Contents

Analysis

Problem and Stakeholders

Outline

My project is an application that helps with scheduling and productivity. You input tasks you need to do, as well as the time they take and the time they must be done by, and the app will make a schedule for you to follow. The schedule is adaptive, tasks can be added or changed at any time and the schedule will adapt. When adding tasks you can set them to repeat on certain days or every interval. To help with following the schedule, the app will send you notifications when you need to change tasks. If a task is taking longer to complete than you entered, you can defer the task for later or extend the task and the schedule will adapt to fit. You can also schedule very long tasks and they will be broken up into chunks and put into the schedule. The app is a website you have to log into the access the schedule, meaning the schedule can be accessed from anywhere on any device.

Justification of a Computational Solution

Schedules have been made by hand for centuries, and a lot of tools and algorithms have been developed to aid in this such as Gantt charts and scheduling diagrams. The issue with making schedules by hand is that it takes a long time, a complex schedule can take up to an hour to create and they are very hard to optimise. This high time cost means that most people don't schedule their time, especially for personal tasks. It also makes

schedules very rigid because if anything changes then the schedule has to be remade from scratch, this means that schedules that are made by hand are very bad at responding to change which is bound to happen at some point.

Essentially, scheduling is a number crunching task, you try every permutation of the list of tasks in order to find one that fufills all deadlines and criteria. For a low to medium amount of tasks, this approach is extremely fast on computers compared to making a schedule by hand, microseconds compared to hours. This means that computed schedules can be very quickly regenerated in response to changes in a user's plans.

As well as this a correct scheduling algorithm would make no mistakes, whereas schedules made by hand often have mistakes which are very hard to fix because of how much time they take to remake.

Stakeholders

Due to the nature of the project, it could appeal to anyone who wants some help with staying on top of lots of different tasks, but my main target audience is students as they both have a lot of different tasks to keep track of, and also the most likely to adopt a digital schedule as the majority of them reguarly use computers throughout the day. To make the app useful for students, I need to make sure common tasks like school, revision, and homework are easy to add so they can set up their schedule quickly. A lot of these are repeating tasks so the app will need to handle those well. Also, the main device of most students is smartphones, so I'll need to make the website responsive and works well on mobile platforms, as well as making sure notifications work on iOS and Android. Adding tasks needs to be easy and fast otherwise people won't use the app. Finallly, the website needs to look aesthetic otherwise people won't use it.

Research

In order to inform the design of my app, I researched existing solutions and conducted a survey and evalutated these to find features my stakeholders will find useful.
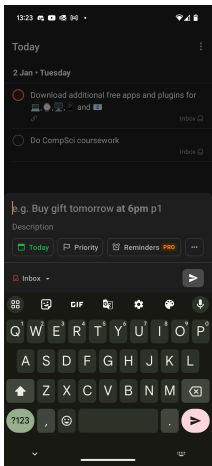
Existing Solutions

Microsoft To Do
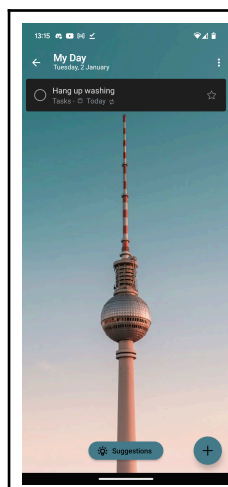
- Easy to add tasks, just press the plus button and type in the task title, but trying to do anything else is very difficult, the repeat feature is hard to use and limited
- "My Day" view is useful as a form of basic schedule
- Tasks don't have a duration or an end time, and there's not a proper schedule so you have to manually schedule reminders
- Integrated well with Microsoft ecosystem, you can attach documents and link to emails
- No way to add large tasks that are split up

Google ToDo

	• Easy to add basic tasks, similar to Microsoft To Do, but no way of repeating tasks automatically and difficult to add details • No form of schedule, just a list of tasks • Tasks don't have a duration or an end time so reminders can't be automated • No way to add large tasks

