Ollscoil Teicneolaíochta an Atlantaigh

Atlantic Technological University

Focus Up

Project Engineering

Year 4

Richard Stephen (G00382880)

Bachelor of Engineering (Honours) in Software and Electronic Engineering

Atlantic Technological University

2022/2023

Declaration

This project is presented in partial fulfilment of the requirements for the degree of Bachelor of Engineering (Honours) in Software and Electronic Engineering at Galway-Mayo Institute of Technology.

This project is my own work, except where otherwise accredited. Where the work of others has been used or incorporated during this project, this is acknowledged and referenced.

Richard Stephen.

Acknowledgements

I would like to thank my supervisor, Michelle Lynch for her support from the start to the end of this project. I would also like to thank my classmates who have given me assistance and suggestions in relation to my project, as well as acting as teammates.

Table of Contents

1 Summary 6

2 Poster 7

2 Introduction 8

3 Technologies and Tools 9

4 Project Architecture 11

5 Project Plan 12

6 Frontend 14

7.1 Structure [5] 14

7.2 Login 15

7.3 Homepage 16

7.4 Buttons 17

7.5 Pomodoro Timer 18

7.6 Note Taking 24

7.7 To-Do-list. [10] 32

8 Backend [11] 36

8.1 Structure 36

8.2 Schemas 38

8.3 Requests and Routing 39

8.3.1 Notes 40

8.3.2 Timer 43

8.3.2 To-do List 44

8.3.3 User Authentication [13] 46

7 Ethics 51

10 Conclusion 52

11 References 53

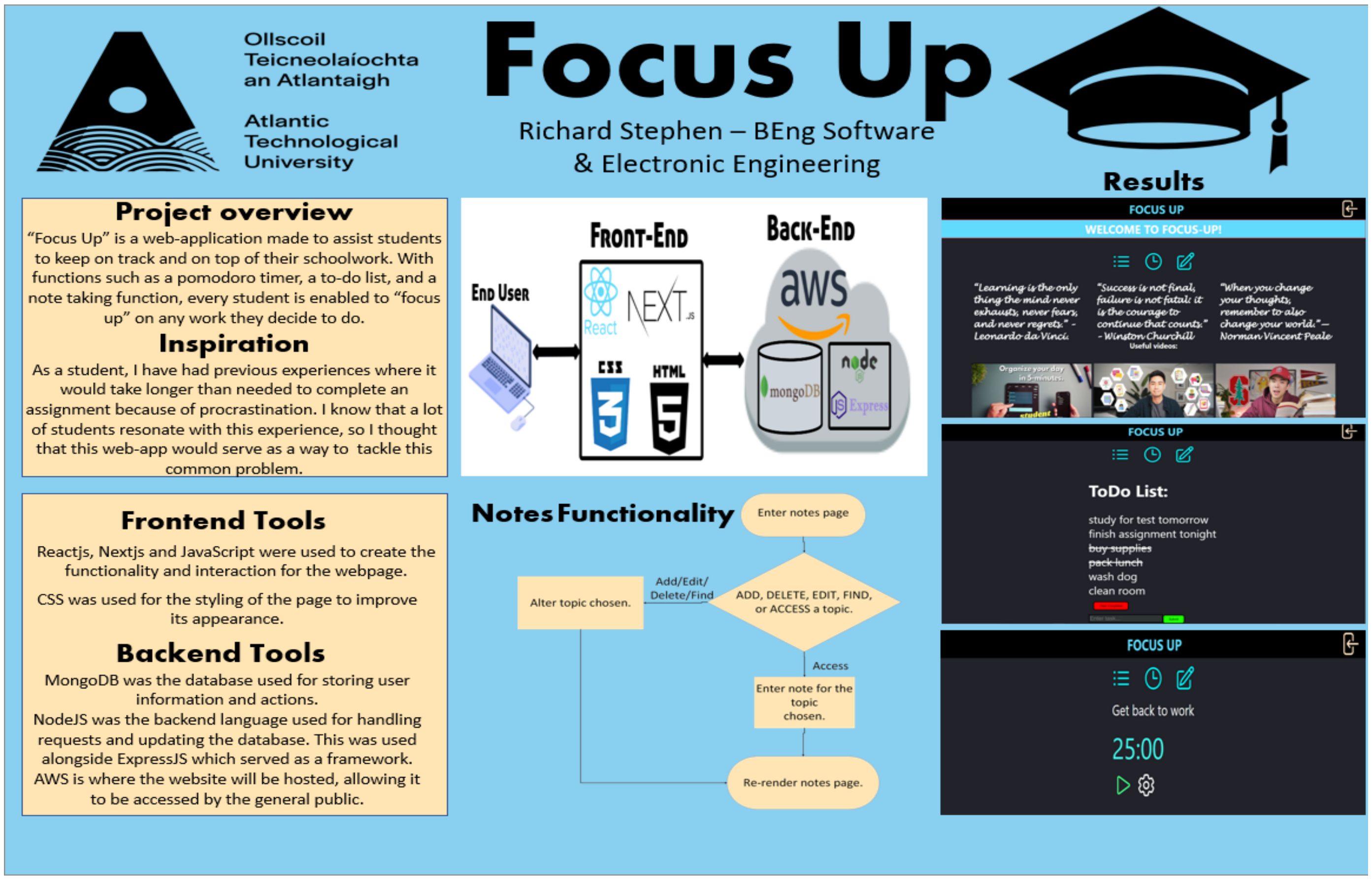
1 Summary

In this modern day, there’s more technology than ever, which is great for human advancement, but in the context of college, it means there is more opportunities for more distractions. I am a victim of said distractions. I often find myself losing track of what I am working on and in turn, end up procrastinating my work. This unnecessarily increases the amount of time taken to complete an assignment, or study for an exam. This is a common problem among students also. There are many web applications and mobile applications that aim to assist students with tackling this problem. One I have used, and continue to use, Remnote, is a great example of one of these applications, and is in fact my inspiration for this project. I wanted to try my own spin on this type of web application as it hits very close to home about my procrastination issue.

I have named my project “Focus Up”, which is a web application aimed at students. The goal of this project is to create a web app that would assist them in their daily studies. It includes features like a pomodoro timer, a to-do list, and a note taking function. The project was developed using the MERN stack. React and Nextjs were used for my frontend code, I used MongoDB for my database storage, and used express and NodeJS to create my backend code. For version control I used GitHub, where I created branches and commits, ensuring that the progress of my project was saved after every significant change made. I used JIRA for project management and scheduling, as well as OneNote, which was used for creating my weekly individual as well as teamwork logs to track my progress.

This project has greatly enhanced my software engineering skills through learning new languages, as well as my management and planning skills through learning how to use the JIRA software. It has also improved my researching skills, as I had to learn any new languages from scratch, like React and Node. This project has also improved my knowledge of the web application development process.

2 Poster



Ollscoil Teicneolaíochta an Atlantaigh

Atlantic Technological University

Focus Up

Richard Stephen – BEng Software & Electronic Engineering

Project overview

"Focus Up" is a web-application made to assist students to keep on track and on top of their schoolwork. With functions such as a pomodoro timer, a to-do list, and a note taking function, every student is enabled to "focus up" on any work they decide to do.

Inspiration

As a student, I have had previous experiences where it would take longer than needed to complete an assignment because of procrastination. I know that a lot of students resonate with this experience, so I thought that this web-app would serve as a way to tackle this common problem.

Frontend Tools

Reactjs, Nextjs and JavaScript were used to create the functionality and interaction for the webpage.

CSS was used for the styling of the page to improve its appearance.

Backend Tools

MongoDB was the database used for storing user information and actions.

NodeJS was the backend language used for handling requests and updating the database. This was used alongside ExpressJS which served as a framework.

AWS is where the website will be hosted, allowing it to be accessed by the general public.

Notes Functionality

Results

FOCUS UP

WELCOME TO FOCUS-UP!

"Learning is the only thing the mind never exhausts, never fears, and never regrets." - Leonardo da Vinci.

"Success is not final, failure is not fatal: it is the courage to continue that counts." - Winston Churchill

"When you change your thoughts, remember to also change your world." — Norman Vincent Peale

Useful videos:

FOCUS UP

ToDo List:

study for test tomorrow
finish assignment tonight
~~buy supplies~~
~~pack lunch~~
wash dog
clean room

FOCUS UP

Get back to work

25:00

Figure 1 – Project Poster

2 Introduction

The primary goal of this project was to develop a web application that would serve as an additional outreach to students who need a convenient app that cover a number of functionalities at once. A project like this felt like it made sense to me as I personally have a problem with procrastinating and losing focus because of the method in which I study. I felt that going from tab to tab to find different features for study opens a small window for a distraction to appear in a search, because currently with the advancement of technology and targeted ads, these distractions become more common. I designed this app with the mind that the user would not need to leave the realms of the webpage to tackle any task they have on hand.

3 Technologies and Tools

3.1 ReactJs

React is a JavaScript frontend framework created by Facebook. It is used to create web applications on computers and laptops. React is a rapidly growing framework that is mostly preferred over plain JavaScript given its ability to support larger scale projects with several components, whereas plain JS would be more suited to very small and simple websites.

3.2 NextJS

NextJS is a lightweight framework for React applications. It allows you to build a React app that uses server-side rendering to store content in advance on the server. This means that users will meet with a pre-rendered webpage, ensuring that they can access the site at a faster speed than if the app was made with just React. Nextjs is still based on React, but simply improves upon it. [1]

3.3 NodeJS

Nodejs is an open-source, cross-platform JavaScript runtime environment and library for running web applications outside the client's browser. I chose NodeJS as my backend language as it is complimenting the MongoDB database, making it an easier process than if I were to use SQL. The code being in JS also makes it easier for developers to understand, as a lot of them are familiar with the JS language already. [2]

3.4 MongoDB

MongoDB is a document-oriented NoSQL database used for high volume data storage (NoSQL meaning that it doesn't use SQL like relational databases). Instead of using tables and rows like in relational databases, MongoDB uses collections and documents. If one should use Nodejs as backend code in their project, it would be wise to use a MongoDB database alongside it as I have previously stated that these two technologies complement each other very well. [3]

3.5 ExpressJS

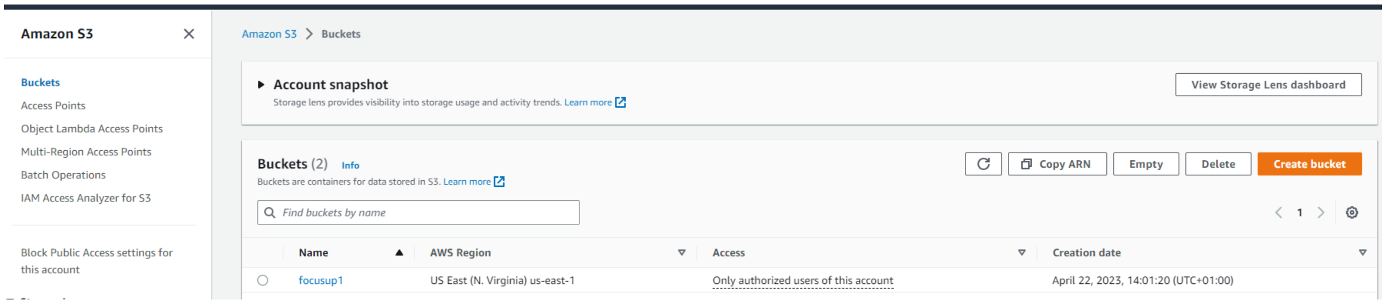
ExpressJS is a Nodejs framework designed for web application quickly and easily, by taking away most of the setup you would need in plain NodeJS, for example, routing. By simply using one of 5 verbs (GET, POST, DELETE, PATCH, PUT) and attaching a URL route, you can set up routing and requests easily and efficiently. As well as that, using express middleware allows you to make changes to request and response objects, giving you more freedom. [4]

3.6 GitHub

GitHub is a cloud-based git repository service used for version control. I used this tool to keep my project up to date as well as saving my progress using branches, commits, and pull requests.

3.7 AWS

AWS stands for Amazon Web Service, which allows for the use of hosting your website on the AWS cloud. This allows the user to run the website with only the ip assigned to its "Instance" [5], which is a server resource provided by third party cloud services. Using an S3 bucket, the user can upload their entire project on the cloud to be hosted. Then, by setting up an EC2 instance, their bucket is assigned an ip address which they can then use as a URL to their website rather than using localhost 3000.


Amazon S3
Buckets
Access Points
Object Lambda Access Points
Multi-Region Access Points
Batch Operations
IAM Access Analyzer for S3
Block Public Access settings for this account
Amazon S3 > Buckets
Account snapshot
View Storage Lens dashboard
Copy ARN
Empty
Delete
Create bucket
Find buckets by name

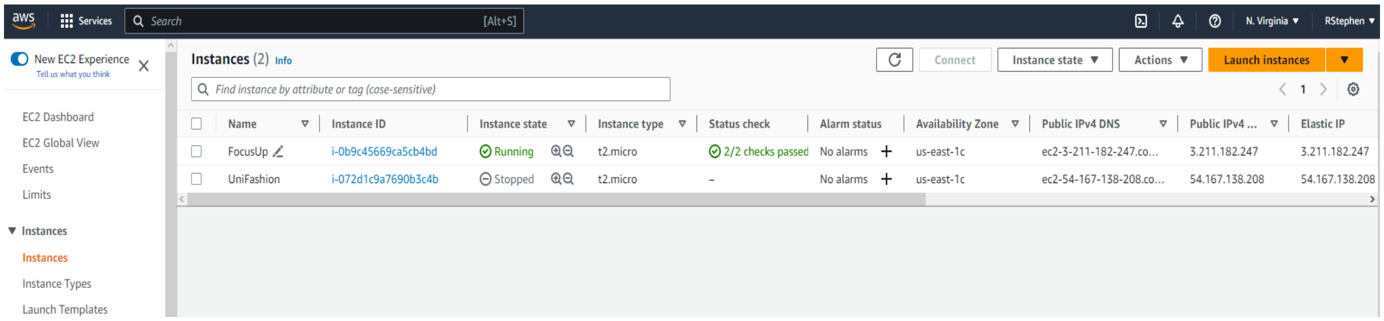
Storage lens provides visibility into storage usage and activity trends. Learn more

Buckets (2) Info

Buckets are containers for data stored in S3. Learn more

Name	AWS Region	Access	Creation date
focusup1	US East (N. Virginia) us-east-1	Only authorized users of this account	April 22, 2023, 14:01:20 (UTC+01:00)

Figure 2- S3 bucket



Instances (2) Info

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS	Public IPv4 ...	Elastic IP
FocusUp	i-0b9c45669ca5cb4bd	Running	t2.micro	2/2 checks passed	No alarms	us-east-1c	ec2-3-211-182-247.co...	3.211.182.247	3.211.182.247
UniFashion	i-072d1c9a7690b3c4b	Stopped	t2.micro	–	No alarms	us-east-1c	ec2-54-167-138-208.co...	54.167.138.208	54.167.138.208

Figure 3 – EC2 instance

4 Project Architecture

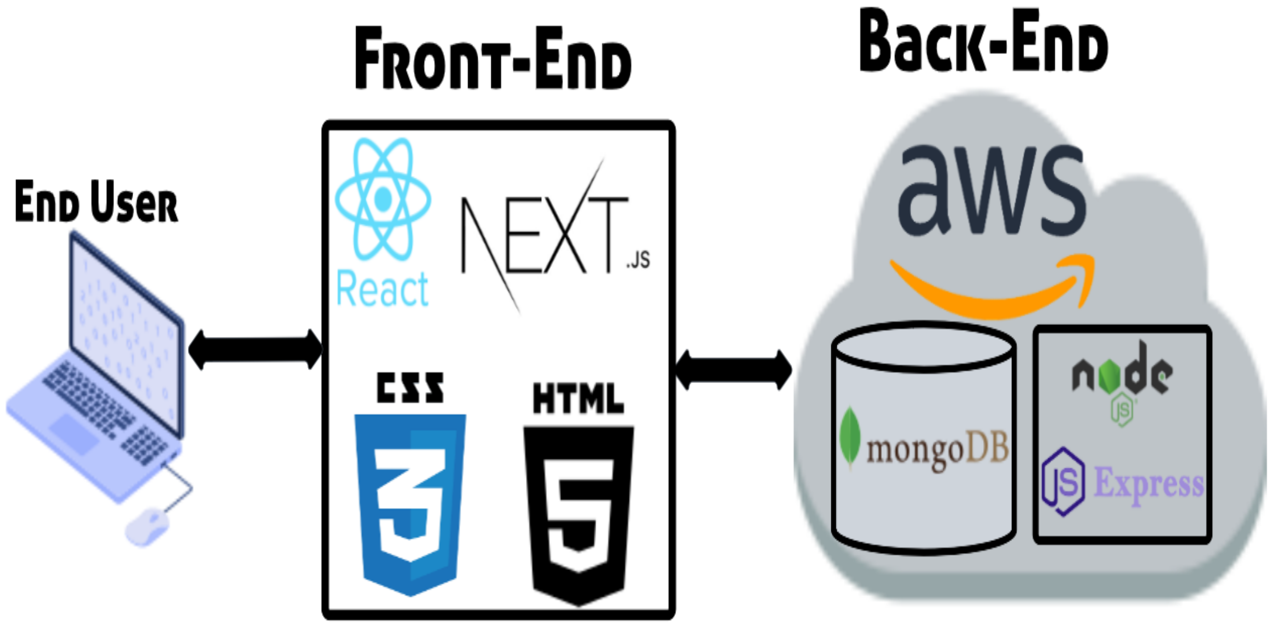
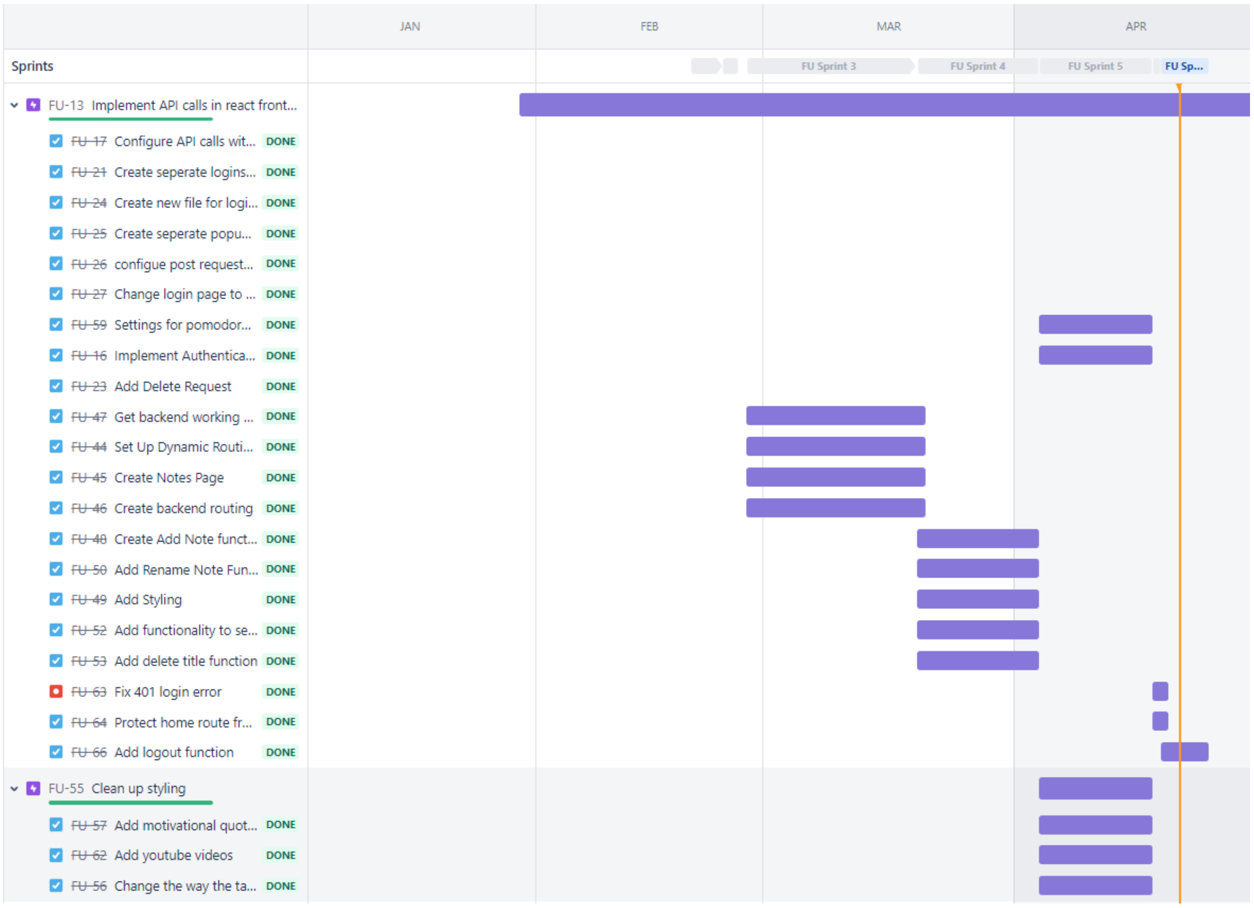


Figure 4-1 Architecture Diagram

5 Project Plan

JIRA is a project management tool used for project planning and task review. This is the tool that I used to plan and manage my project, taking big tasks and breaking them down into smaller ones.

