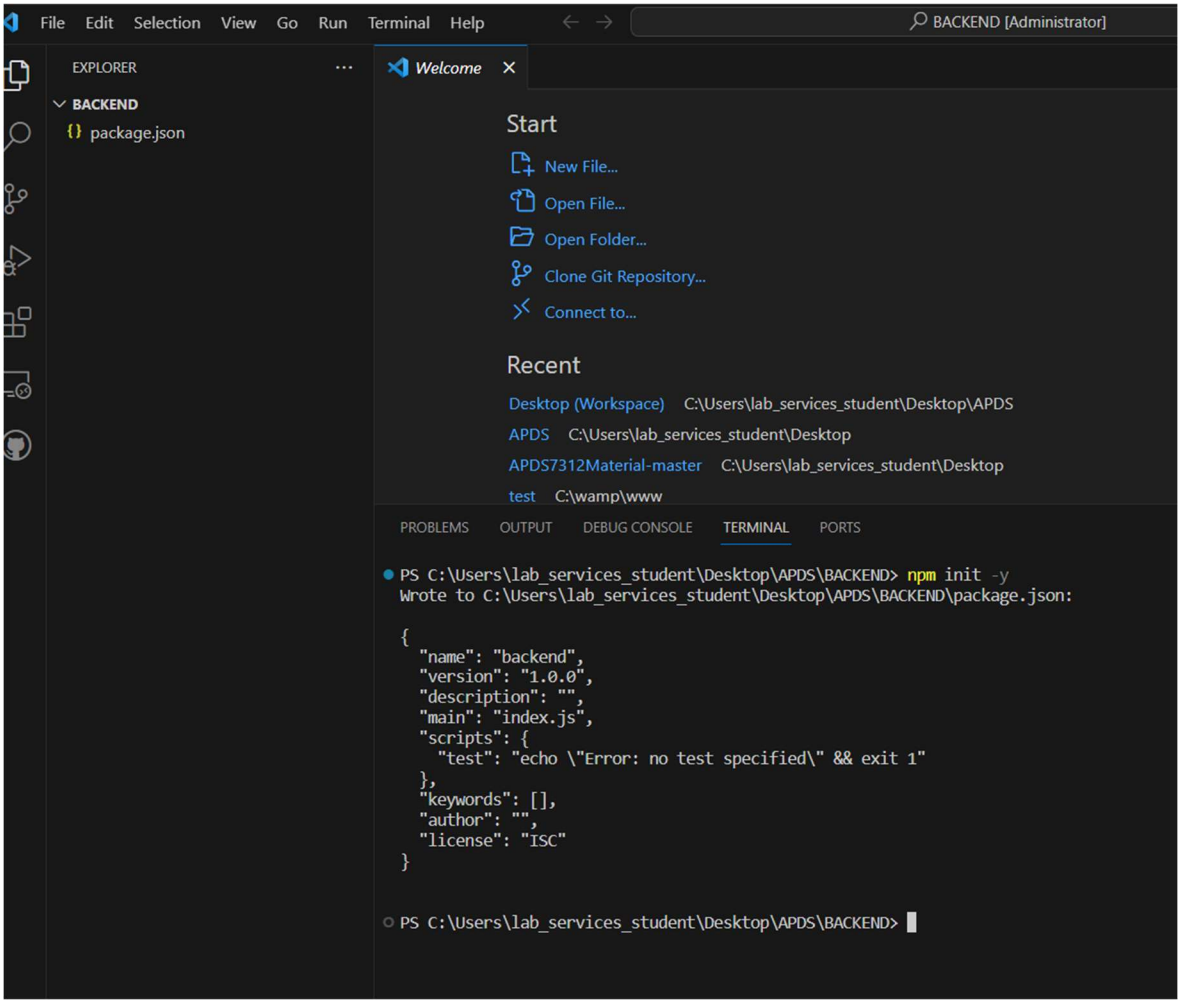
[Run JavaScript Everywhere - Node.js](nodejs.org/en) (2024)



In a new terminal window, make sure you are in the backend folder that you created earlier.

- Type this command (npm stands for node package manager https://docs.npmjs.com) npm init -y

This will create a package.json file. It lists all the dependent packages your projects require which are stored in a folder called node_modules.



- Add a package to your project using the “npm” install command
  (i=install, D=dev only dependencies)

  npm i -D nodemon

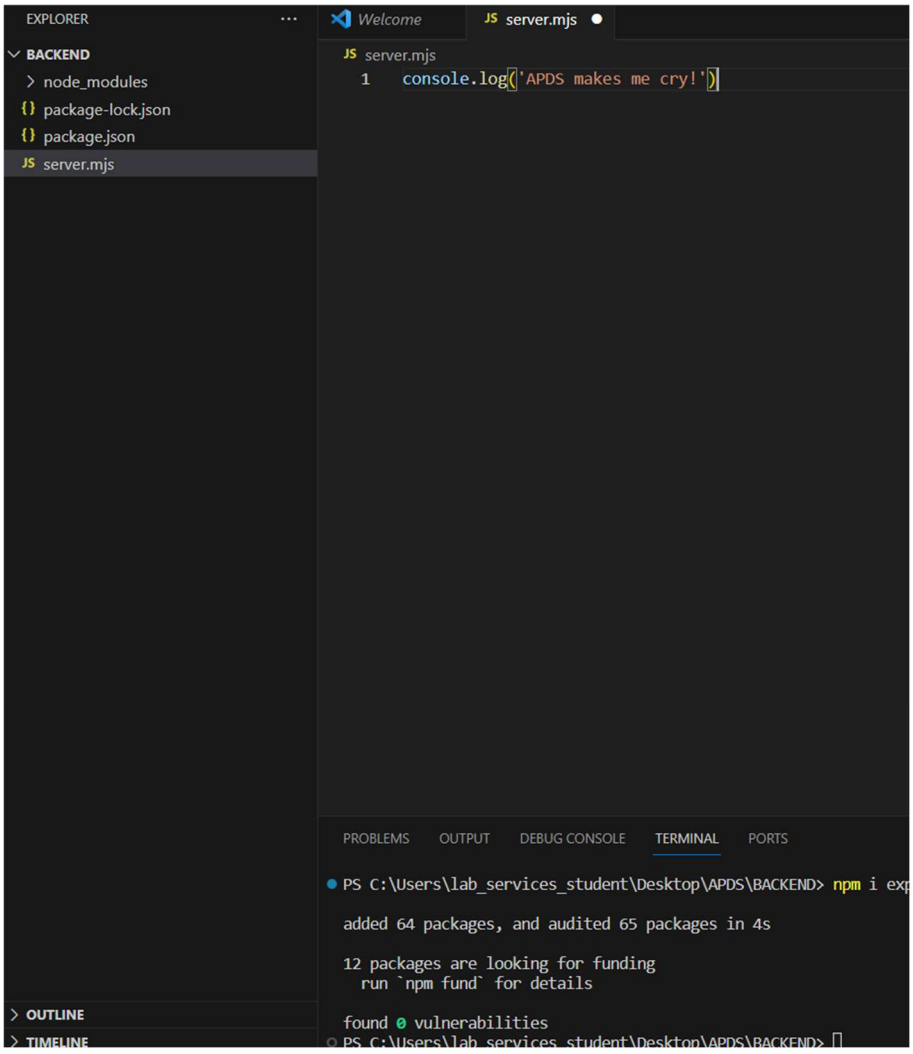
https://www.npmjs.com/package/nodemon
“nodemon is a tool that helps develop node.js based applications by automatically restarting the node application when file changes in the directory are detected.”

You can see below this now appears at the bottom of the package.json file along with its version.

- In addition, change the package.json “scripts” section as below. This will allow us to run our backend in either development or production environments.

- Create a new file in your backend folder “server.js” and insert a console.log command as below. Remember this command – you will use it extensively throughout this exercise to debug.

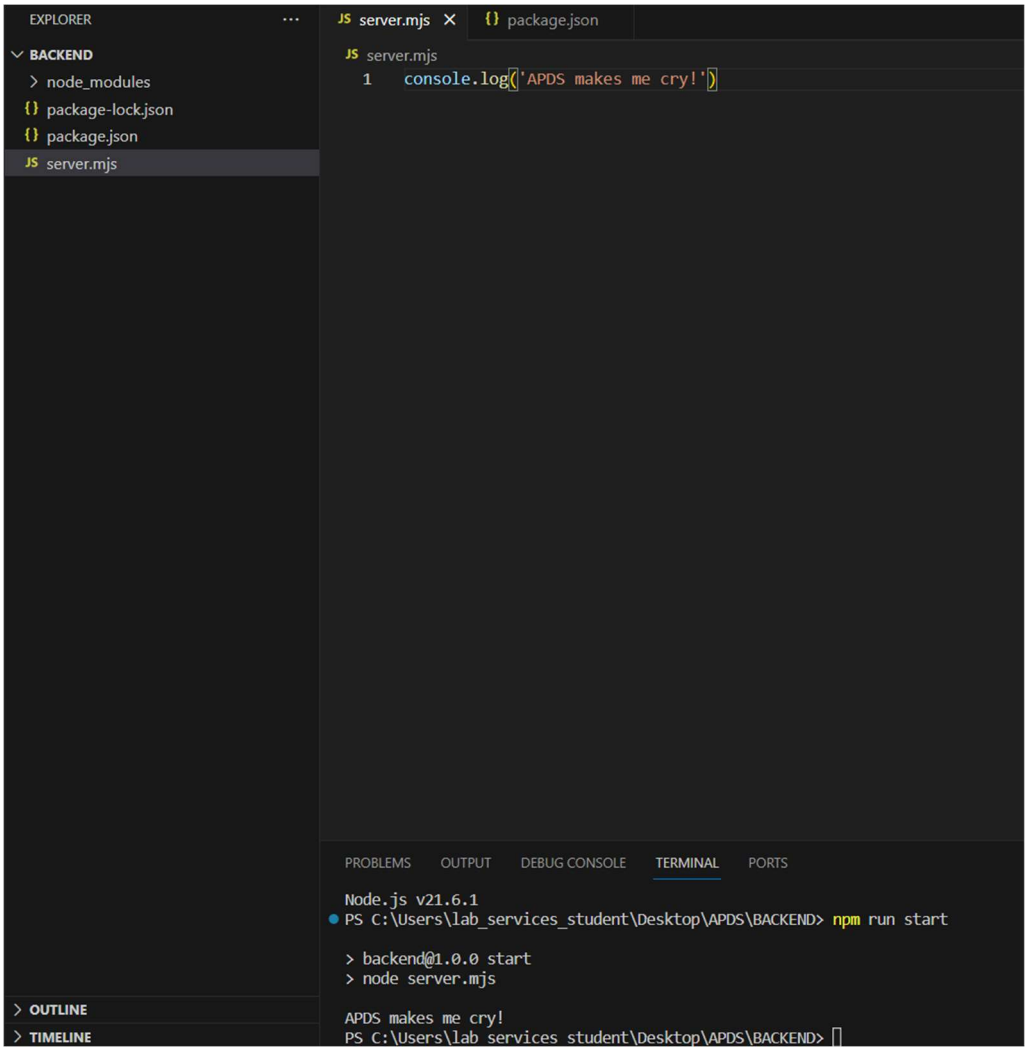
Make sure to “Save All” files from the menu. “Save” only saves the current tab you are on. This is a common issue that students encounter.



- Run the code as follows

  npm run start

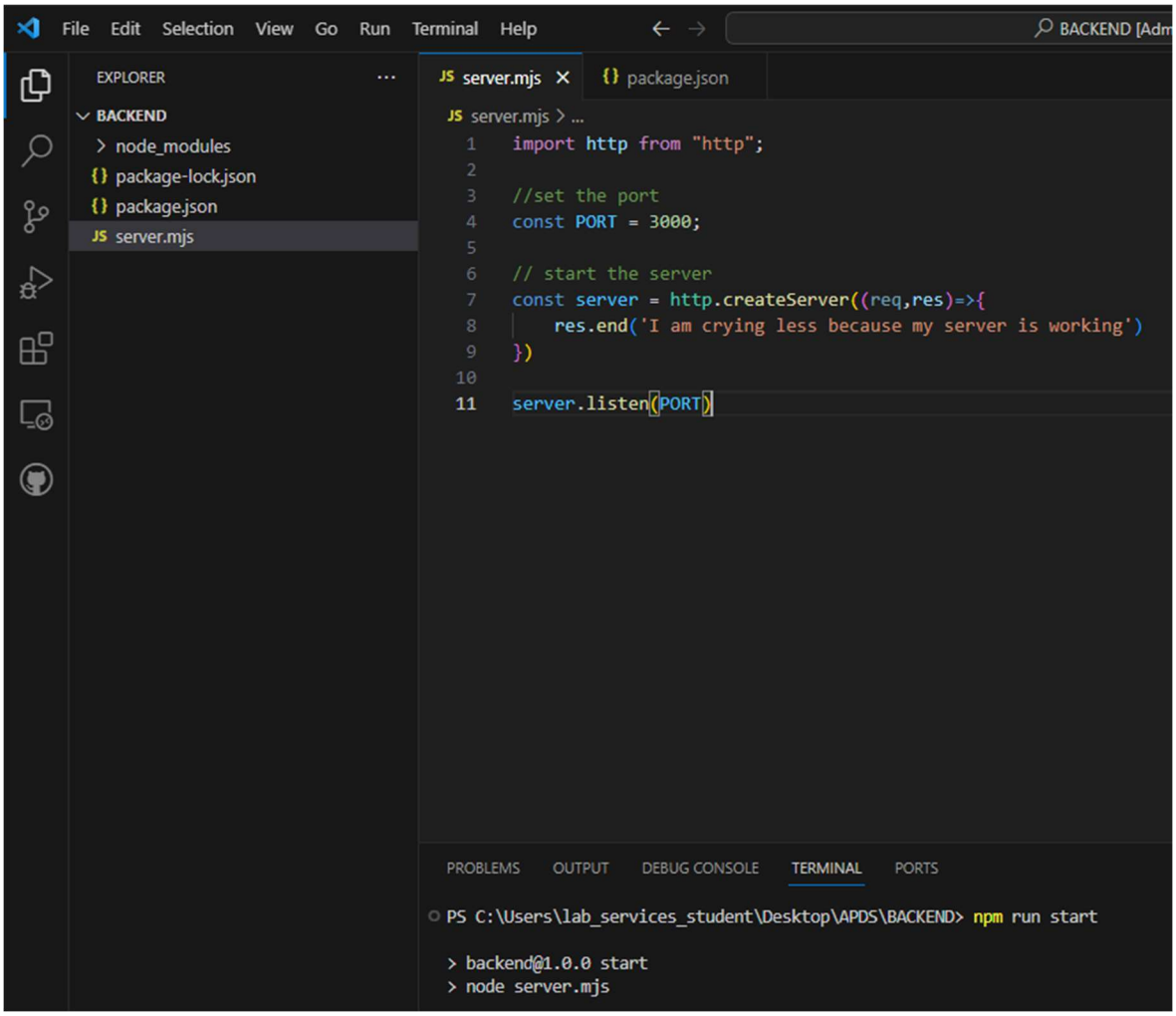
This runs “node server” as per your scripts tag in “package.json”
You can see in the terminal that it displays our log.



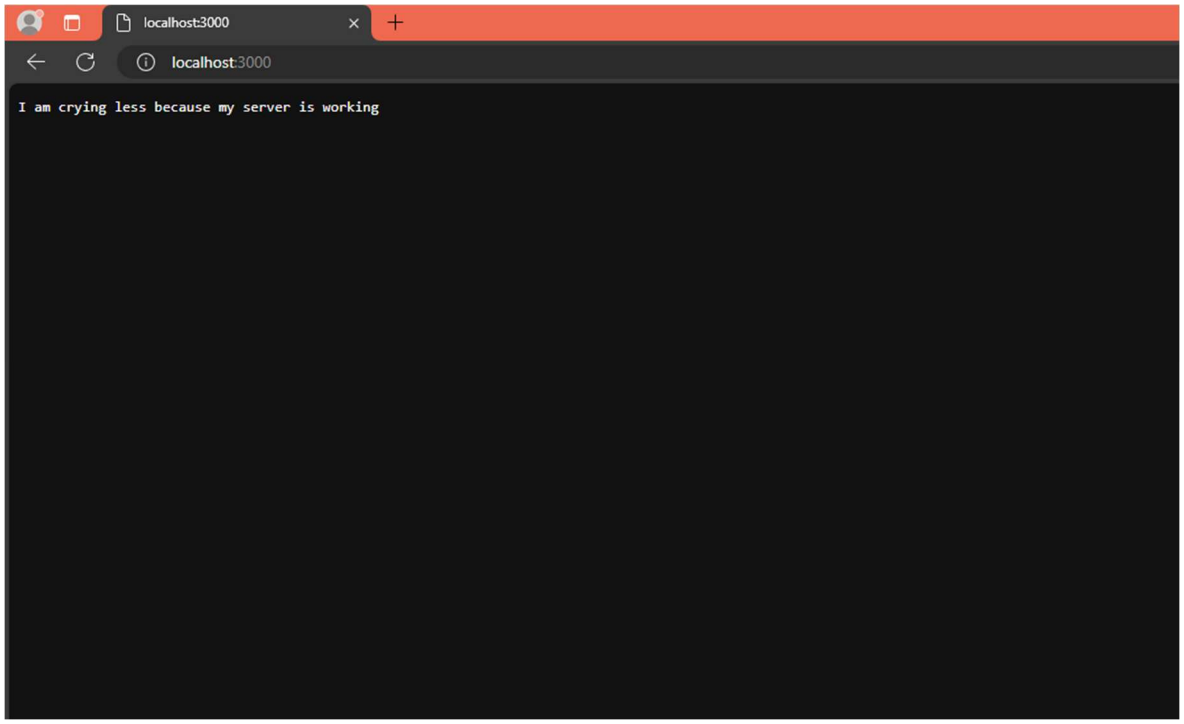
Our code is not however running as a HTTP server which listens on a port and is accessed with a web browser.

- Change server.js as follows.

MERN Stack Explained | MongoDB (Accessed 12 July 2024).