Todoist

	• Best looking UI out of the 3, lots of colour • Very complicated, there are lots of different types of tasks and it wasn't explained • Easy to add tasks, just need a title, but many basic features like reminders and repeating are locked behind a paid version • No form of schedule, just a list of tasks • Tasks don't have a duration or an end time so reminders can't be automated • No way to add large tasks

Summary of Existing Solutions

App	Feature	Will I be adding it?	Justification
Microsoft To Do and Google ToDo	Easy to add tasks	Yes	The easier tasks are to add, the higher the chance that users will add them
Microsoft To Do	My Day view	Yes	Makes it easy to work out what needs to be done compared to a list of tasks
Microsoft To Do	Integration with other software	No	This feature is not feasible for the project (see limitations)
Todoist	Colourful UI	Yes	The good UI design made it

			very easy to understand the quite complicated task UI

The main thing I noticed while researching existing solutions is that there aren't any major apps that provide a scheduling feature, they all stop at a list of tasks with optional reminders. Google ToDo can integrate with Google Calendar, but its not automated and manually scheduling tasks is long and complicated to do by hand. If my app had this feature it would set it apart from the competition. Also the ability to repeat tasks was missing from nearly all of the apps I tried, so this is definately something I need to implement.

Survey

I created a survey and sent it out to some student peers, I got 8 repsonses in total with the following answers:

Do you currently use a todo/scheduling app?

8 responses

- Yes
- No

87.5%
12.5%
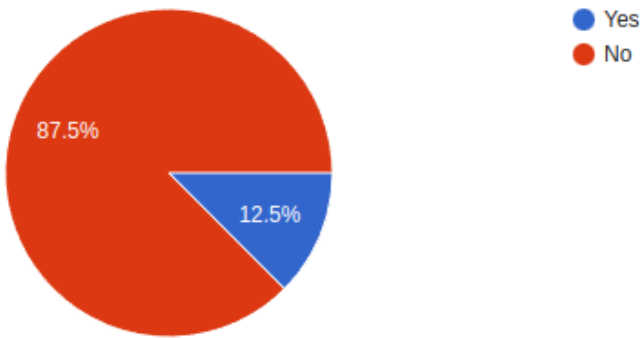
Would you prefer a schedule or a list of tasks?

8 responses

- Schedule
- List of Tasks

37.5%
62.5%
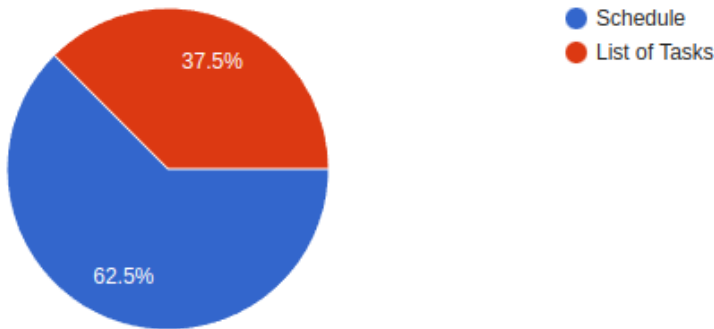
When adding tasks, how much required information would you be willing to enter?

8 responses

Category	Value
Title	8 (100%)
Deadline	7 (87.5%)
Duration	5 (62.5%)
Description	4 (50%)
Priority	8 (100%)
Category e.g. Revision, Chores	5 (62.5%)
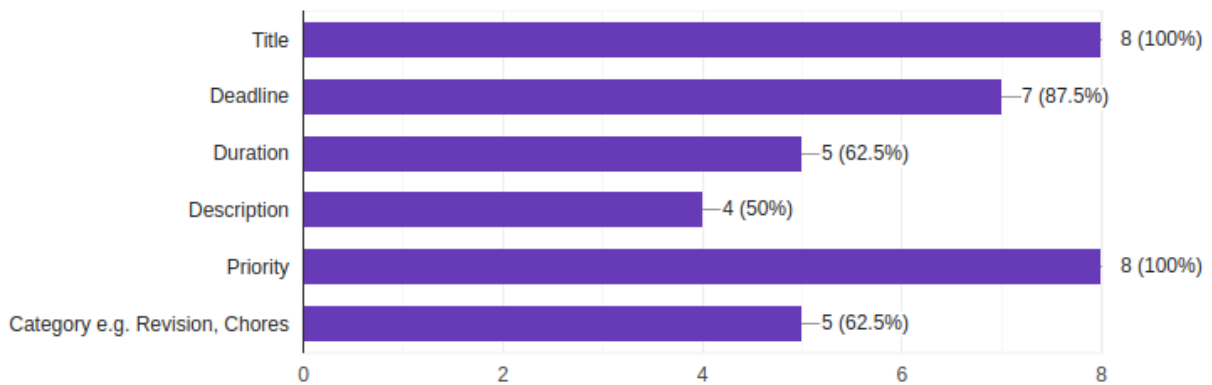
Do you find reminder notifications useful?

