



Ollscoil
Teicneolaíochta
an Atlantaigh

Atlantic
Technological
University

Project Engineering

Year 4

Johnson Shogbaike

Bachelor of Engineering (Honours) in Software
and Electronic Engineering

Atlantic Technological University

2023/2024

Declaration

The purpose of this report is to present a comprehensive analysis of the STAR application. It will go over inspiration, research and making of the application, as well as the features it provides and potential future development.

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 otherwise accredited. Where the work of others has been used or incorporated during this project, this will be acknowledged and referenced.

- Johnson Shogbaike

Acknowledgements

I would like to show my appreciation to Michelle Lynch, Paul Lennon, David Newman, Niall O'Keefe and Brian Costello for the help and guidance that they provided me with throughout this year when it came to this project.

I would also like to thank Enoch Abiodun and Connor Ngouanna. This year, I worked on group projects with those two and those projects provided me with insights that helped me in the making of this application.

Table of Contents

Project Engineering	1
Year 4	1
Johnson Shogbaike	1
Bachelor of Engineering (Honours) in Software and Electronic Engineering	1
Atlantic Technological University	1
2023/2024	1
Declaration	2
Acknowledgements	3
Summary	6
1. Introduction	6
2. Poster	8
3. Project Architecture	9
4. Background and Research:	10
Background research:	Error! Bookmark not defined.
5. Technologies Used:	11
Front-End Application:	11
Backend:	11
Database:	11
6. Features	12
7. Planning	12
8. Tasks	13
8.1. Models	13
8.2. Components	14
8.2.1. TodoForm	14
8.2.2. TodoProgress	16
8.2.3. Status	16
8.2.4. TodoPriority	18
8.2.5. DeleteBlock	18
8.3. API Routes	19
8.3.1. Edit Route	19
8.3.2. Create Task Route	19
8.3.3. Delete Task Route	20
9. Users	21

9.1.	Models	21
9.2.	Components.....	21
9.2.1.	SignupForm	21
9.2.2.	LoginForm	23
9.3.	API Routes	24
9.3.1.	Check User Existence	24
9.3.2.	Create User	24
9.3.3.	Login.....	25
10.	Difficulties Faced and Problem Solving	29
10.1.	Learning Curve:.....	29
10.2.	Front-end Application to Full Stack Application:	29
10.3.	CORS:	29
11.	Ethics.....	30
12.	What I learnt from this project.....	31
13.	Conclusion	32
14.	References	33
15.	Appendix.....	Error! Bookmark not defined.

Summary

The goal of this project is to provide students with a user-friendly platform that allows them to easily track academic tasks and manage their progress. The application allows the user to create, delete and edit tasks, and integrate with the student's timetable.

The aimed completion of the project was 7 months, with the possibility of adding extra features. The project was aimed to have the basic features of a task manager, with the addition of progress managing and timetable integration.

This project was originally planned to be a React-native application (a mobile application), but I decided to keep the project as a web application for a few reasons:

- **Accessibility:** while the same cannot be said for mobile applications, this tool being a web application allows it to be easily accessible to any application with an internet connection.
- **Usage:** ideally, this is a tool that would be used alongside your studies and assignments. Rather than having the app on another device, I thought it would be better to have it all on the same device.
- **Performance:** from my understanding, even though react-native applications offer amazing performance

This report will go over all the technologies used like Next.js and Node.js, and I will go over what I learned from this experience and some further development that I could do with the application.


1. Introduction

Coming back from work placement in third year going straight into my final year, I had a tough time organising my academic tasks efficiently. I was having difficulties balancing life and school adequately, and overall, just had poor management over my scheduling and tracking all the tasks that I need to get done. I was sure that I was not the only one who has had these problems, and I am sure I will not be the last. So, I started to ask the question. "How can I make my school life easier?"

That is when I produced the idea to surround my project around just that. Making school life easier. The Student Task Assistant Resource (STAR) is a tool that allows users to do just that. It is a web application, which facilitates the basic functionality of a task managing application, allows users to monitor the progress of each task and integrate their college timetables into the app.

This report will detail all the planning that went into the creation of this app, the research done, technologies used, and a showcase of the result of all the research and work done.

2. Poster




Ollscoil
Teicneolaíochta
an Atlantaigh

Atlantic
Technological
University

STAR

Student Task Assistant Resource

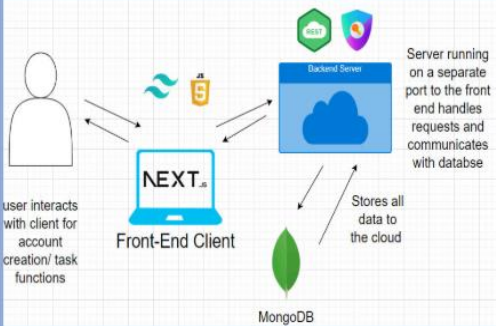


Introduction

When I came back to school life from my work experience, I had a very hard time getting back into the swing of things. I began to ask myself the question "how could I make my school life easier?"

The STAR web app is an application designed for students, and allows for timetable integration into the app. It has the basic features of a task manager, (with task creation, progress features, etc. The app is an all-in-one task monitoring tool that can be utilised in everyday school life.