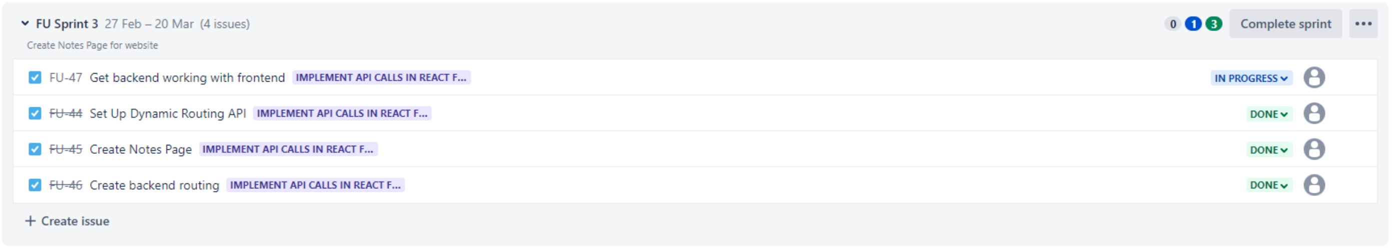


FU-13 Implement API calls in react front...
- ~~FU-17~~ Configure API calls wit... DONE
- ~~FU-21~~ Create seperate logins... DONE
- ~~FU-24~~ Create new file for logi... DONE
- ~~FU-25~~ Create seperate popu... DONE
- ~~FU-26~~ configue post request... DONE
- ~~FU-27~~ Change login page to ... DONE
- ~~FU-59~~ Settings for pomodor... DONE
- ~~FU-16~~ Implement Authentica... DONE
- ~~FU-23~~ Add Delete Request DONE
- ~~FU-47~~ Get backend working ... DONE
- ~~FU-44~~ Set Up Dynamic Routi... DONE
- ~~FU-45~~ Create Notes Page DONE
- ~~FU-46~~ Create backend routing DONE
- ~~FU-48~~ Create Add Note funct... DONE
- ~~FU-50~~ Add Rename Note Fun... DONE
- ~~FU-49~~ Add Styling DONE
- ~~FU-52~~ Add functionality to se... DONE
- ~~FU-53~~ Add delete title function DONE
- ~~FU-63~~ Fix 401 login error DONE
- ~~FU-64~~ Protect home route fr... DONE
- ~~FU-66~~ Add logout function DONE

FU-55 Clean up styling
- ~~FU-57~~ Add motivational quot... DONE
- ~~FU-62~~ Add youtube videos DONE
- ~~FU-56~~ Change the way the ta... DONE

Figure 2 - Roadmap

Each block of work is separated into sprints, where you give yourself a set amount of time to get a certain amount of work done.



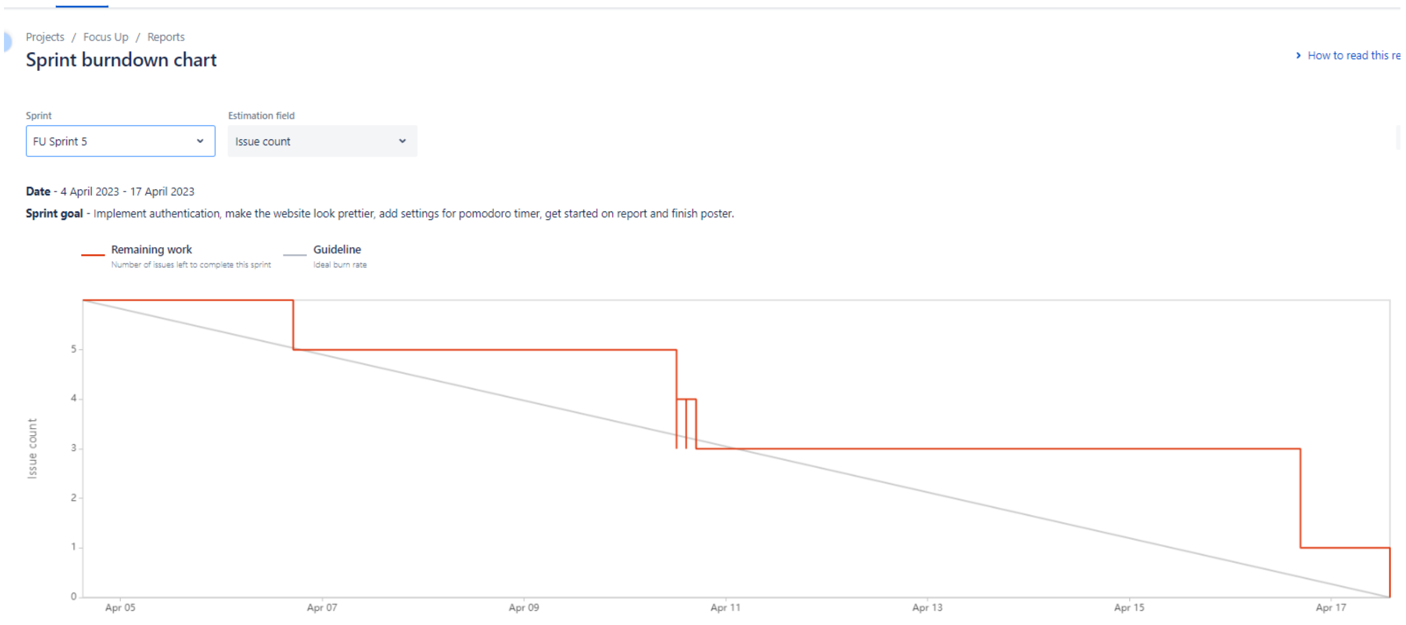
FU Sprint 3 27 Feb – 20 Mar (4 issues)

Create Notes Page for website

- FU-47 Get backend working with frontend IMPLEMENT API CALLS IN REACT F... IN PROGRESS
- ~~FU-44~~ Set Up Dynamic Routing API IMPLEMENT API CALLS IN REACT F... DONE
- ~~FU-45~~ Create Notes Page IMPLEMENT API CALLS IN REACT F... DONE
- ~~FU-46~~ Create backend routing IMPLEMENT API CALLS IN REACT F... DONE

Figure 3 – Sprint Setup

When the sprint is done, you will receive a burndown chart, which details your rate of work throughout the duration of the sprint. Burndown charts help you see how much work you can do in a certain amount of time, enable efficient work for the next sprint.



Date - 4 April 2023 - 17 April 2023

Sprint goal - Implement authentication, make the website look prettier, add settings for pomodoro timer, get started on report and finish poster.

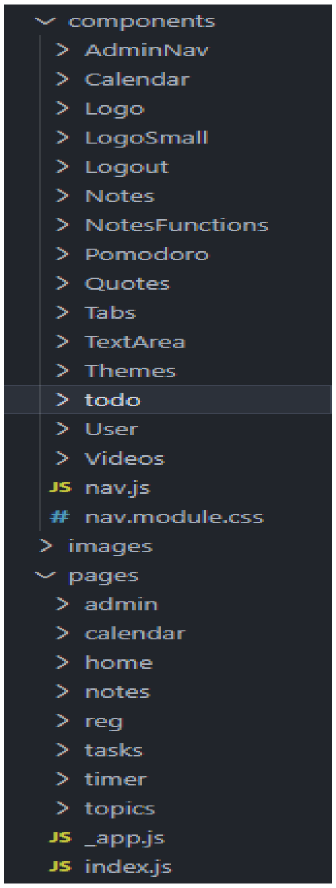
Figure 4 – Burndown chart

6 Frontend

In this section I will be talking about the intricacies of my code and why it works the way it does. I will talk about the frontend in this section and the backend in the next section.

7.1 Structure [5]

The aspects of the app that contribute greatly to the frontend are the components and pages. Components are made in a react app to reduce copy and pasting code if it is needed in other parts of the app, rather, wrapping them in a component enabling them to be called instead of having to be re-written.



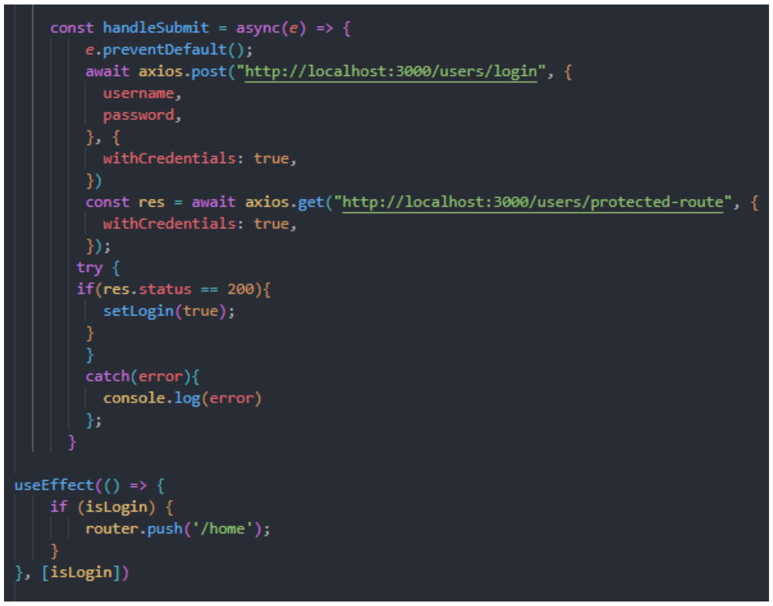
```
components
  AdminNav
  Calendar
  Logo
  LogoSmall
  Logout
  Notes
  NotesFunctions
  Pomodoro
  Quotes
  Tabs
  TextArea
  Themes
  todo
  User
  Videos
  nav.js
  nav.module.css
images
pages
  admin
  calendar
  home
  notes
  reg
  tasks
  timer
  topics
  _app.js
  index.js
```

Figure 5 – File Structure

NextJS recognises any folder under the parent “pages” folder as a url. For example, the home folder is recognised by NextJS as “/home”, and in the subsequent files, the content is rendered.

7.2 Login

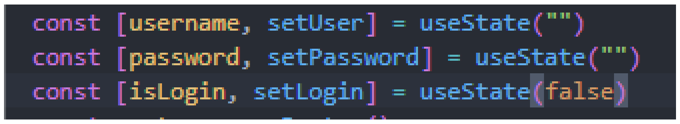
The first page you will see when you open the app is the login page. Here, you have the option to either sign up for a new account, or login to an existing account. This can be done via the use of a REST API [6].



```
    const handleSubmit = async(e) => {
        e.preventDefault();
        await axios.post("http://localhost:3000/users/login", {
          username,
          password,
        }, {
          withCredentials: true,
        })
        const res = await axios.get("http://localhost:3000/users/protected-route", {
          withCredentials: true,
        });
       try {
       if(res.status == 200){
          setLogin(true);
        }
        }
        catch(error){
          console.log(error)
        };
      }

useEffect(() => {
    if (isLogin) {
        router.push('/home');
    }
}, [isLogin])
```

Figure 6 – Login Handler

In my handleSubmit function, a POST request is made to the fetched route. The contents of the POST request include the username and password as the body, which are typed in as an input. When the credentials are submitted, a GET request to the protected-route URL is made. If the username and password are correct, a status 200 is received which will set the usestate variable stored in setLogin to true. There is a useEffect function that will only run when the isLogin usestate variable changes, which is what the square brackets indicates. In the if statement, if the isLogin variable is set to true, the user will be redirected to my homepage.

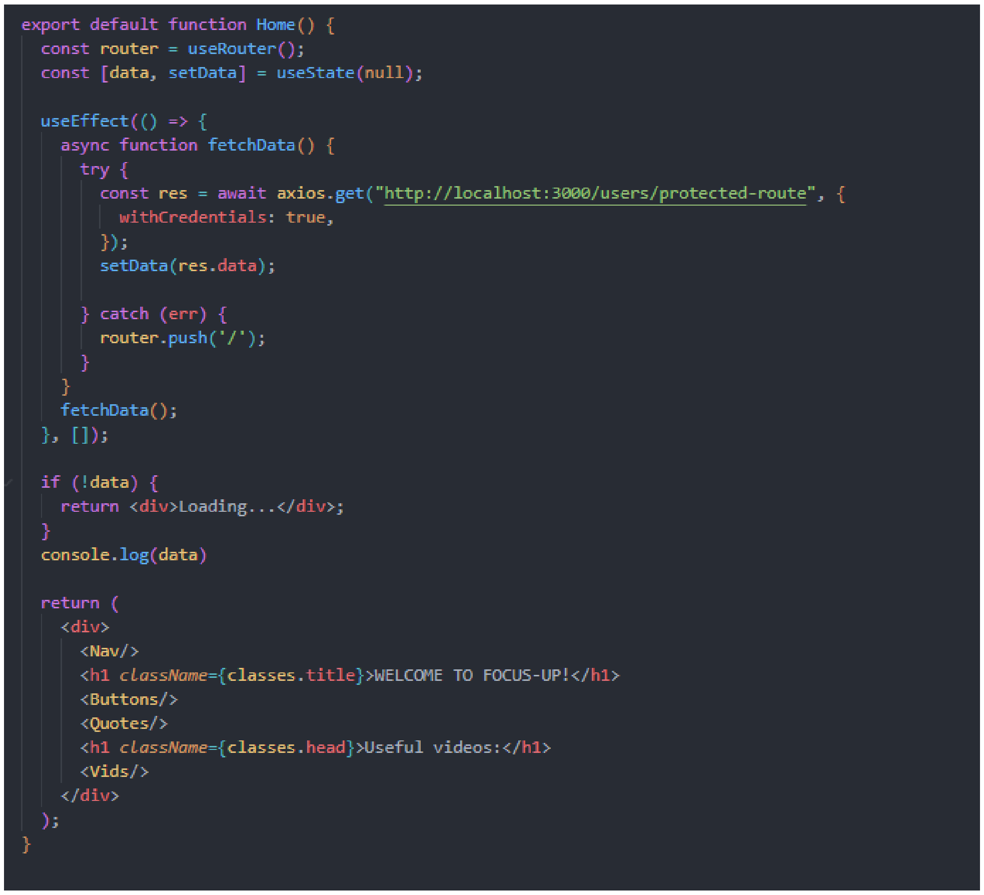


```
const [username, setUser] = useState("")
const [password, setPassword] = useState("")
const [isLogin, setLogin] = useState(false)
```

Figure 7 – Usestate variables

I spoke of usestate variables earlier but didn't explain how they work. Usestate [7] allows you to easily change the state of a variable. The first parameter is the variable itself, and the second parameter is used for setting the state of the variable. The usestate variable can be initialized to anything you like, in this case 2 empty strings and a false Boolean.

7.3 Homepage



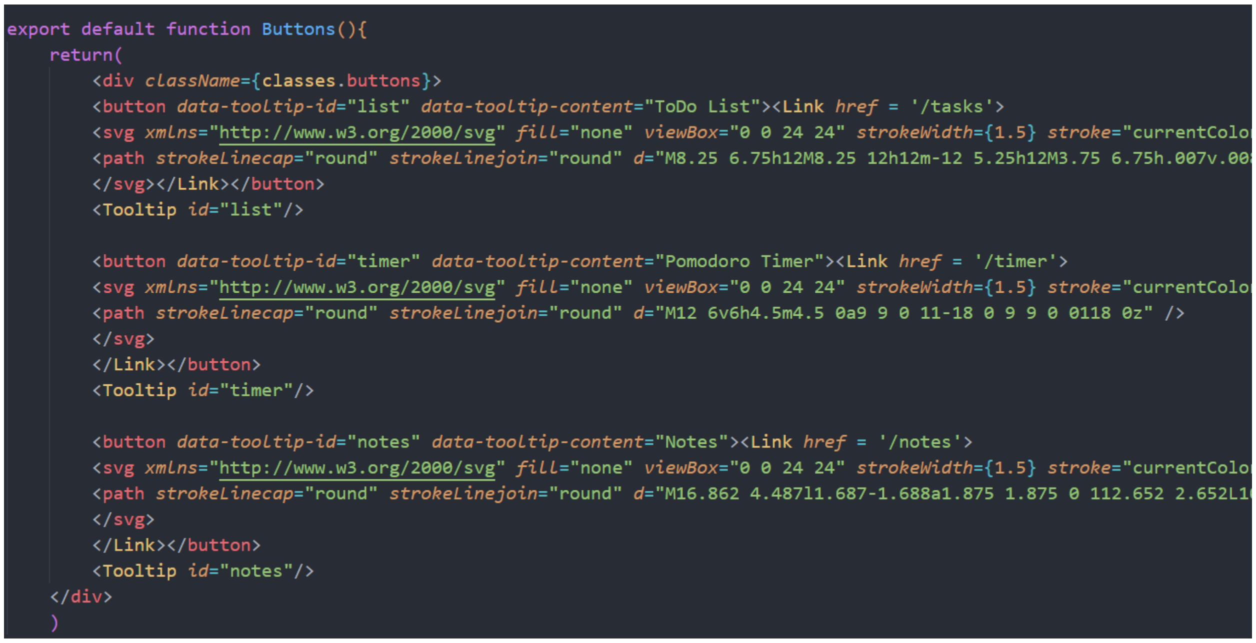
```
export default function Home() {
  const router = useRouter();
  const [data, setData] = useState(null);

  useEffect(() => {
    async function fetchData() {
      try {
        const res = await axios.get("http://localhost:3000/users/protected-route", {
          withCredentials: true,
        });
        setData(res.data);

      } catch (err) {
        router.push('/');
      }
    }
    fetchData();
  }, []);

  if (!data) {
    return <div>Loading...</div>;
  }
  console.log(data)

  return (
    <div>
      <Nav/>
      <h1 className={classes.title}>WELCOME TO FOCUS-UP!</h1>
      <Buttons/>
      <Quotes/>
      <h1 className={classes.head}>Useful videos:</h1>
      <Vids/>
    </div>
  );
}
```

Figure 8 – Route protection

This homepage is a protected route. What this means that unless you are logged into an account, you cannot access this homepage, I will talk about the intricacies when talking about the backend. I used this same get request before to check if a user can login. The condition to login was for a status 200 to be returned, which means we retrieve any data that is sent from that get request. I stated a data usestate variable and set it to null as default. In my try section of my try/catch, I'm attempting to set my data variable to whatever data is being sent from the request. This will only happen if I receive a status 200. My catch deals with if I get an error instead, in which case I will be redirected to the login page. In my return statement you can see I'm using several components, Nav is my navbar, Buttons is the buttons that will send you to different pages, Quotes is a component filled with quotes I have stored, and Vids is where I have YouTube videos linked to study guides.