8 responses

- Yes
- No

100%
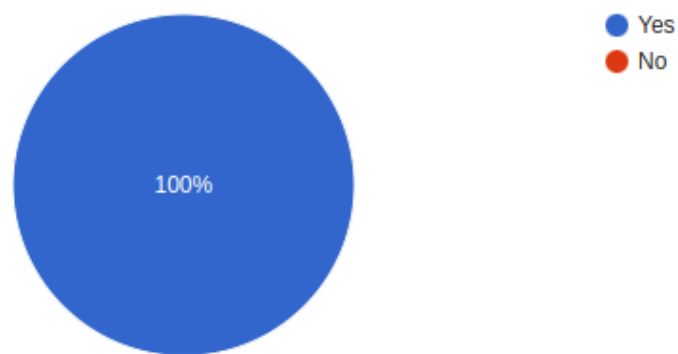
What platforms would you use for a productivity app?

responses

Platform	Value
PC	7 (87.5%)
Laptop	6 (75%)
Smartphone	8 (100%)
Tablet	4 (50%)
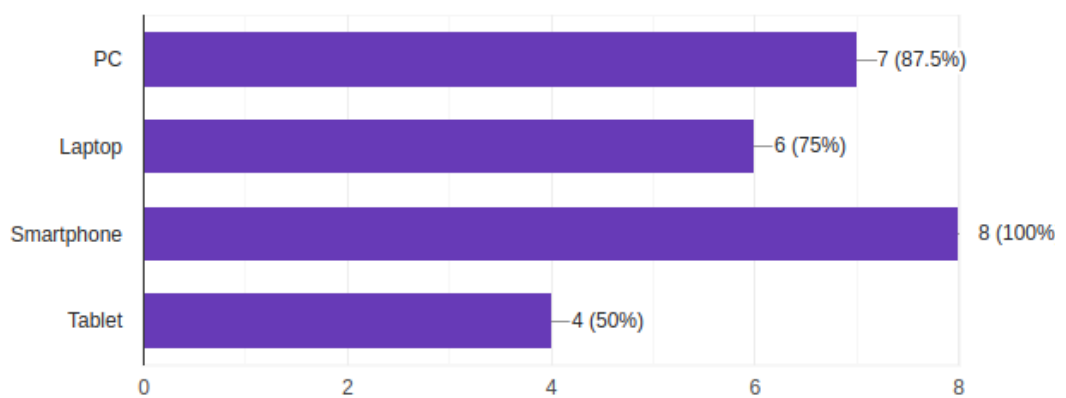
If the app only worked on what platform, what should it be?

8 responses

- ● PC
- ● Laptop
- ● Smartphone
- ● Tablet

62.5%
25%
12.5%
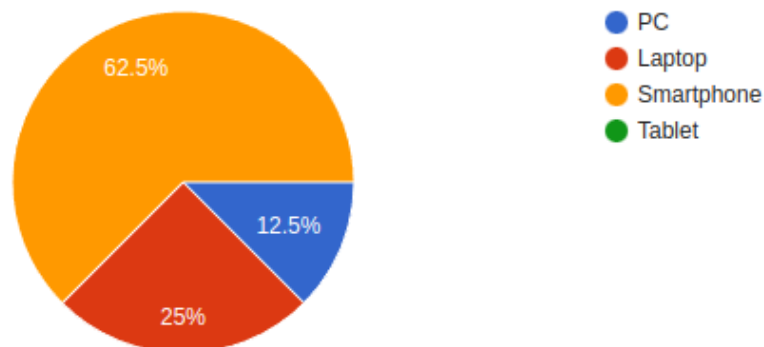
How important is a good looking UI for you?