Architecture Diagram



user interacts with client for account creation/ task functions

Front-End Client




Backend Server

Server running on a separate port to the front end handles requests and communicates with database

MongoDB

Stores all data to the cloud

Results


Features

- Task creation and deletion
- Task categorization
- Login/Signup/Logout/Authentication
- Client to server communication
- Profiles
- Easy-to-use UI

Technologies and skills used

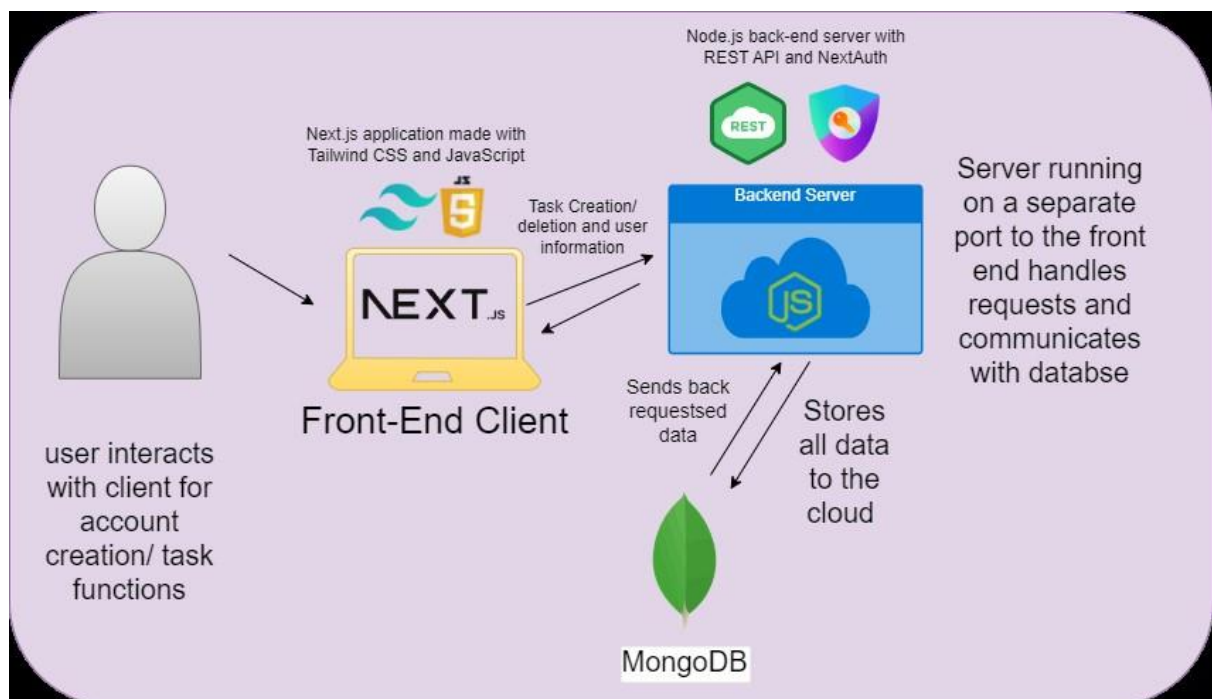
- JavaScript, Tailwind CSS, Next JS
- Node JS, REST
- Express, Database Transactions
- Web Design
- Next Authentication and Cookie Generation
- Time Management
- Project Planning and development
- Navigation

By Johnson
Shogbaiké
BEng in Software and Electronic
Engineering (Honours)



Scan here for code!

3. Project Architecture



4. Background and Research:

Prior to making this application, I conducted so research on what the current market provides to students regarding task management applications. I focused on these main things while conducting this research:

1. What apps out there are like the one I want to create?
2. How do these applications' function?
3. Are these apps aimed at students?
4. What can I do to make my application different from what is already out on the market?

The main applications the drew my attention when conduction this research was Todoist, TickTick and Microsoft ToDo



[1] Todoist is a task management tool that allows you to create digital to-do lists online and organize them into various categories. It also synchronizes across all platforms (phone, tablet, desktop) and a system has been put in place to allow users to easily collaborate with one another.

[2] TickTick is an easy-to-use time management tool that accommodates modules for Time Management, Calendar, Eisenhower Matrix, and Habit Tracker. TickTick is also cross-platform and sends users reminders.

[3] Microsoft ToDo allows users to create task lists and synchronise them across multiple devices so if a user create a task on their laptop, they can view of their tablet, phone, or home desktop. It sets users up for success by helping them manage, prioritize, and complete goals [3].

I have gone through the web searching for task managing apps, but these apps were the closest to what I wanted to create. These three task management tools are amazing, but I felt that they do not truly encompass what my application does, which is the aim to make this tool for the students. The timetable integration feature that I had wanted to create add into the app is what sets the STAR apart from other task management applications. After completing this application, I had to do more discover what would be required for this application to function the way I had envisioned it.

5. Technologies Used:

5.1. Front-End Application:

5.1.1. Next.js:

The front-end of this application uses Next.js version 14.1.0 as a framework. [4] Next.js is a React framework used for building full stack web applications. Some of the key features of Next.js include routing, rendering, data fetching, styling, optimizations, and TypeScript. Except for TypeScript, it was necessary that I used all these the primary features and much more.

5.1.2. JavaScript:

The front-end is coded in JavaScript which controls the client-side functionality of the application.

5.1.3. Tailwind CSS:

Tailwind CSS is used for the styling and design of the application. [5] Tailwind CSS is an open-source CSS framework. The main feature of this library is that, unlike other frameworks like Bootstrap, it does not provide a series of predefined classes for elements such as buttons or tables. Instead, it creates a list of “utility” CSS classes that can be used to style each element by mixing and matching. I decided to use Tailwind over traditional CSS due its highly customizable nature and reusability.