7.4 Buttons

As previously mentioned, there is a buttons component which contains buttons that will redirect you to different pages.



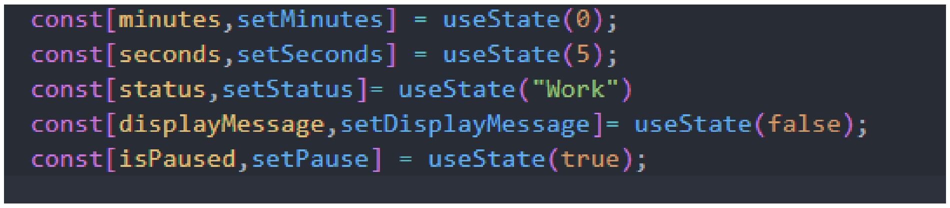
```
export default function Buttons(){
    return(
        <div className={classes.buttons}>
        <button data-tooltip-id="list" data-tooltip-content="ToDo List"><Link href = '/tasks'>
        <svg xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24" strokeWidth={1.5} stroke="currentColo
        <path strokeLinecap="round" strokeLinejoin="round" d="M8.25 6.75h12M8.25 12h12m-12 5.25h12M3.75 6.75h.007v.00
        </svg></Link></button>
        <Tooltip id="list"/>

        <button data-tooltip-id="timer" data-tooltip-content="Pomodoro Timer"><Link href = '/timer'>
        <svg xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24" strokeWidth={1.5} stroke="currentColo
        <path strokeLinecap="round" strokeLinejoin="round" d="M12 6v6h4.5m4.5 0a9 9 0 11-18 0 9 9 0 0118 0z" />
        </svg>
        </Link></button>
        <Tooltip id="timer"/>

        <button data-tooltip-id="notes" data-tooltip-content="Notes"><Link href = '/notes'>
        <svg xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24" strokeWidth={1.5} stroke="currentColo
        <path strokeLinecap="round" strokeLinejoin="round" d="M16.862 4.487l1.687-1.688a1.875 1.875 0 112.652 2.652L1
        </svg>
        </Link></button>
        <Tooltip id="notes"/>
    </div>
    )
```

Figure 9 – Button components

It contains 3 buttons, one sends you to the tasks/todo list page, one sends you to the pomodoro timer page, and the 3rd one sends you to the notes page. Every button is wrapped around a “Link”, which is part of the NextJS library. In each link, a route is provided in which the component is linked to. Each button is also wrapped in a tooltip which gives a short description of where you are going when hovered.

7.5 Pomodoro Timer

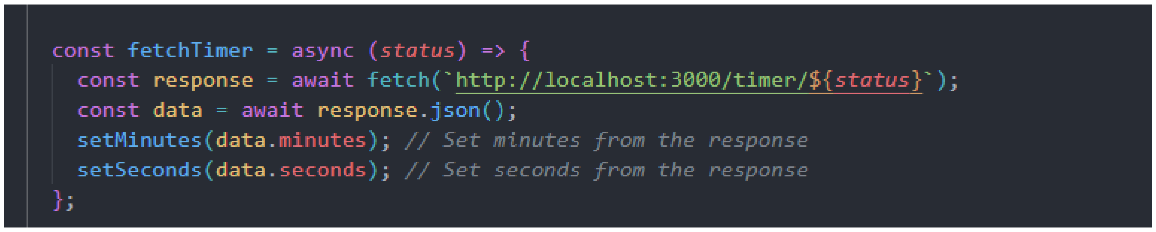
The first feature of the app I will cover is the pomodoro timer [8]. A pomodoro timer is a method of sectioning out study times. The standard format is 25 minutes of study then 5 minutes of rest. This is for you to not burnout when studying for long hours.



```
const[minutes,setMinutes] = useState(0);
const[seconds,setSeconds] = useState(5);
const[status,setStatus]= useState("Work")
const[displayMessage,setDisplayMessage]= useState(false);
const[isPaused,setPause] = useState(true);
```

Figure 10 – Usestate variables

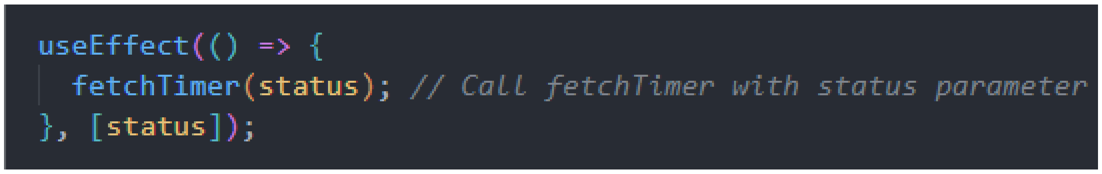
To begin, there are usestate variables defined at the top of my file, minutes and seconds store the minutes and seconds for the timer. Status stores the current work status, there are 2 statuses, “Work” and “Break”. Switching between said statuses will alter the time shown on the pomodoro timer. There is a display message which is tied to a Boolean that only shows when true. Finally, there’s an isPaused variable which toggles true and false when paused and unpaused.



```
const fetchTimer = async (status) => {
  const response = await fetch(`http://localhost:3000/timer/${status}`);
  const data = await response.json();
  setMinutes(data.minutes); // Set minutes from the response
  setSeconds(data.seconds); // Set seconds from the response
};
```

Figure 11 – Get timer info

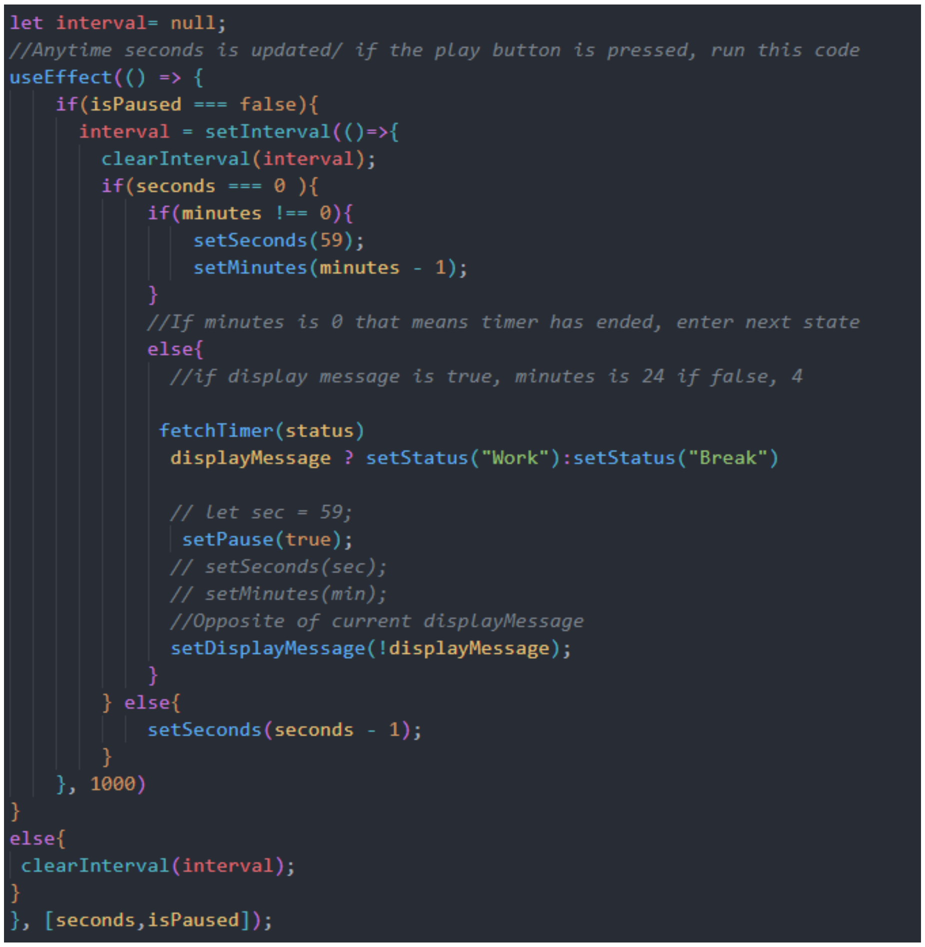
There is a fetchTimer function that makes a GET request to my stored timer in the backend. The variables stored in that stored timer are assigned to the seconds and minutes usestate variables based on the status passed into the function.



```
useEffect(() => {
  fetchTimer(status); // Call fetchTimer with status parameter
}, [status]);
```

Figure 12 – Useffect hook

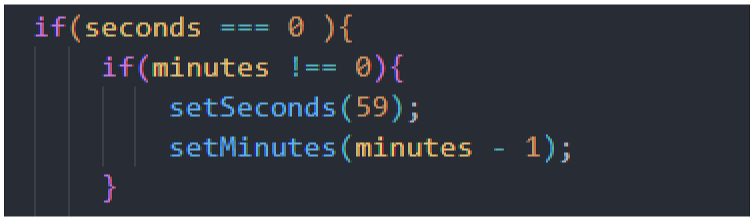
This useEffect function will call the fetchTimer function when the status changes, which changes which time is shown.



```
let interval= null;
//Anytime seconds is updated/ if the play button is pressed, run this code
useEffect(() => {
    if(isPaused === false){
      interval = setInterval(()=>{
        clearInterval(interval);
        if(seconds === 0 ){
            if(minutes !== 0){
                setSeconds(59);
                setMinutes(minutes - 1);
            }
            //If minutes is 0 that means timer has ended, enter next state
            else{
              //if display message is true, minutes is 24 if false, 4

             fetchTimer(status)
              displayMessage ? setStatus("Work"):setStatus("Break")

              // let sec = 59;
               setPause(true);
              // setSeconds(sec);
              // setMinutes(min);
              //Opposite of current displayMessage
              setDisplayMessage(!displayMessage);
            }
        } else{
            setSeconds(seconds - 1);
        }
    }, 1000)
}
else{
 clearInterval(interval);
}
}, [seconds,isPaused]);
```

Figure 13 – Pomodoro timer

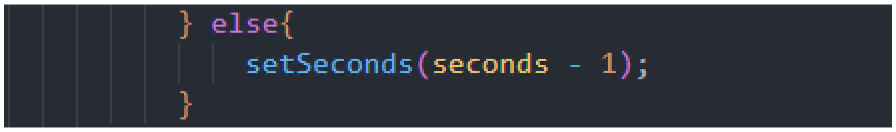
The code is wrapped in a useEffect function which will only run if the seconds or isPaused variable changes. If the isPaused variable is false, that means the timer isn't paused, so the code can run. An interval variable is declared outside the function as null and it stores a reference to the setInterval function, what this setInterval function does is runs the code inside the function after a set amount of time, in this case 1000 milliseconds or one second. Before any of the functional code executes, the interval is cleared with the clearInterval function to ensure that 2 intervals are not running at the same time, as that leads to a very weird acting timer.



```
if(seconds === 0 ){
    if(minutes !== 0){
        setSeconds(59);
        setMinutes(minutes - 1);
    }
```

Figure 14 – Nested if statement

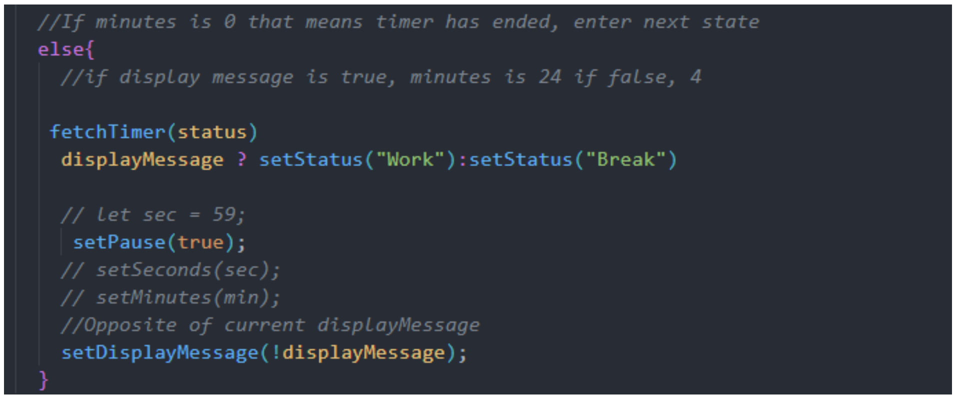
There are nested if statements to set the seconds and minutes. So, if seconds is equal to 0 and minutes is not equal to zero then the seconds is reset to 59 and minutes goes down by 1.



```
    } else{
        setSeconds(seconds - 1);
    }
```

Figure 15 – Else statement

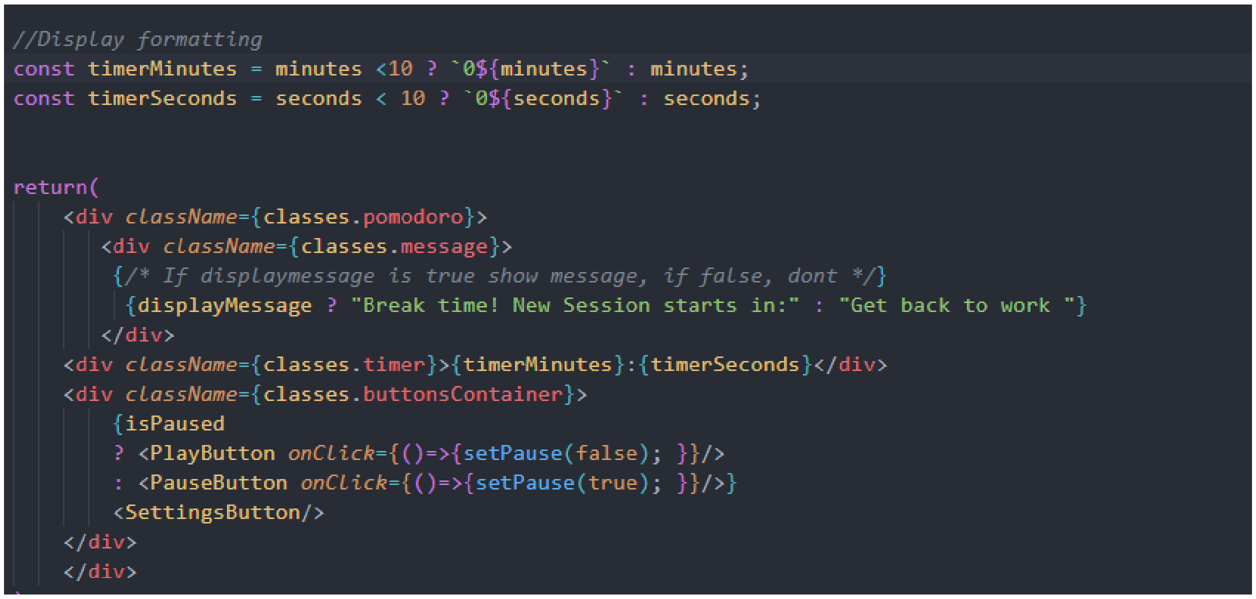
But if seconds is not equal to 0, deduce it by 1 until it is.



```
//If minutes is 0 that means timer has ended, enter next state
else{
  //if display message is true, minutes is 24 if false, 4

 fetchTimer(status)
  displayMessage ? setStatus("Work"):setStatus("Break")

  // let sec = 59;
   setPause(true);
  // setSeconds(sec);
  // setMinutes(min);
  //Opposite of current displayMessage
  setDisplayMessage(!displayMessage);
}
```

Figure 16 – Changing work status.

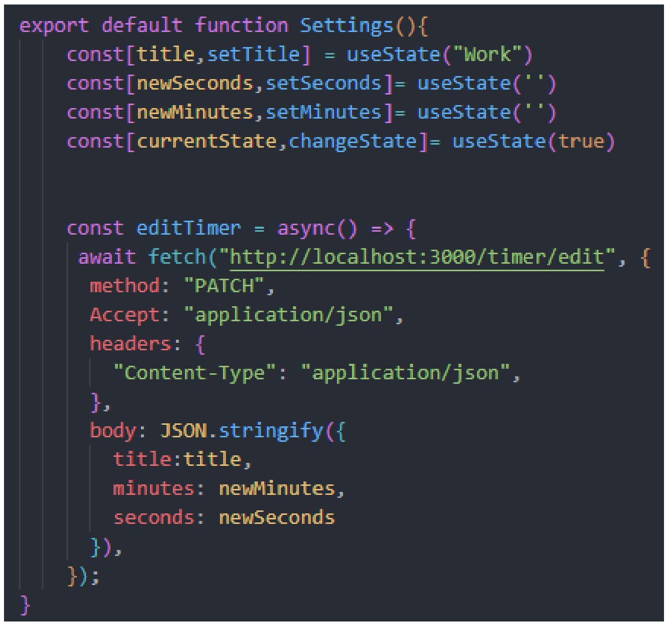
If minutes is equal to 0, that means the timer has ended. In this else statement, the status is being set based on if the display message is true or false. This is done by the ternary operator "?". The code on the left executes on a true statement and the code on the right executes on a false statement. When the code enters this else statement, the displayMessage state variable is set to the opposite of its current state, which is what "!" does. The isPaused variable is set to true to give the user the chance to start whenever they want. Finally, the status is sent to the fetchTimer function where previously stated, changes the time shown based on the status passed into it.



```
//Display formatting
const timerMinutes = minutes <10 ? `0${minutes}` : minutes;
const timerSeconds = seconds < 10 ? `0${seconds}` : seconds;


return(
    <div className={classes.pomodoro}>
      <div className={classes.message}>
       {/* If displaymessage is true show message, if false, dont */}
        {displayMessage ? "Break time! New Session starts in:" : "Get back to work "}
      </div>
    <div className={classes.timer}>{timerMinutes}:{timerSeconds}</div>
    <div className={classes.buttonsContainer}>
        {isPaused
        ? <PlayButton onClick={()=>{setPause(false); }}/>
        : <PauseButton onClick={()=>{setPause(true); }}/>}
        <SettingsButton/>
    </div>
    </div>
```

Figure 17 – Render timer

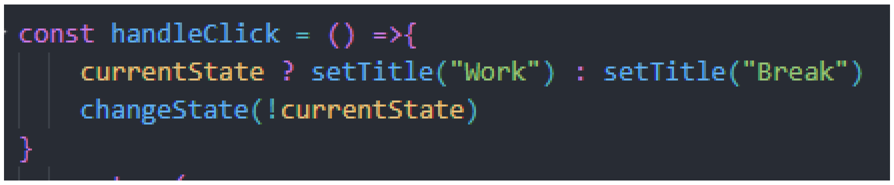
The ternary operator is used to show what message shows up based on if displayMessage is true or not. There are 3 buttons, a play button, a pause button and a settings button, the play button is show if the isPaused state is true and the pause button shows if the isPaused state is false. The play and pause buttons set the isPaused state variable to true or false and the settings button opens a popup where you can edit how much time you want for work and breaktime.



```
export default function Settings(){
    const[title,setTitle] = useState("Work")
    const[newSeconds,setSeconds]= useState('')
    const[newMinutes,setMinutes]= useState('')
    const[currentState,changeState]= useState(true)


    const editTimer = async() => {
     await fetch("http://localhost:3000/timer/edit", {
      method: "PATCH",
      Accept: "application/json",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({
        title:title,
        minutes: newMinutes,
        seconds: newSeconds
      }),
    });
}
```

Figure 18 – Function to edit timer.