Copy

8 responses
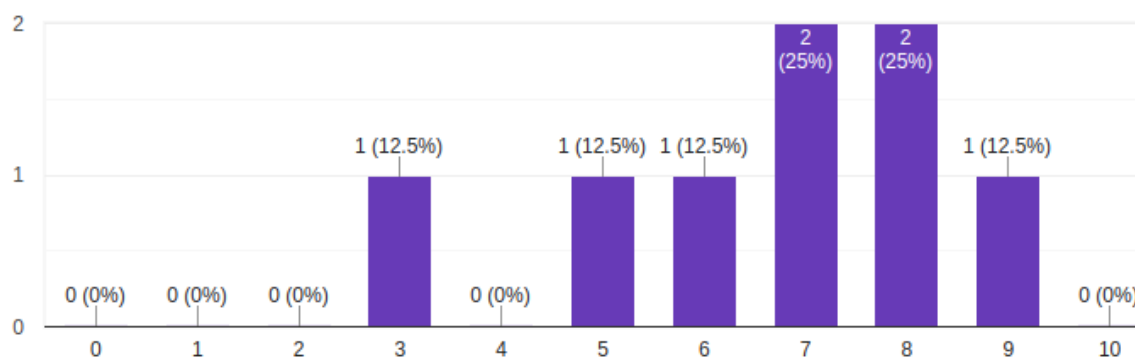
My first takeaway from this survey is that only 1 person is currently using a todo/scheduling app, I spoke to some of the participants after they took the interview, and all of them explained how they'd like to use one, but felt that all the existing solutions were missing features or were poor fits for their lifestyle. This is really promising as it shows that there is a demand that hopefully this project will fill. Some of the participants also agreed to try out my app once it's completed and give feedback.

The participants slightly preferred a schedule over a list of tasks, although this was very split so I will need to implement both for my website. When I spoke to some of the participants after, most of them said that schedules took too long to make by hand, or that they were too inflexible, this would be fixed by automatically generating a schedule like I'm planning to do.

Most participants said that they'd be happy inputting a title, deadline and priority for each task, so these are inputs I can design the scheduling algorithm around. Just under half of the participants said they wouldn't want to enter a duration, when I spoke to them after the survey I found the main reason was because they find it hard to guess how long something will,

this is a very important input for a scheduling algorithm, so I'll need to think of a way to address this.

All participants said they found reminder notifications useful, this will be one of my highest priority features. All candidates said that they would use the app on smartphones, so they should be my main platform, this was closely followed by PCs and laptops, so I will need to support those platforms too. I will not test for tablet support, but it might work out of the box depending on UI design. When asked for the most important platform for them, the majority of participants responded with smartphones, so this further highlights the importance of supporting smartphones.

Most participants said that a good looking UI was important for them, so I will need to make sure my UI is aesthetic.

Summary of Survey
• There is a large demand for a feature-complete todo/scheduling app among students
• Most people would like to use automatic scheduling
• Smartphones are the most important platform, followed by PCs/laptops
• Reminder notifications are one of the most important features
• UI needs to look good

Features of the Proposed Solution

Source	Feature	Priority	Explanation
Own Idea	Automatic Scheduling	High	This means that users don't need to keep deadlines and their todo list in their head to pick what tasks they need to do. Nearly all of the survey participants said they wanted this feature so I have made it high priority
Research	Todo List View	High	This allows users to easily view all of their tasks so they can be modified or deleted, nearly all similar apps had this feature so I've given it a high priority
Stakeholder	Repeated Tasks	High	Since my main stakeholder is students, who have lots of repeated tasks such as school and revision sessions, the app will need to handle repeated

			tasks well, so I've given it a high priority
Research	Reminder Notifications	High	Nearly all of the people who filled out the survey said reminder notifications were important to them, so I've given them a high priority
Research	Delaying tasks/ Take a break button	Medium	Many survey participants who I spoke to afterwards explained they would like a way of delaying tasks or taking a break for a length of time, this should be easy to add to the automatic scheduling system but not essential, so I've given it a medium priority
Research	API	Low	While speaking to survey participants, a few brought up making an API to get things like the current/upcoming task so they can use it in othe projects, I would to make this but it is not essential so I've given it a low priority
Own Idea	Schedule View	Low	Being able to view the entire schedule at once might be a nice feature, but its quite hard to make and since the schedule can completely change I'm not sure if it would be useful, so I've given it a low priority