5.2. Back-End Server:

The backend of this application is where server-side components responsible for fetching and storing data into the database, as well as handling requests from the front-end. The backend is the bridge between the client-side and the database.

5.2.1. Node.js:

The app’s backend uses Node.js version 20.9.0 as its runtime. [5] Node.js is an open source and cross-platform JavaScript runtime environment. It allows developers to run JavaScript code on the server.

5.2.2. Express.js:

The backend is also coded JavaScript, but also uses the Express.js framework. [6] Express is a minimal and flexible Node.js web application framework that provides a robust set of features for the web and mobile applications. Express provides a thin layer of fundamental web application features, without obscuring Node.js features.

5.2.3. Bcrypt.js & JWT:

The backend handles all authentication and verification features of the application, such as bcrypt.js to hash the password and JWTwebtoken to create a session.

5.3. Database:

MongoDB is the database this application uses for storing user information and other important data, such as tasks. [7] MongoDB is a source-available, cross-platform, document-oriented database program. Classified as a NoSQL database product, MongoDB utilizes JSON-like documents with optimal schemas.

6. Features

Here are the features this tool uses:

6.1. Task functions:

Of course, with a task managing application, like this one, its necessary to be able to create tasks. Tasks can be created, deleted, and edited, and this is all saved to the database.

6.2. Progress Monitoring:

While this functionality is a part of the task functionality, each task has a progress bar and statuses that allow users to view how their tasks are progressing, as well as include updates with each increase in progress (through editing).

6.3. User Creation and login:

Users can create their account and login into the app, which will create a session for them every time they are logged in. The user's information, and details are protected.

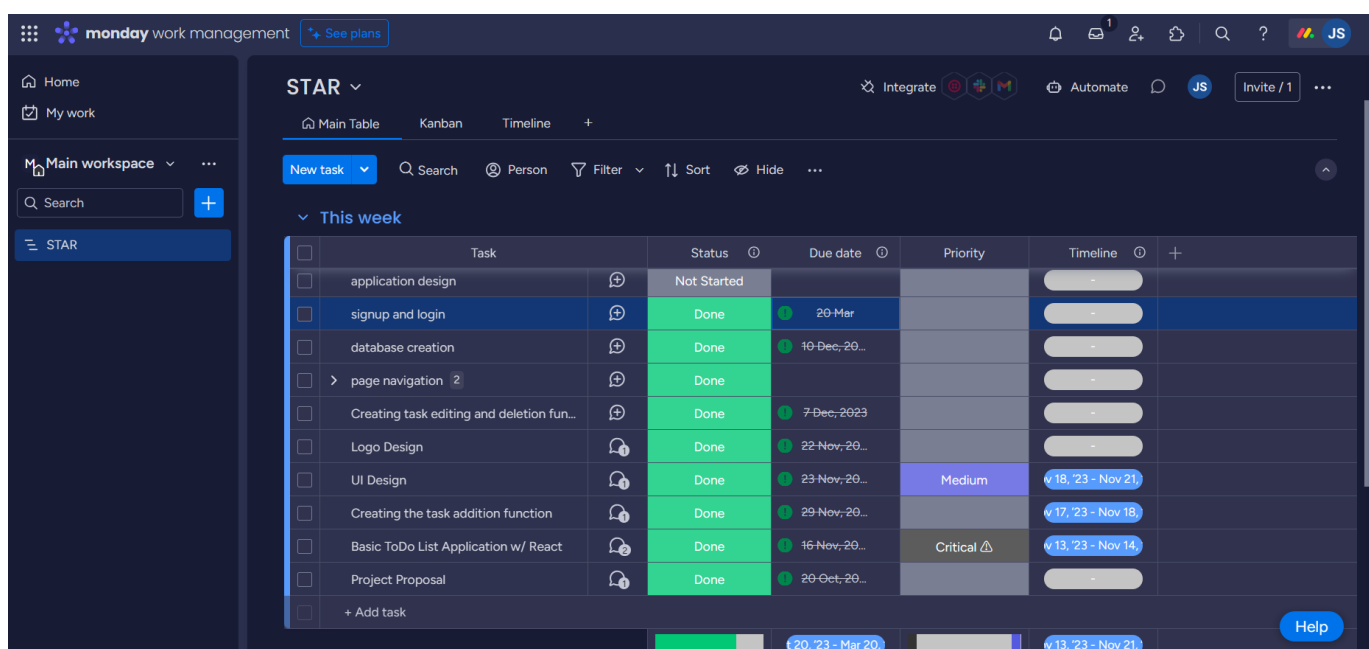
6.4. Timetable integration:

A user can navigate to the upload timetable page and input their timetable into the application. 📅

7. Planning

Planning is a key aspect in the time management of this project. Monday is a powerful **project management system** — a complete Work OS designed to help your team complete projects efficiently, collaborate effectively, and grow online. We provide you with all the tools and features you need to deliver great products and projects [13].

Project planning played a crucial role in this project as the time spent on this project had to be balanced out and effectively managed with all my other modules. I used Monday over other project management tools because I found it easy to use and understand.



8. Tasks

This section will be used to explain the code and logic behind all the task functionality of the application.