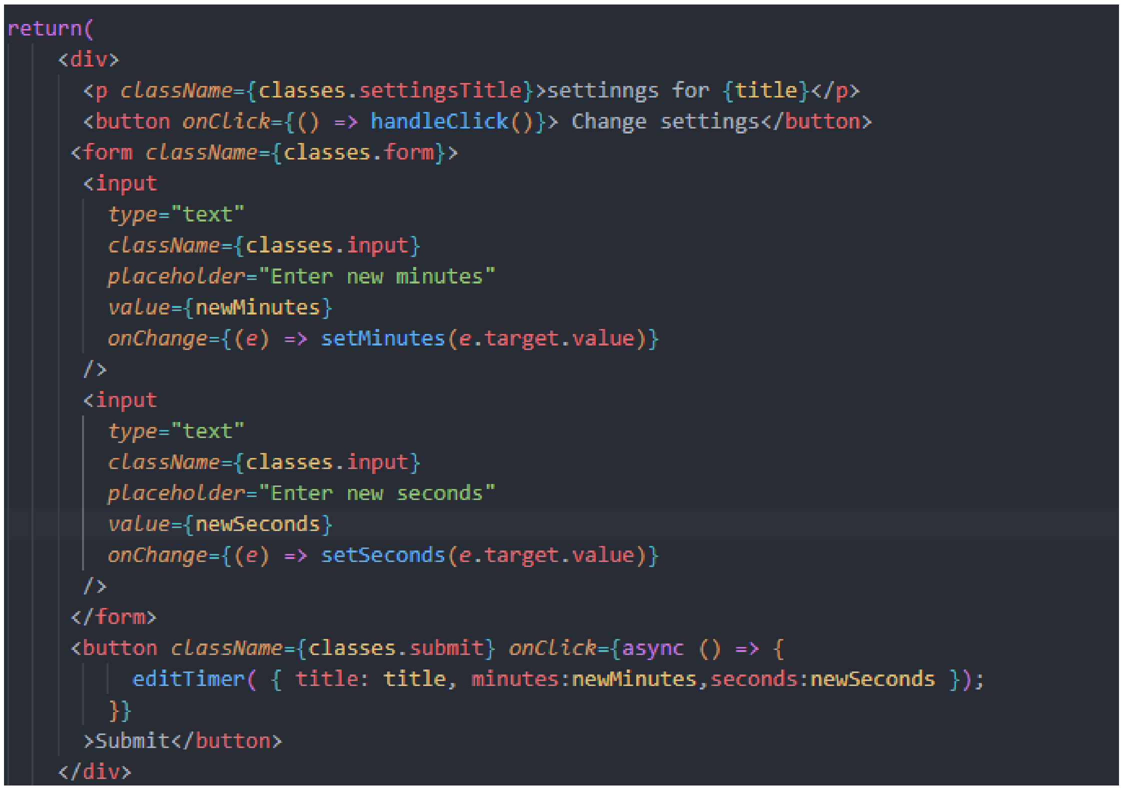
In the settings component, there are 4 usestates. One for holding the work state to be changed, two for setting the minutes and seconds, and one for switching the current state. There is an editTimer function that contains a PATCH request that accepts a title, minutes, and seconds as the body of the request. The request will search for a database object with the title passed in and will alter the minutes and seconds of that object to the values inputted.



```
const handleClick = () =>{
    currentState ? setTitle("Work") : setTitle("Break")
    changeState(!currentState)
}
```

Figure 19 – State handler

In the handleClick function, the title state variable is switched based on the state of the currentState variable, which also switches when the function is called.

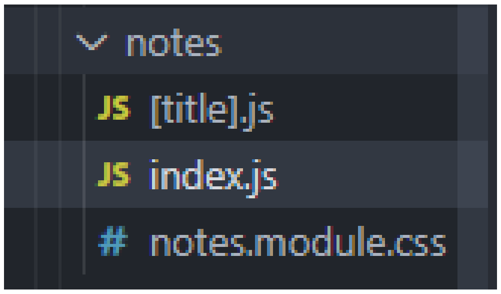


```
return(
    <div>
      <p className={classes.settingsTitle}>settinngs for {title}</p>
      <button onClick={() => handleClick()}> Change settings</button>
     <form className={classes.form}>
      <input
        type="text"
        className={classes.input}
        placeholder="Enter new minutes"
        value={newMinutes}
        onChange={(e) => setMinutes(e.target.value)}
      />
      <input
        type="text"
        className={classes.input}
        placeholder="Enter new seconds"
        value={newSeconds}
        onChange={(e) => setSeconds(e.target.value)}
      />
     </form>
     <button className={classes.submit} onClick={async () => {
          editTimer( { title: title, minutes:newMinutes,seconds:newSeconds });
        }}
      >Submit</button>
    </div>
```

Figure 20 – Render Components

The button at the top of the return statement calls the handleClick function which executes all the code inside the function. The user then has 2 input fields to enter the minutes and seconds they want. When the submit button is clicked, the editTimer function is called, sending the values of title, minutes, and seconds as the parameters requested in the body.

7.6 Note Taking

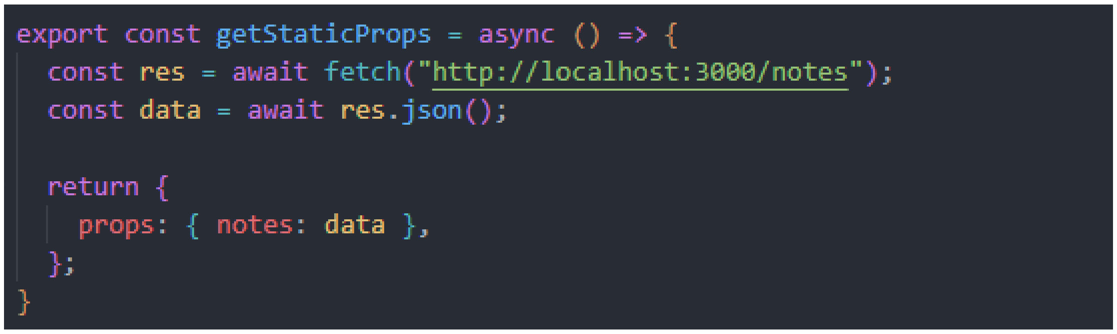


```
notes
  [title].js
  index.js
  notes.module.css
```

Figure 21 – Paging setup

The note taking feature allows you to create, update, find and delete a note topic. You are also able to go into each individual note topic and create a note which is completely unique to that topic.

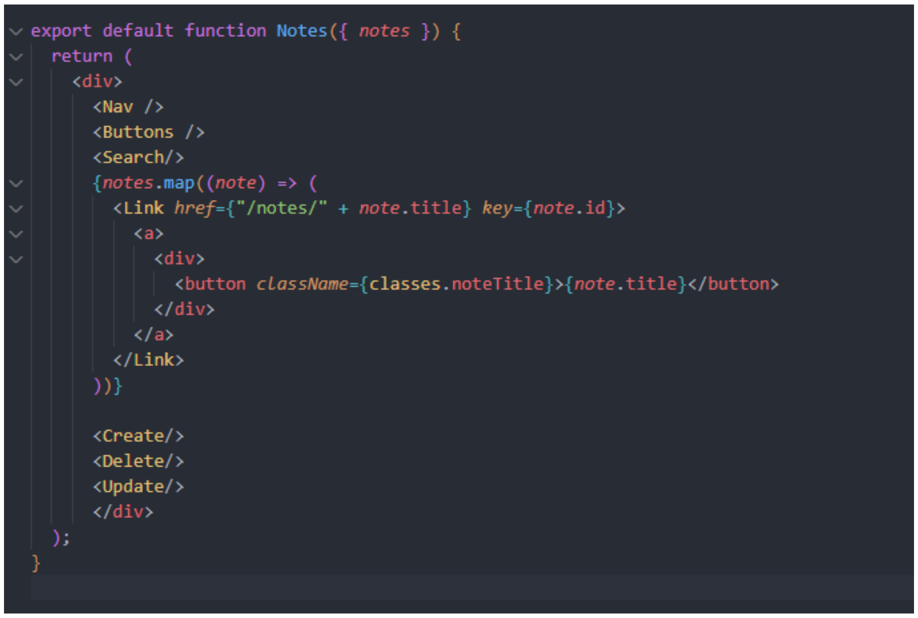
This folder has a slightly different structure to other page folders. As well as containing the file for the main page, it also includes another file with its name enclosed in square brackets. This is what is called dynamic routing [9] in NextJS. What this means is that pages can be rendered based on data passed into it. For example, the default notes page is "/notes", but if I wanted to access my math notes, I would have to go to the "/notes/math" URL.



```
export const getStaticProps = async () => {
  const res = await fetch("http://localhost:3000/notes");
  const data = await res.json();

  return {
    props: { notes: data },
  };
}
```

Figure 22 – getStaticProps function

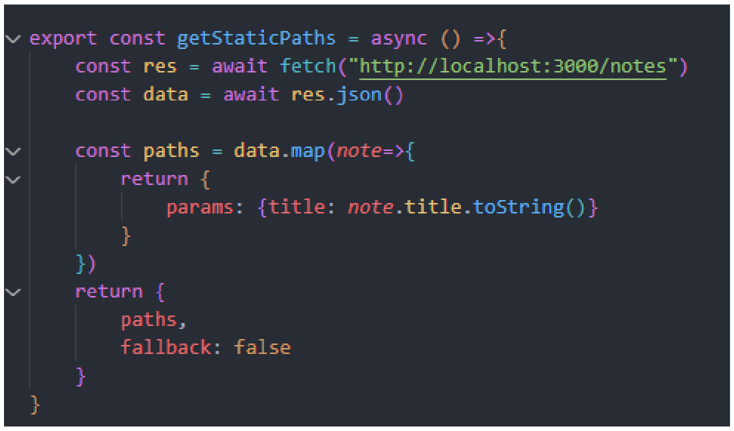
In the index.js file, a function getStaticProps is defined. The getStaticProps function is a tool used by Nextjs to pre-render pages at build time, enabling faster render times for pages when visited. In this code, a get request is being made to the "/notes" route, which fetches all the note topics stored in the database. The response is then stored in a variable called data, and that data is returned as a prop with a "notes" key.



```
export default function Notes({ notes }) {
  return (
    <div>
      <Nav />
      <Buttons />
      <Search/>
      {notes.map((note) => (
        <Link href={"/notes/" + note.title} key={note.id}>
          <a>
            <div>
              <button className={classes.noteTitle}>{note.title}</button>
            </div>
          </a>
        </Link>
      ))}

      <Create/>
      <Delete/>
      <Update/>
      </div>
  );
}
```

Figure 23 – Render List of pages

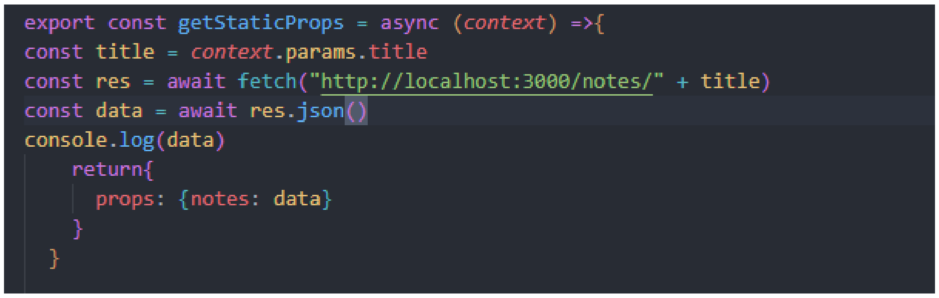
The notes prop is passed into the Notes component, where it is inserted into a mapping function [10]. This mapping function goes through every note stored in the database and renders it as a link to its own page, with the name of the note being an extension of the "/notes" URL. Every rendered component must be unique hence why each component is attached with an individual key, that being its objectId which is stored in the database.



```
export const getStaticPaths = async () =>{
    const res = await fetch("http://localhost:3000/notes")
    const data = await res.json()

    const paths = data.map(note=>{
        return {
            params: {title: note.title.toString()}
        }
    })
    return {
        paths,
        fallback: false
    }
}
```

Figure 24 – getStaticPaths function

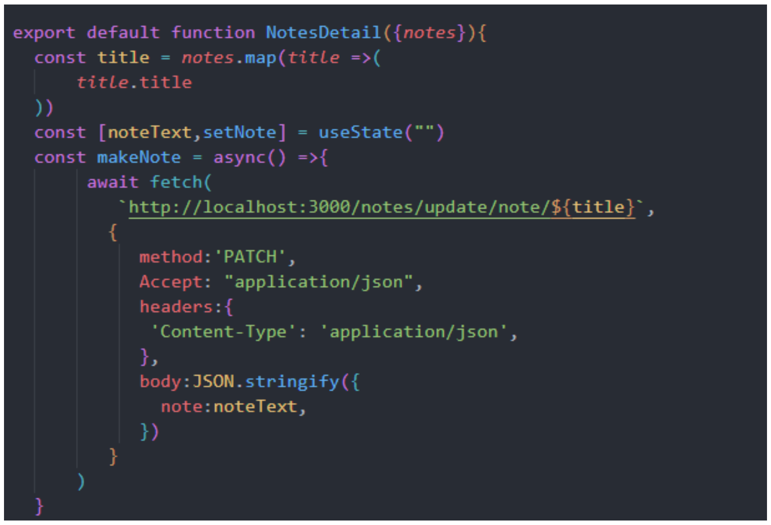
The getStaticPaths function is used for pre-rendering dynamic pages. The function itself attempts to make the path that is requested based on the data it fetches. What's happening in this function is that the note objects are fetched and stored in a variable. The data in this variable is then mapped to an array where each note title has a params key and is converted into a string, which is all stored in a paths variable. The array of paths and a fallback Boolean are returned as an object. Fallback is set to false so that the page returns a 404 error if the path requested by a user does not exist.



```
export const getStaticProps = async (context) =>{
const title = context.params.title
const res = await fetch("http://localhost:3000/notes/" + title)
const data = await res.json()
console.log(data)
    return{
      props: {notes: data}
    }
  }
```

Figure 25 – getStaticProps function

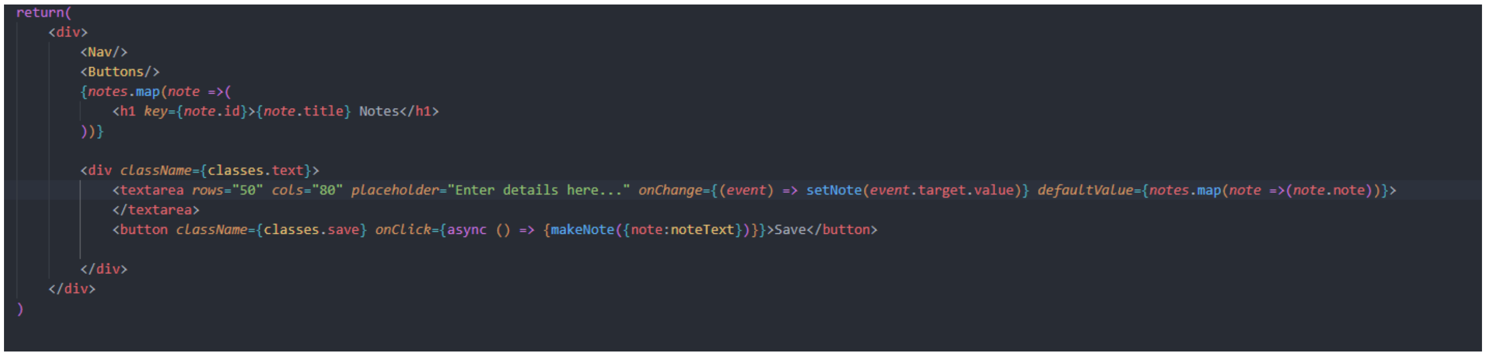
The getStaticProps function fetches the data for the page requested. The context parameter passed into the function contains the params object, which also contains the title parameter in the url. This result is passed to a variable called title. The get request fetches the data needed for the specific title passed into the url, which is stored in a variable. That variable gets passed down to the component that will render the data.



```
export default function NotesDetail({notes}){
  const title = notes.map(title =>(
      title.title
  ))
  const [noteText,setNote] = useState("")
  const makeNote = async() =>{
      await fetch(
        `http://localhost:3000/notes/update/note/${title}`,
      {
          method:'PATCH',
          Accept: "application/json",
          headers:{
           'Content-Type': 'application/json',
          },
          body:JSON.stringify({
            note:noteText,
          })
      }
    )
  }
```

Figure 26 – Function to edit note

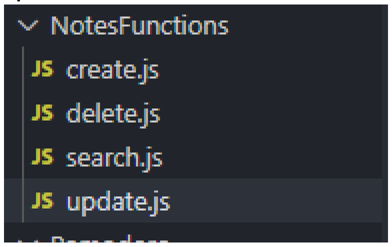
The NotesDetail component renders the page. The notes prop defined in getStaticProps is passed down into the component, and using a mapping function to extract the title, it is assigned to a variable called title. A usestate variable is defined that will handle the inputted notes. In the makeNote function, a PATCH request is made to the specific url which has the title passed into it. The title passed in to the url is the title to be edited, and the noteText inside the body is what parameter is edited, in this case it’s the note parameter.



```
return(
    <div>
        <Nav/>
        <Buttons/>
        {notes.map(note =>(
            <h1 key={note.id}>{note.title} Notes</h1>
        ))}

        <div className={classes.text}>
            <textarea rows="50" cols="80" placeholder="Enter details here..." onChange={(event) => setNote(event.target.value)} defaultValue={notes.map(note =>(note.note))}>
            </textarea>
            <button className={classes.save} onClick={async () => {makeNote({note:noteText})}}>Save</button>

        </div>
    </div>
)
```

Figure 27 – Render note taking area.

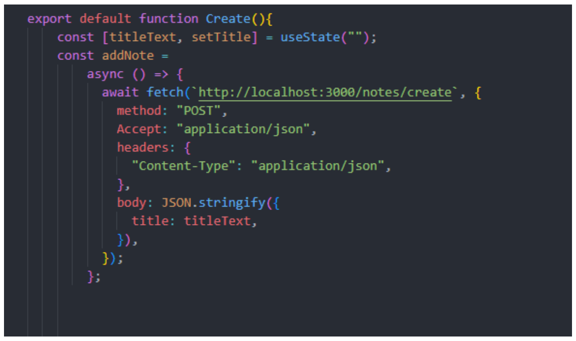
The data in the notes prop is mapped to a temporary array to display the name of the note title. In the textarea, the user can input notes. The setNote usestate setter being set to event.target.value means that the value of noteText is updated every time the user inputs a key. The textarea also contains a defaultValue prop which sets its value to the mapped notes object, extracting the note parameter. Finally, there's a button that calls the makeNote function when clicked, passing down the inputted noteText into the function.



```
NotesFunctions
    JS create.js
    JS delete.js
    JS search.js
    JS update.js
```

Figure 28 – Note functions structure.

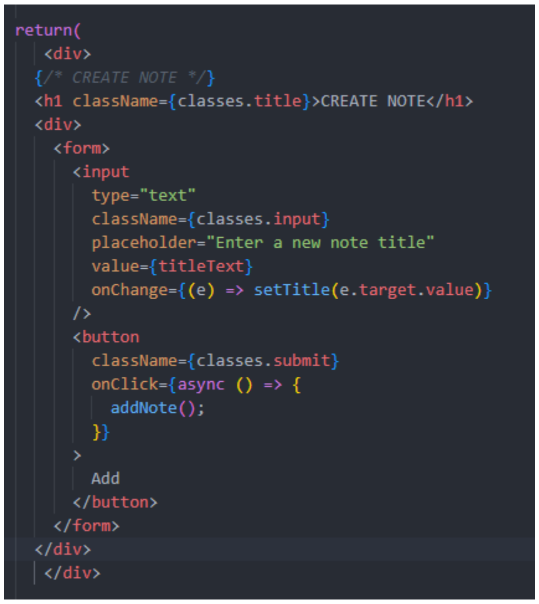
Each function for altering the note titles were stored in a component called NotesFunctions.



```
export default function Create(){
    const [titleText, setTitle] = useState("");
    const addNote =
        async () => {
          await fetch(`http://localhost:3000/notes/create`, {
            method: "POST",
            Accept: "application/json",
            headers: {
              "Content-Type": "application/json",
            },
            body: JSON.stringify({
              title: titleText,
            }),
          });
        };
```

Figure 29 – POST request to create note.