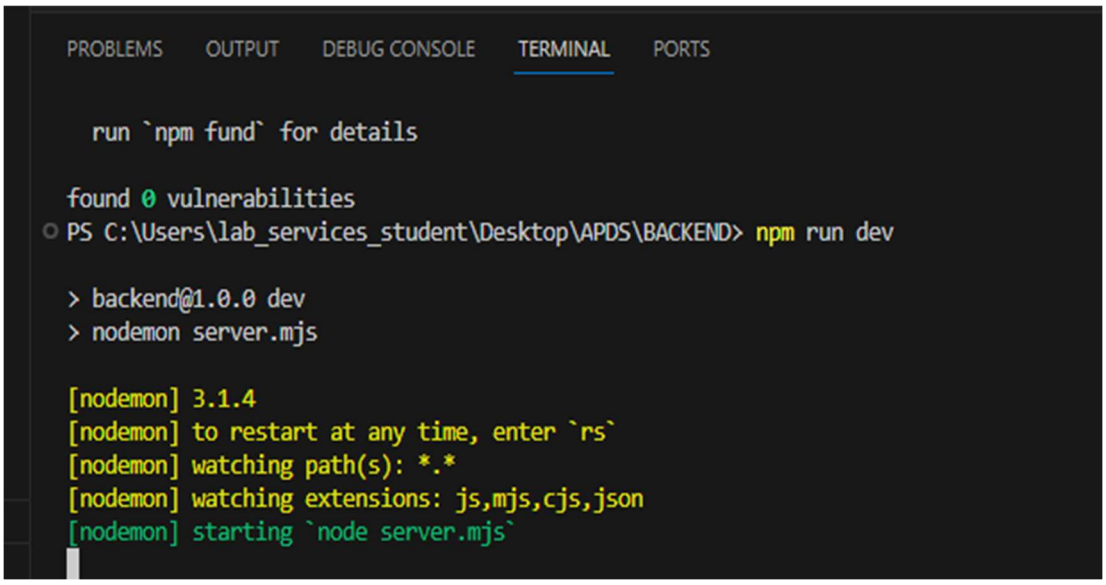


This time we must open a web browser to see the output.



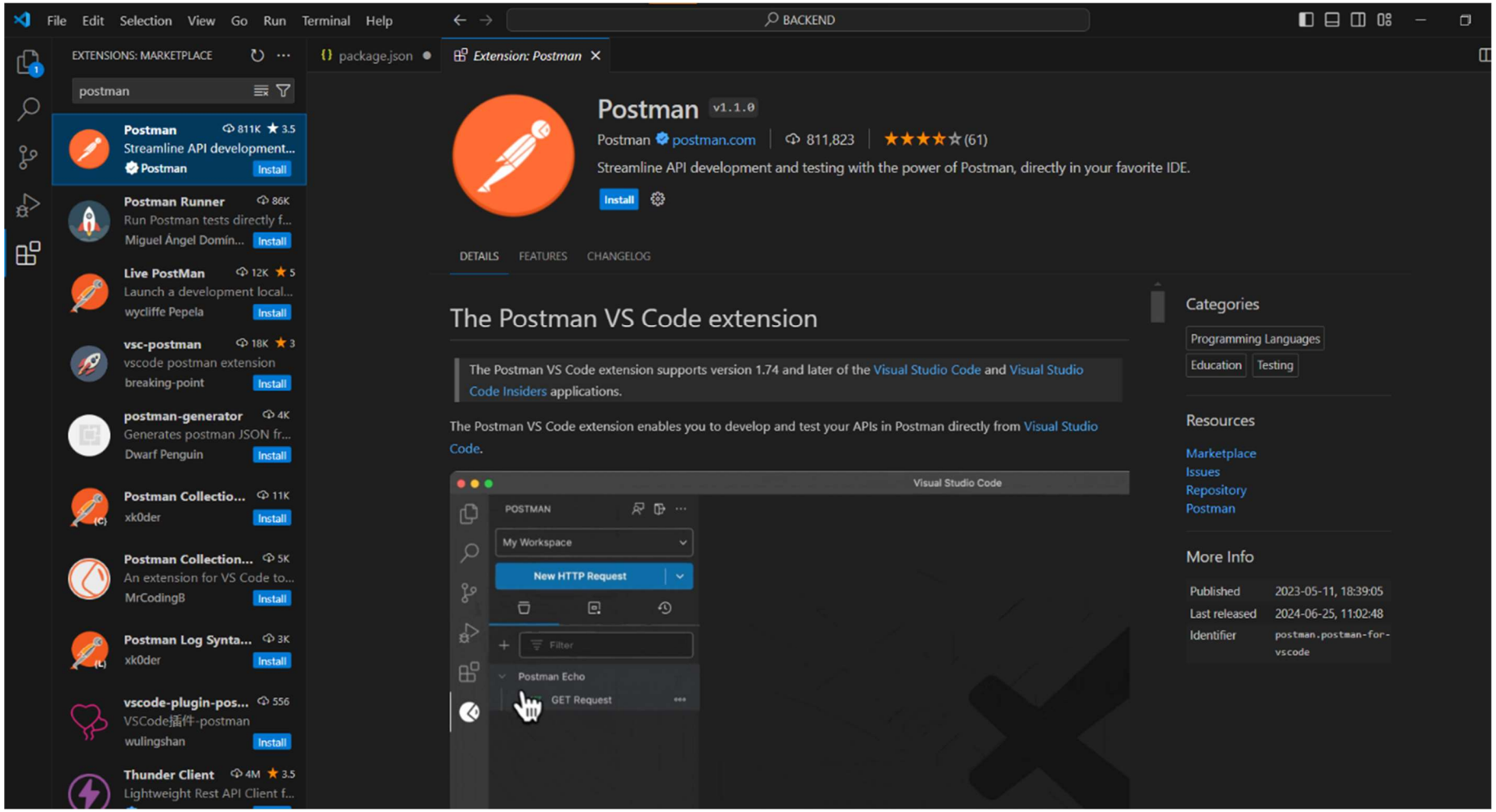
From now on, instead of running “npm run start” we run (as we would in production)
npm run dev

This will restart the server every time we make a change to our code and save – thanks to nodemon
(and the fact that we changed our scripts tag in package.json)

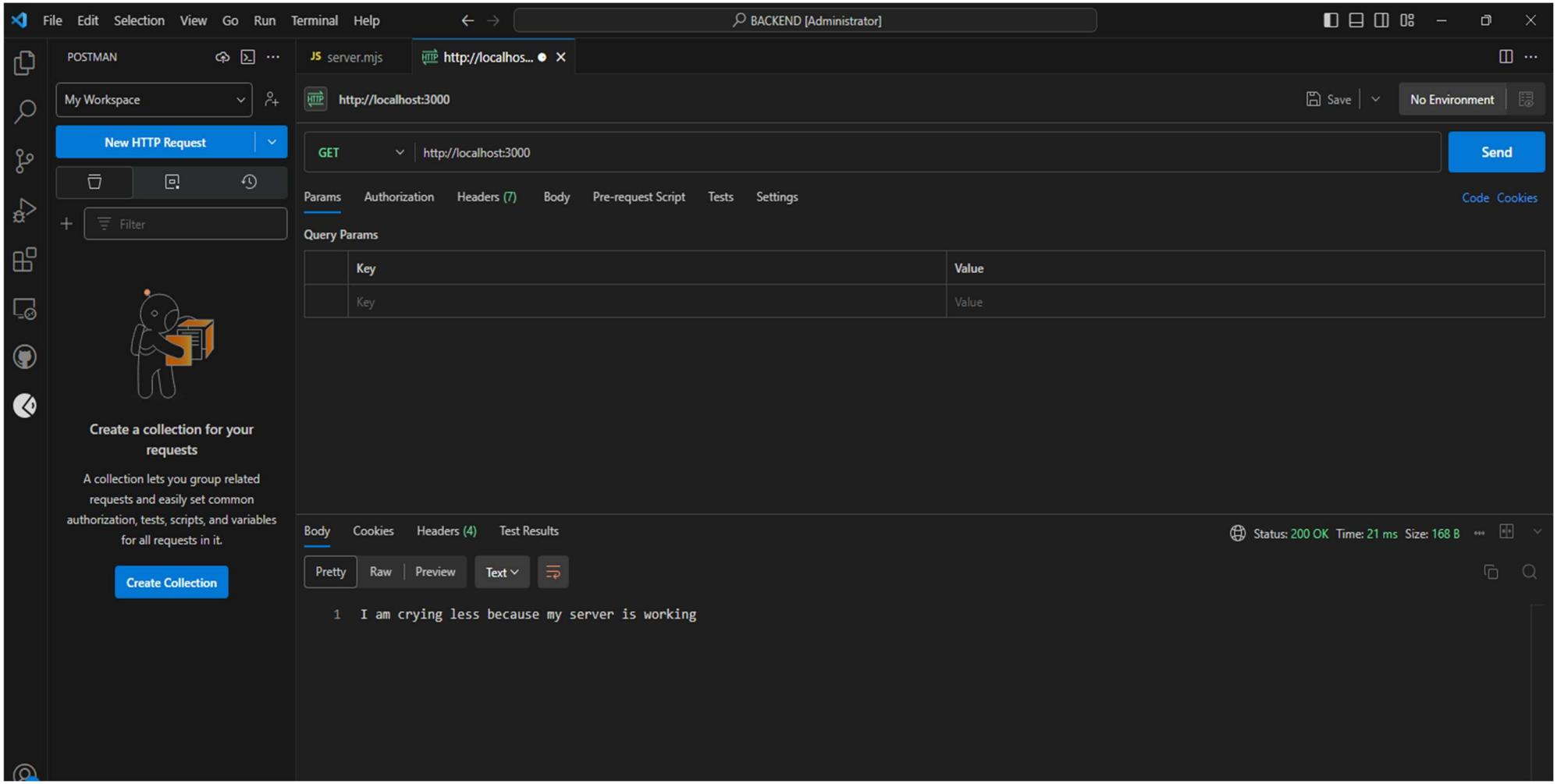


Before we continue, we must install a Visual Studio Code extension called “Postman”(you can also download the desktop app). This will allow us to call code in our backend API server without a frontend (or web browser)!

[Get started in Postman | Postman Learning Center](#) Accessed 10 July 2024.

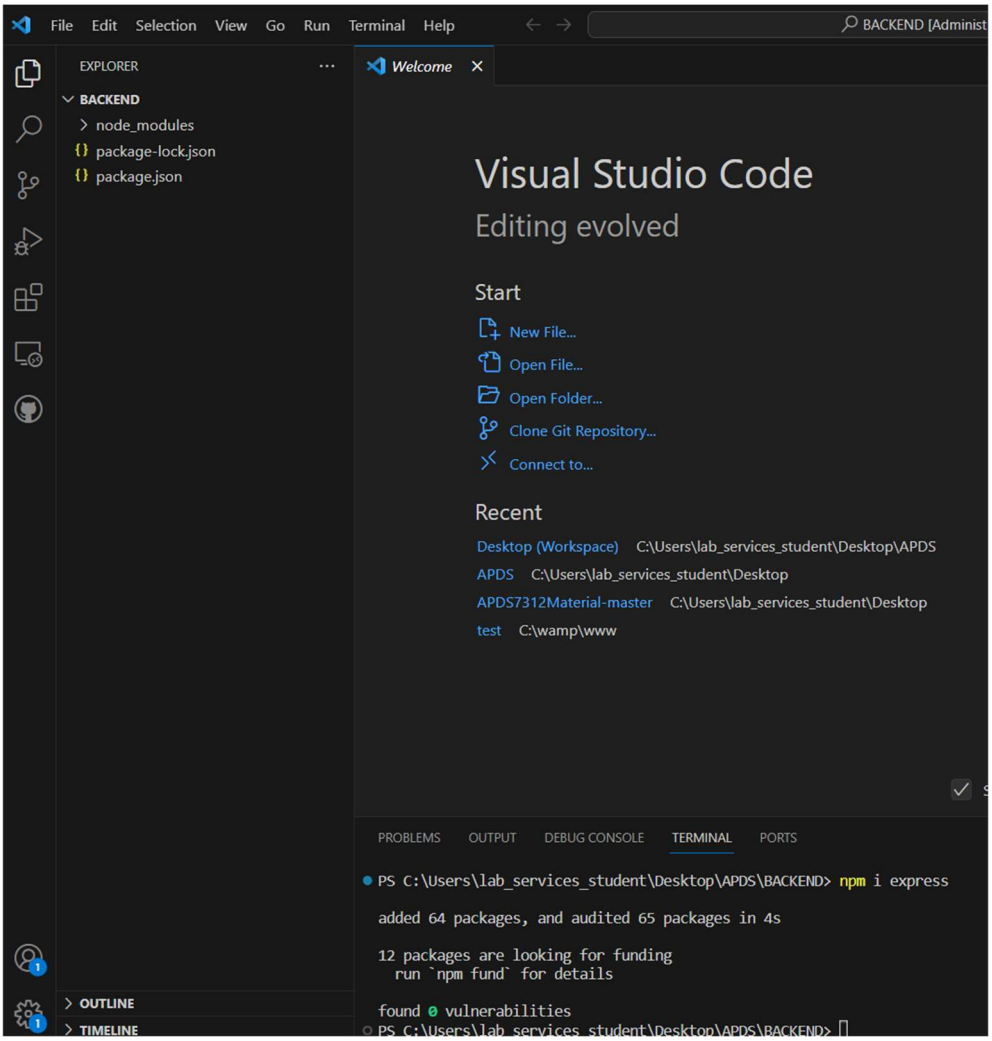


Once the Postman extension is installed, we need to login, once logged in you can click new request and perform a get request to your server's port.

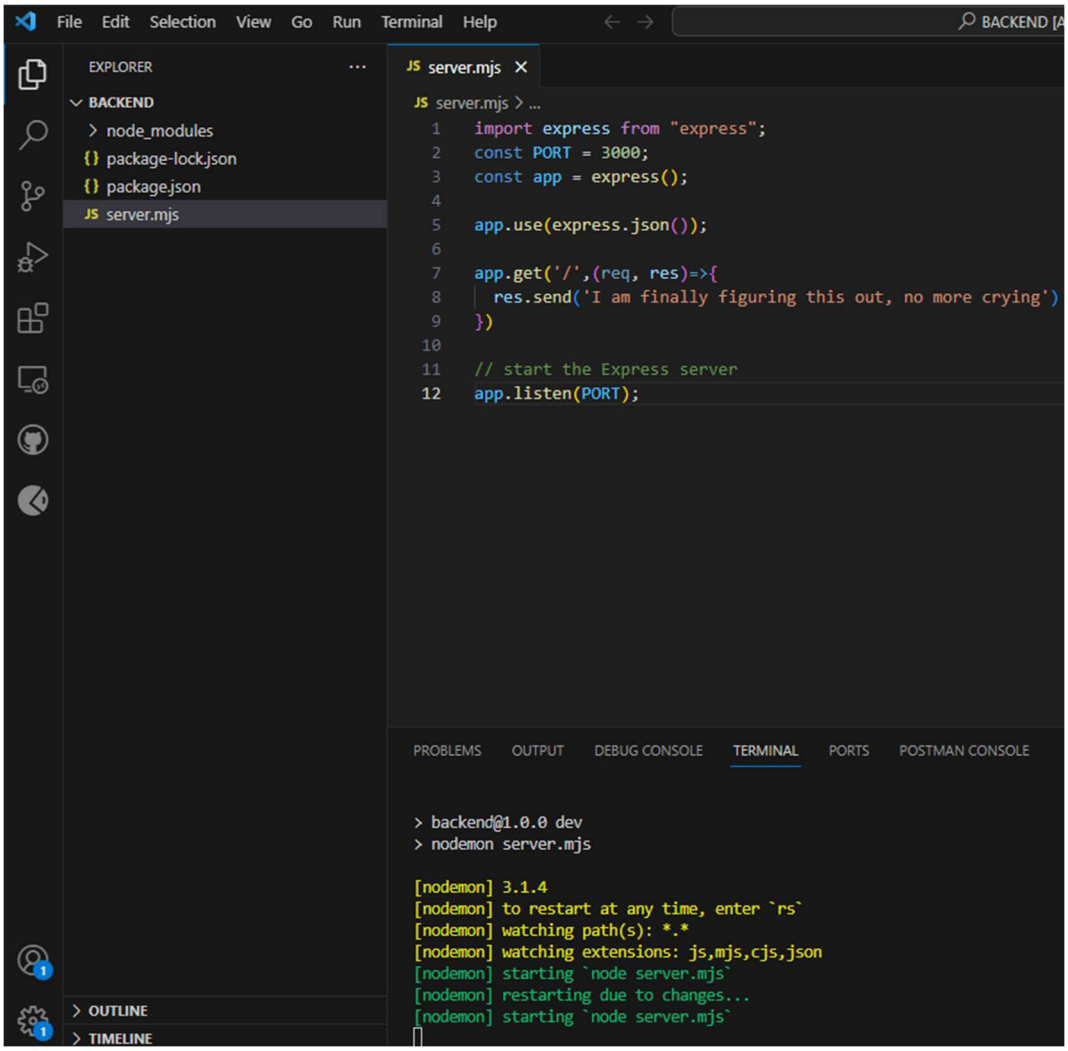


Now hand coding our “server.mjs” in this way would be very tedious. Fortunately for us “there’s a package for that” called Express (yes, the “E” in MERN, along with N for Node).

- We can install it by running the following command in the terminal
  npm I express



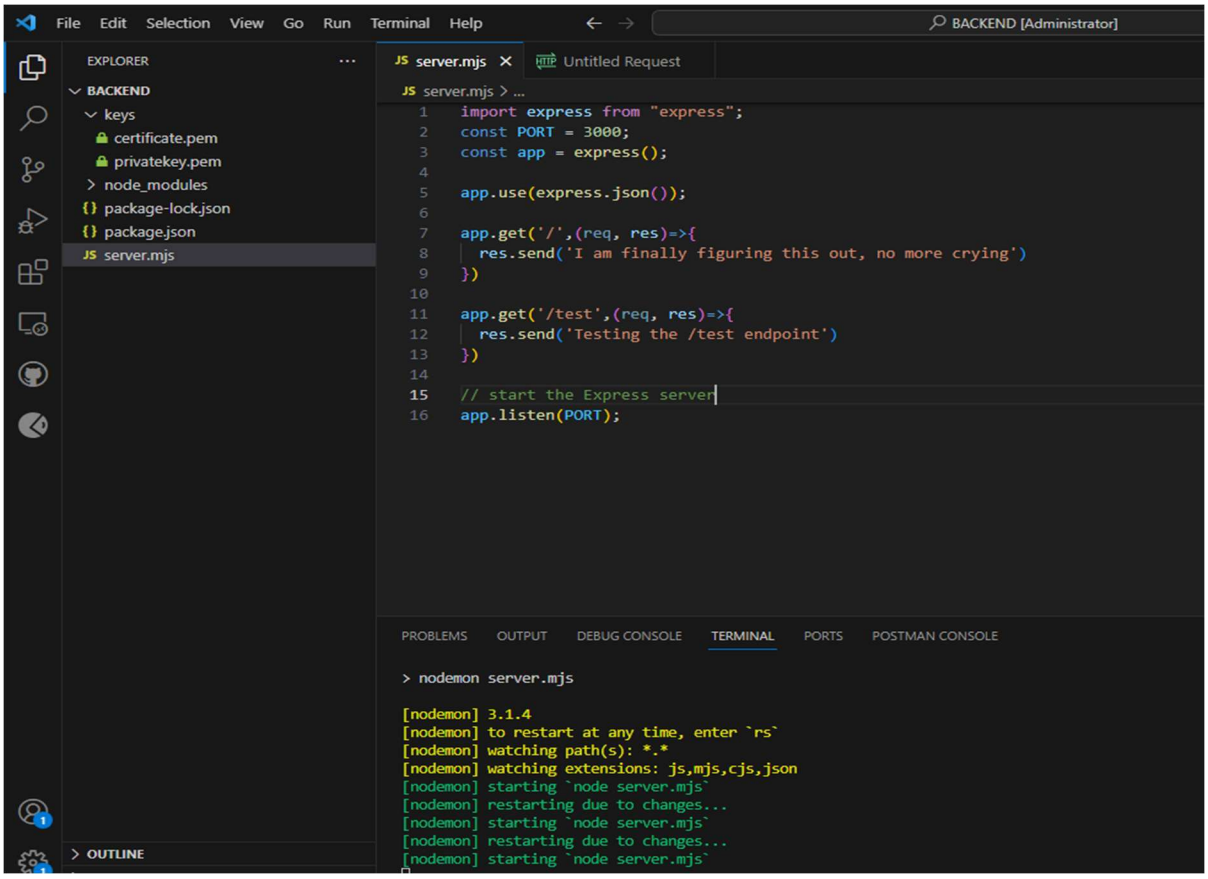
- Change server.jms to look as follows in the screenshot below.