Limitations

The first limitation is that the user must be connected to the internet in order to access their task list/ schedule, I may be able to get some sort of local caching working, but if another device makes a change the the schedule it will not be updated until the first device reconnects to the internet.

Another limitation is that you cannot attach documents to tasks like you can in other solutions, this is because it adds a lot of complexity and its not a feature I or any of my peers I spoke to would use. You will also not be able to share your schedule with others.

In addition, I am only planning on supporting smartphones, PCs and laptops, tablet support will not be tested as I don't have a tablet to test with. I also don't plan on adding any integrations with other services such as email apps or GitHub as this adds a lot of complexity to the project, however I may look at doing this if I take this project further.

Finally, there will be no way to get a data dump of your task list to backup or store locally, and there will be no way of accessing the history of your schedule as this would be very expensive to store on the server.

Hardware and Software Requirements

PC/Laptop Minimum Requirements:

Item	Justification
Monitor	Users will need to be able to see the visual output of the website in order to use it
Keyboard	The user will need to be able to type information about the tasks into the web app, this could be done with a mouse and on-screen keyboard
Mouse	A mouse will be needed to interact with the web app, to select tasks and press buttons
Windows 10/11 or macOS 10.15 or a 64-bit Linux OS	The user will need a supported operating system to run an internet browser to access the website. These are the supported operating systems for Chrome
SSE3 capable processor/ Intel Pentium 4 and up	The user will need a supported CPU to run an internet browser. This is the requirement for Chrome
4GB of RAM or more	The user will need enough memory to run a browser smoothly, this is taken from the Chrome requirements
1GB of harddrive space	The user will need to have a web browser installed which usually takes up 1GB

Mobile Minimum Requirements:

Item	Justification
Touchscreen	Users will need to be able to see the website and interact with it through a touchscreen

Android 8.0+ or iOS 10.0+	Minimum supported OSes for Chrome/Safari respectively
2GB of RAM or more	This is listed as Chrome and Safari's minimum memory requirements, this is lower than on PCs likely because mobile browsers are more memory optimised
1GB of hardrive space	The user will need to have a web browser installed which usually takes up 1GB

Success Criteria

Inputs

Criteria	Explanation	Measure
The user should be able to add tasks with a title, deadline, duration, repeat scheduling and priority easily on both desktop and mobile platforms	Users need to be able to add tasks as they think of them, else they might forget and it won't be added to their schedule. If adding tasks is complicated or takes too long then people just won't add them	During evaluation, I will ask those who tested the app how easy they felt it was to add tasks
The user should be able to edit tasks and delete them	Tasks can change as users realise they might take longer than estimate, or the deadline might move, or someone else might do the task for them, the app needs to facilitate this by letting users edit and delete tasks	During evaluation, I will check if I implemented this feature, as well as asking testers whether they were aware of the feature
The user should be able to mark tasks as complete easily	Tasks need to be easily marked as complete else users might forget which would mean the schedule will desync	During evaluation, I will ask testers how easy they felt it was to mark a task as complete
The user should be able to take a break for any length of time, which will clear their schedule	If users are taking a break, the app needs to not schedule any tasks for that time else the schedule will desync	During evaluation, I will check I added this feature, as well as if testers found it worked and was useful