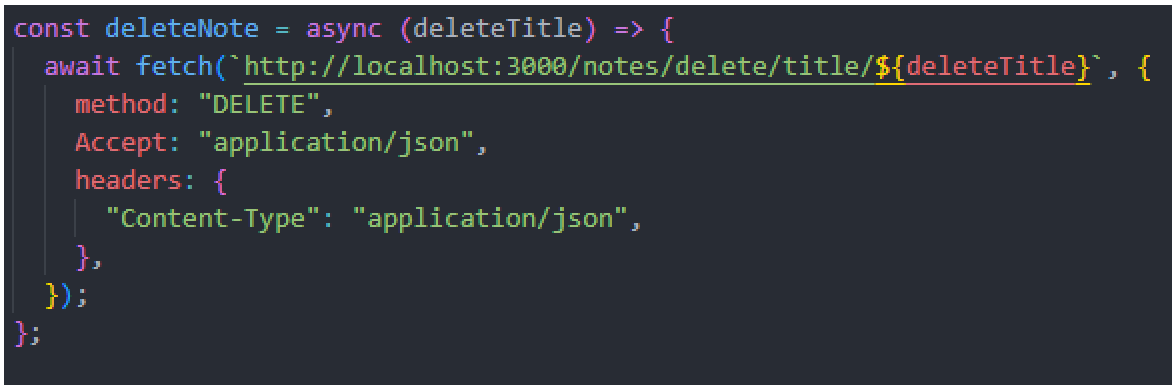
In the create component, a POST request is made to the route with the body of the POST request being the title of the note. The title of the note will be assigned the value of the titleText usestate variable.



```
return(
  <div>
  {/* CREATE NOTE */}
  <h1 className={classes.title}>CREATE NOTE</h1>
  <div>
    <form>
      <input
        type="text"
        className={classes.input}
        placeholder="Enter a new note title"
        value={titleText}
        onChange={(e) => setTitle(e.target.value)}
      />
      <button
        className={classes.submit}
        onClick={async () => {
          addNote();
        }}
      >
        Add
      </button>
    </form>
  </div>
  </div>
```

Figure 30 – Input and send data to request.

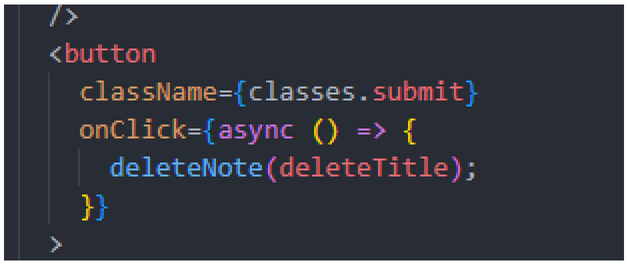
Inside the return statement there is an input where the user can input the name of the title they want to create. The title is set to whatever is typed into the input and it updates with every new character entered. There is also a button component that calls on the addNote function when pressed, assigning the title parameter the value of the inputted titleText usestate variable and creating a note object with that name.



```
const deleteNote = async (deleteTitle) => {
  await fetch(`http://localhost:3000/notes/delete/title/${deleteTitle}`, {
    method: "DELETE",
    Accept: "application/json",
    headers: {
      "Content-Type": "application/json",
    },
  });
};
```

Figure 31 – DELETE request to delete topic.

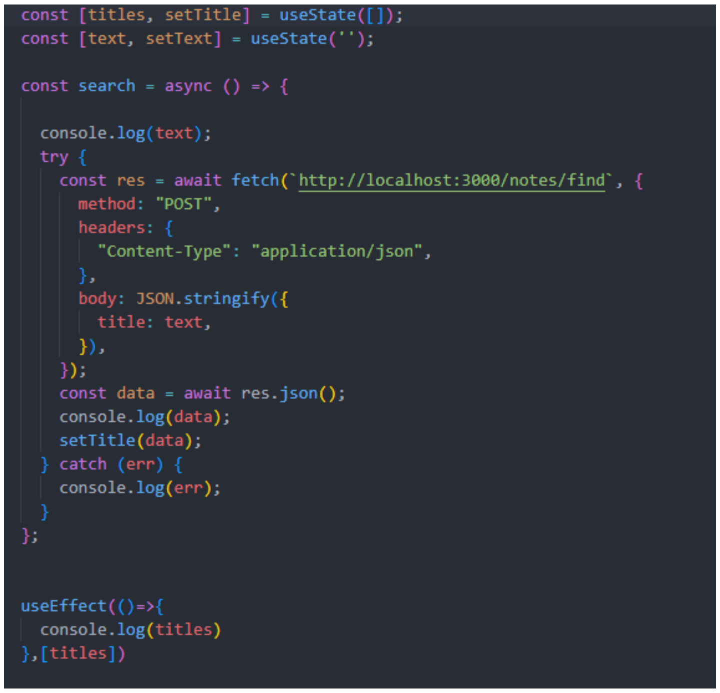
The delete function deletes a note based on the title of the note passed into it. The title is then passed into the URL where the title will be deleted.



```
/>
<button
  className={classes.submit}
  onClick={async () => {
    deleteNote(deleteTitle);
  }}
>
```

Figure 32 – Data sent to DELETE request.

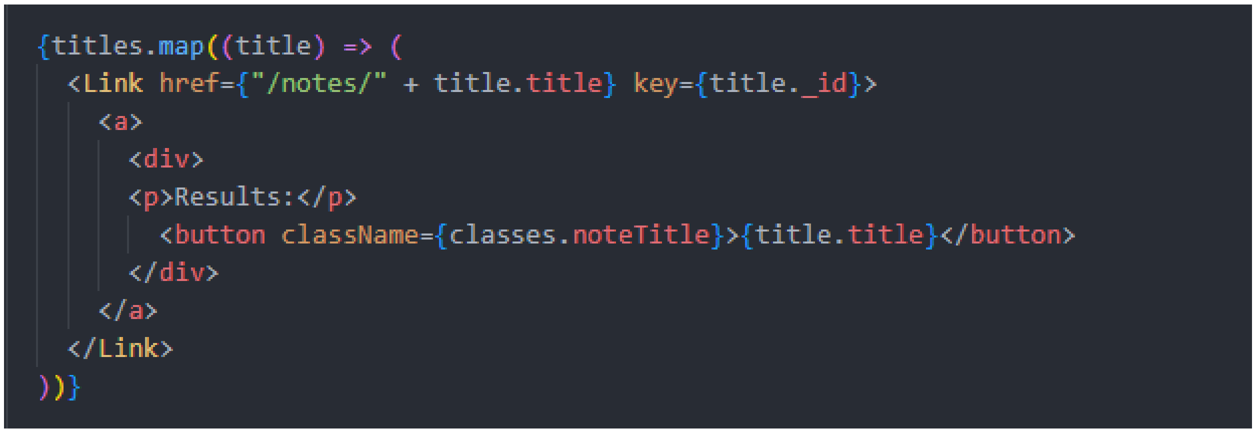
The button component calls the deleteNote function but also passes in the value of deleteTitle as an argument to the function, where it will be deleted.



```
const [titles, setTitle] = useState([]);
const [text, setText] = useState('');

const search = async () => {

  console.log(text);
  try {
    const res = await fetch(`http://localhost:3000/notes/find`, {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({
        title: text,
      }),
    });
    const data = await res.json();
    console.log(data);
    setTitle(data);
  } catch (err) {
    console.log(err);
  }
};


useEffect(()=>{
  console.log(titles)
},[titles])
```

Figure 33 – POST request to find topic.

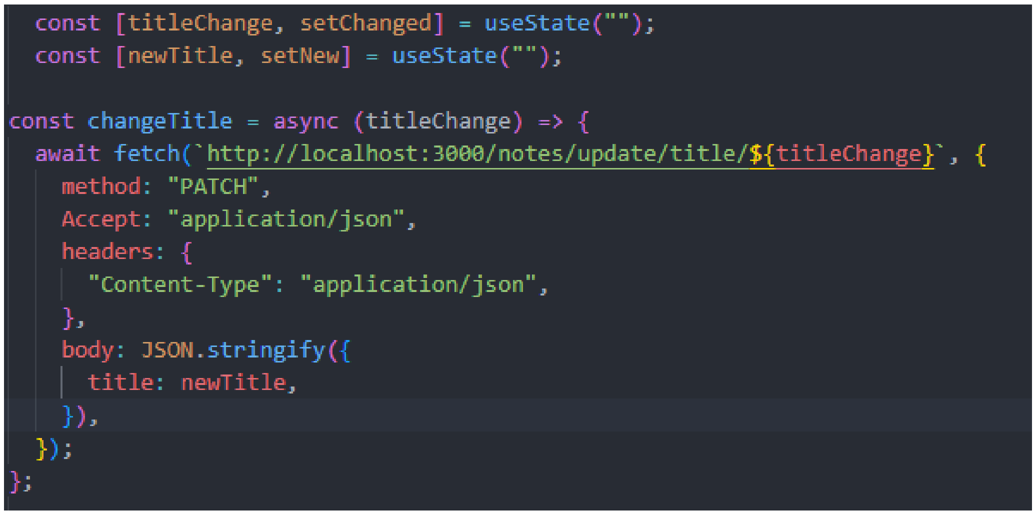
In the search component, there contains a search function which makes a POST request to its fetched route. The title is the body of the request, and it is assigned the value of the text usestate variable. The result of this search should be a list of titles that share the name that was searched, and this result is stored in a variable called data. The titles usestate variable (which is initialized to an empty array) is set to the value of the data variable.



```
{titles.map((title) => (
  <Link href={"/notes/" + title.title} key={title._id}>
    <a>
      <div>
      <p>Results:</p>
        <button className={classes.noteTitle}>{title.title}</button>
      </div>
    </a>
  </Link>
))}
```

Figure 34 – Render found topics.

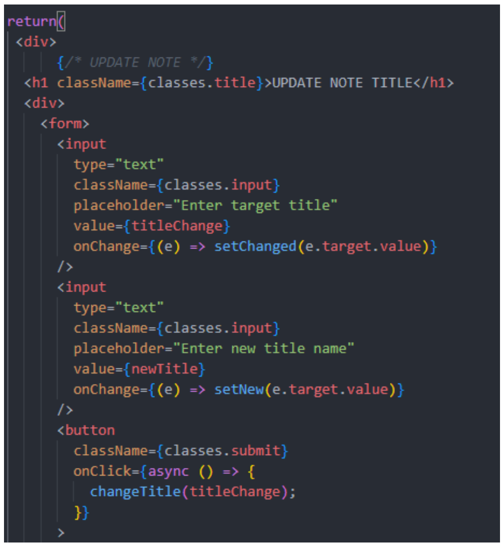
The titles usestate variable is put into a mapping function where each object stored has their title displayed. The titles are button components that are wrapped in a Link component. When clicked, the user will be sent to the dynamic route for that title.



```
  const [titleChange, setChanged] = useState("");
  const [newTitle, setNew] = useState("");

const changeTitle = async (titleChange) => {
  await fetch(`http://localhost:3000/notes/update/title/${titleChange}`, {
    method: "PATCH",
    Accept: "application/json",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify({
      title: newTitle,
    }),
  });
};
```

Figure 35 – PATCH request to change topic name.

In the update component, there is a changeTitle function that accepts the titleChange usestate variable as an argument. This titleChange variable is then passed into the fetched URL where a PATCH request will be performed on it. The titleChange variable represents the current title to be changed, and the newTitle variable, which is accepted as the body of the request, is the name the title will change to.



```
return(
 <div>
     {/* UPDATE NOTE */}
  <h1 className={classes.title}>UPDATE NOTE TITLE</h1>
  <div>
    <form>
      <input
        type="text"
        className={classes.input}
        placeholder="Enter target title"
        value={titleChange}
        onChange={(e) => setChanged(e.target.value)}
      />
      <input
        type="text"
        className={classes.input}
        placeholder="Enter new title name"
        value={newTitle}
        onChange={(e) => setNew(e.target.value)}
      />
      <button
        className={classes.submit}
        onClick={async () => {
          changeTitle(titleChange);
        }}
      >
```

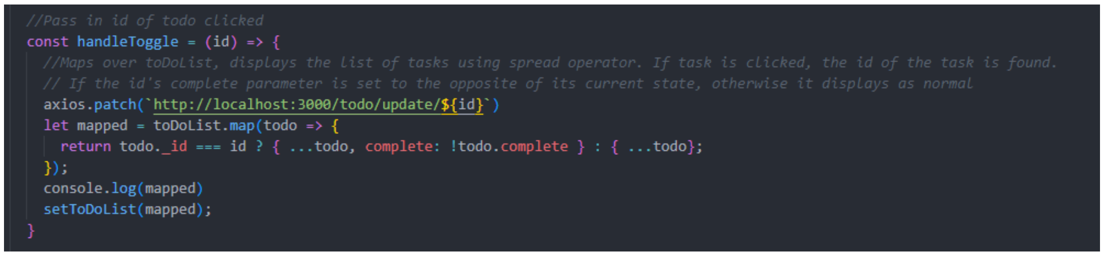
Figure 36 – Send data to PATCH request.

There are 2 input components, one for setting the name of the title to be changed, and one for the new name of the title. When the button is pressed, the titleChange variable is passed in as an argument to the changeTitle function, where it will be updated to the value of the newTitle usestate variable.

7.7 To-Do-list. [11]

The to-do list feature allows users to add and remove tasks.

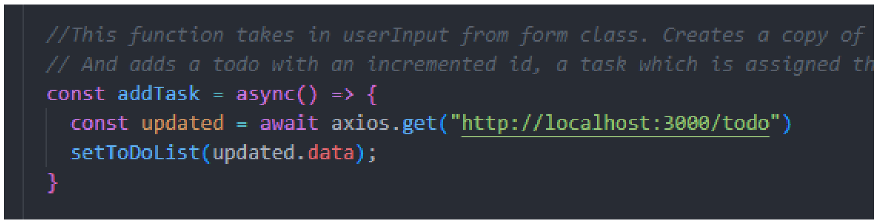
Here the todolist is set to an empty array using a usestate. In the getList function, the tasks from the todolist stored in the database are fetched. The result of that get request is set to the value of the todolist usestate variable. The getList function is then put inside a useEffect function where it will run the code inside the function when the page renders and at no other time.



```
//Pass in id of todo clicked
const handleToggle = (id) => {
  //Maps over toDoList, displays the list of tasks using spread operator. If task is clicked, the id of the task is found.
  // If the id's complete parameter is set to the opposite of its current state, otherwise it displays as normal
  axios.patch(`http://localhost:3000/todo/update/${id}`)
  let mapped = toDoList.map(todo => {
    return todo._id === id ? { ...todo, complete: !todo.complete } : { ...todo};
  });
  console.log(mapped)
  setToDoList(mapped);
}
```

Figure 37 – Task state handler

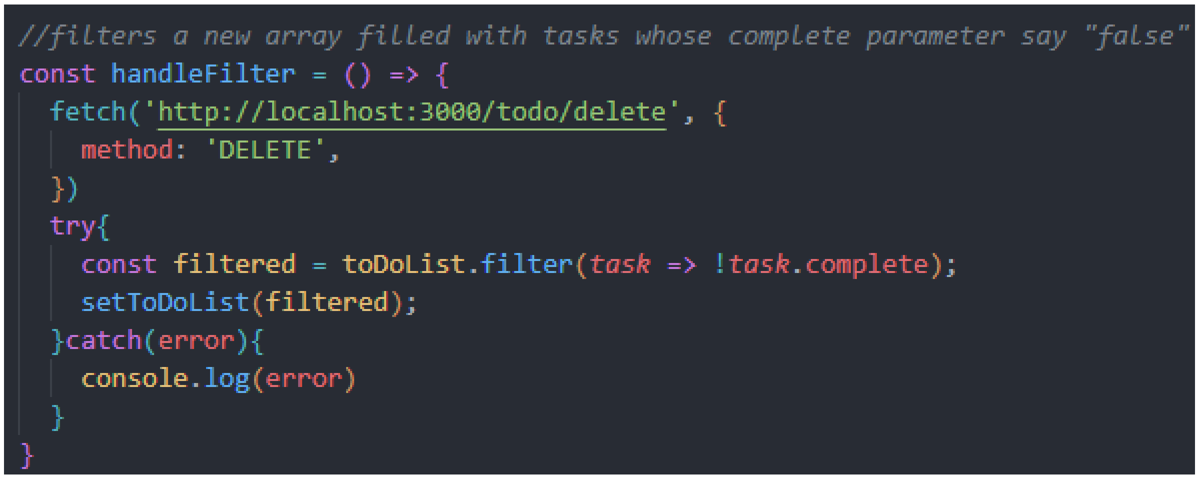
The handleToggle function switches a to-do’s state between complete and incomplete. When a task is clicked, the handleToggle function is called, passing in its id. In the function, the fetch request reads the id into its URL. In the backend the id passed into the URL gets its “complete” parameter switched to its opposite. The todolist is then inserted into a mapping function. In here, the id of all the tasks in the todolist are compared to the id of the task passed in. If the ids don’t match, a new object of the task is created with its original parameters. If the ids do match though, a new object of the task is created but with an overwritten complete parameter. This parameter is switched to the opposite of its current value. The results of mapping function are assigned to a variable and the todolist is then set to the updated list.



```
//This function takes in userInput from form class. Creates a copy of
// And adds a todo with an incremented id, a task which is assigned th
const addTask = async() => {
  const updated = await axios.get("http://localhost:3000/todo")
  setToDoList(updated.data);
}
```

Figure 38 – Function to update list

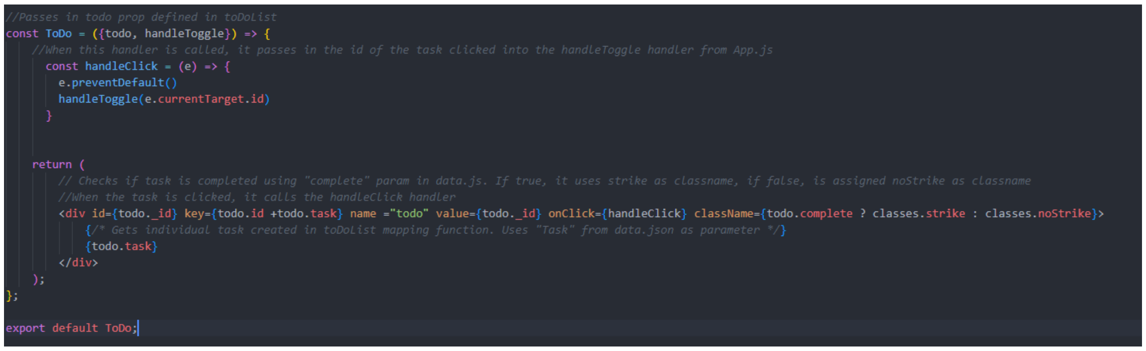
The addTask function handles updating the todolist with the new task added. The adding of tasks is handled in another file. In this function, the updated todolist is assigned to a variable. The todolist usestate variable is then set to the results of the get request.



```
//filters a new array filled with tasks whose complete parameter say "false"
const handleFilter = () => {
  fetch('http://localhost:3000/todo/delete', {
    method: 'DELETE',
  })
  try{
    const filtered = toDoList.filter(task => !task.complete);
    setToDoList(filtered);
  }catch(error){
    console.log(error)
  }
}
```

Figure 39 – HandleFilter function

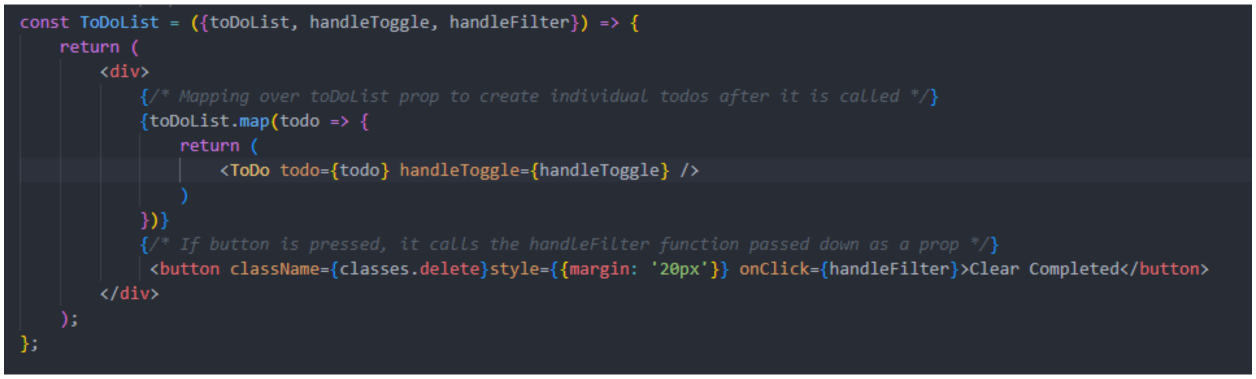
The handleFilter function deletes any tasks that are marked complete. The url is fetched and a DELETE request is performed on it. That delete request deletes all completed tasks from the database. In the try block, the .filter() method is performed on the toDoList, which returns a new array of all the non complete tasks. The toDoList is then set to the result of the method.



```
//Passes in todo prop defined in toDoList
const ToDo = ({todo, handleToggle}) => {
  //When this handler is called, it passes in the id of the task clicked into the handleToggle handler from App.js
    const handleClick = (e) => {
      e.preventDefault()
      handleToggle(e.currentTarget.id)
    }


  return (
    // Checks if task is completed using "complete" param in data.js. If true, it uses strike as classname, if false, is assigned noStrike as classname
    //When the task is clicked, it calls the handleClick handler
    <div id={todo._id} key={todo.id +todo.task} name ="todo" value={todo._id} onClick={handleClick} className={todo.complete ? classes.strike : classes.noStrike}>
      {/* Gets individual task created in toDoList mapping function. Uses "Task" from data.json as parameter */}
      {todo.task}
    </div>
  );
};

export default ToDo;
```

Figure 40 – Render a task.