8.1. Models

The Todo model is in the backend of my application. It defines the task data that is going to be saved to MongoDB.

```
const mongoose = require("mongoose")
const connectToMongoDB = require("../dbConfig/dbConfig")
connectToMongoDB()
const todoSchema = new mongoose.Schema({
```

All models created in this project require “mongoose.” Mongoose is an Object Data Modelling (ODM) library for MongoDB and Node.js. It manages relationships between data, provides schema validation, and is used to translate between objects in code and the representation of those objects in MongoDB [9].

The variable “connectToMongoDB” is a function imported from the dbConfig file. As the name suggests, it allows the schema to connect to MongoDB.

“todoSchema” is the name of the mongoose model that defines the data that MongoDB will be storing.

```
{
  title: {
    type: String,
    required: [true, "Please add task title"]
  },
  description: {
    type: String,
    required: [true, "Please add task description"]
  },
  category: {
    type: String,
    required: [true, "Please add task category"]
  },
  priority: {
    type: Number,
    required: [true, "Please add task priority"]
  },
  progress: Number,
  status: String,
  active: Boolean,
},
{
  timestamps: true,
}
```

Here are all the fields that are expected in each task created. “Timestamps: true” was added for preference. This is just to show the time that a task was created / edited.

```
const Todo = mongoose.models.Todo || mongoose.model("Todo", todoSchema);
module.exports = Todo;
```

This code is what creates the task in the database. The first line checks if there is a model named “Todo” in the Mongoose models. If it exists, the variable “Todo” is assigned to it. If it does not exist, a new Mongoose model with the name “Todo” is created and the schema “todoSchema.”

The model is exported as “Todo” so it can be used in other files.

8.2. Components

I created components that would help with the functionality and appearance of the tasks.

8.2.1. TodoForm

The component “TodoForm” is responsible for a form to create or update tasks.

```
"use client"

import React, { useState } from "react";
import { useRouter } from "next/navigation";
```

Defining the component with the “use client” header allows the code to be rendered on the browser and access React hooks, such as “useState” and “useEffect.” The “useState” hook is used for managing states in functional components. React imports all the necessary dependencies needed for this specific component.

The import “useRouter” is another React hook that provides access to the router object. It allows navigation within the navigation.

```
const EDITMODE = todo._id === "new" ? false : true;
```

The “TodoForm” component has been made in a way that its dynamic, meaning that depending on the context, it has different functionalities. The variable “EDITMODE” is used to check for a task id. Task ids are set to new if they are new, otherwise, the actual task id would be fetched from the database. It is important to know if the id is set to “new” or not, as that will determine the api call that will be made when creating or updating a task.

```
const [formData, setFormData] = useState({
  title: todo.title || "", // Set title to empty string if undefined
  description: todo.description || "", // Set description to empty string if undefined
  priority: todo.priority || 1, // Set priority to 1 if undefined
  progress: todo.progress || 0, // Set progress to 0 if undefined
  status: todo.status || "not started", // Set status to "not started" if undefined
  category: todo.category || "Project", // Set category to "Project" if undefined
});
```

The above piece of code uses a “useState” hook to make the state of the variables “formData” and “setFormData.” As stated above, this form is made to be dynamic, which would explain why formData is written this way. All the fields necessary to create a task are set so that on making a new task, they are defaulted, but on updating a task, it sets each field to that of the task retrieved from the database.

```
const handleChange = (e) => {  
  const { name, value } = e.target;  
  setFormData((prevState) => ({  
    ...prevState,  
    [name]: value,  
  }));  
};
```

This code handles any changes made in the input fields. I used inputs for each field required to make a task and set the value to “formData.fieldname.” For example, for the title field, whatever is inputted into that field is set to “formData.title.”

```
<label>Title</label>  
<input  
  id="title"  
  name="title"  
  type="text"  
  onChange={handleChange}  
  required={true}  
  value={formData.title}  
>
```

All the input fields are placed within these form tags, with an “onSubmit” of “handleSubmit”.

```
<form className="flex flex-col gap-3 w-1/2" onSubmit={handleSubmit}>
```

At the end of the form there is an input with the class of “btn,” which was a class name that I had defined before creating the field. The class can be reused for whenever I need to create a button. This highlights one of the reasons I used Tailwind CSS, which was its reusability.

```
<input  
  type="submit"  
  className="btn"  
  value={EDITMODE ? "Update Task" : "Create Task"}  
>
```

The button input is of type submit, which submits the entire form and allows the function “handleSubmit” to take it from there. the value of this input is set to either “Update Task “or “Create Task”, depending on whether the form is in “EDITMODE” or not.

When the form is submitted, it calls the handleSubmit function which, depending on whether the form is in EDITMODE or not, will make a POST or PUT request to the API located in the backend of the application.

8.2.2. TodoProgress

This is the code that allows a user to set the progress of a task that they have created. The prop “progress” is passed into the function and dynamically adjusts the width of the progress bar based on that prop.

```
const TodoProgress = ({ progress }) => {
  return (
    <div
      className="w-full
        ■ bg-gray-200
        rounded-full h-2.5"
    >
      <div
        className="■ bg-blue-600 h-2.5 rounded-full"
        style={{ width: `${progress}%` }}
      ></div>
    </div>
  );
};
```

8.2.3. Status

The “status” prop is passed into this function. Depending on the status chosen from the status options on the form (done, started, not started, stuck, almost done) a unique colour is set in the status display.


```
const Status = ({ status }) => {
  const getColor = (status) => {
    let color = "bg-slate-700";
    switch (status.toLowerCase()) {
      case "done":
        color = "bg-green-200";
        return color;
      case "started":
        color = "bg-yellow-200";
        return color;
      case "not started":
        color = "bg-red-200";
        return color;
      case "stuck":
        color = "bg-orange-300";
        return color;
      case "almost done":
        color = "bg-teal-300";
    }
    return color;
  };
};
```