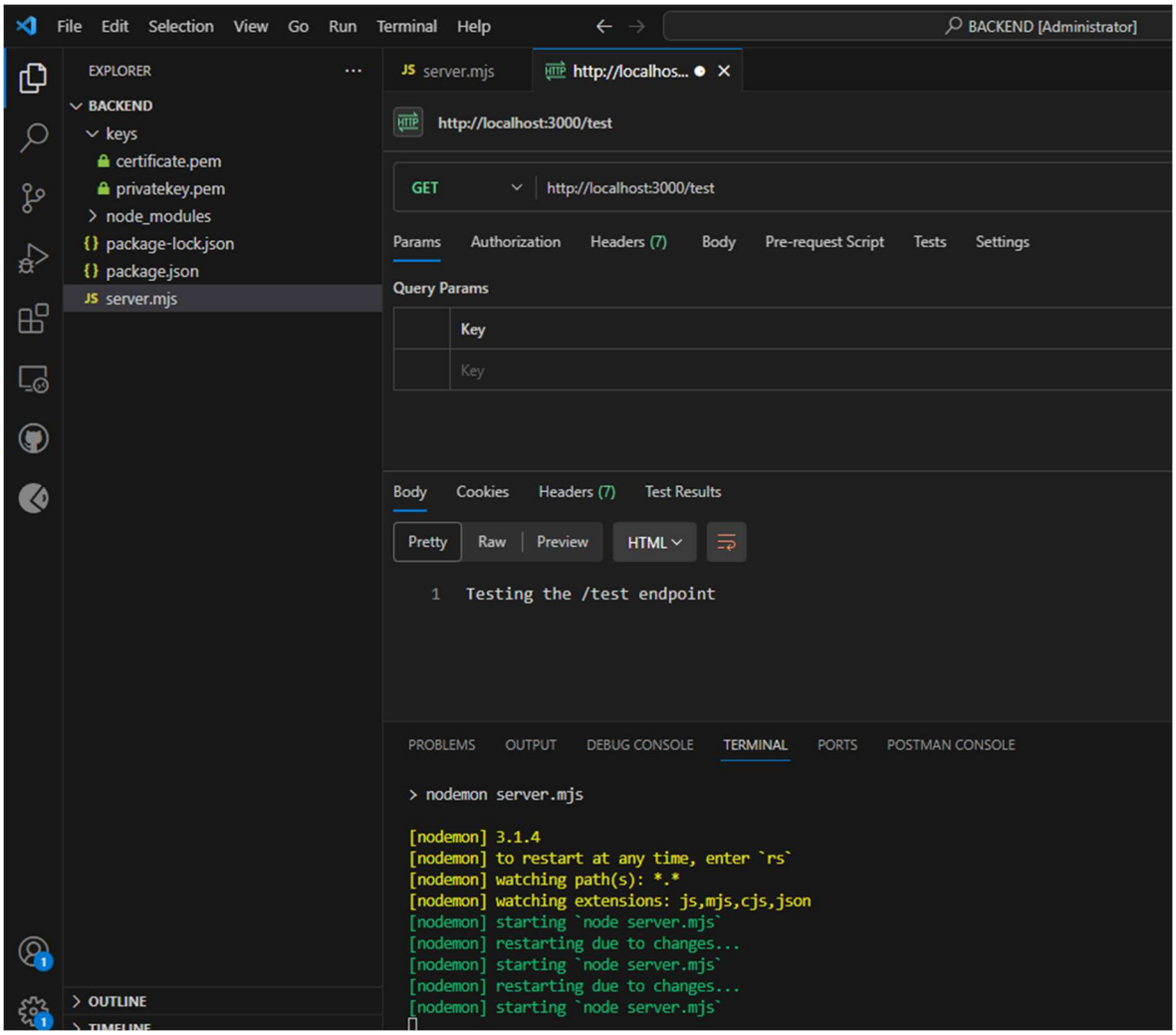
- Send Request" to test your server in Postman again. https://expressjs.com/en/starter/hello-world.html [Accessed 13 July 2024]

Update server.mjs as follows to create a testing endpoint.

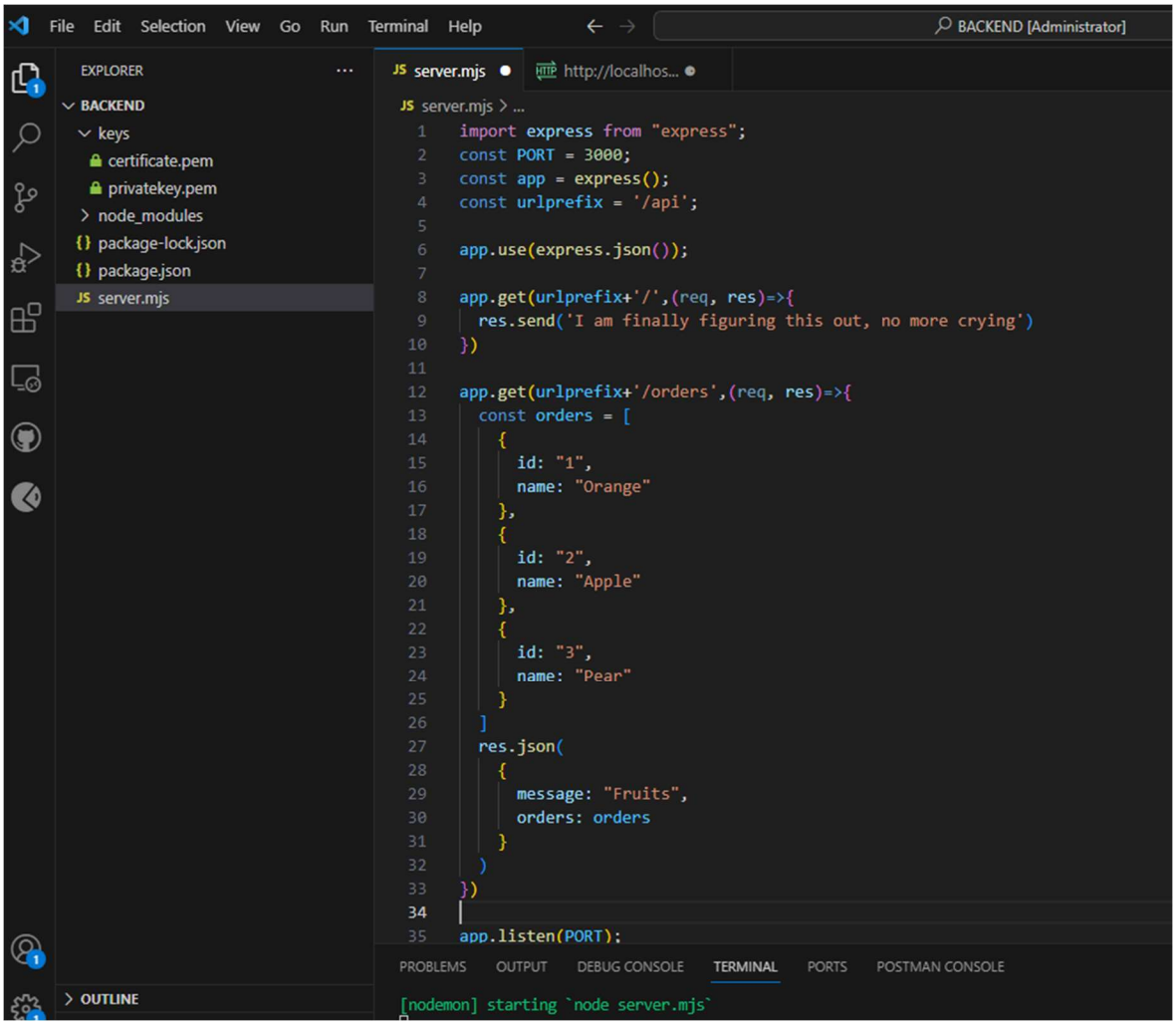
Both these endpoints will now work.
http://localhost:3000/test [Accessed 13 July 2024].



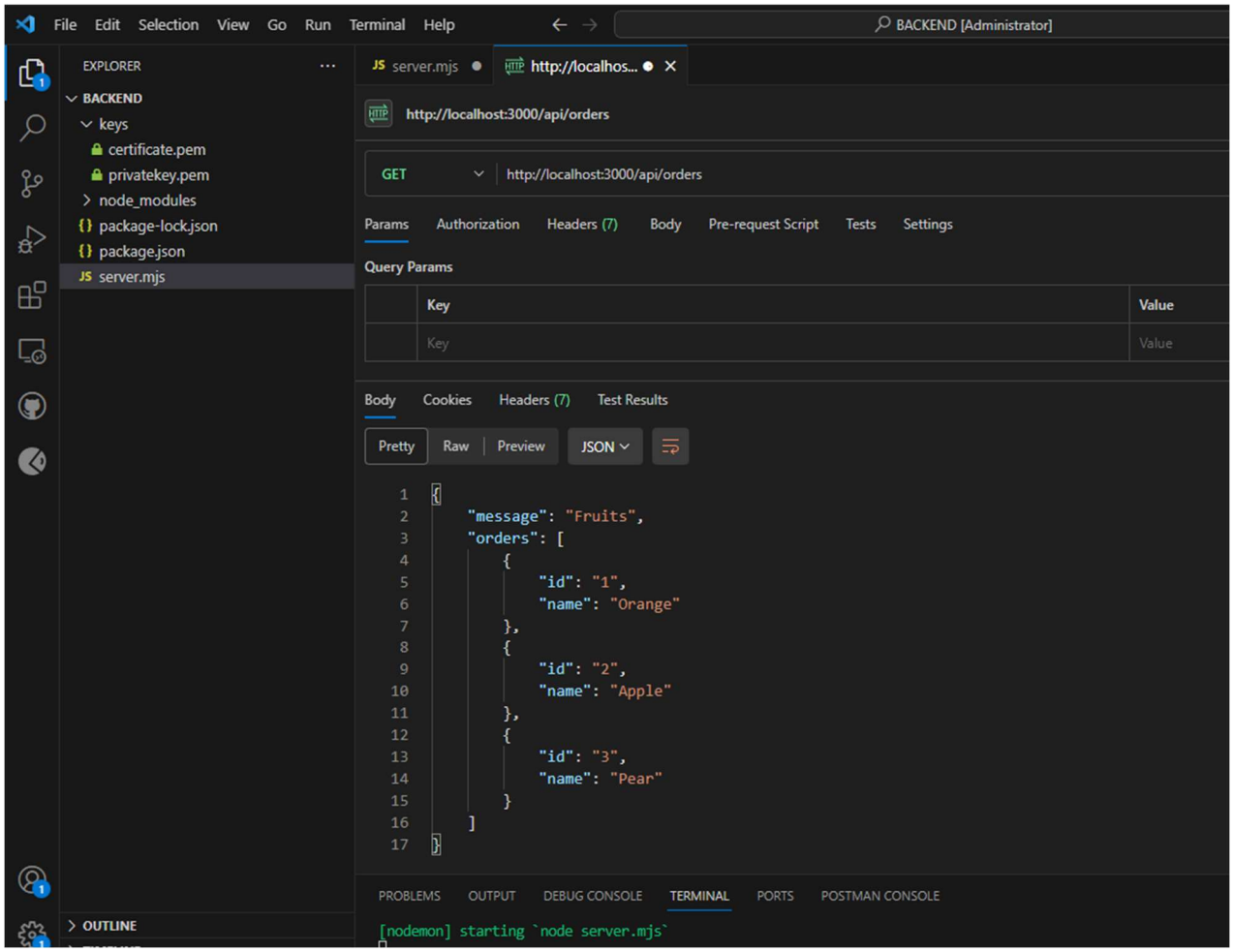
Do not forget to test with Postman



Now let's return a more complex string like JSON and modify our endpoint



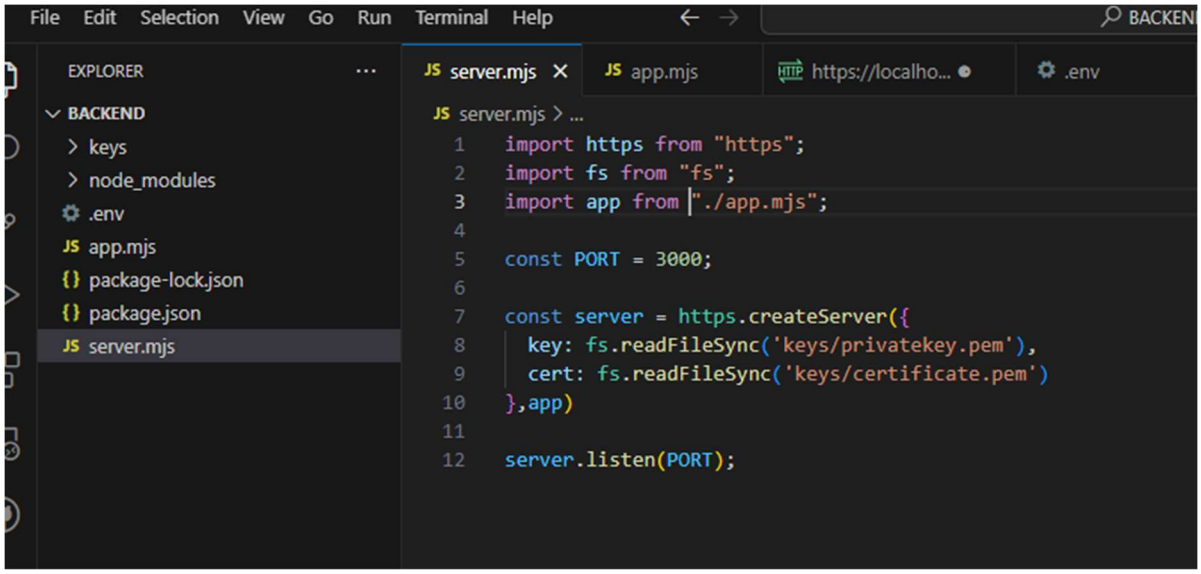
Now Let's Test to see if our updates worked.



In order to start securing our server with SSL we need to:

- Start by creating a new folder called “keys” and copying the two .pem files you created earlier using OpenSSL into it.

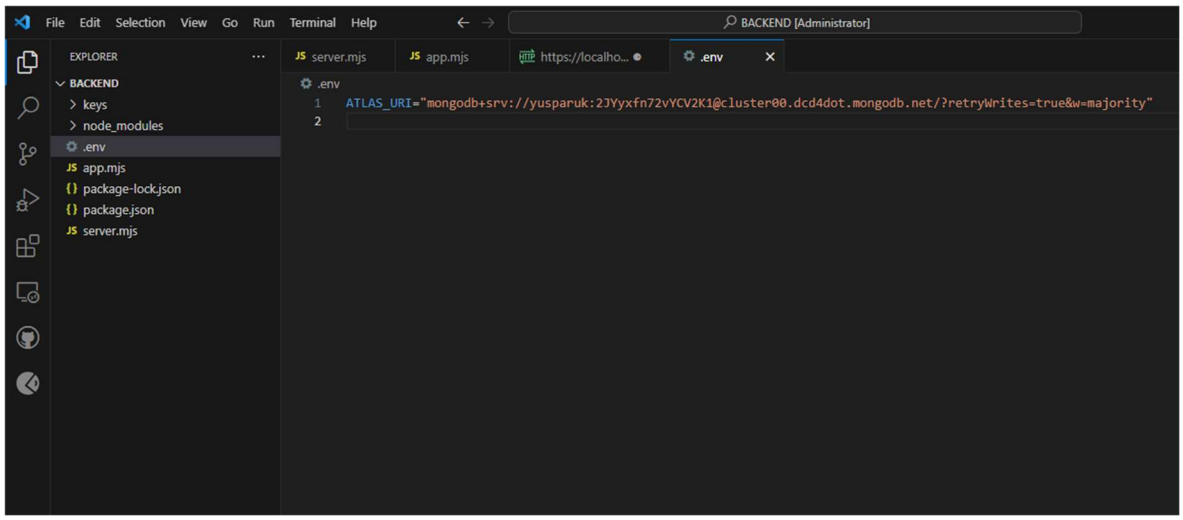
- Then change your server.js as below.



2.4 *Integrating with MongoDB*

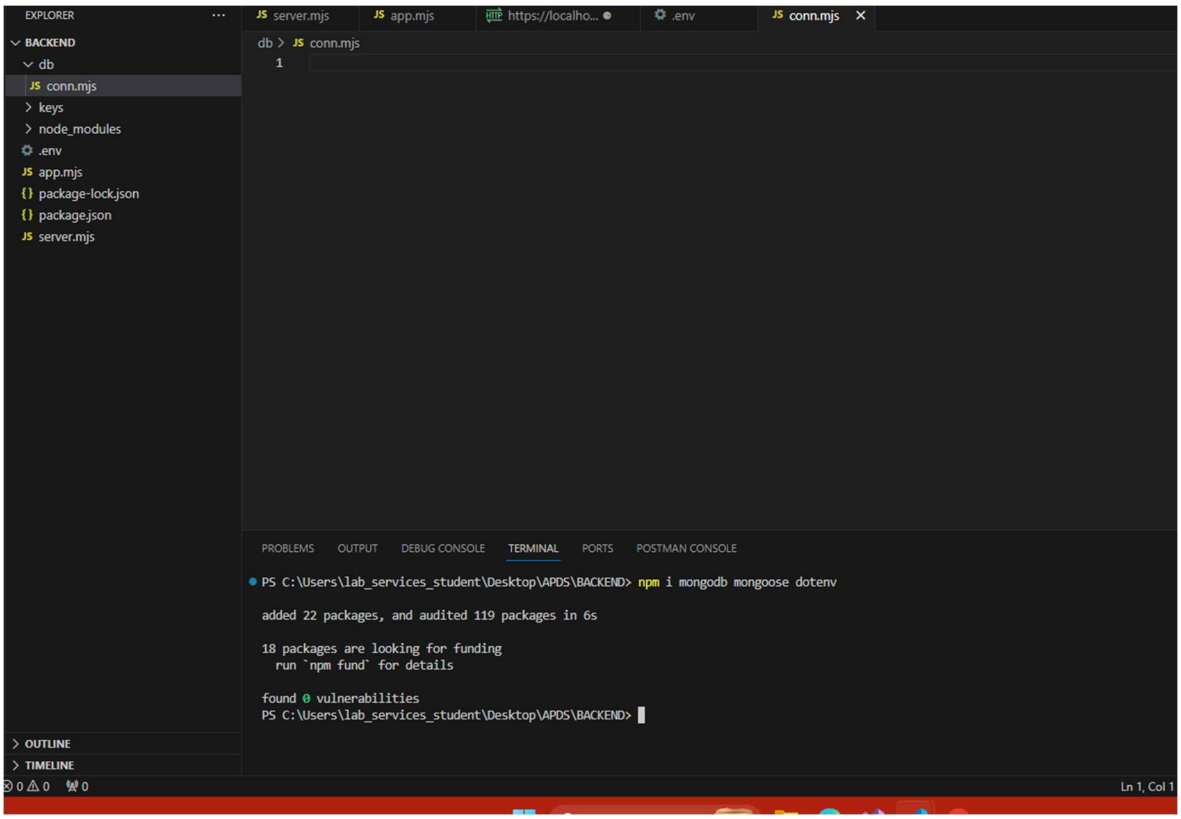
- To connect to MongoDB, we need to save our connection string as an environment variable.