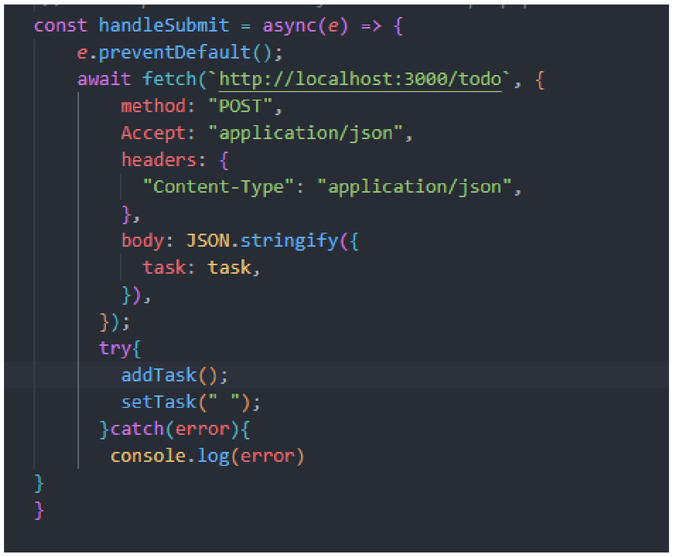
A toDo class was made defined with 2 props, todo and handleToggle. In the return statement, a component is created containing props. The important ones include the onClick handler, which will call the handleClick function, which will pass the id of the task into the handleToggle prop. The className prop checks the “complete” parameter of the task. If it’s completed, the classname will be set to the CSS module that puts a strikethrough through the task. If not, the task will be set to the CSS module that doesn’t put a strikethrough through the task. Inside of the div it renders the task.



```
const ToDoList = ({toDoList, handleToggle, handleFilter}) => {
    return (
        <div>
            {/* Mapping over toDoList prop to create individual todos after it is called */}
            {toDoList.map(todo => {
                return (
                    <ToDo todo={todo} handleToggle={handleToggle} />
                )
            })}
            {/* If button is pressed, it calls the handleFilter function passed down as a prop */}
             <button className={classes.delete}style={{margin: '20px'}} onClick={handleFilter}>Clear Completed</button>
        </div>
    );
};
```

Figure 41 – Render a list of tasks.

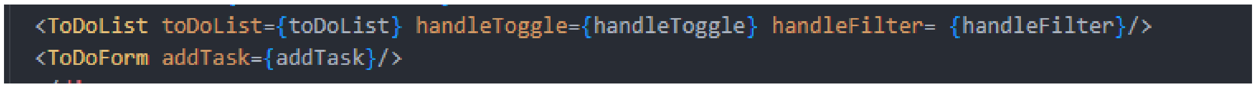
The toDo component is used inside of the ToDoList component, along with its props. The mapping function maps over the todolist that is assigned to the todolist prop, rendering a task every time it iterates. The task read from the todolist is assigned to the todo prop, the handleToggle prop declared in the ToDoList component is assigned to the handleToggle prop that exists in ToDo. When the “Clear Completed” button is clicked, it calls on the handleFilter prop defined in the component.



```
const handleSubmit = async(e) => {
    e.preventDefault();
    await fetch(`http://localhost:3000/todo`, {
        method: "POST",
        Accept: "application/json",
        headers: {
          "Content-Type": "application/json",
        },
        body: JSON.stringify({
          task: task,
        }),
      });
      try{
        addTask();
        setTask(" ");
      }catch(error){
       console.log(error)
}
}
```

Figure 42 – POST request to create task.

The ToDoForm component that contains an addTask prop. A usestate defaulted to an empty string is defined. In the handleSubmit function, a POST request to the route is made, taking in the task usestate variable as a parameter. The addTask function passed down as a prop to the component is called.



```
<ToDoList toDoList={toDoList} handleToggle={handleToggle} handleFilter= {handleFilter}/>
<ToDoForm addTask={addTask}/>
```

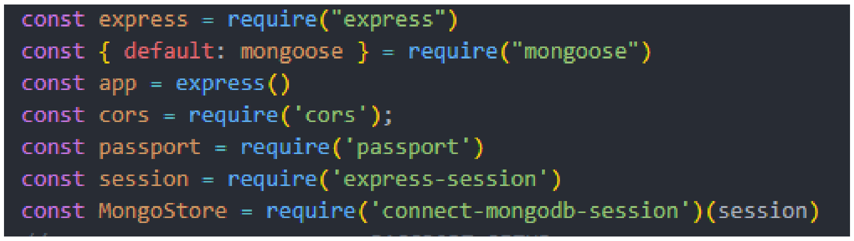
Figure 43 – Components with props

In the App.js file, the TodoList component is passed in the todolist usestate variable, the handletoggle handler and the handleFilter handler are passed down as props into the component. Any call to the props passed into the component call the handlers in the component call the functions in the main app code. In the ToDoForm component, the addTask handler is passed in as a prop to the component, following the same rules as the ToDoList component.

8 Backend [12]

8.1 Structure

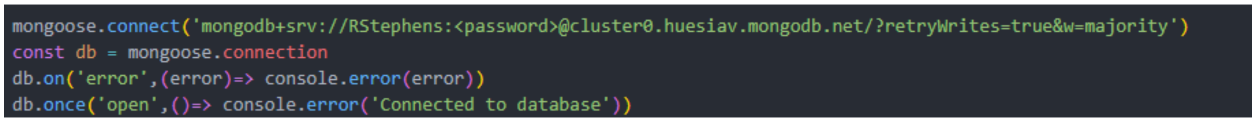
In order for any changes made on the front-end to persist when the user leaves the page or when the page refreshes, a database is needed to store all relevant data.



```
const express = require("express")
const { default: mongoose } = require("mongoose")
const app = express()
const cors = require('cors');
const passport = require('passport')
const session = require('express-session')
const MongoStore = require('connect-mongodb-session')(session)
```

Figure 44 - Required libraries.

The server.js file is where the setup for the server is located. The express library allows the use of express middleware, Mongoose enables the action connecting to a database, cors [13], which stands for Cross-Origin Resource Sharing, allows requests from the browser to the server, which both have different domains, express-session is used for creating a user session cookie, usually used for authentication, and connect-mongodb-session is included as a part of the express session parameters so that the session can be saved to the MongoDB database.



```
mongoose.connect('mongodb+srv://RStephens:<password>@cluster0.huesiav.mongodb.net/?retryWrites=true&w=majority')
const db = mongoose.connection
db.on('error',(error)=> console.error(error))
db.once('open',()=> console.error('Connected to database'))
```

Figure 45 – Database connection

The database is hosted on mongo atlas, which provides a link where the developer can use to connect to their database. If the database fails to connect, an error message is sent in the console, but if the database connects successfully, a success message is sent in the console.

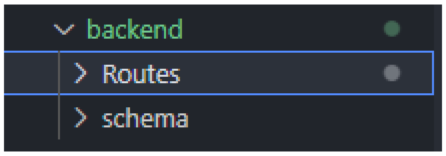
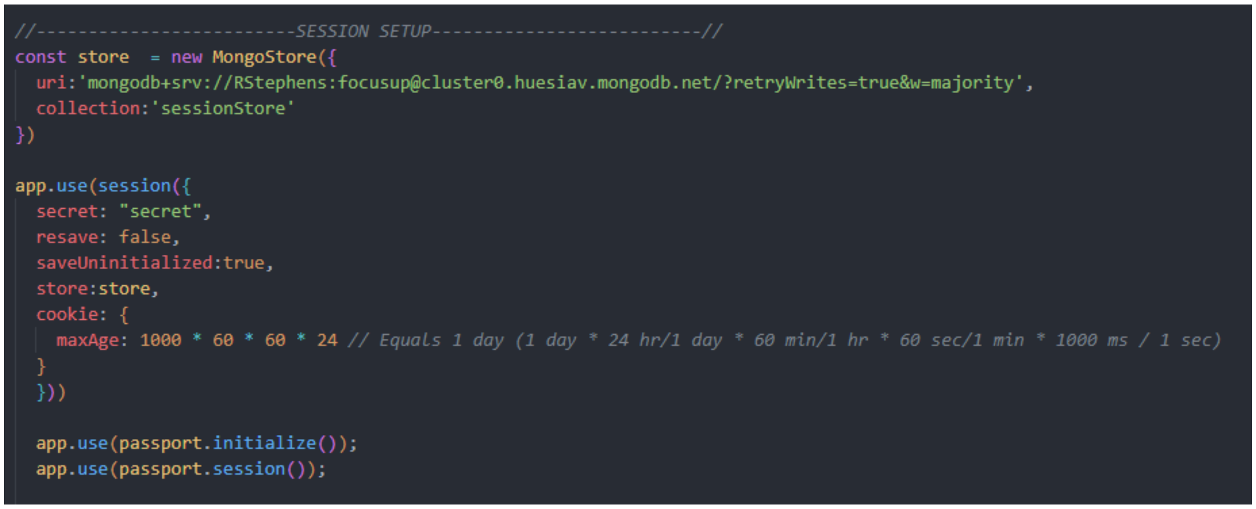


Figure 46 – Backend setup

The backend functions are in the backend folder. The routes folder contains all the routing code, and the schema folder contains all the collections which are tables where the information is stored.



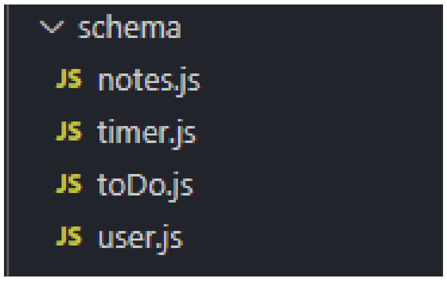
```
//------------------------SESSION SETUP-------------------------//
const store  = new MongoStore({
  uri:'mongodb+srv://RStephens:focusup@cluster0.huesiav.mongodb.net/?retryWrites=true&w=majority',
  collection:'sessionStore'
})

app.use(session({
  secret: "secret",
  resave: false,
  saveUninitialized:true,
  store:store,
  cookie: {
    maxAge: 1000 * 60 * 60 * 24 // Equals 1 day (1 day * 24 hr/1 day * 60 min/1 hr * 60 sec/1 min * 1000 ms / 1 sec)
  }
  }))

  app.use(passport.initialize());
  app.use(passport.session());
```

Figure 47

Session management is set up using the express-session library. When a session is created and attached to a user, every request is attached with the specified session for the user. The session contains a secret key, a resave option, a saveUninitialized option, a session store, in this case the session is stored in a mongoDB collection, and a maximum cookie age, which is set to 1 day.

The passport.initialize function is used to initialize passport for the application, and the passport.session function allows session based authentication.

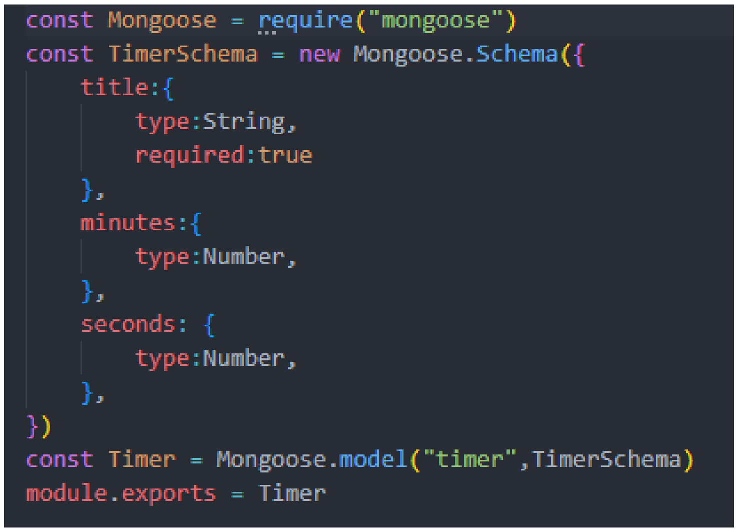
8.2 Schemas



```
schema
  notes.js
  timer.js
  toDo.js
  user.js
```

Figure 48 – Schema structure

There are 4 schemas set up in the project, each of them representing a structure for an object in its respective collection.

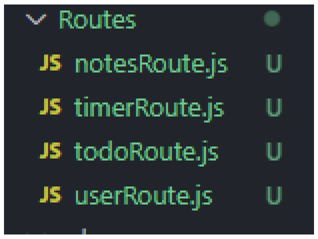


```
const Mongoose = require("mongoose")
const TimerSchema = new Mongoose.Schema({
    title:{
        type:String,
        required:true
    },
    minutes:{
        type:Number,
    },
    seconds: {
        type:Number,
    },
})
const Timer = Mongoose.model("timer",TimerSchema)
module.exports = Timer
```

Figure 49 – Schema setup

Here is timer schema as an example of how they are set up.” Mongoose.Schema” is what creates the schema, and inside its parameters are defined. Each parameter is given an expected type, in this example, the title is attached with the required parameter, and it is set to true, meaning that if there’s a case where information needs to be entered relating to this schema, the title is not allowed to be blank.

8.3 Requests and Routing

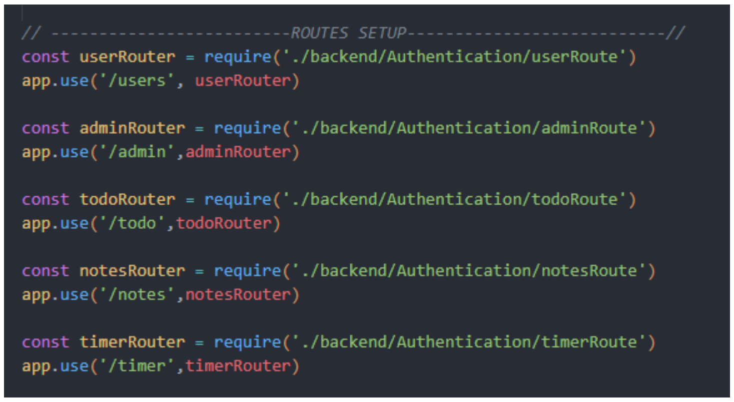


```
Routes
  notesRoute.js   U
  timerRoute.js   U
  todoRoute.js    U
  userRoute.js    U
```

Figure 50 – Routes structure

Every request that is sent to the server will always be directed to a specific route. Routes are needed to filter and sort where the requests are headed, else the server won't know what to do with the request it's given.

All the routes are imported into the server.js file. The set of routes in each file path are assigned to a variable that holds them. This variable is passed in to the app.use() method as a reference to the set of routes. The app.use() method also defines a path, where all relevant requests will be directed to.

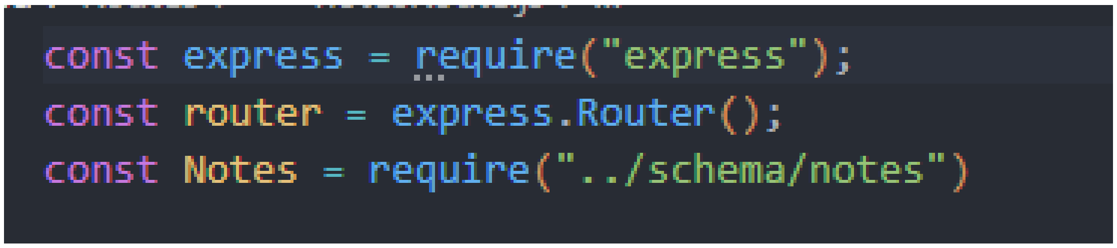


```
// -------------------------ROUTES SETUP---------------------------//
const userRouter = require('./backend/Authentication/userRoute')
app.use('/users', userRouter)

const adminRouter = require('./backend/Authentication/adminRoute')
app.use('/admin',adminRouter)

const todoRouter = require('./backend/Authentication/todoRoute')
app.use('/todo',todoRouter)

const notesRouter = require('./backend/Authentication/notesRoute')
app.use('/notes',notesRouter)

const timerRouter = require('./backend/Authentication/timerRoute')
app.use('/timer',timerRouter)
```

Figure 51 – Creating default routes.

There are 5 different types of requests that can be made, GET, POST, PATCH, PUT, and DELETE, all of which are designed for specific purposes. GET is used if you want to fetch data from the database. POST is used if you want to send a body of data to the database, where you can perform an array of functions on. PATCH is used if you want to alter a parameter in a database object. PUT is used if you want to replace the entire database object with what is entered in the body. Lastly, DELETE is used if you want to delete a database object from the collection. It is usually best practice to only use each request for its intended purpose, as it boasts increased readability when debugging and not doing so provides potential security risks.

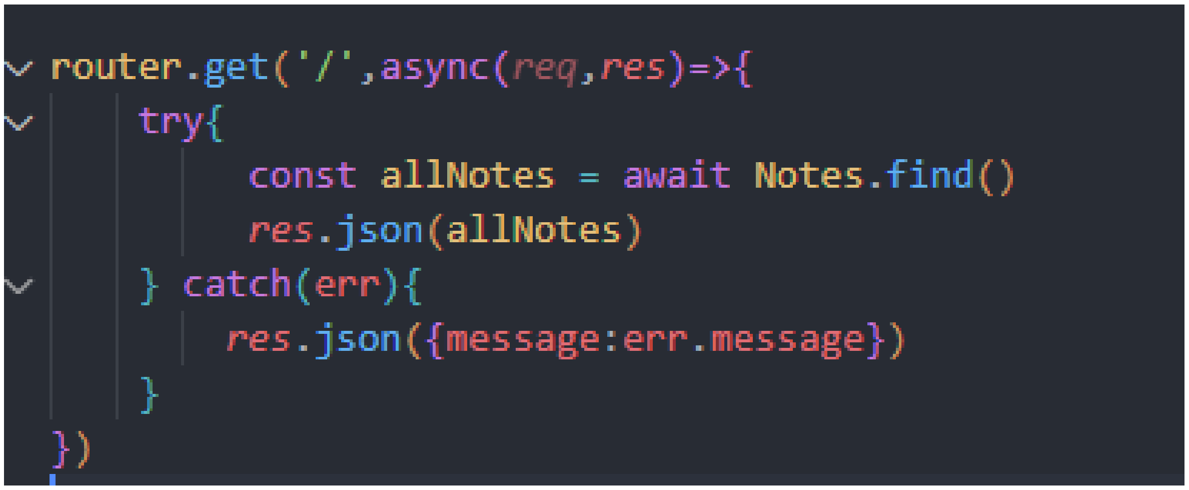
8.3.1 Notes



```
const express = require("express");
const router = express.Router();
const Notes = require("../schema/notes")
```

Figure 52 – Required libraries and files.