8.2.4. TodoPriority

The priority display of the application is simple. The priority of each task ranges from one to five, with five being the highest priority. Depending on what the priority of any task is set to, the flame icon will change its colour to red, otherwise it remains a slate colour.

```
const TodoPriority = ({ priority }) => {
  return (
    <div className="flex justify-start align-baseline">
      <FontAwesomeIcon
        icon={faFire}
        className={`pr-1 ${priority > 0 ? "text-red-400" : "text-slate-400"}`}
      />
      <FontAwesomeIcon
        icon={faFire}
        className={`pr-1 ${priority > 1 ? "text-red-400" : "text-slate-400"}`}
      />
    </div>
  )
}
```

8.2.5. DeleteBlock

When the trash icon was deleted, it would make a delete request to the API. The task would be found by its id and all information and data regarding that task would be deleted.

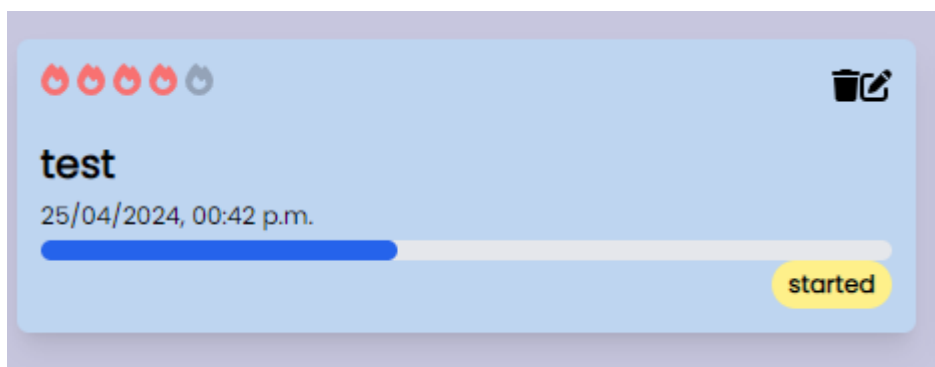
```
const DeleteBlock = ({ id }) => {
  const router = useRouter();
  const deleteTask = async () => {
    const res = await fetch(`http://localhost:5000/todos/${id}`, {
      method: "DELETE",
    });
    if (res.ok) {
      router.refresh();
    }
  };
}
```

8.3. API Routes

8.3.1. Edit Route

```
router.put("/:id", async (req, res) => {
  const { id } = req.params;
  const formData = req.body;
  try {
    const updatedTask = await Todo.findByIdAndUpdate(id, formData, {
      new: true,
    });
    if (!updatedTask) {
      console.error("Task not found for update:", id);
      return res.status(404).json({ message: "Task not found" });
    }
    console.log("Task Updated:", updatedTask);
    return res.status(200).json({ message: "Task Updated", updatedTask });
  } catch (error) {
    console.error("Error updating task:", error);
    return res.status(500).json({ message: "Error", error });
  }
});
```

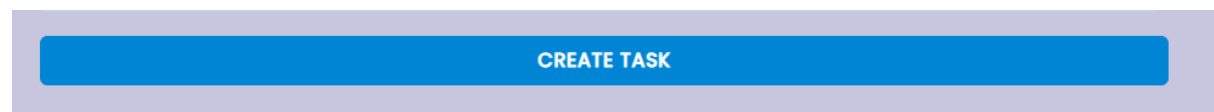
When the edit icon on the task card display is clicked, it calls this get route. It finds that specific task by its id and retrieves the data. That data is then filled into the todoForm.



8.3.2. Create Task Route

```
router.post("/", async (req, res) => {
  try {
    const formData = req.body;
    const newTodo = await Todo.create(formData);
    console.log("Task Created:", newTodo);
    return res.status(201).json({ message: "Task Created" });
  } catch (error) {
    console.error("Error creating task:", error);
    return res.status(500).json({ message: "Error", error });
  }
});
```

This route is called whenever the create task button at the end of the todoform is clicked. A task will only be created when all fields of the task form are filled. It awaits the data being sent to the task and then creates it.

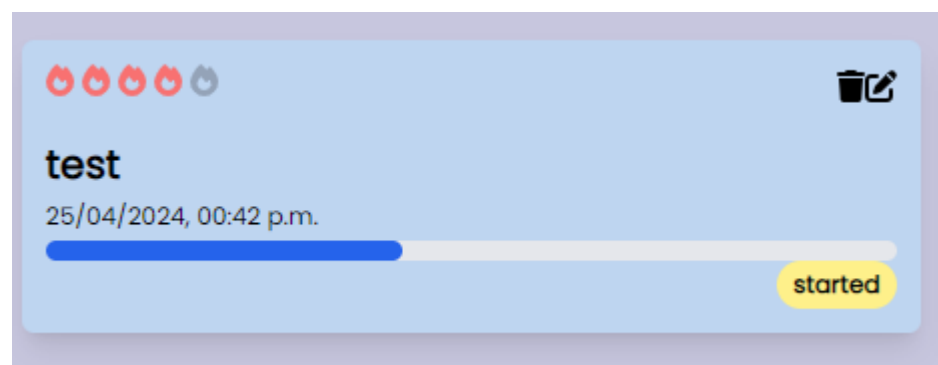


The await operator is used to wait for a promise and get its fulfilment value [10]. In this case, the Promise that is being awaited is the creation of the new task.