We create a file called .env and save the connection string within

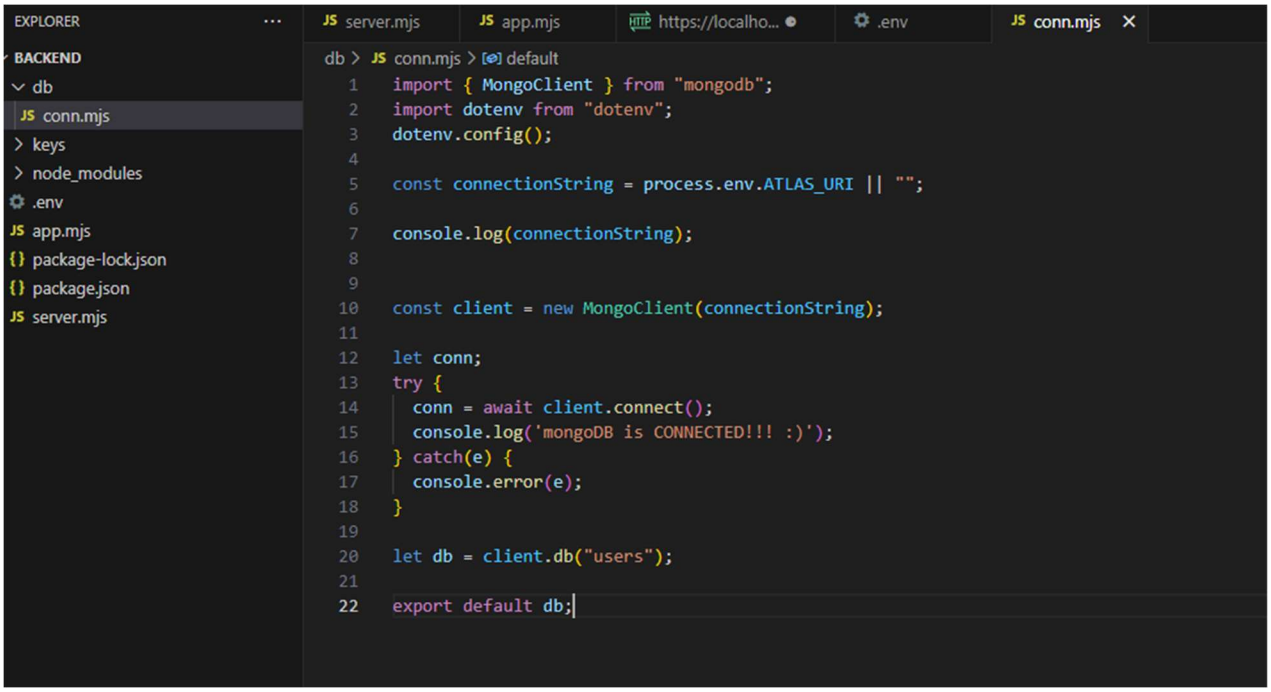


- Next we create a folder called db with our connection file within.
- We also install new packages using the terminal.

This file will connect our app to the database.



We can now update the connection file to connect us to the database.
We use MongoClient to connect us to the database and dotenv to load the environment variables.



We are now moving on to Routing, we use routing to route different endpoints to functions.
In this case we can create a routes folder and a post.mjs file.

Here we created the route “/” to get all the posts.
And “/upload” to create a post.

BACKEND > routes > JS post.mjs > ...

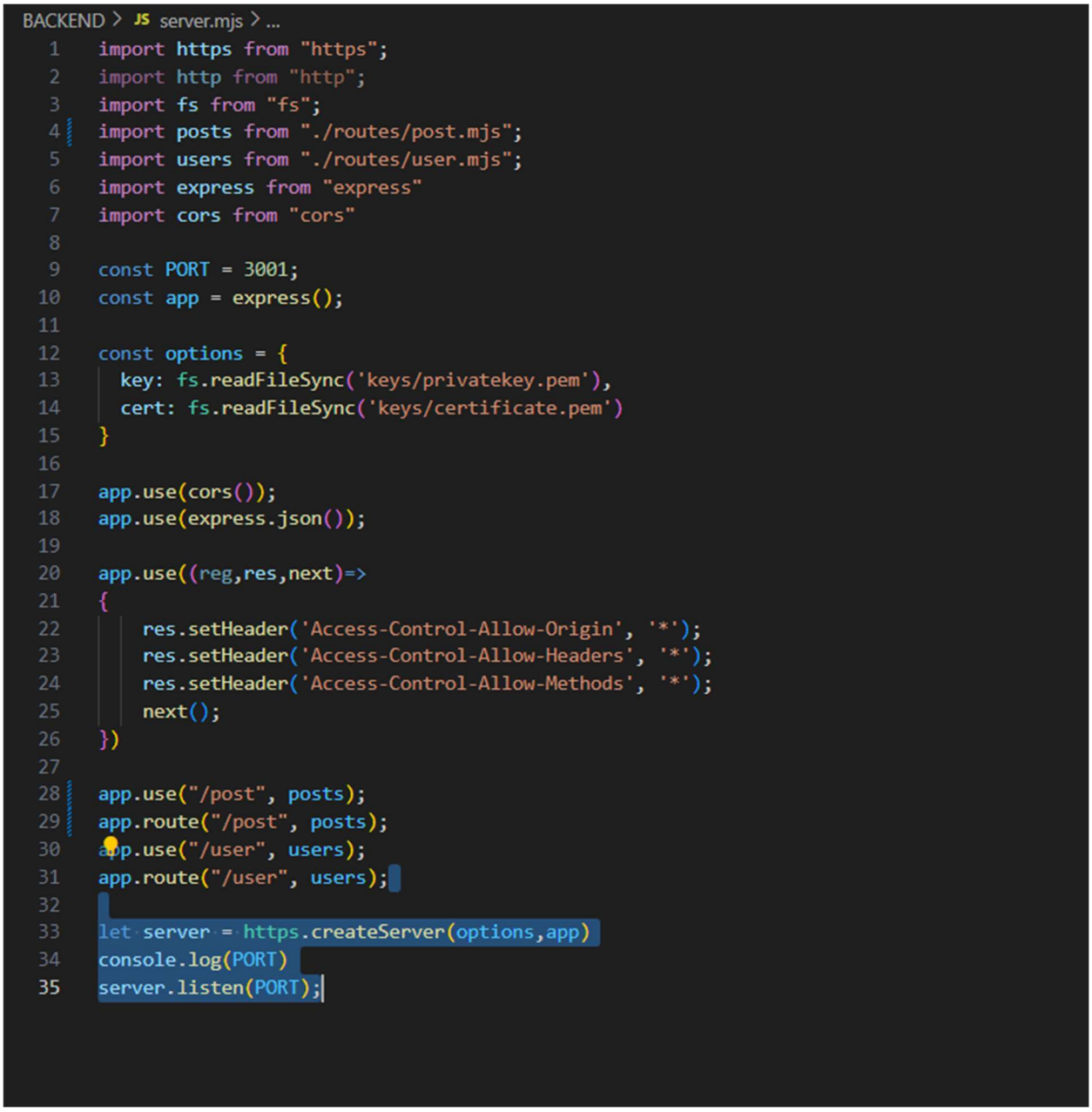
```
1   import express from "express";
2   import db from "../db/conn.mjs";
3   import { ObjectId } from "mongodb";
4
5   const router = express.Router();
6
7   // Get all the records.
8   router.get("/", async (req, res) => {
9     let collection = await db.collection("posts");
10    let results = await collection.find({}).toArray();
11    res.send(results).status(200);
12  });
13
14  // Create a new record.
15  router.post("/upload", async (req, res) => {
16      let newDocument = {
17        user: req.body.user,
18        content: req.body.content,
19        image: req.body.image
20      };
21      let collection = await db.collection("posts");
22      let result = await collection.insertOne(newDocument);
23      res.send(result).status(204);
24  });
25
```

Our post takes 3 inputs, a username, content and a picture which will be saved as string in the db.

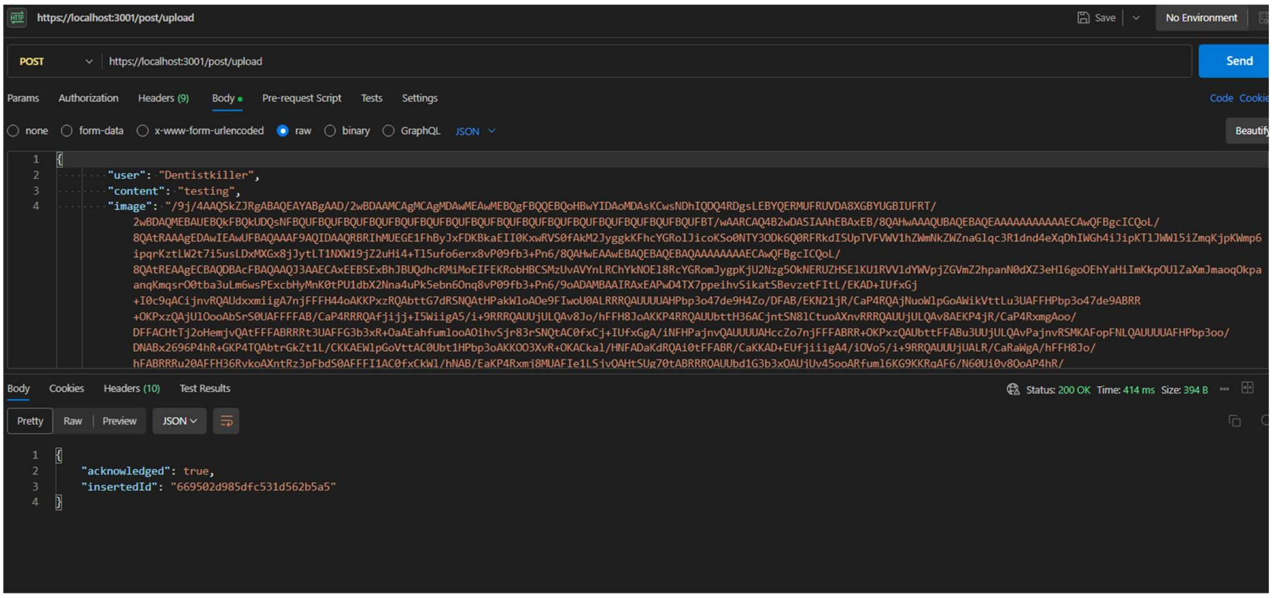
Now we can update our server.mjs

We are including cors to allow our server to accept requests from other endpoints, here we have everything set to “*” which means all.

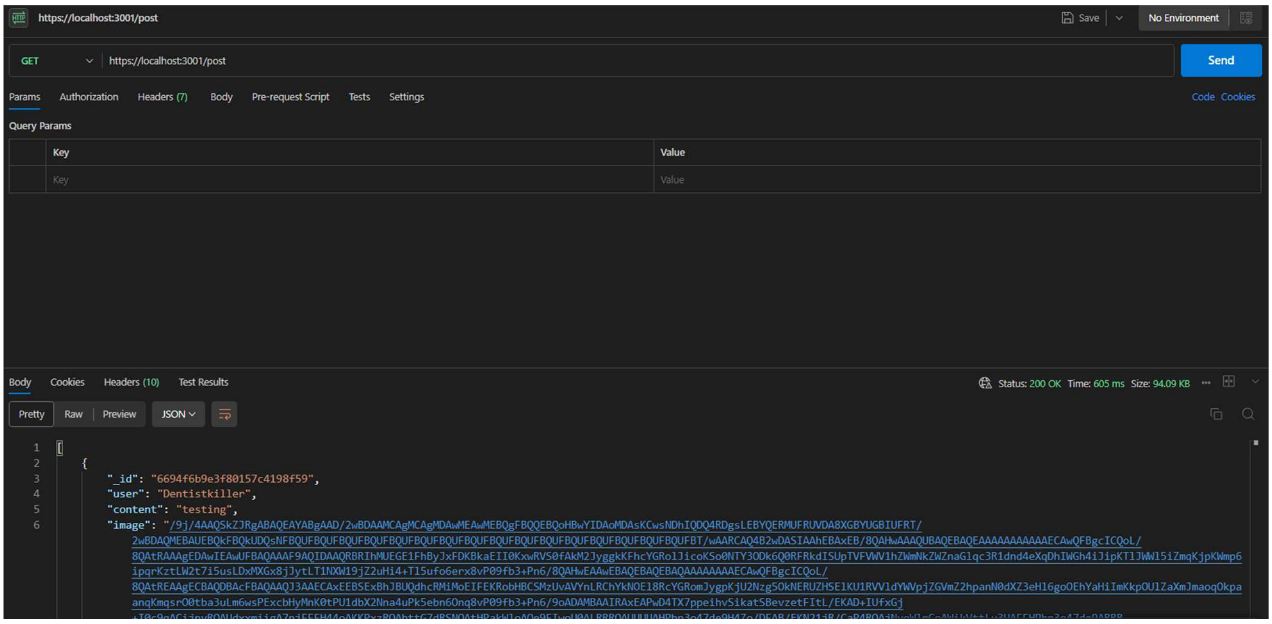
Then we import our posts from the post.mjs file and we allow our app to use the routes.



Now we can test creating a new post using a POST request.



And we can test if it updated by performing a GET request.
Note for now input a random string to test for the image.

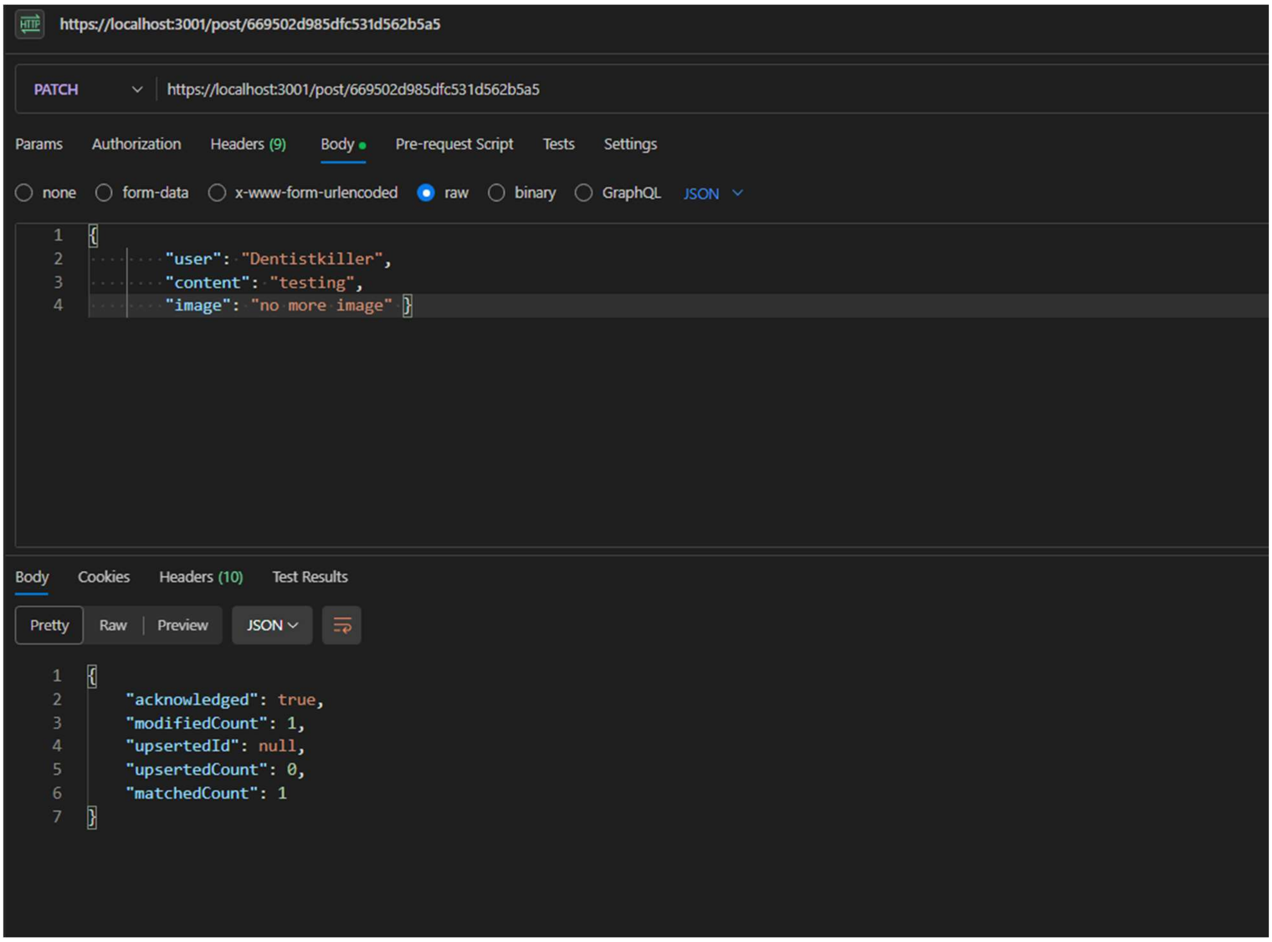


Since we have first 2 endpoints working, we can now update our post.mjs with other operations, like edit (PATCH),and DELETE, as well as to find a specific record by its ID.