Outputs

Criteria	Explanation	Measure
The user should be able to see a list of tasks	The users will need to manage their tasks,	During evaluation, I will ask for feedback

and their information easily	this could include editting them, deleting them and more. This information needs to be nicely presented to the user	on how tasks were presented to testers
The user should be able to quickly see their current task and upcoming tasks on their schedule	Users could want to check what task they need to do, or prepare for upcoming ones	During evaluation, I will ask testers on how the current and upcoming tasks were presented
The user should be sent reminder notifications when they need to start a new task	As I found in my survey, users want to be reminded of their task when they need to start it, so they don't have to constantly check the app	During evaluation, I will ask for feedback on how reliable and useful the reminder notification were
The website should be aesthetic and intuitive to use	As I found in the survey, UIs that are both aesthetic and easy to use are quite important for most people	I will ask testers about whether the UI was easy to use and whether it looked good on their devices

Processing

Criteria	Explanation	Measure
When a user makes any change to their task list, it should be stored on the server	Users might want to use the app on multiple devices, meaning tasks have to be synced across them. For this a user's task list must be stored in a database with APIs for getting and modifying the task list	I will ask testers if they found task syncing and persistence to be reliable, or if they lost any task information between devices
A schedule should be generated for the users task list	As I found by talking to people that took the survey, many people want a personal schedule to follow, but don't want to have to schedule it themselves because it takes too long and isn't flexible. By generating a schedule on-the-fly	I will ask testers how they found the generated schedule

	it becomes nearly instant and flexible as another schedule can be generated at any time	
When a user makes any change to their task list or decides to take a break, the schedule should be regenerated	The schedule need to be synchronised with a user's task list, otherwise its not useful, so it must be regenerated on any change	I will ask testers how reliable they felt the schedule regeneration was
Generated schedules should meet all deadlines and prioritise tasks correctly	If a schedule doesn't fit the user's task list, then its not very useful	I will check generated schedules against a user's task list when implementing the algorithm

Design

Decomposition

Full Diagram
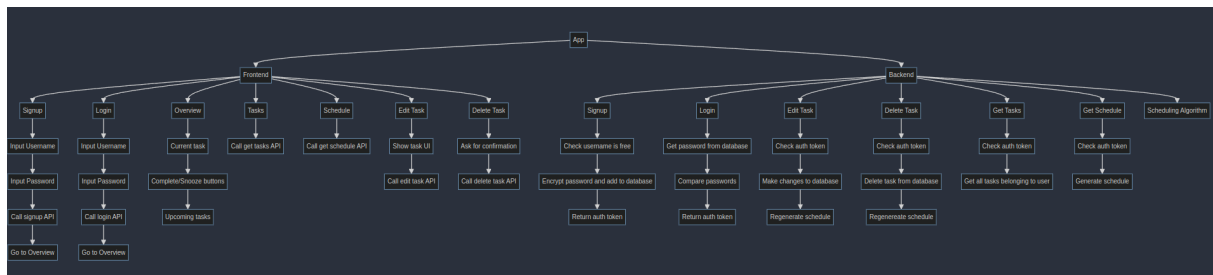
Frontend
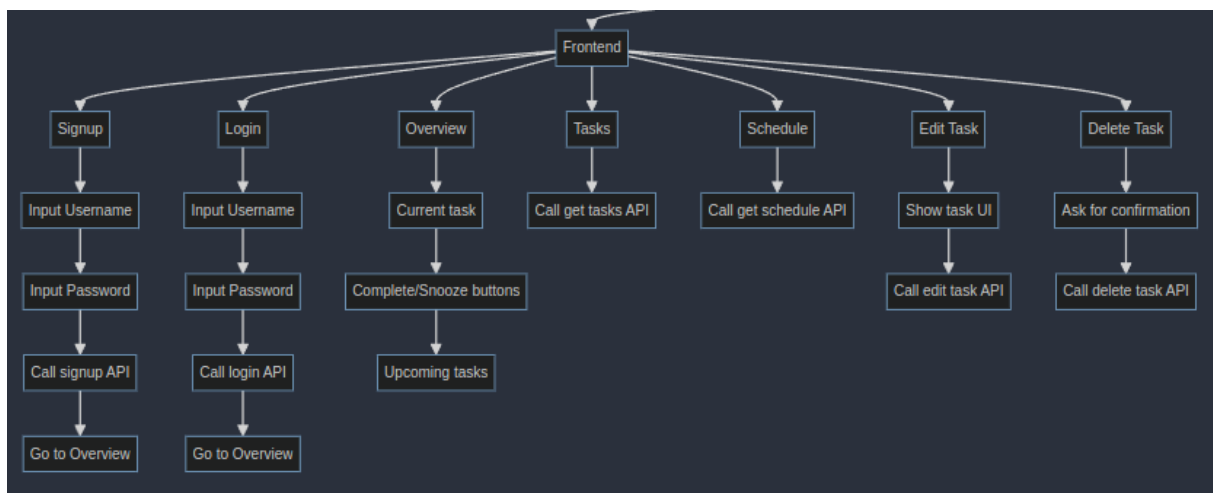
This shows the different pages and operations I will need to implement on the frontend, I chose to break down the project into pages and the flow on each page as they are a natural way of breaking up web apps. I also split the project into frontend (client-side) and backend (server-side), this is because the code will not be able to be shared between them easily, so its important to work out where each feature belongs.