8.3.3. Delete Task Route

```
router.delete("/:id", async (req, res) => {
  const { id } = req.params;
  try {
    await Todo.findByIdAndDelete(id);
    console.log("Task Deleted:", id);
    return res.status(200).json({ message: "Task Deleted" });
  } catch (error) {
    console.error("Error deleting task:", error);
    return res.status(500).json({ message: "Error", error });
  }
});
```

When the trash icon on the task card is deleted, it called this route. This route finds the task by its id and using the function “findByIdAndDelete()” it deletes that task from the database.



9. Users

9.1. Models

```
const userSchema = new mongoose.Schema(  
  {  
    username: {  
      type: String,  
      required: [true, "Please provide a username"],  
    },  
    email: {  
      type: String,  
      required: [true, "Please provide an email"],  
    },  
    password: {  
      type: String,  
      required: [true, "Please provide a password"],  
    },  
  },  
);
```

Here is the Schema for the User Model. A username, email and password are required. With the user, whenever someone logs into their account, it generates a JWT token to:

1. Create a session and
2. Authenticate the user.

This authentication is only done on login.

9.2. Components

9.2.1. SignupForm

Signup

Username
your-username

Email
your-email

Password
password

NO SIGNUP

[Already signed up? Log in here](#)

```
"use client";

import Link from "next/link";
import { useRouter } from "next/navigation";
import React, { useEffect, useState } from "react";
```

Here are the imports used in the signup form. Unlike the todoform, the signupform uses the useEffect hook. This is ordinarily used for data fetching and DOM manipulation, but in my code, I use it to set the submit form button to disabled whenever the username, email & password are not all filled in. This allows for the form to also have dynamic attributes.

Fig. 1 Button Disabled.

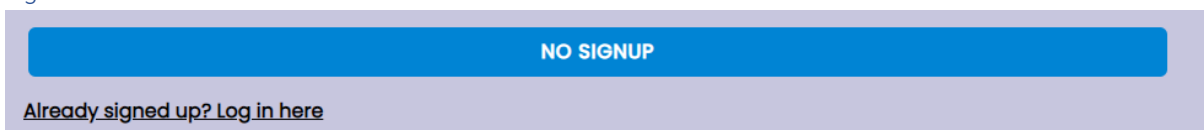


Fig. 2 Button Enabled



```
const router = useRouter();
const [username, setUsername] = useState("");
const [email, setEmail] = useState("");
const [password, setPassword] = useState("");
```

By default, the username, email, and password are set to being empty. In the form, the email, username, and password fields are assigned using input fields.

```
e.preventDefault();
try {
  setLoading(true);
  const resT = await fetch("http://localhost:5000/signup/checkUser", {
    method: "POST",
    body: JSON.stringify({ email }),
    headers: {
      "Content-Type": "application/json",
    },
  });
  if (!resT.ok) {
    throw new Error("Error signing up");
  }
}
```

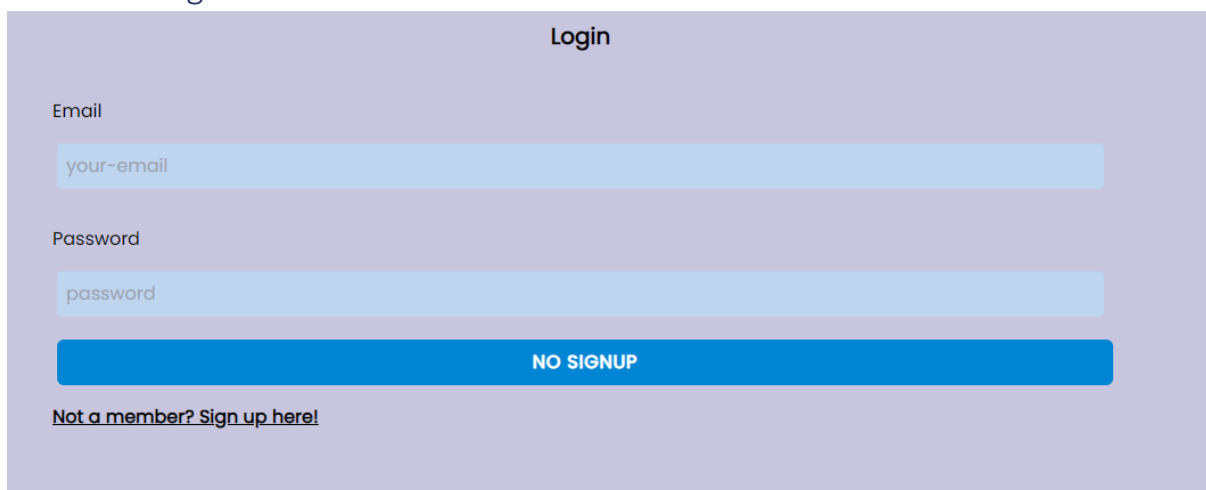
When the SignupForm is submitted, it calls the handleSubmit function, which firstly calls this API route. This route checks if the user's email already exists in the database. If the user exists, it returns an error message. This allows each user's email to be unique.

```
const res = await fetch("http://localhost:5000/signup/", {
  method: "POST",
  body: JSON.stringify({ username, email, password }),
  headers: {
    "Content-Type": "application/json",
  },
});

if (!res.ok) {
  throw new Error("Error signing up");
}
```

If the user does not exist, the function continues to this route. This route is responsible for creating the new user and stores their information into MongoDB.