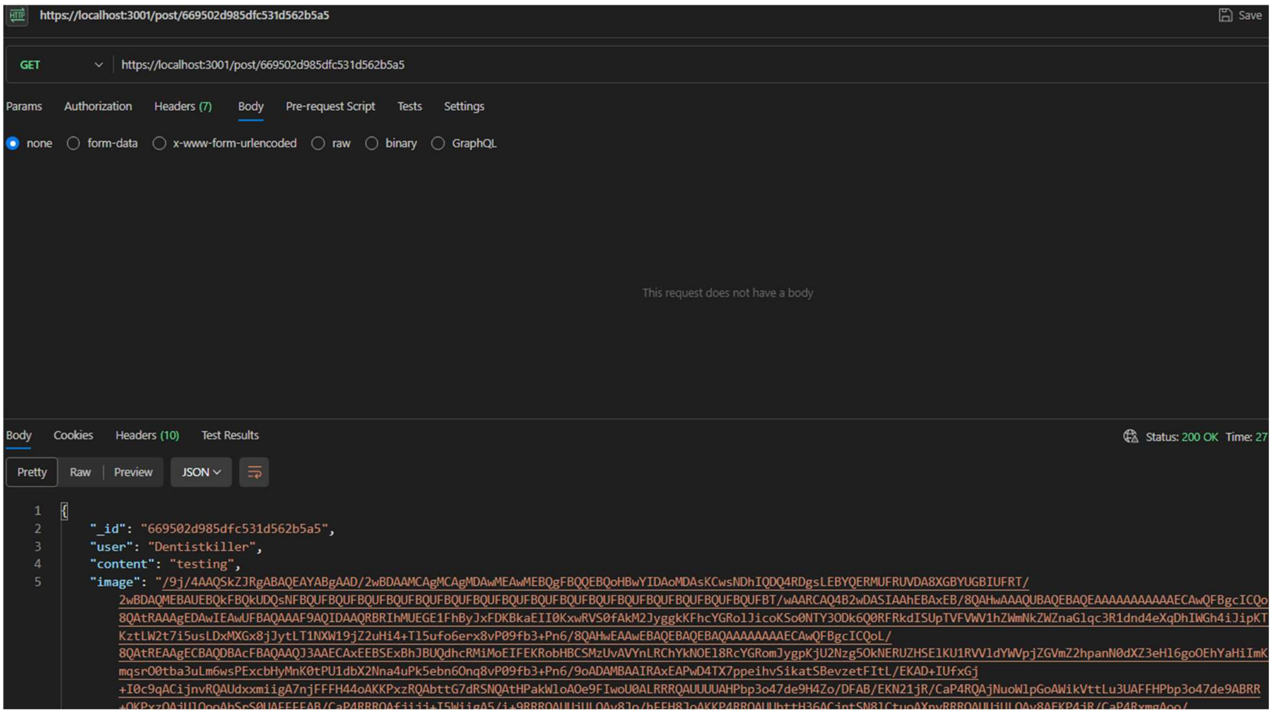
BACKEND > routes > JS post.mjs > ...

```
27  //Update a record by id
28  router.patch("/:id",  async (req, res) => {
29    const query = { _id: new ObjectId(req.params.id) };
30    const updates =  {
31      $set: {
32        name: req.body.name,
33        comment: req.body.comment
34      }
35    };
36
37    let collection = await db.collection("posts");
38    let result = await collection.updateOne(query, updates);
39
40    res.send(result).status(200);
41  });
42
43  // Gets a single record by id
44  router.get("/:id",  async (req, res) => {
45    let collection = await db.collection("posts");
46    let query = {_id: new ObjectId(req.params.id)};
47    let result = await collection.findOne(query);
48
49    if (!result) res.send("Not found").status(404);
50    else res.send(result).status(200);
51  });
52
53  // Delete a record
54  router.delete("/:id", async (req, res) => {
55    const query = { _id: new ObjectId(req.params.id) };
56
57    const collection = db.collection("posts");
58    let result = await collection.deleteOne(query);
59
60    res.send(result).status(200);
61  });
62
63  export default router;
```

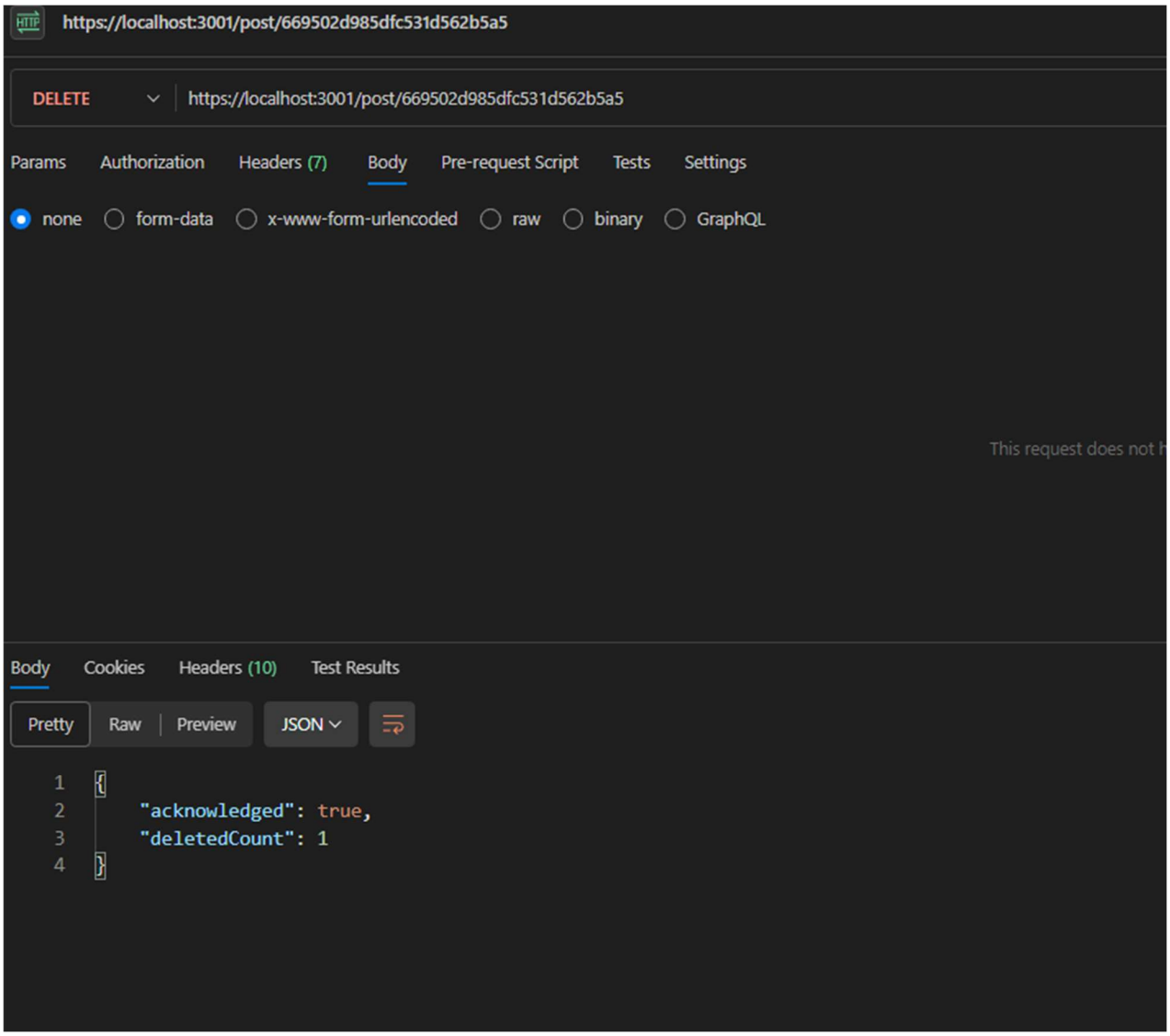
Testing the Patch method.

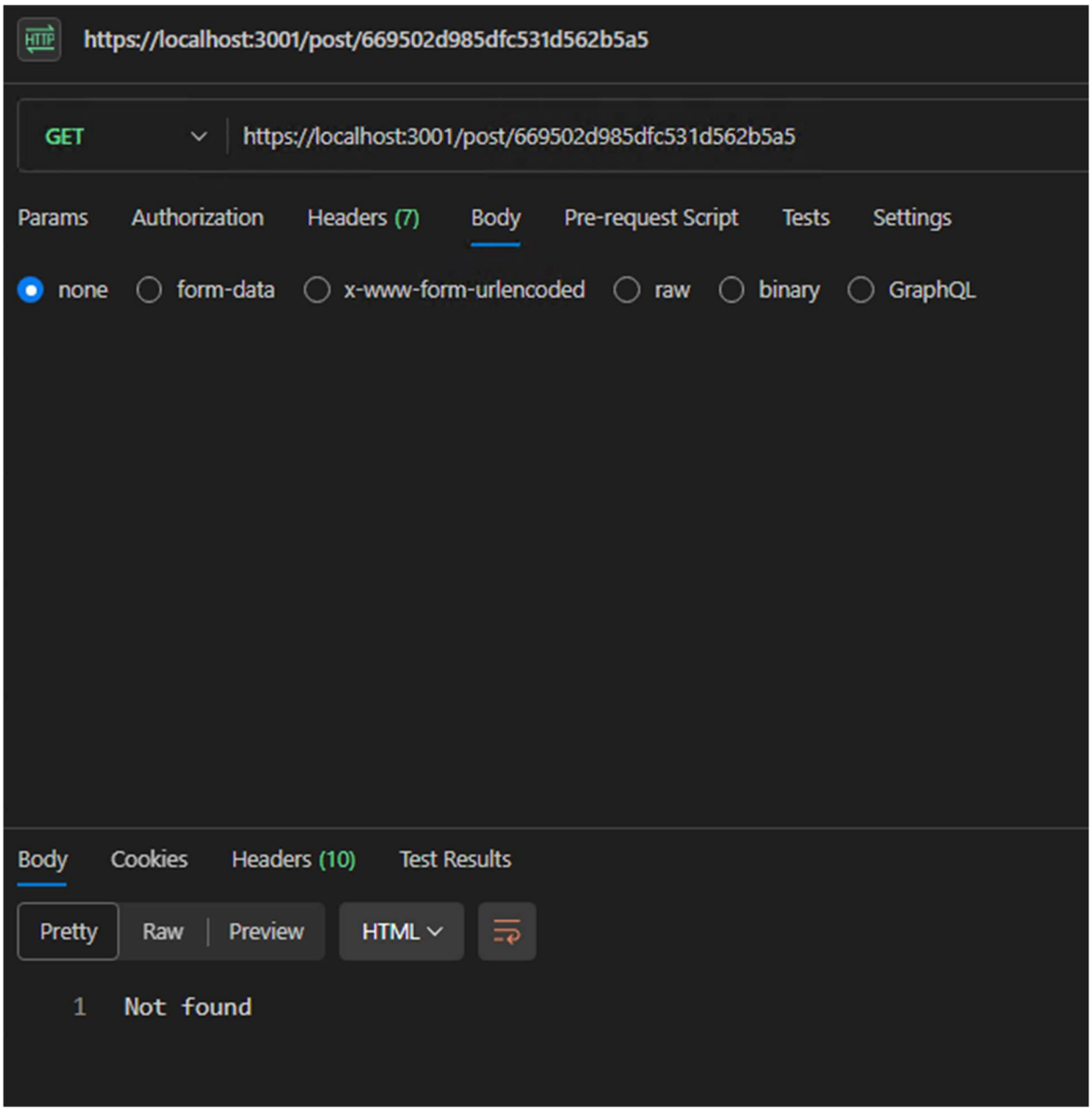


Testing to get specific updated records.



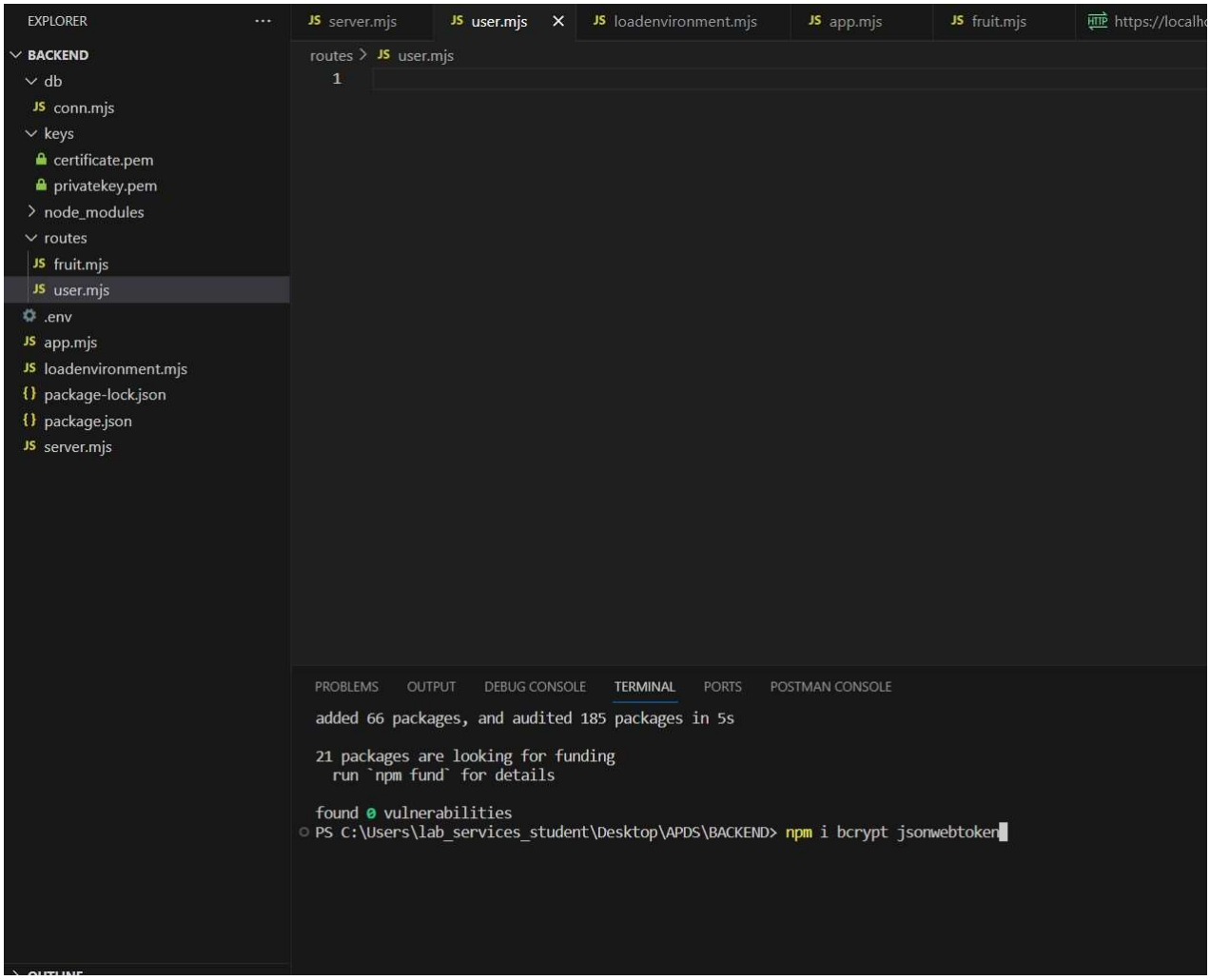
Testing Delete method.

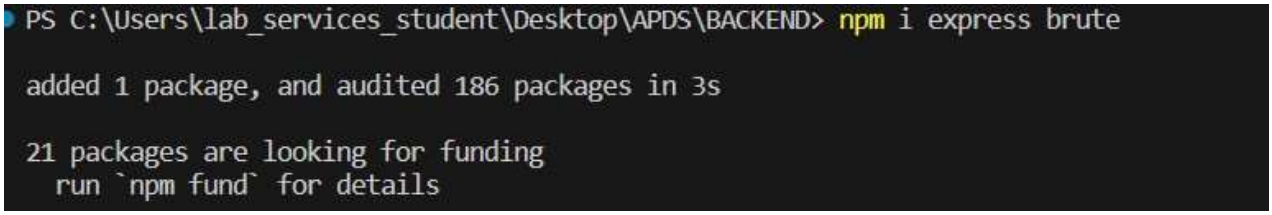




We can see its deleted since we cant find a record with the id anymore.

Moving on to our User Routes.
We can start by creating a user.mjs file in the routes folder,
And at the terminal we can install our required packages.





We are using:

Jsonwebtoken(JWT), a secure token authentication system.
Bcrypt, which is a powerful encryption tool.
Express Brute, for bruteforce prevention.

In creating out signup method, we use bcrypt to hash the password.

routes > JS user.mjs > ...

```
1   import express from "express";
2   import db from "../db/conn.mjs";
3   import { ObjectId } from "mongodb";
4   import bcrypt from "bcrypt";
5   import jwt from "jsonwebtoken";
6   import ExpressBrute  from "express-brute";
7
8   const router = express.Router();
9
10  var store = new ExpressBrute.MemoryStore();
11  var bruteforce = new ExpressBrute(store);
12
13  //sign up
14  router.post("/signup", async (req, res) => {
15      const password = bcrypt.hash(req.body.password,10)
16      let newDocument = {
17      name: req.body.name,
18      password: (await password).toString()
19      };
20      let collection = await db.collection("users");
21      let result = await collection.insertOne(newDocument);
22      console.log(password);
23      res.send(result).status(204);
24  });
```

We can also add our login method.

We using bruteforce. Prevent to prevent bruteforce attacks
Once a user is successfully logged in they get assigned a token encoded with their credentials.