Backend
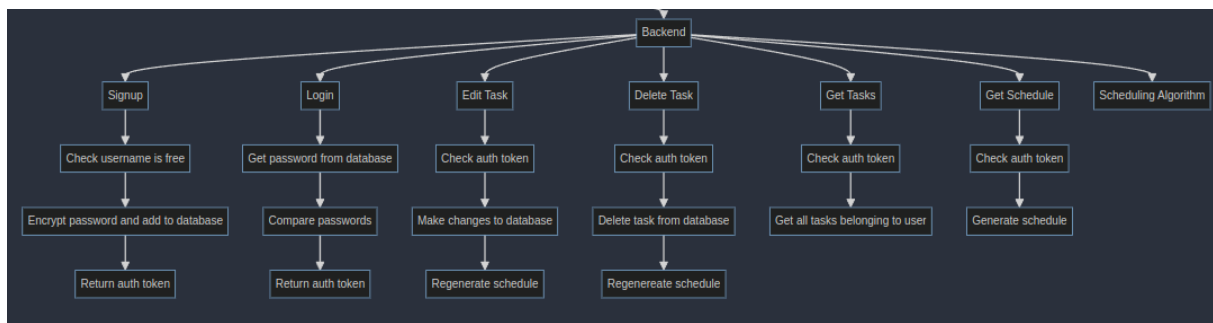
This shows the different API routes I will need to implement on the backend, these will be used by the frontend. I've left out the scheduling algorithm as I cover it properly below in the algorithms section. I decomposed the needed APIs by operation so each route only had one thing to

worry about, this would make it easy to reuse the APIs across the project in a modular way.

Interface Mock-Ups

Overview Page

This page will be the first page the user sees after logging in to their account. It shows the current task if there is one, some upcoming tasks and some buttons used to complete/snooze the task, or go on a break. If a button is pressed, it will trigger an API call to edit the current task. It also has the navbar in the top left to navigate between this, the task list and the schedule view. All pages share this

Task List Page

This page is a grid-like view of a users tasks, it will be sorted by scheduled start time, but the sorting can be changed, by clicking on a task it will pop up the task editing UI.

Schedule Page

This page shows the users schedule for the day, I chose to only make it the current day as the dynamic schedule will probably move tasks around quite a lot, and I don't want users to see a schedule that will likely change. By clicking on a task it opens up the task editting UI

Signup/Login Page

This page is what users will use to sign up for and log in to their account, from here the user will be redirected to the overview page. When the submit button is pressed the login/signup API route will be called.