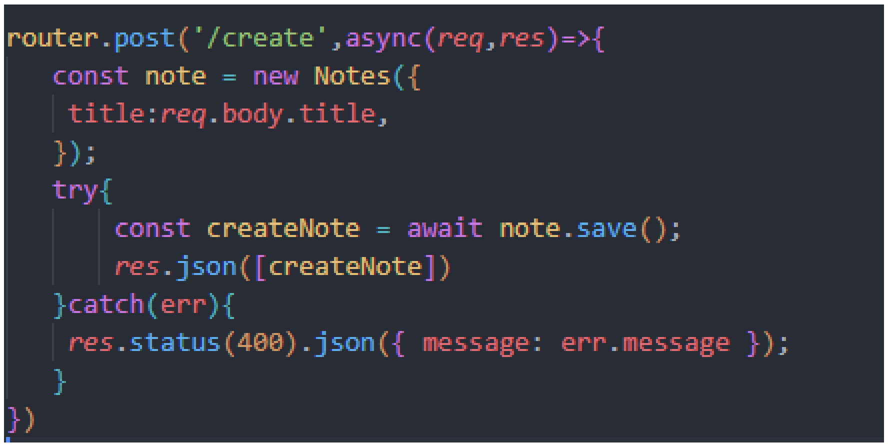
Every file used for routing imports express so that it can use express middleware, the express.Router() method which is used to handle requests, and their respective schema which contains the parameters for each database object.



```
router.get('/',async(req,res)=>{
    try{
        const allNotes = await Notes.find()
        res.json(allNotes)
    } catch(err){
        res.json({message:err.message})
    }
})
```

Figure 53 – GET request to fetch all.

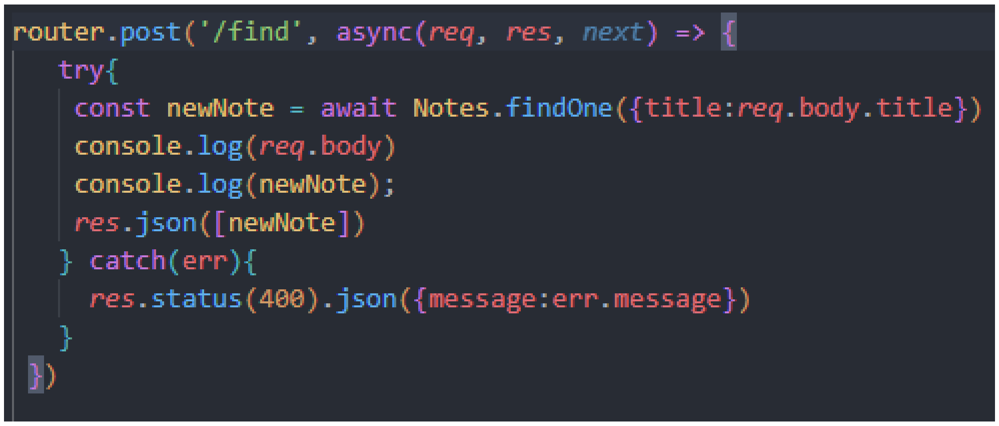
The first request defined is the get request. The first parameter of the function is the route that the request will be sent to, so every get request must be sent to the "/notes" path, as the slash with nothing else defines its default path. The second parameter handles what to do when the request is made. Since the methods performed on a schema return a promise, its good practice to wrap them in an asynchronous function because results aren't guaranteed to be immediate. This also allows you to handle the result without blocking any other code running. The await keyword used here means that the code will only move to the next line after it gets a result. In this function, the .find() method is performed on the Notes schema, which will return all objects inside of the collection.



```
router.post('/create',async(req,res)=>{
   const note = new Notes({
    title:req.body.title,
   });
   try{
       const createNote = await note.save();
       res.json([createNote])
   }catch(err){
    res.status(400).json({ message: err.message });
   }
})
```

In this POST request, the first parameter now contains a name beside the slash, representing another path in the route. This means that to send a post request with this specific function, they must be sent to the "/notes/create" path.

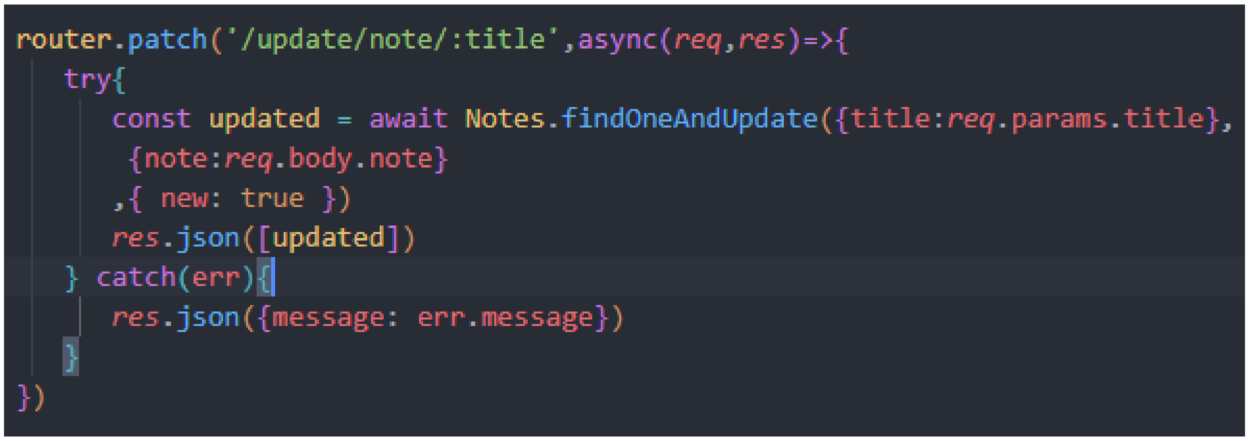
This post request creates a new note object that is saved to the database. A new instance of the Notes schema is made, which only asks for the title parameter in the body of the request. There is another parameter in the schema, but since it is not marked as required it is not always needed when creating a note object. The .save() method is only able to be performed on the instance of a model which is why a new one was made. In this case it is performed on the note instance, which saves the body of the request in the database as a new object.



```
router.post('/find', async(req, res, next) => {
   try{
    const newNote = await Notes.findOne({title:req.body.title})
    console.log(req.body)
    console.log(newNote);
    res.json([newNote])
   } catch(err){
     res.status(400).json({message:err.message})
   }
 })
```

Figure 54 – POST request to find topic.

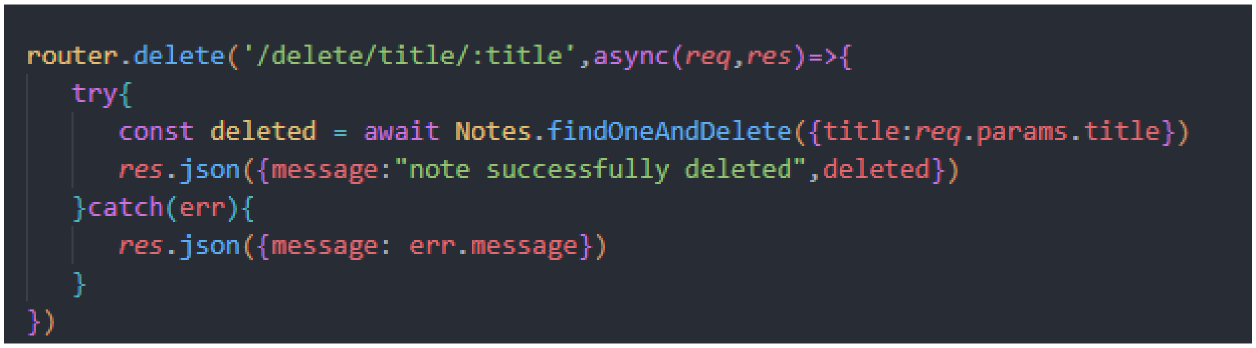
This post request is a search function for finding a title name. The .findOne() method is performed on the Notes schema which requests the title to be searched for in the body. This method can be done on the model itself because it's not an instance method.



```
router.patch('/update/note/:title',async(req,res)=>{
    try{
      const updated = await Notes.findOneAndUpdate({title:req.params.title},
       {note:req.body.note}
      ,{ new: true })
      res.json([updated])
    } catch(err){
      res.json({message: err.message})
    }
})
```

Figure 55 – PATCH request to change note.

In this patch requests path parameter, there is a colon beside the title path. This means that the route is looking for a title parameter to be entered or passed into the URL to perform methods on. The findOneAndUpdate() method takes 2 parameters, the first parameter is finding the object based on its title, and the second parameter is editing the note parameter of the found object.

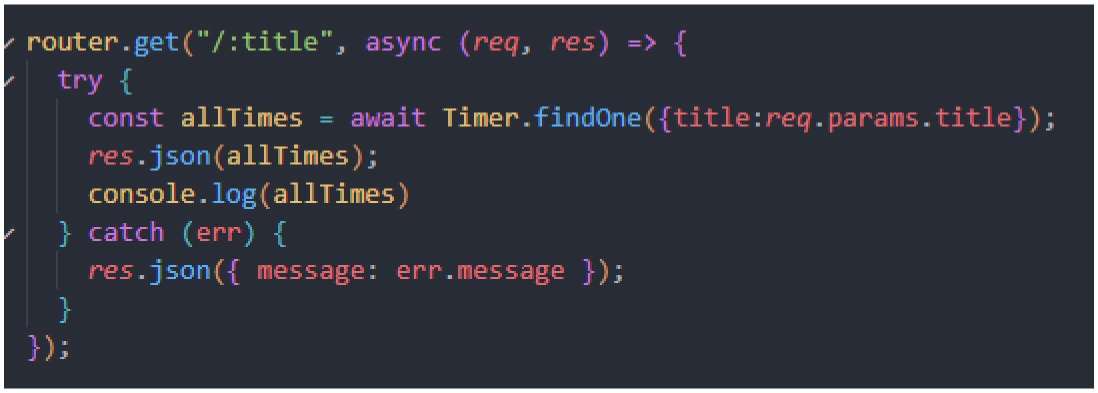


```
router.delete('/delete/title/:title',async(req,res)=>{
    try{
      const deleted = await Notes.findOneAndDelete({title:req.params.title})
      res.json({message:"note successfully deleted",deleted})
    }catch(err){
      res.json({message: err.message})
    }
})
```

Figure 56 – DELETE request to delete topic.

In this DELETE request, the findOneAndDelete() method is performed on the schema. It looks for the name of the title passed into the title parameter specific path and deletes it. This method only requires one argument.

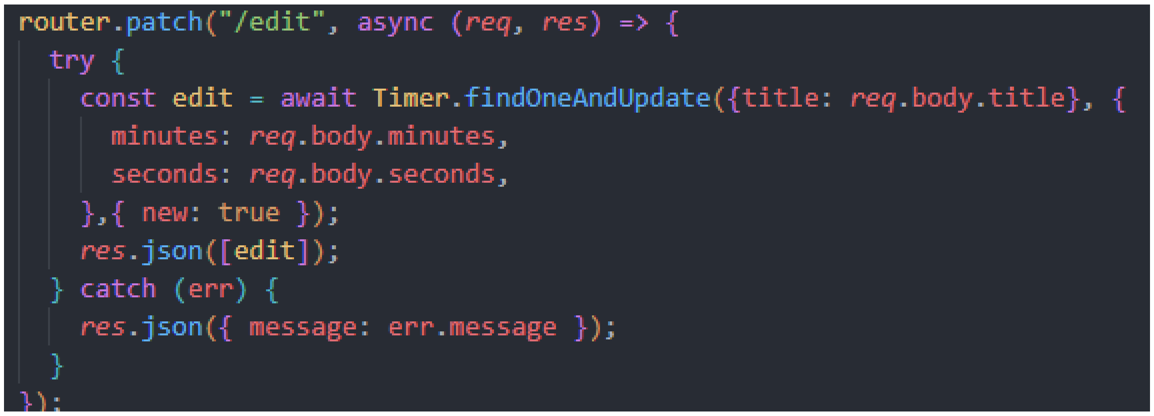
8.3.2 Timer



```
router.get("/:title", async (req, res) => {
  try {
    const allTimes = await Timer.findOne({title:req.params.title});
    res.json(allTimes);
    console.log(allTimes)
  } catch (err) {
    res.json({ message: err.message });
  }
});
```

Figure 57 – GET request to find by title.

The GET request for this route is looking for a title parameter to be passed into its path. Using the findOne() method the object of the passed in title will be found. The titles have already been set up and cannot be changed. These titles are "Work" and "Break".

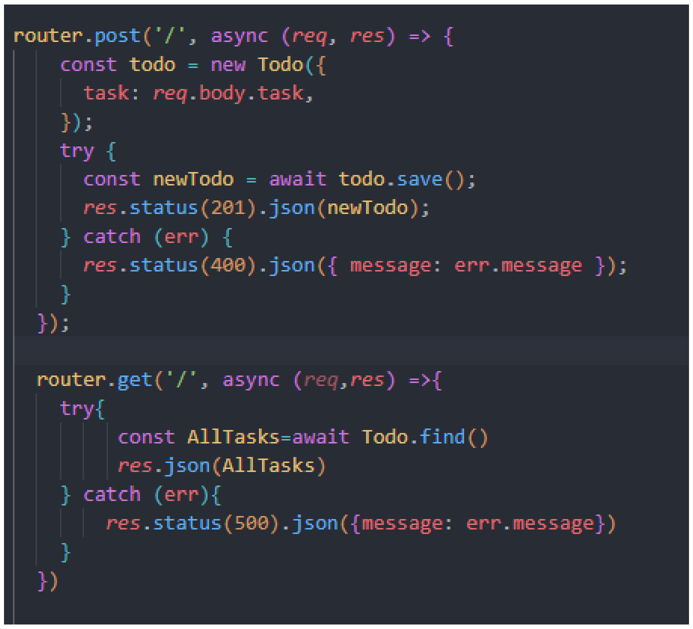


```
router.patch("/edit", async (req, res) => {
  try {
    const edit = await Timer.findOneAndUpdate({title: req.body.title}, {
      minutes: req.body.minutes,
      seconds: req.body.seconds,
    },{ new: true });
    res.json([edit]);
  } catch (err) {
    res.json({ message: err.message });
  }
});
```

Figure 58 – PATCH request to edit timer.

In the PATCH request, the findOneAndUpdate() method is performed on the Timer model. It first looks for the title of the object as a body to the request, and then updates the minutes and seconds to what is entered as a body to the request.

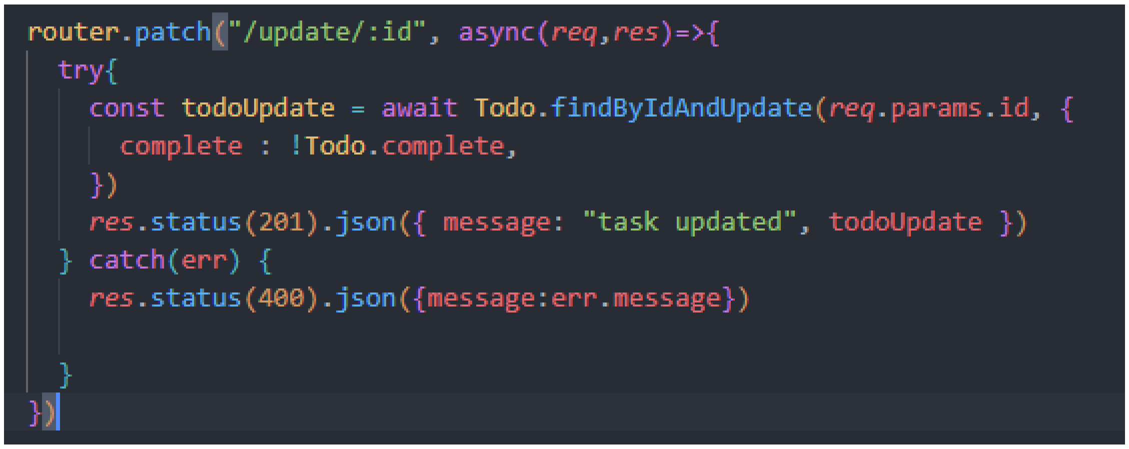
8.3.2 To-do List



```
router.post('/', async (req, res) => {
    const todo = new Todo({
      task: req.body.task,
    });
    try {
      const newTodo = await todo.save();
      res.status(201).json(newTodo);
    } catch (err) {
      res.status(400).json({ message: err.message });
    }
  });

  router.get('/', async (req,res) =>{
    try{
        const AllTasks=await Todo.find()
        res.json(AllTasks)
    } catch (err){
        res.status(500).json({message: err.message})
    }
  })
```

Figure 59 – POST and GET requests to create and fetch all.

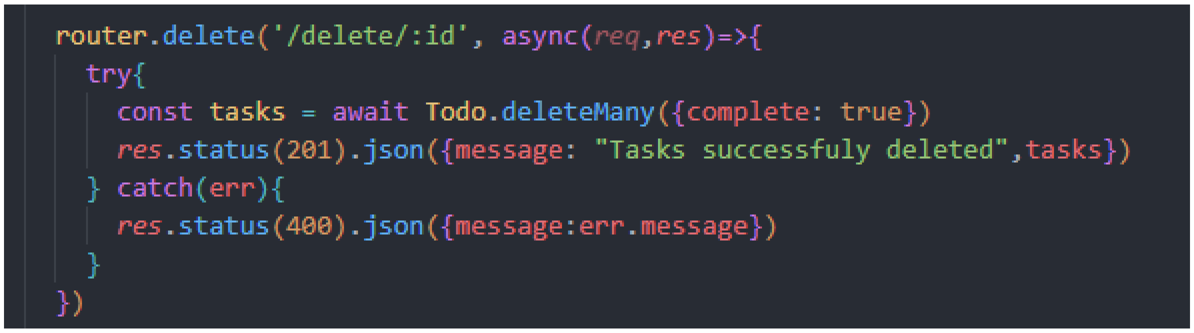
The POST and GET requests are something we've seen before and are very standard. The POST request adds a new Todo object with a requested task to the database, and the GET request returns all the ToDo objects stored in the database. In the schema there is a “complete” parameter that is set to a Boolean defaulted at false, so every new ToDo object created is defaulted to not being completed.



```
router.patch("/update/:id", async(req,res)=>{
  try{
    const todoUpdate = await Todo.findByIdAndUpdate(req.params.id, {
      complete : !Todo.complete,
    })
    res.status(201).json({ message: "task updated", todoUpdate })
  } catch(err) {
    res.status(400).json({message:err.message})

  }
})
```

Figure 60 – PATCH request to edit state by id.

In the PATCH request, the path is waiting for the object id to be passed into it. Using the findByIdAndUpdate() method, the object id is searched for, and the complete parameter of the found id is set to the opposite of its current value.



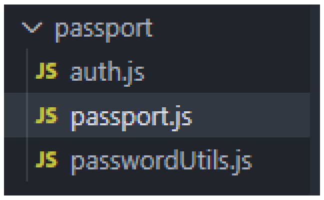
```
router.delete('/delete/:id', async(req,res)=>{
  try{
    const tasks = await Todo.deleteMany({complete: true})
    res.status(201).json({message: "Tasks successfuly deleted",tasks})
  } catch(err){
    res.status(400).json({message:err.message})
  }
})
```

Figure 61 – DELETE request to delete completed tasks.

The DELETE request also looks for a passed in object id. The deleteMany() method is performed on the Todo model, which looks for any object that has a complete parameter of true and deletes all of them rather than one.