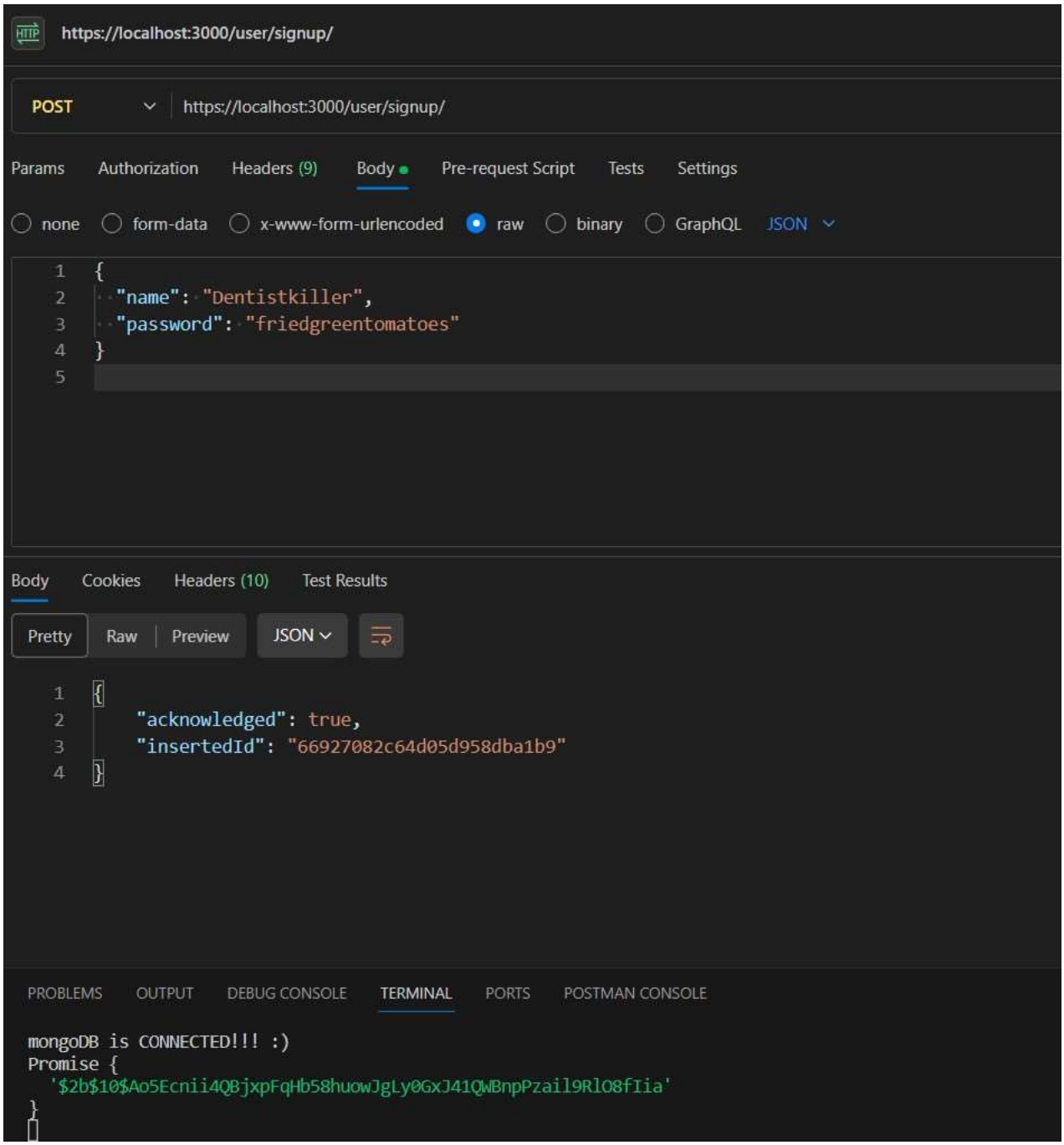
```
25
26  //login
27  router.post("/login",bruteforce.prevent, async (req, res) => {
28      const { name, password } = req.body;
29      console.log(name + " "+ password)
30
31      try {
32        const collection = await db.collection("users");
33        const user = await collection.findOne({ name });
34
35        if (!user) {
36          return res.status(401).json({ message: "Authentication failed" });
37        }
38
39        // Compare the provided password with the hashed password in the database
40        const passwordMatch = await bcrypt.compare(password, user.password);
41
42        if (!passwordMatch) {
43          return res.status(401).json({ message: "Authentication failed" });
44        }
45        else{
46          // Authentication successful
47        const token = jwt.sign({username:req.body.username,password:req.body.password},"this_secret_should_be_longer_than_it_is",{expiresIn:"1h"})
48        res.status(200).json({ message: "Authentication successful", token: token, name: req.body.name });
49        console.log("your new token is", token)
50        }
51      } catch (error) {
52        console.error("Login error:", error);
53        res.status(500).json({ message: "Login failed" });
54      }
55    });
56
57  export default router
```

We can now also update the server file to allow for user routes.

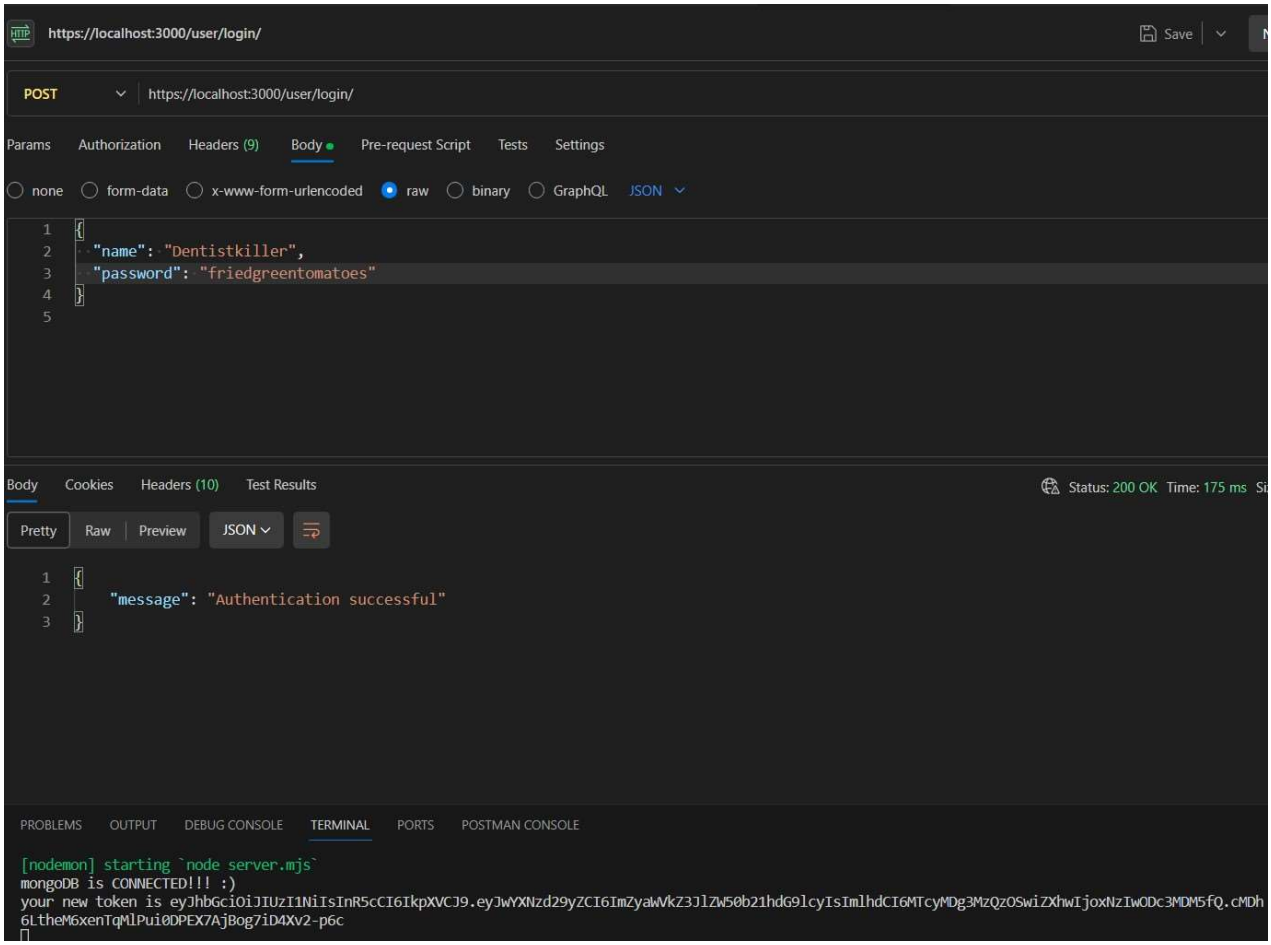
server.mjs > app.use() callback

```
import https from "https";
import http from "http";
import fs from "fs";
import fruits from "./routes/fruit.mjs";
import users from "./routes/user.mjs";
import express from "express"
import cors from "cors"

const PORT = 3000;
const app = express();

const options = {
  key: fs.readFileSync('keys/privatekey.pem'),
  cert: fs.readFileSync('keys/certificate.pem')
}

app.use(cors());
app.use(express.json());

app.use((reg,res,next)=>
{
    res.setHeader('Access-Control-Allow-Origin', '*');
    res.setHeader('Access-Control-Allow-Headers', '*');
    res.setHeader('Access-Control-Allow-Methods', '*');
    next();
})

app.use("/fruit", fruits);
app.route("/fruit", fruits);
app.use("/user", users);
app.route("/user", users);

let server = https.createServer(options,app)

server.listen(PORT);
```

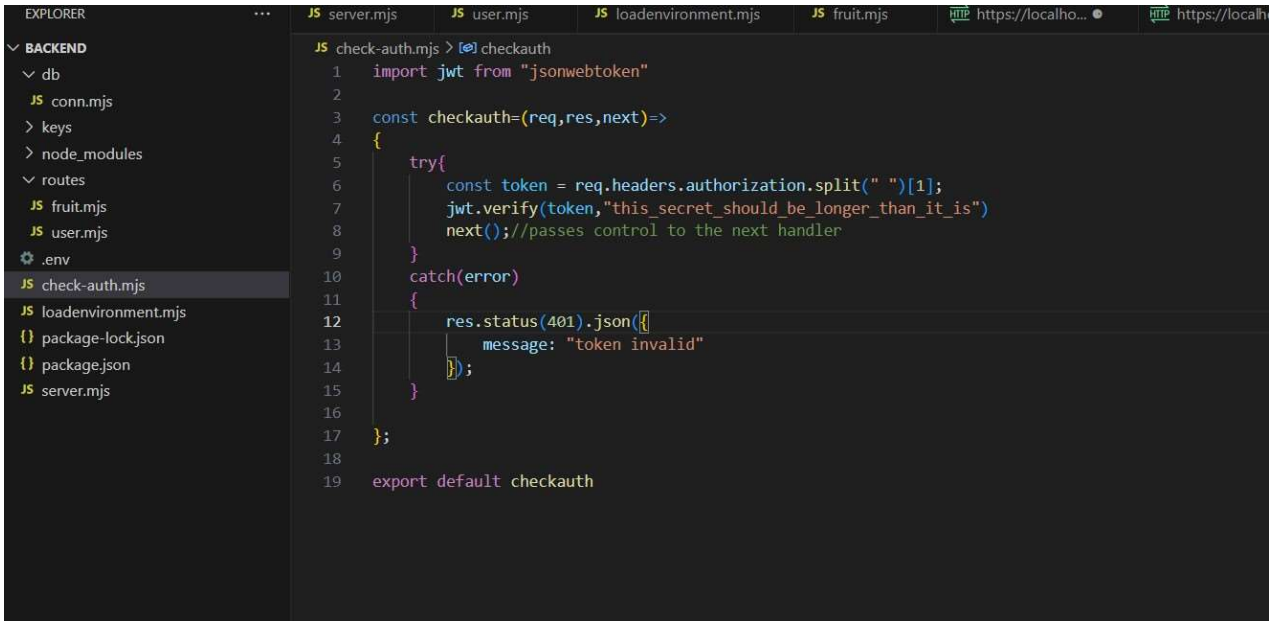
Testing our login and registration methods.



On successful login you can see the generated token.



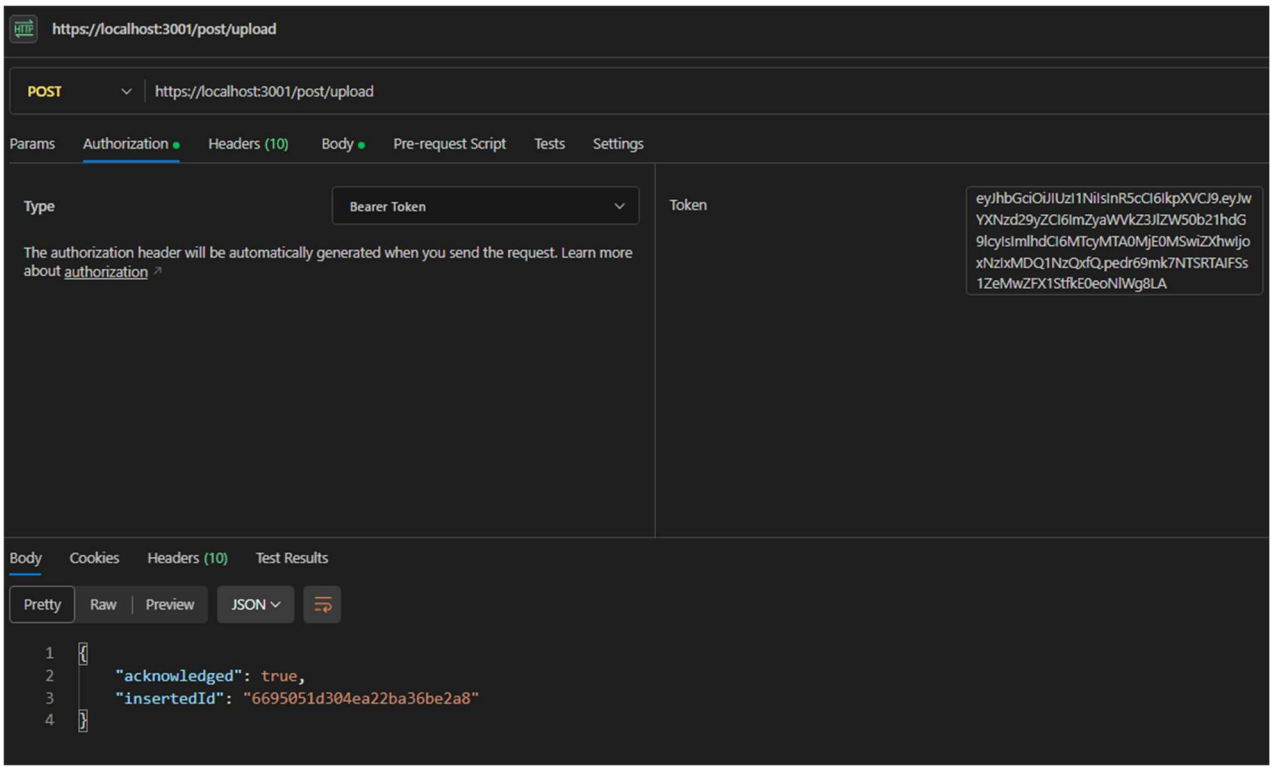
Now that we have our token we need to use it to verify if someone's session is valid, to do this we need to create a class to check if a user is authenticated.



We can now update our post.mjs routes to include the checkauth when a request is being made.

```
15  // Create a new record.
16  router.post("/upload", checkauth, async (req, res) => {
17      let newDocument = {
18        user: req.body.user,
19        content: req.body.content,
20        image: req.body.image
21      };
22      let collection = await db.collection("posts");
23      let result = await collection.insertOne(newDocument);
24      res.send(result).status(204);
25  });
26
27  //Update a record by id
28  router.patch("/:id", checkauth, async (req, res) => {
29    const query = { _id: new ObjectId(req.params.id) };
30    const updates =  {
31      $set: {
32        name: req.body.name,
33        comment: req.body.comment
34      }
35    };
36
37    let collection = await db.collection("posts");
38    let result = await collection.updateOne(query, updates);
39
40    res.send(result).status(200);
41  });
42
```

Now to test in postman we need to include the token generated from logging as a header in your request, we select authorization -> Bearer token.



You should now have a fully functioning backend application.

3 Frontend

3.1 First React Application

The frontend development we use is React – the “R” from MERN
. https://react.dev/ [Accessed 13 July 2024].

Learn more by following their docs. https://react.dev/learn/start-a-new-react-project
[Accessed 13 July 2024].

Introduction to React

React is a widely adopted JavaScript library developed by Facebook for creating user interfaces, especially for single-page applications. It enables developers to build interactive and efficient UIs through a component-based architecture. Here is a detailed overview of the fundamental concepts of React:

1 Components

Components are the fundamental building blocks of a React application. They are self-contained units that define parts of the user interface. React supports two main types of components:

Functional Components: These are JavaScript functions that accept inputs (known as props) and return React elements. They are generally used for components that do not require internal state management or lifecycle methods.

Class Components: These are ES6 classes that extend React.Component and include a render method which returns React elements. Class components can manage internal state and utilise lifecycle methods to handle various stages of a component's existence.

2 JSX (JavaScript XML)

JSX is a syntax extension for JavaScript that allows developers to write HTML-like structures within JavaScript code. JSX enhances readability and writing efficiency, as it merges the concerns of HTML and JavaScript into a single, coherent structure. This syntax is compiled into standard JavaScript function calls that describe the user interface.

3 Props

Props, short for "properties," are immutable attributes used to pass data and event handlers from parent components to child components. Props enable dynamic rendering of components based on the data provided and facilitate component reusability and composition.

4 State

State represents internal data that affects how a component behaves and renders. Unlike props, state is mutable and managed within the component itself. It can be updated in response to user actions or other events, which triggers re-rendering to reflect changes in the user interface.

5 Lifecycle Methods

Lifecycle methods are special functions in class components that are invoked at various stages of a component’s lifecycle. These methods allow developers to execute code at specific points, such as when a component mounts, updates, or unmounts, enabling the management of side effects and component behaviour.

6 Hooks

Hooks are functions introduced in React 16.8 that enable functional components to use state and lifecycle features, which were previously only available in class components. Essential hooks include useState for managing state, useEffect for handling side effects, and useContext for accessing context values.

7 Context API

The Context API is a feature that allows the sharing of global data across the component tree without the need to pass props manually through every level. It is used for managing global state or settings, such as user authentication status or application theme.

8 React Router

React Router is a library used for implementing routing in React applications. It facilitates the definition of different routes and navigation between components or pages, enabling the creation of complex, multi-view applications.