9.2.2. LoginForm



Just like the SignupForm, this Loginform is dynamic. All fields must be filled to be able to login form for the submit button to be enabled.

```
setLoading(true);
const res = await fetch("http://localhost:5000/login/", {
  method: "POST",
  body: JSON.stringify({ email, password }),
  headers: {
    "Content-Type": "application/json",
  },
});

if (!res.ok) {
  throw new Error("Error logging in");
}
```

On submission, the email and password are sent to the API login route. It checks to see if the email from the login form exists already and then cross-checks that entered password against the password under that user in the Database.

9.3. API Routes

9.3.1. Check User Existence

```
router.post("/checkUser", async (req, res) => {
  try {
    console.log("POST RAN ***");
    const { email } = req.body;
    const foundUser = await User.findOne({ email });
    if (foundUser) {
      console.log("if found user");
      return res.status(400).json({ message: "User already exists" });
    } else {
      console.log("if not found user");
      return res.status(200).json({ message: "User does not exist" });
    }
  } catch (error) {
    return res.status(500).json({ message: error.message });
  }
});
```

This is the first route that is fetched on Signup. The email is de-structured from the request body. Using the findOne() function, all users in the “users” collection in MongoDB are checked.

users				
Storage size:	Documents:	Avg. document size:	Indexes:	Total index size:
20.48 kB	8	200.00 B	1	36.86 kB

If there is an identical email to the one used for the signup, the creation of that user is denied, and an error is returned.

9.3.2. Create User

```
router.post("/", async (req, res) => {
  try {
    console.log("POST 2 RTAN");
    const reqBody = await req.body;
    await connectToMongoDB();
    const { username, email, password } = reqBody;
```

After the new user is checked, the username, email and password are passed to this route. It receives req.body, which parses the incoming request.


```
const hashedPassword = await bcrypt.hash(password, 10);
```

Using the bcrypt.js library, the password is hashed with a salt factor of 10. This is to protect the password and the user from being hacked.

```
const newUser = await User.create({
  username,
  email,
  password: hashedPassword,
});
```

A new user is created, using the email and username from the request body. Instead of using the password directly from the request body, the hashedPassword is set as the password. This new user is then saved to the database.

9.3.3. Login

```
router.post("/", async (req, res) => {
  try {
    await connectToMongoDB();
    const { email, password } = req.body;
    console.log("LoginPOST RAN");

    console.log(email);
    console.log(password);
    const existingUser = await User.findOne({ email }).select("password");
    if (!existingUser) {
      return res.status(400).json({ message: "User does not exist" });
    }

    const passCheck = await bcrypt.compare(password, existingUser.password);
    if (!passCheck) {
      return res.status(400).json({ message: "Incorrect Password" });
    }
  }
});
```

The email and password are received as a request body from the login form. The email is checked in the database to see if it exists, and the password is then checked using bcrypt.js to compare it to the password under the email that was just checked. If the user email does not exist or the password is incorrect, a JSON response with a status code of 400, and an error message is sent by the route handler.

```
const token = jwt.sign({ id: existingUser._id }, process.env.JWT_SECRET, {
  expiresIn: "1d",
});
delete existingUser.password;

res.cookie("jwt", token, {
  httpOnly: true,
  maxAge: 24 * 60 * 60 * 1000,
});
```

On login, a JWT token is created using the function “`jwt.sign()`”. The user id and the JWT Secret is passed into this function. The JWT secret is stored as an environment variable and is used to sign the token. The token is signed to serve as a key validation method to guard the user’s information against malicious tampering.

The password under the “existing User” object is then removed to prevent information leakage. This serves as an extra layer of security. While this is just a minor part of the overall project, it is important for user info to be protected and for anyone who uses this web application to feel assured that their information will not be stolen.

10. Timetable

Components

When inserting a timetable, it must be in .xlsx or .xls format. I created an input that allows the file to be selected from a user's machine.

```
<div className="container mx-auto py-4">
  <input
    type="file"
    accept=".xlsx, .xls"
    className="border border-gray-300 rounded p-2 mb-4"
    onChange={handleFileUpload}
  />
```

The function "handleFileUpload" is called when an excel file is chosen. This is the code responsible for converting information from the file to data that the app can read.

```
const [data, setData] = useState([]);
const handleFileUpload = (e) => {
  const reader = new FileReader();
  reader.readAsBinaryString(e.target.files[0]);
  reader.onload = (e) => {
    const data = e.target.result;
    const workbook = XLSX.read(data, { type: "binary" });
    const sheetName = workbook.SheetNames[0];
    const sheet = workbook.Sheets[sheetName];
    const parsedData = XLSX.utils.sheet_to_json(sheet);
    setData(parsedData);
  };
};
```

Data and setData are both initialised to empty arrays. On handleFileUpload, a new instance of the function "FileReader()" is created and set to the name "reader". The contents of the file are read as a binary string. Once the file is fully loaded, it goes through several functions to grab each row of information from the file. That information "parsedData" is then set to "data".

```

{data.length > 0 && (
  <table className="table-auto">
    <thead>
      <tr>
        {Object.keys(data[0]).map((key) => (
          <th key={key} className="border border-gray-300 p-2">
            {key}
          </th>
        ))}
      </tr>
    </thead>
    <tbody>
      {data.map((row, index) => (
        <tr key={index}>
          {Object.values(row).map((value, index) => (
            <td key={index} className="border border-gray-300 p-2">
              {value}
            </td>
          ))}
        </tr>
      ))}
    </tbody>
  </table>
)}

```

The data length is then checked in the return statement. If there is data in the file ("data.length > 0"), the data is formatted in a table. The headers are separated into table head ("thead") and the table body ("tbody"). The styling makes the table more presentable.