8.3.3 User Authentication [14]

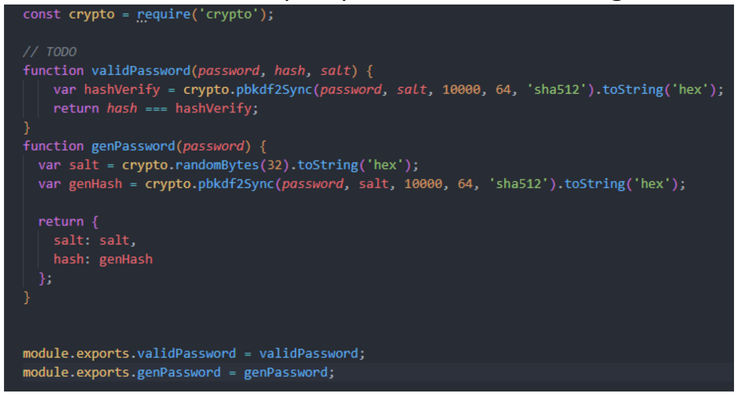
To implement user authentication, Passport.js was used. Passport.js is an authentication middleware used in Node.js used by the widely general public.



```
passport
  auth.js
  passport.js
  passwordUtils.js
```

Figure 62 – Passport Structure

The passport middleware is stored in the passport folder containing 3 files.

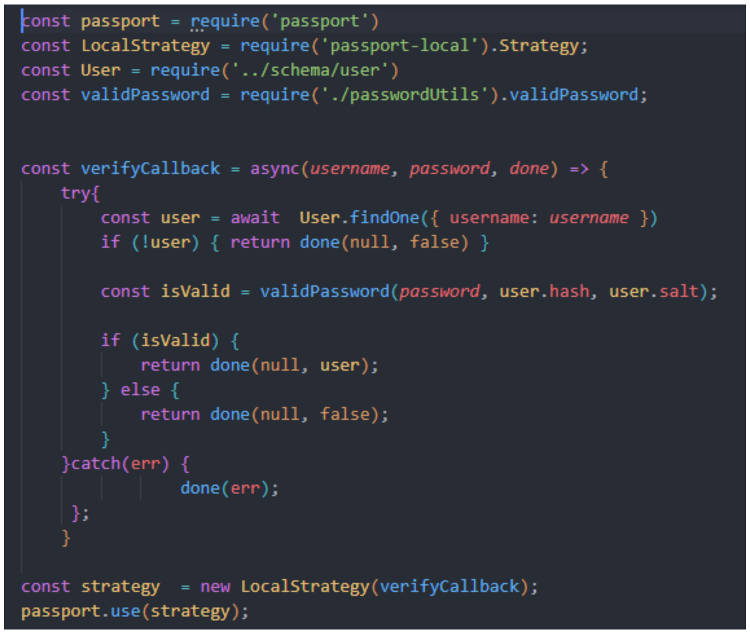


```
const crypto = require('crypto');

// TODO
function validPassword(password, hash, salt) {
    var hashVerify = crypto.pbkdf2Sync(password, salt, 10000, 64, 'sha512').toString('hex');
    return hash === hashVerify;
}
function genPassword(password) {
  var salt = crypto.randomBytes(32).toString('hex');
  var genHash = crypto.pbkdf2Sync(password, salt, 10000, 64, 'sha512').toString('hex');

  return {
    salt: salt,
    hash: genHash
  };
}


module.exports.validPassword = validPassword;
module.exports.genPassword = genPassword;
```

Figure 63 – Password encryption

In the passportUtils file, a password passed in is hashed, meaning that it gets encrypted so that potential hackers cannot see the sensitive information. The crypto library is imported, which is a library for password encryption. The genPassword() function accepts a passed in password as an argument to the function. A 32-bit salt value is created, then converted to a hexadecimal string so that it can be used in cryptographic functions. Next, a hash is created using a hashing function. The password parameter serves as the password the user enters, the salt value generated is passed in which is attached to the password, then the password is hashed 10,000 times (the higher the number, the more secure it is). The result is turned into a string. The function returns an object that contains a salt and hash parameter.

The validPassword function tests to see if the entered password matches the user's password in the database when logging in. The password entered by the user, the generated salt and generated hash are passed into the function. The entered password is hashed, then compared to the existing hashed password to see if it matches, which returns a true or false.

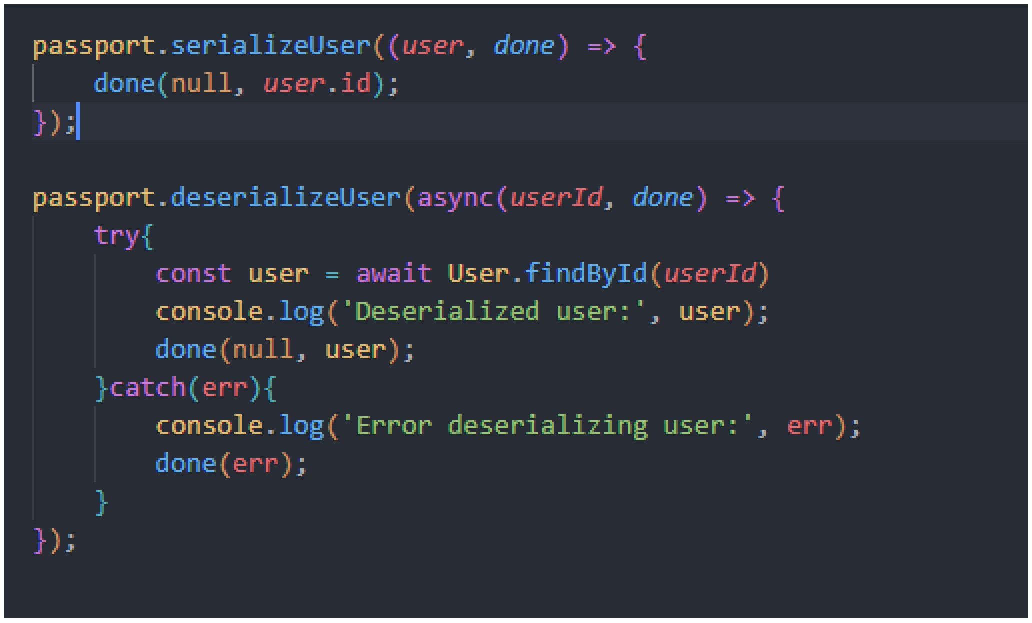


```
const passport = require('passport')
const LocalStrategy = require('passport-local').Strategy;
const User = require('../schema/user')
const validPassword = require('./passwordUtils').validPassword;


const verifyCallback = async(username, password, done) => {
    try{
        const user = await  User.findOne({ username: username })
        if (!user) { return done(null, false) }

        const isValid = validPassword(password, user.hash, user.salt);

        if (isValid) {
            return done(null, user);
        } else {
            return done(null, false);
        }
    }catch(err) {
                done(err);
     };
    }

const strategy  = new LocalStrategy(verifyCallback);
passport.use(strategy);
```

Figure 64 – Local strategy

In the passport.js file, the passport local strategy is created, which is the basic form of authentication, requiring a username and a password. When the user logs in, the local strategy is called, which takes the inputted username and password and passes them into the verifyCallback function. In this function, the server attempts to find the entered username if it exists in the database. If they don't exist, the server returns a false value to the done function, the first parameter indicates if there's an error or not, and the second parameter indicates if authentication has passed or failed. If the user does exist, a variable is assigned to the validPassword function, which is imported. The inputted password is passed in, as well as the users existing salt and hash values. This function will assign the variable with a true or false value. If the passwords match, the user object is passed into the done function, indicating that authentication was successful, else a false value will be passed to the done function.

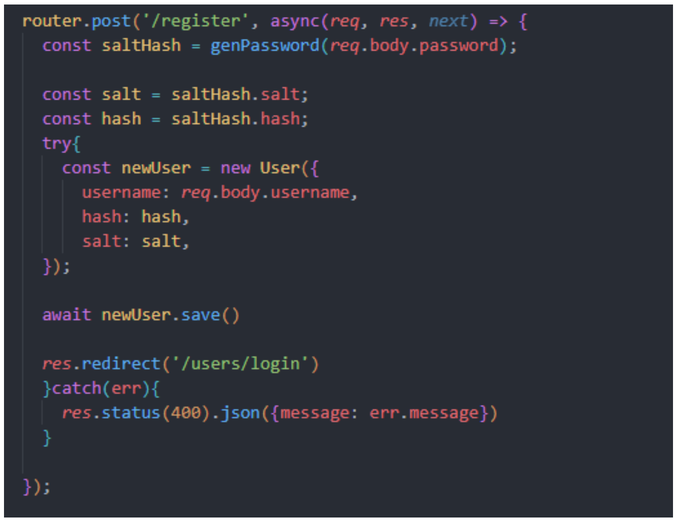
The verifyCallback function is passed into a new instance of the imported LocalStrategy function, then passport will use the local strategy when the user attempts to login, calling the verifyCallback function.



```
passport.serializeUser((user, done) => {
    done(null, user.id);
});

passport.deserializeUser(async(userId, done) => {
    try{
        const user = await User.findById(userId)
        console.log('Deserialized user:', user);
        done(null, user);
    }catch(err){
        console.log('Error deserializing user:', err);
        done(err);
    }
});
```

Figure 65 – Serialization and deserialization

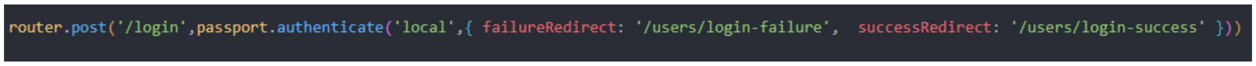
The serializeUser function grabs the user from the database if authenticated and grabs the id of the user and stores it in the session. The deserializeUser function gets the id of the user stored in the session and matches it to a user's id in the database, the function returns the object of the user found. These two functions are primarily used for session management.



```
router.post('/register', async(req, res, next) => {
  const saltHash = genPassword(req.body.password);

  const salt = saltHash.salt;
  const hash = saltHash.hash;
  try{
    const newUser = new User({
      username: req.body.username,
      hash: hash,
      salt: salt,
  });

  await newUser.save()

  res.redirect('/users/login')
  }catch(err){
    res.status(400).json({message: err.message})
  }

});
```

Figure 66 – Register POST route.

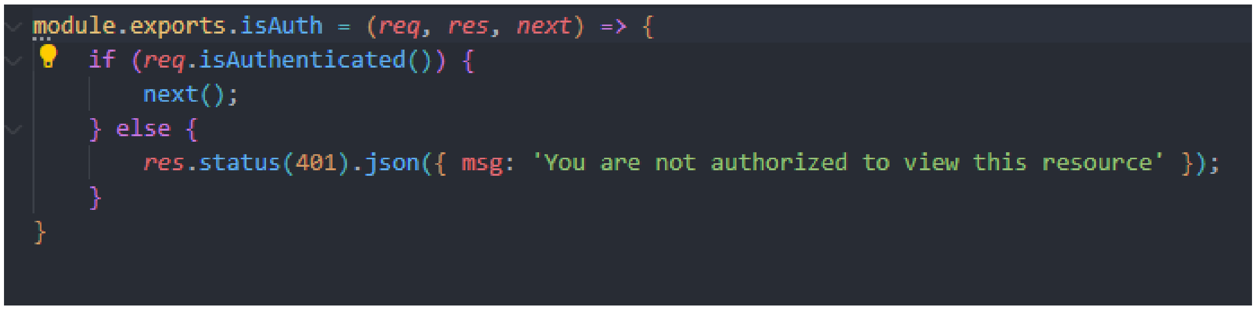
In the user route, there is a POST request for the registration of a user. When the user enters a username and a password, the password is passed into the genPassword function imported from the passportUtils class and assigned to a variable. The salt and hash generated and assigned to 2 different variables, which are used as object parameters in the new instance of the User model. The user is then saved to the database using the .save method. The user is then redirected to the login route in the server, which was used for testing.



```
router.post('/login',passport.authenticate('local',{ failureRedirect: '/users/login-failure',  successRedirect: '/users/login-success' }))
```

Figure 67- Login POST route

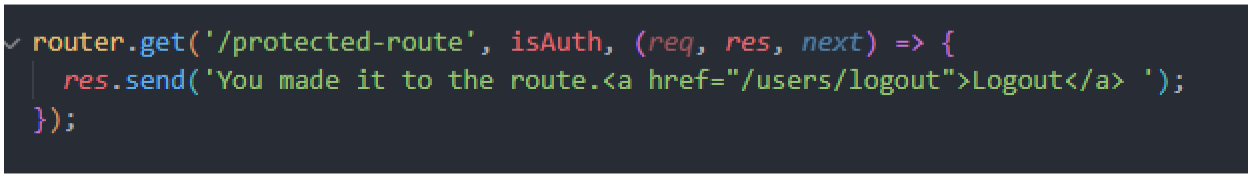
The login POST request uses the passport.authenticate function to authenticate its users. The first parameter indicates the strategy to be used, in this case the local strategy. Passport will look for the local strategy to invoke which in turn will call on the verifyCallback function. The second parameter represents the behaviour of the function. If authentication succeeds, the user is sent to the login-success route in the backend, if not, they are sent to the login-failure route.



```
module.exports.isAuth = (req, res, next) => {
    if (req.isAuthenticated()) {
        next();
    } else {
        res.status(401).json({ msg: 'You are not authorized to view this resource' });
    }
}
```

Figure 68 – Authentication checking

In the auth.js file, a function called isAuth is created. This checks if the user is authenticated before allowing them to access a route. If they are authenticated, the program moves on to the next piece of middleware, but if not, an error message is shown.



```
router.get('/protected-route', isAuth, (req, res, next) => {
  res.send('You made it to the route.<a href="/users/logout">Logout</a> ');
});
```

Figure 69 – Protected route

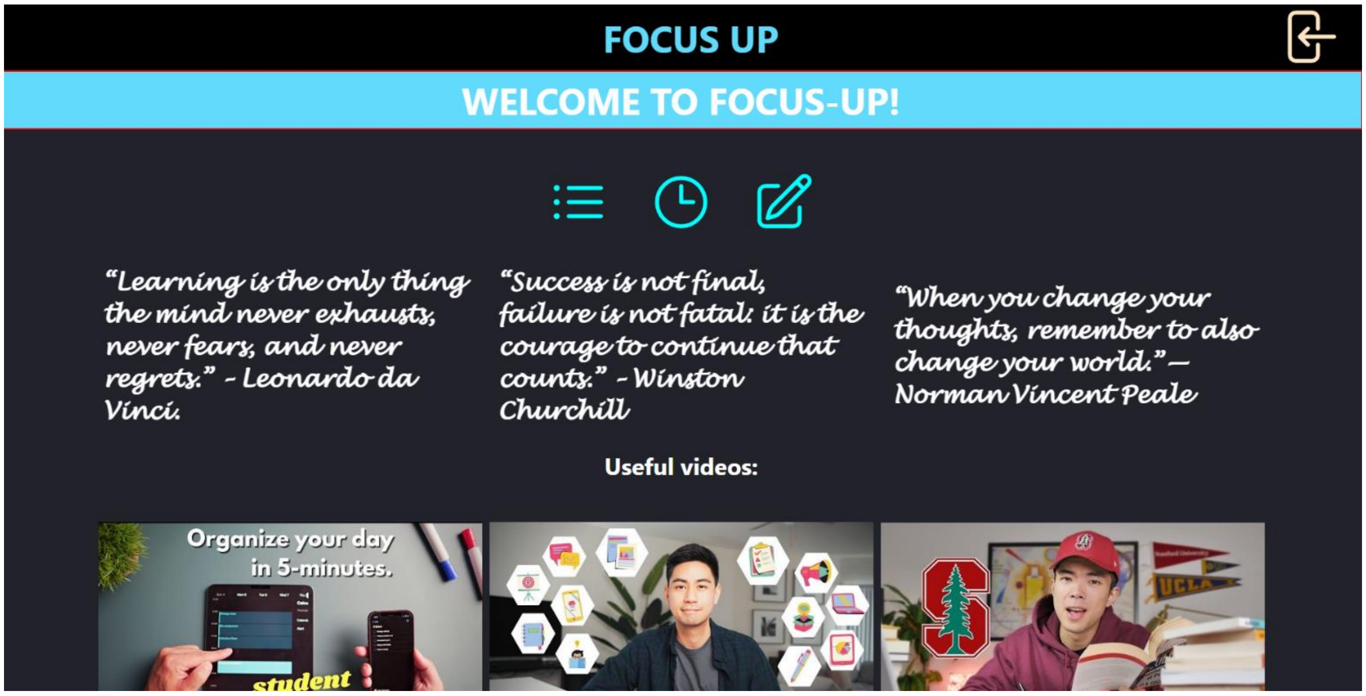
Back in the user route, the isAuth function is imported into the file and passed in as a parameter to the GET request of this route. If the user is authenticated, the user will be able to access the route, else, they will receive the error message defined in the isAuth function.

7 Ethics

Include a short section on ethical considerations in your project or in the field of study of your project.

10 Conclusion

In conclusion, I feel that I achieved a lot of the goals that I had set out to do. I was able to tackle the challenge of learning new languages and creating a functioning project which in turn has improved my knowledge in full stack development greatly. As well as that I have now become more familiar with project planning and version control using JIRA and GitHub respectively.



FOCUS UP

WELCOME TO FOCUS-UP!

"Learning is the only thing the mind never exhausts, never fears, and never regrets." - Leonardo da Vinci.

"Success is not final, failure is not fatal: it is the courage to continue that counts." - Winston Churchill

"When you change your thoughts, remember to also change your world." — Norman Vincent Peale

Useful videos:

Figure 70 – Front Page

This project could use some improvements and further development. I plan to further work on this project after the semester to further my understanding of the languages I'm learning.

11 References

[1] D. S. Gardón, "SnipCart," [Online]. Available: https://snipcart.com/blog/next-js-vs-react. [Accessed 17 April 2023].

[2] T. Sufiryan, "simplilearn," [Online]. Available: https://www.simplilearn.com/tutorials/nodejs-tutorial/what-is-nodejs. [Accessed 17 April 2023].

[3] D. Taylor, "Guru99," [Online]. Available: https://www.guru99.com/what-is-mongodb.html. [Accessed 17 April 2023].

[4] Express, "Express," [Online]. Available: https://expressjs.com/en/guide/using-middleware.html#middleware.router. [Accessed 17 April 2023].

[5] Academind, "Youtube," [Online]. Available: https://www.youtube.com/watch?v=MFuwkrseXVE&t=1203s. [Accessed 23 october 2022].

[6] W. D. Simplified, "Youtube," [Online]. Available: https://www.youtube.com/watch?v=fgTGADljAeg. [Accessed 24 January 2023].

[7] Academind, "Youtube," [Online]. Available: https://youtu.be/Dorf8i6lCuk?t=4368. [Accessed 1 November 2022].

[8] A. Popovic, "Youtube," [Online]. Available: https://www.youtube.com/watch?v=9z1qBcFwdXg. [Accessed 6 November 2022].

[9] T. N. Ninja, "Youtube," [Online]. Available: https://youtu.be/WPdJaBFquNc. [Accessed 5 march 2023].

[10] Pankaj_Singh, "Geeksforgeeks," [Online]. Available: https://www.geeksforgeeks.org/javascript-array-map-method/. [Accessed 27 February 2023].

[11] C. Kopecky, "educative," [Online]. Available: https://www.educative.io/blog/react-hooks-tutorial-todo-list. [Accessed 1 November 2022].

[12] W. d. Simplified, "Youtube," [Online]. Available: https://www.youtube.com/watch?v=fgTGADljAeg. [Accessed 24 January 2023].

[13] npm. [Online]. Available: https://www.npmjs.com/package/cors. [Accessed 15 March 2023].

[14] freeCodeCamp, "Youtube," [Online]. Available: https://www.youtube.com/watch?v=F-sFp_AvHc8. [Accessed 27 March 2023].