9 Virtual DOM

The Virtual DOM is a lightweight, in-memory representation of the actual DOM. React uses the Virtual DOM to efficiently manage and apply updates to the real DOM by comparing the current and previous states and only making necessary changes. This approach enhances performance and ensures that updates are executed in the most efficient manner.

https://www.w3schools.com/react/react_intro.asp [Accessed 10 July 2024]

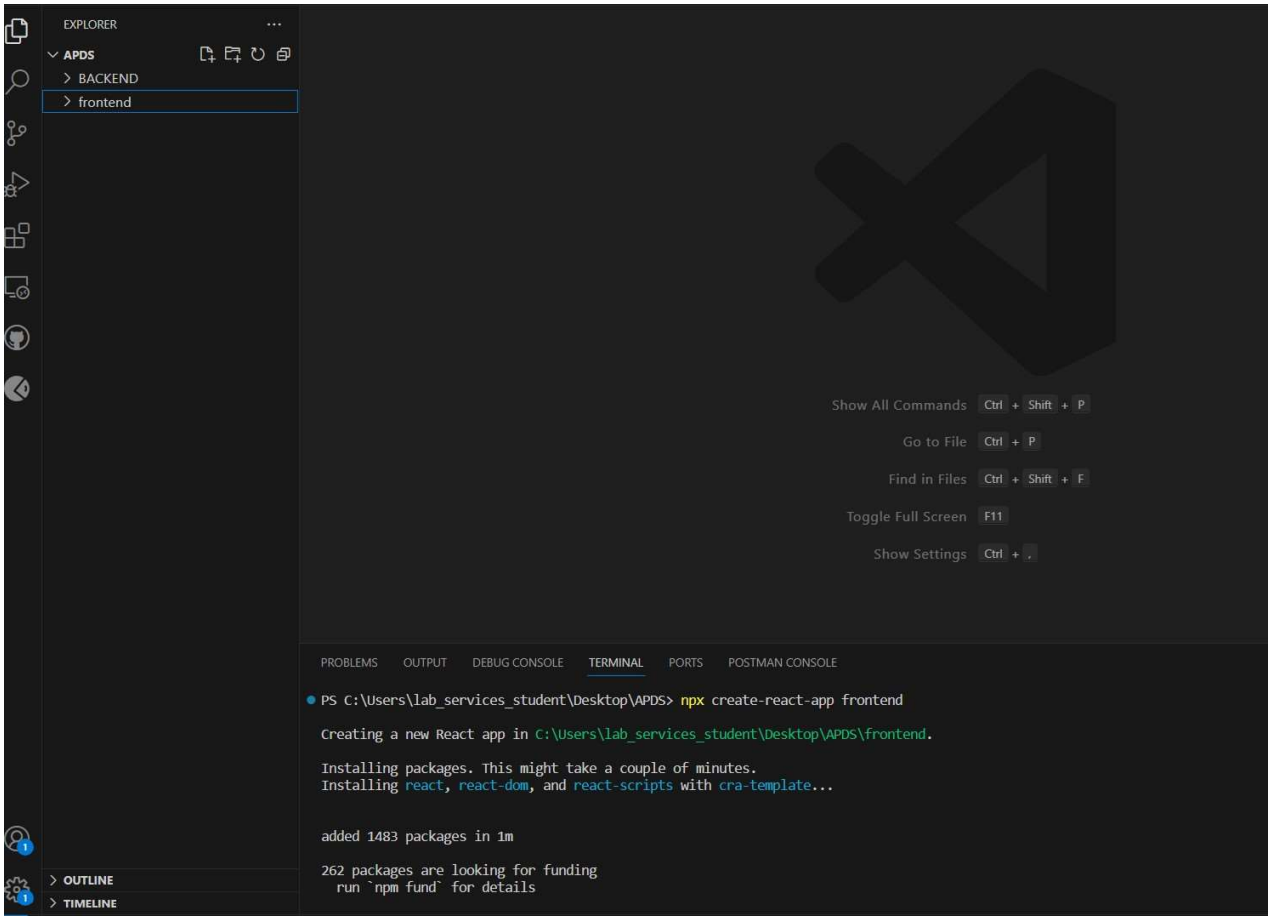
Run the following command inside a terminal to create a new application/project called “frontend”.

Note you must be in a folder that is generally accessible to you, I created an APDS folder containing both frontend and backend folders.

- npx create-react-app frontend

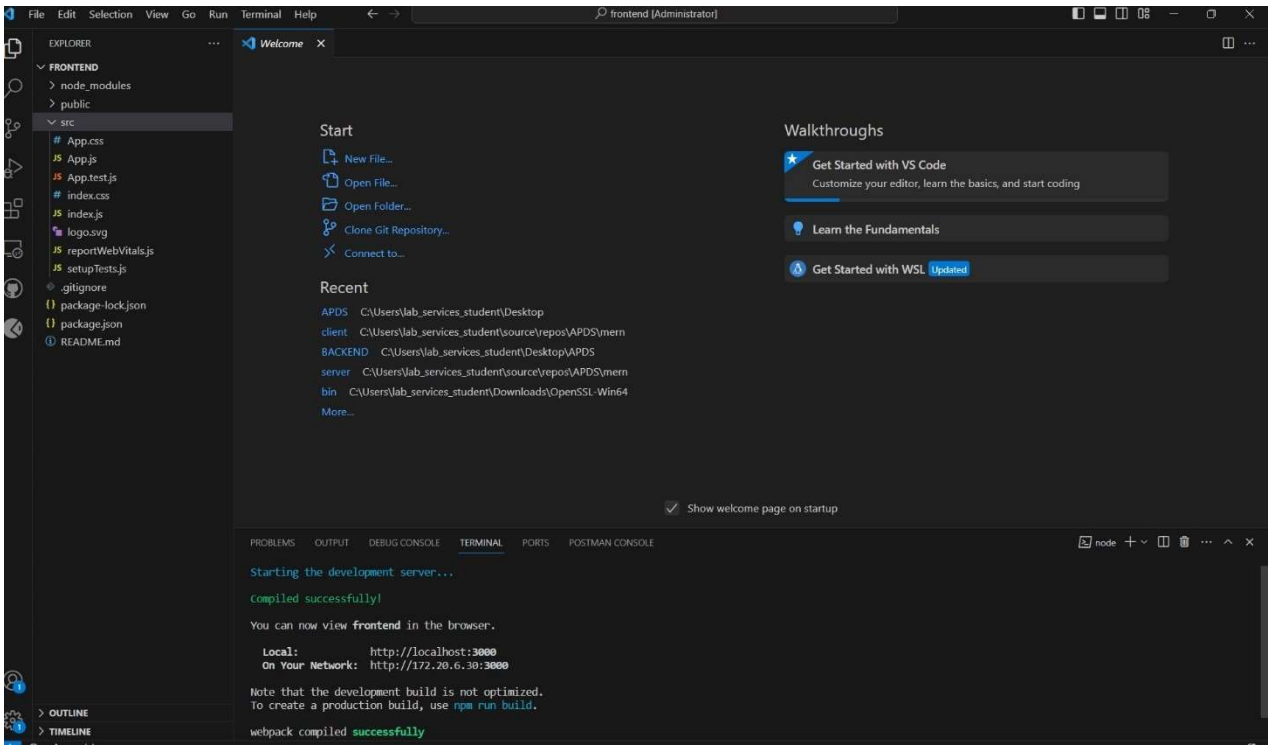
This creates a “barebones” application for your project in the “frontend” folder and installs the required packages in a folder called “node_modules”. Accept the default for any questions.

NOTE node_modules can be safely deleted and recreated – useful to reduce the size of the project for copying and moving.

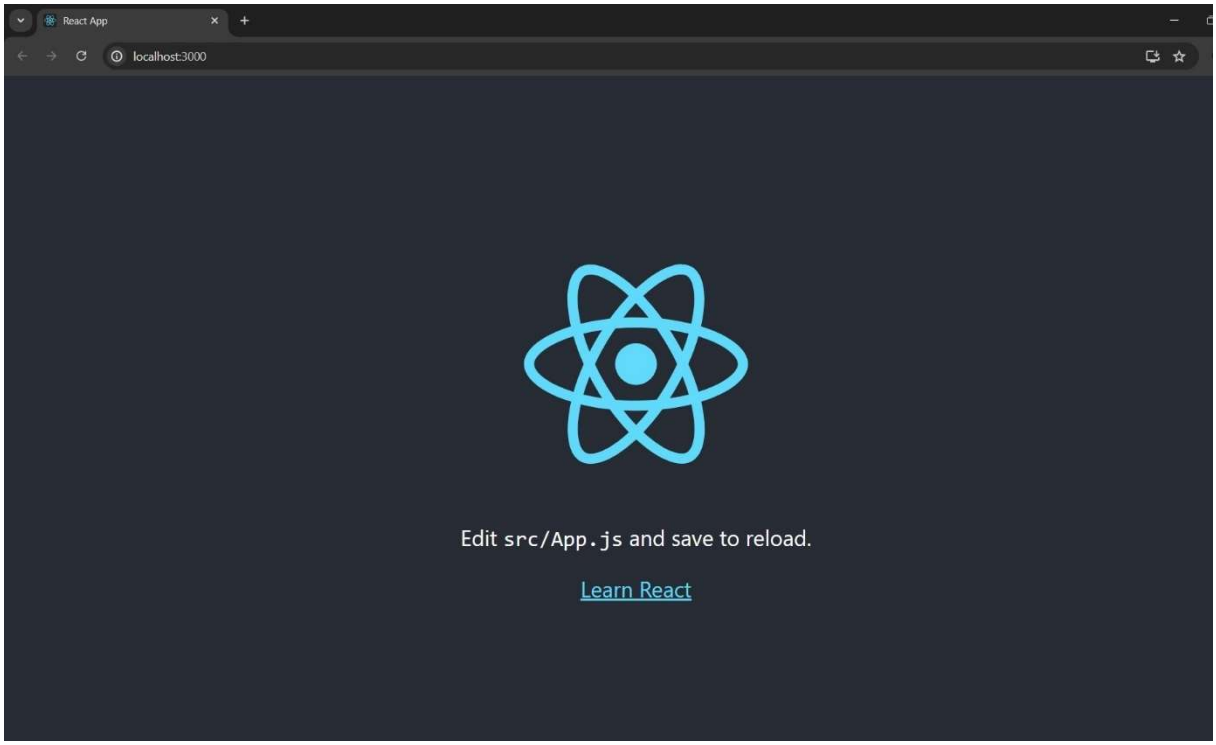


Once the folder is created, we can open the folder in Visual Studio.

We can navigate to the frontend colder in the terminal by using “cd frontend” and then we run “npm start” to run our react app.

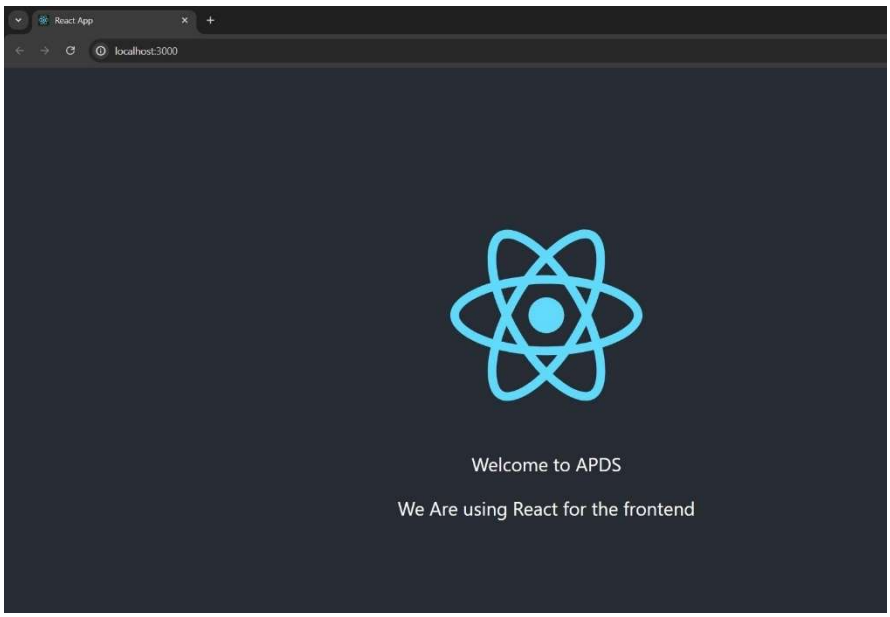


What our browser will look like.



Now let's edit our app.js to display something else.

```
src > App.js > ...
1   import logo from './logo.svg';
2   import './App.css';
3
4   function App() {
5     return (
6       <div className="App">
7         <header className="App-header">
8           <img src={logo} className="App-logo" alt="logo" />
9           <p>
10            Welcome to APDS
11          </p>
12          <a>
13            We Are using React for the frontend
14          </a>
15        </header>
16      </div>
17    );
18  }
19
20  export default App;
21
```

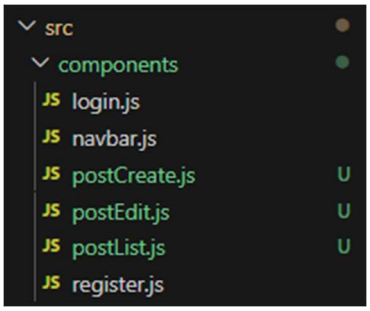


Run the following command to install the router, this will connect the backend api to the front

end.

```
PS C:\Users\lab_services_student\Desktop\APDS\frontend> npm i react-router-dom

added 3 packages, and audited 1549 packages in 5s

262 packages are looking for funding
  run `npm fund` for details

8 vulnerabilities (2 moderate, 6 high)

To address all issues (including breaking changes), run:
  npm audit fix --force

Run `npm audit` for details.
```

Now we can start creating our frontend by creating our components.
We will create a folder and within that create files for all our components.

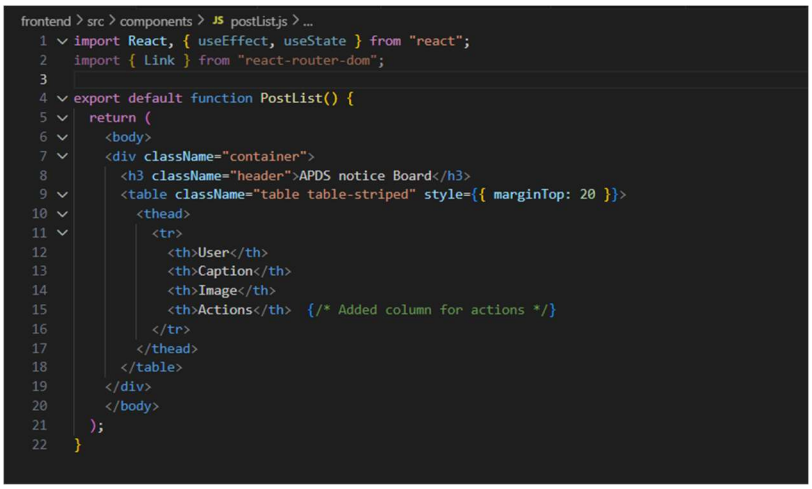


The provided React code sets up routing for the application, including paths for viewing the post list, editing a post, creating a new post , user registration, and user login, along with a navbar that is displayed on all pages.

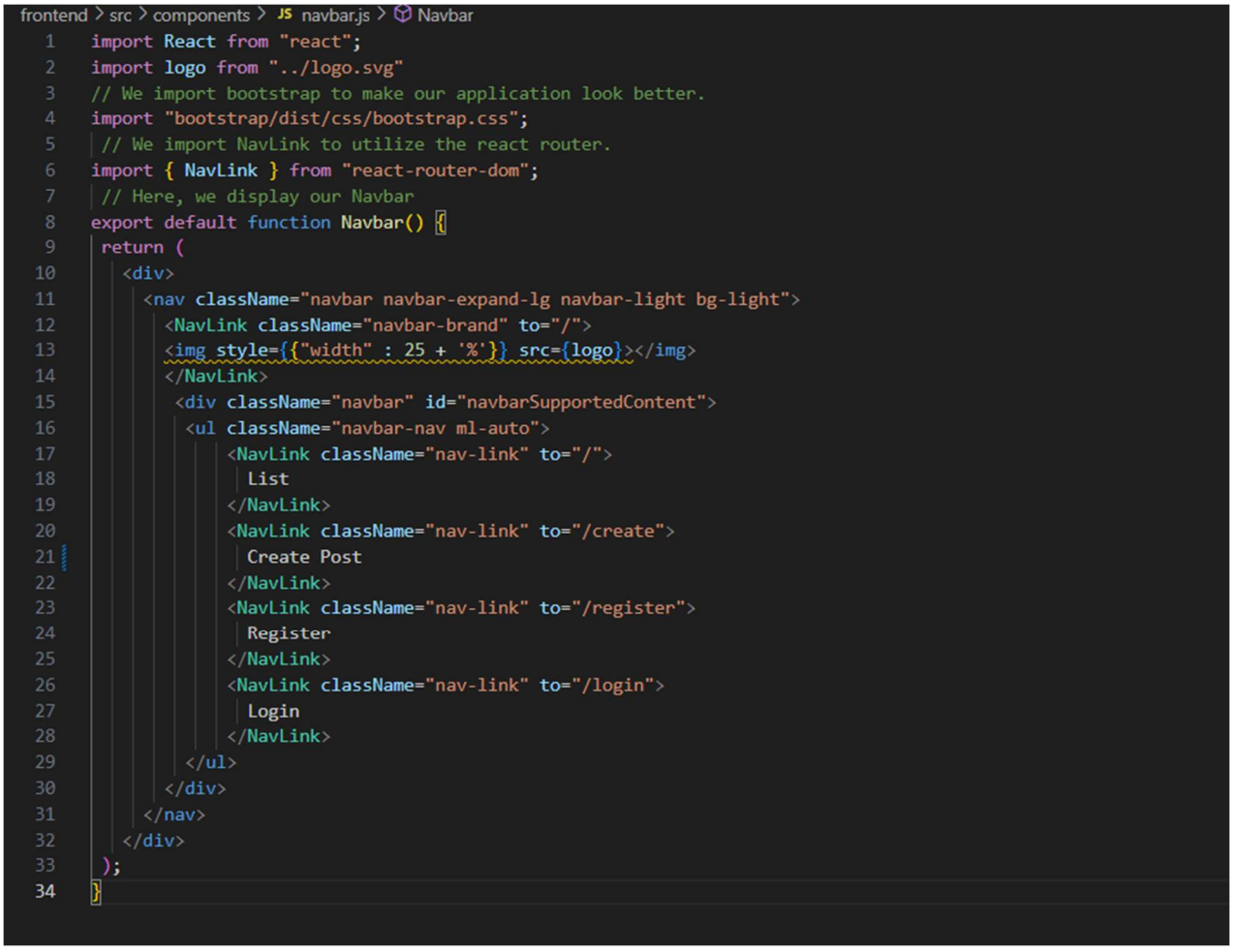
frontend > src > JS App.js > App

```
import './App.css';
import React from "react";
// We use Route in order to define the different routes of our application
import { Route, Routes } from "react-router-dom";
// We import all the components we need in our app
import Navbar from "./components/navbar";
import PostList from "./components/postList";
import EditPost from "./components/postEdit";
import CreatePost from "./components/postCreate";
import Register from "./components/register";
import Login from "./components/login";

const App = () => {
  return (
    <div>
      <Navbar />
      <Routes>
        <Route exact path="/" element={<PostList />} />
        <Route path="/edit/:id" element={<EditPost />} />
        <Route path="/create" element={<CreatePost />} />
        <Route path="/register" element={<Register />} />
        <Route path="/login" element={<Login />} />
      </Routes>
    </div>
  );
};

export default App;
```

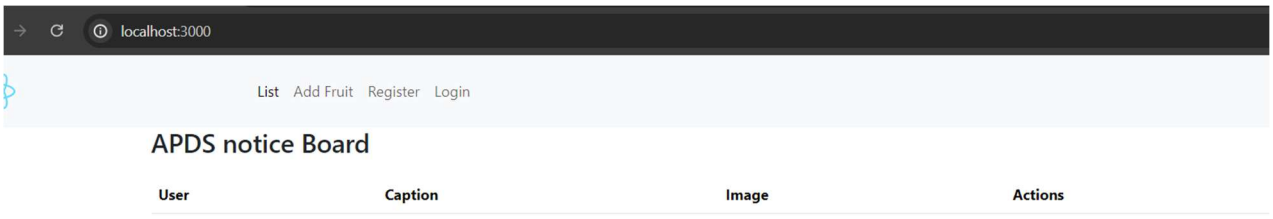
We can now update our defaults “/” page with is the postList.js
This function returns the HTML used to design the UI, creating the display.