11. Difficulties Faced and Problem Solving

11.1. Learning Curve:

As someone who did not really have the most experience in front end or backend development, there was a steep learning curve in learning exactly what I need to do to make this project. When doing this project, I was sure to keep the Full Stack Development module open and online for solutions. This majorly assisted me in this project.

11.2. Front-end Application to Full Stack Application:

Initially I was following along with a YouTube video where a programmer was creating a ticketing application. [11] The application was inspired and built off that. But the application was a Front-end Application and had no server. I spent a lot of time researching how to create a back end and what that backend would require. Thankfully, I found another [12] video that thought me how to create a REST API. I was not sure if it were a REST API that my application needed so I investigated the benefits of using a REST API versus other types of backends and found that this would be best suited for me. Learning Express was a bit awkward, but there was no learning curve. So, with that, I was able to convert my application from just having a frontend with no server to a proper application.

11.3. CORS:

After creating the backend server for this project, I ran into a bug. I was not able to make requests to the backend server. It was because the front end of the application was running on a different port to the backend. After doing so research, the issue was that I did not enable CORS (). After enabling CORS (), I was quickly able to fix the issue.

12. Teamwork

Teamwork was particularly important with this project. I always consulted my classmates on viable solutions to code bugs and project issues and their advice would always be helpful. Up until the end of the first term, I participated in stand-ups and spoke on the progress of my project and my goals for the week. I also would go on teams' calls every so often with my classmates to work on our respective project and offer solutions to one another. ☺

13. Ethics

When coding this project, these were the ethical considerations I considered:

1. **Protection:** I considered the importance of user information protection, as I want this application to be a trusted one. This led me to finding bcrpt.js and hashing all user passwords, as well as adding extra stop gaps in the routes to ensure user information could not be stolen.

14. What I learnt from this project

There are many things I learnt while creating this project:

1. Time Management: effectively managing time for projects and balancing it with all the other projects and assignments was crucial. The use of JIRA as a time management tool played an especially significant role in the completion of this project.
2. Teamwork: whenever I had issues with my project, I always had my classmates to assist me in finding a solution to the problem. Each one of my classmates were extremely helpful and always gave good advice. I learnt that having a good relationship with your peers or your team members is crucial when working on projects, especially group projects.
3. Front-End Development: Prior to this project, the only experience I had with web programming was doing some CSS for a group project I did in 3rd Year. From this project I learnt about:
 - The React Framework,
 - Next.js,
 - JavaScript,
 - Tailwind CSS,
4. Back-End Development: Unlike for front-end development, I had absolutely no experience in back-end development. I was able to learn a lot of back-end development from the Full Stack Development Module and from many of the YouTube videos and documents that I looked to for help. I learnt:
 - Node.js,
 - REST API Creation,
 - Bcrypt.js,
 - Generating JWT Tokens
 - Page protection
 - Express.js
5. MongoDB: I learned how to use MongoDB Atlas (Online Database Environment) and MongoDB Compass (Local Database Environment), as with this project, I used both.

15. Conclusion

With all the technologies learn from online research and subjects studied from this school year I was able to create an application that allows students to track any tasks in their appropriate categories and track the task progress. This application allows the user to set up a new username, email, and password, and all their information is protected. While there is room for improvement, I am proud of the application I made and hope to continue to work on it after I leave college and develop it even further, by allowing the app to be available as an application on mobile and hosting it on AWS.

Thank you for reading. Have a wonderful day!

16. References

- [1] Art of Manliness, [Using Todoist to Be More Productive | Art of Manliness](#)
- [2] TickTick, what is TickTick?, [🔖 Beginner's Guide \(ticktick.com\)](#)
- [3] Microsoft, Welcome to Microsoft ToDo, [Welcome to Microsoft To Do - Microsoft Support](#)
- [4] Next JS, Introduction & Main Features, [Docs | Next.js \(nextjs.org\)](#)
- [5] Wikipedia, Tailwind CSS, [Tailwind CSS - Wikipedia](#)
- [6] GeeksForGeeks, Node.js Tutorial, [Node.js Tutorial | Learn NodeJS - GeeksforGeeks](#)
- [7] Express.js, [Express - Node.js web application framework \(expressjs.com\)](#)
- [8] Wikipedia, MongoDB, [MongoDB - Wikipedia](#)
- [9] Introduction to Mongoose for MongoDB, Nick Karnik, [Introduction to Mongoose for MongoDB \(freecodecamp.org\)](#)
- [10] MDN Web Docs, Await, [await - JavaScript | MDN \(mozilla.org\)](#)
- [11] freeCodeCamp.org, Next.js/Tailwind CSS tutorial, [Next.js, Tailwind CSS, and MongoDB Project Tutorial – Ticketing App \(youtube.com\)](#)
- [12] Web Dev Simplified, build a REST API with Node.js, [Build A REST API With Node.js, Express, & MongoDB - Quick \(youtube.com\)](#)
- [13] Monday blog, what is Monday.com, <https://monday.com/blog/product/how-to-use-monday-com/>