The navbar.js also needs to be updated to allow navigation between the different pages.



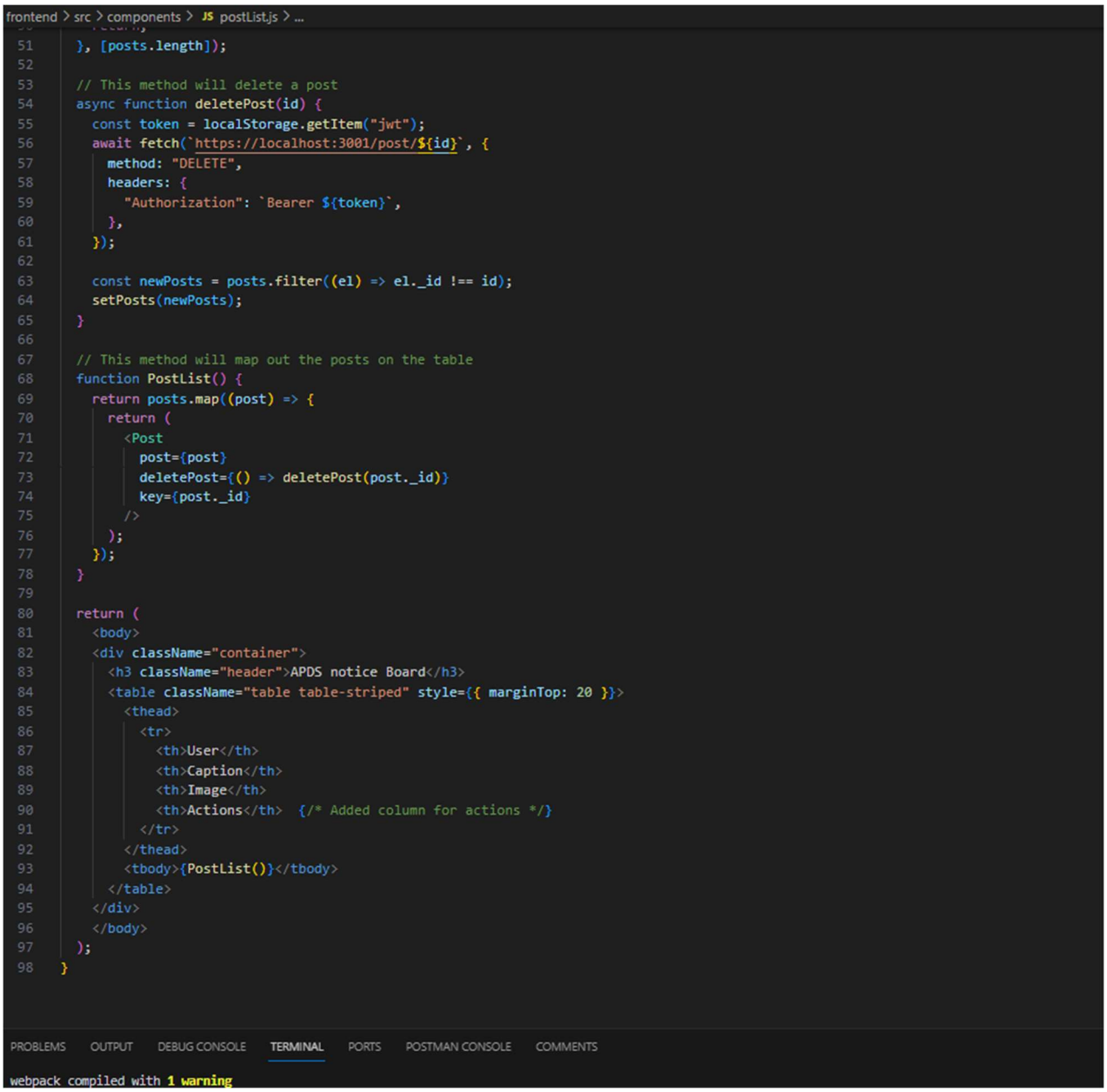
This is what the app should look like at this point.



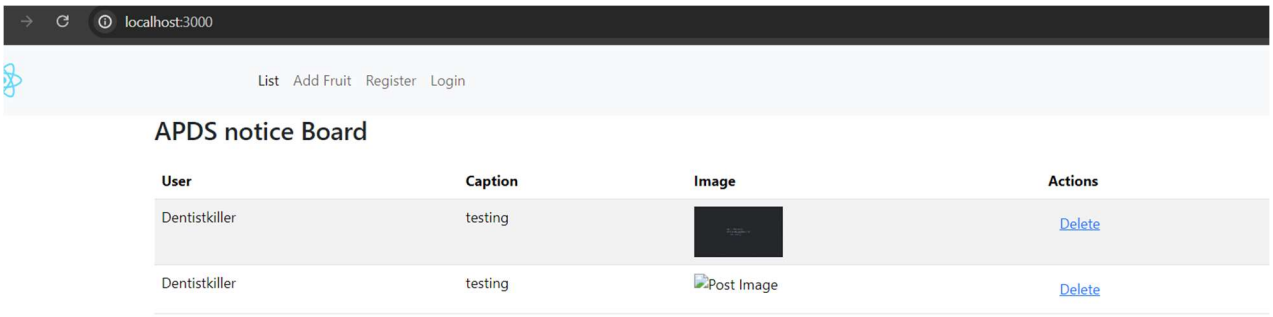
Now to add functionality to this design.
This code fetches a list of posts from a database and displays them in a table with an option to delete each post, using hooks `useEffect` for data fetching and `useState` for state management.

frontend > src > components > JS postList.js > ...

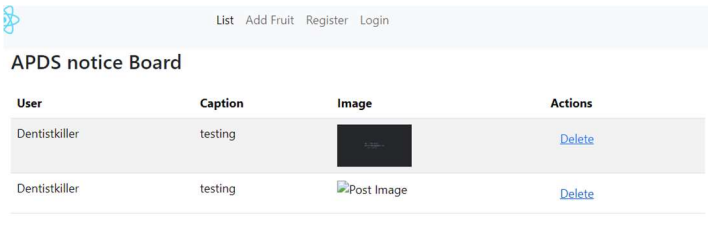
```jsx
import React, { useEffect, useState } from "react";
import { Link } from "react-router-dom";
import '../App.css';

const Post = (props) => (
  <tr>
      <td>{props.post.user}</td>
    <td>{props.post.content}</td>
    <td>
      {props.post.image && (
        <img
          src={`${props.post.image}`}  // Convert base64 string to image
          alt="Post Image"
          style={{ maxWidth: '100px', maxHeight: '100px', objectFit: 'cover' }}  // Ensure the image fits within the size limits
        />
      )}
    </td>
    <td>
      <button className="btn btn-link"
        onClick={() => {
          props.deletePost(props.post._id);
        }}
      >
        Delete
      </button>
    </td>
  </tr>
);

export default function PostList() {
  const [posts, setPosts] = useState([]);

  // This method fetches the posts from the database.
  useEffect(() => {
    async function getPosts() {
      const response = await fetch(`https://localhost:3001/post/`);

      if (!response.ok) {
        const message = `An error occurred: ${response.statusText}`;
        window.alert(message);
        return;
      }

      const posts = await response.json();
      setPosts(posts);
    }

    getPosts();

    return;
  }, [posts.length]);
```



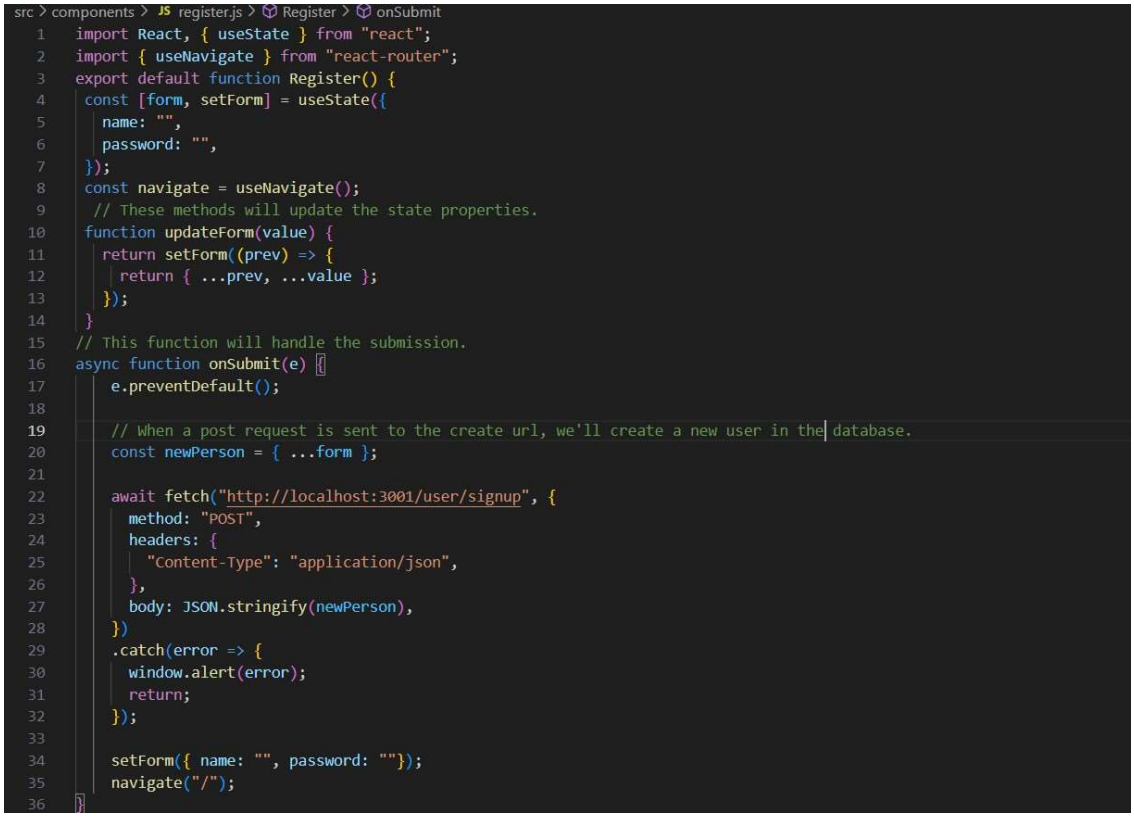
When accessing the website you can now see the site update with data from the database.



If a delete is attempted, since the user is not logged in, it will return as unauthorized.
We use the developer tools in your selected browser to help us test in React.



In order to authorize a user, we need to start with updating the signup.js file.



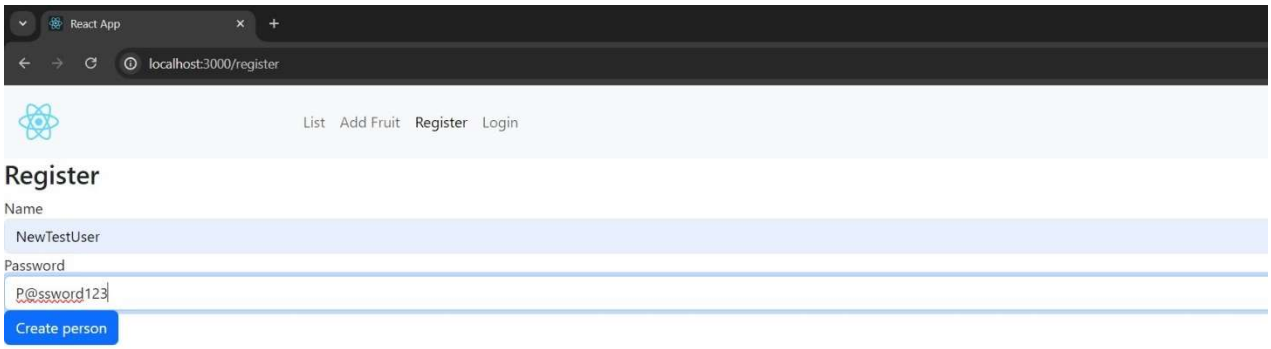

```
  // This following section will display the form that takes the input from the user.
  return (
    <div>
      <h3>Register</h3>
      <form onSubmit={onSubmit}>
        <div className="form-group">
          <label htmlFor="name">Name</label>
          <input
            type="text"
            className="form-control"
            id="name"
            value={form.name}
            onChange={(e) => updateForm({ name: e.target.value })}
          />
        </div>
        <div className="form-group">
          <label htmlFor="password">Password</label>
          <input
            type="text"
            className="form-control"
            id="password"
            value={form.position}
            onChange={(e) => updateForm({ position: e.target.value })}
          />
        </div>

        <div className="form-group">
          <input
            type="submit"
            value="Create person"
            className="btn btn-primary"
          />
        </div>
      </form>
    </div>
  );
}
```


We will also need to update the login.js file.

frontend > src > components > JS login.js > ...

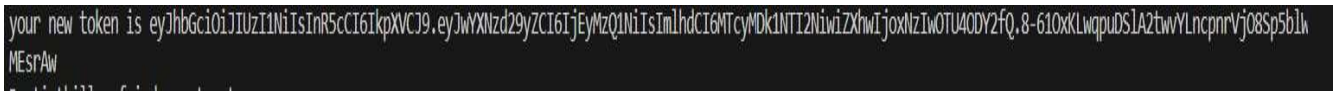
```
import React, { useState } from "react";
import { useNavigate } from "react-router";
export default function Login() {
 const [form, setForm] = useState({
   name: "",
   password: "",
 });
 const navigate = useNavigate();

 function updateForm(value) {
   return setForm((prev) => {
     return { ...prev, ...value };
   });
 }
// This function will handle the submission.
async function onSubmit(e) {
    e.preventDefault();

    // Passes the form data to the API on our backend.
    const newPerson = { ...form };

    const response = await fetch("https://localhost:3001/user/login", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(newPerson),
    })
    .catch(error => {
      window.alert(error);
      return;
    });

    const data = await response.json();
    const { token, name } = data;
    console.log(name + " " + token)

    // Save the JWT to localStorage
    localStorage.setItem("jwt", token);

    // Optionally, save the username if needed
    localStorage.setItem("name", name);

    setForm({ name: "", password: ""});
    navigate("/");
}
  // HTML for the login page.
 return (
   <div>
     <h3>Login</h3>
     <form onSubmit={onSubmit}>
       <div className="form-group">
         <label htmlFor="name">Name</label>
         <input
           type="text"
           className="form-control"
           id="name"
           value={form.name}
           onChange={(e) => updateForm({ name: e.target.value })}
         />
       </div>
       <div className="form-group">
         <label htmlFor="password">Password</label>
         <input
           type="text"
           className="form-control"
           id="password"
           value={form.password}
           onChange={(e) => updateForm({ password: e.target.value })}
         />
       </div>

       <div className="form-group">
         <input
           type="submit"
           value="Login"
           className="btn btn-primary"
         />
       </div>
     </form>
   </div>
 );
}
```

Our login and registration should now look something like this.



A successful registration should display in the developer console.

When you login it should generate the token. To ensure the token is generated check the log in the sever after pressing login.



Now on to creating the posts.
Update postCreate.js like so.

Notice how in the response header we now have to include the token in order to perform the checkauth function and validate whether the user is authenticated.

```
return (
  <div className="container">
    <h3 className="header">Create New Post</h3>
    <form onSubmit={onSubmit} className="form">
      <div className="form-group">
        <label htmlFor="user">User</label>
        <input
          type="text"
          className="form-control"
          id="user"
          value={form.user}
          disabled  // Make the user field read-only
        />
      </div>
      <div className="form-group">
        <label htmlFor="content">Content</label>
        <input
          type="text"
          className="form-control"
          id="content"
          value={form.content}
          onChange={(e) => updateForm({ content: e.target.value })}
        />
      </div>
      <div className="form-group">
        <label htmlFor="image">Image</label>
        <input
          type="file"
          className="form-control"
          id="image"
          accept="image/*"
          onChange={handleImageChange}
        />
      </div>
      <div className="form-group">
        <input
          type="submit"
          value="Create Post"
          className="btn btn-primary"
        />
      </div>
    </form>
  </div>
);
```

```
frontend > src > components > JS postCreate.js > CreatePost
import React, { useState, useEffect } from "react";
import { useNavigate } from "react-router-dom";
import '../App.css';

export default function CreatePost() {
  const [form, setForm] = useState({
    user: "",
    content: "",
    image: "",
  });
  const navigate = useNavigate();

  // Retrieve the user from localStorage when the component mounts
  useEffect(() => {
    const savedUser = localStorage.getItem("name");  // Make sure the user is saved in localStorage as a string
    if (savedUser) {
      setForm((prev) => ({
        ...prev,
        user: savedUser,
      }));
    } else {
      // Redirect to login or another page if there is no user
      navigate("/login");  // Redirect to login if user data is missing
    }
  }, [navigate]);

  function updateForm(value) {
    return setForm((prev) => {
      return { ...prev, ...value };
    });
  }

  // Function to handle image file change
  async function handleImageChange(e) {
    const file = e.target.files[0];
    if (file) {
      try {
        const reader = new FileReader();
        reader.onloadend = () => {
          const base64String = reader.result.split(",")[1]; // Remove the '' part
          updateForm({ image: base64String });
        };
        reader.readAsDataURL(file);
      } catch (error) {
        window.alert("Error reading image file.");
      }
    }
  }

  // Function to handle form submission
  async function onSubmit(e) {
    e.preventDefault();

    const token = localStorage.getItem("jwt");

    const newPost = {
      user: form.user,
      content: form.content,
      image: form.image
    };

    try {
      const response = await fetch("https://localhost:3001/post/upload", {  // Changed to http for local development
        method: "POST",
        headers: {
          "Content-Type": "application/json",
          "Authorization": `Bearer ${token}`,
        },
        body: JSON.stringify(newPost),
      });

      if (!response.ok) {
        throw new Error("Network response was not ok");
      }

      const result = await response.json();
      console.log("Post created:", result);
      //setForm({ user: form.user, content: "", image: "" });  // Reset form but keep user
      navigate("/");
    } catch (error) {
      window.alert(error);
    }
  }

```

4 Further Development

- The user interface in this guide has been left deliberately simple to teach you the core concepts. Explore ways to enhance this by using alternate HTML, CSS and relook at Angular Material https://react.dev/learn/describing-the-ui [Accessed 11 July 2024].
- Use Regex to help setting and maintaining a strong password policy. https://www.w3schools.com/js/js_regexp.asp [Accessed 11 July 2024].
- Clean up your debugging console.log commands or wrap these in a method in which they can be turned on or off.
- Have a look at the DOM sanitizer to sanitize (secure) your output https://angular.io/guide/security Have you considered the other aspects in this link? [Accessed 11 July 2024.]
- Do not re-invent the wheel! Research additional packages to assist in security e.g., Prevent brute force attacks https://www.npmjs.com/package/express-brute [Accessed 11 July 2024].
- General express security https://www.npmjs.com/package/helmet [Accessed 11 July 2024].
- Server logging https://www.npmjs.com/package/morgan [Accessed 11 July 2024].