Edit Task Popup

This will pop up from the bottom of the screen whenever the user click on a task in any of the 3 main pages. To simplify input validation, I've chosed to use on dropdowns, native date and time pickers as much as possible. I'm not sure if the title editting by clicking on it will be obvious so I might change that design if its an issue

Algorithms

Signup

On the frontend this algorithm needs to run some basic validation and then call the signup backend API with the inputted username and password. The server will then check the username if the username is free, if it is then it creates a new account, generates a login token and returns it to the frontend which will then save it, before going to the overview page.

```
function signup_frontend(username, password) {
  if username.is_empty() or password.is_empty() {
    return;
  }

  token = signup_backend(username, password);
  // Error handling
  save_login_token(token);
  goto("overview")
}

function signup_backend(username, password) {
  existing_users = get_users();
  if existing_users.contains(username) {
    return ERR_USERNAME_TAKEN;
  }

  user = add_user(username, password);
  token = generate_login_token(user);
  return token;
}
```

Login

On the frontend this algorithm needs to run some basic validation and then call the lgoin backend API with the inputted username and password. The server will then check the username and password are correct, if it is then it generates a login token and returns it to the frontend which will then save it, before going to the overview page.

```
function login_frontend(username, password) {
  if username.is_empty() or password.is_empty() {
    return;
  }

  token = login_backend(username, password);
  // Error handling
  save_login_token(token);
  goto("overview")
}

function login_backend(username, password) {
  user = get_user(username); // Get user with username from database, null if doesn't exist

  if user == null {
    return ERR_USER_DOES_NOT_EXIST;
  }

  if password != user.password {
    return ERR_PASSWORD_INCORRECT;
  }

  token = generate_login_token(user);
  return token;
}
```

Get Tasks API

This algorithm is used by the frontend to fetch the list of tasks to display in the todo list view.

```
function get_tasks(token) {
  user = get_user_from_token(token);
  tasks = get_tasks().filter(task => task.owner == user);
  return tasks;
}
```

Get Schedule API

This algorithm is used by the frontend to fetch the schedule to display the current and upcoming tasks in the overview page.

```
function get_schedule(token) {
  user = get_user_from_token(token);
  schedule = get_schedules().filter(schedule => schedule.owner == user)[0]; // A user will
only have one schedule
  return schedule;
}
```

Edit Task API

This algorithm edits a task stored in the database, the schedule must be regenerated to keep it up to date

```
function edit_task(token, oldTask, newTask) {
  user = get_user_from_token(token);
  if user != oldTask.owner { return ERR_UNAUTHORISED; }
```

```
    edit_task_query(oldTask, newTask);
    generate_schedule(user);
}
```

Delete Task API

This algorithm deleted a task stored in the database, the schedule must be regenerated to keep it up to date

```
function delete_task(token, task) {
  user = get_user_from_token(token);
  if user != task.owner { return ERR_UNAUTHORISED; }
  delete_task_query(task);
  generate_schedule(user);
}
```

Generate Schedule

This algorithm takes a list of tasks and returns a schedule with every task somewhere on it, this algorithm will be developed properly as I iterate on the implemented version, so for now I've picked a very simple algorithm which will be optimised in the future. All it currently does is sort the tasks by the deadline and then packs them into the schedule.

```
struct ScheduledTask {
  start: Time,
  end: Time,
  task: Task
}

type Schedule = Array<ScheduledTask>

function generate_schedule(tasks: Array<Task>, day_start: Time, day_end: Time) -> Schedule {
  sorted = tasks.sort(a, b => a.deadline - b.deadline);
  start = now();
  schedule = [];
  for task in sorted {
    if day_end < (start + task.duration) { // If the task would finish after the day's ended,
move it to the beginning of the next day
      start = day_start;
    }

    schedule.push(ScheduledTask { start, end: start + task.duration, task }); // Add the task
to the schedule
    start += task.duration; // Advance the start time by the task's duration
  }
  return schedule;
}
```

This algorithm has a lot of flaws, like it doesn't pack tasks a the end of a day and doesn't nessesarily meet all deadlines, but it will work the majority of the time, and I can improve it once the rest of the product is built.

Data Types

```
struct User {
  username: String,
  password: String,
}

struct Task {
  title: String,
  duration: Duration,
```

```
    deadline: Time,
    owner: User
}

struct ScheduledTask {
    start: Time,
    end: Time,
    task: Task
}

struct Schedule {
    tasks: Array<ScheduledTask>,
    owner: User
}
```

Validation

Most validation has been included in algorithms planning, but I will need
to ensure User references in the Schedule and Task struct are valid when I
use them to ensure the program is robust.

Testing

I will take an iterative approach to testing where after each feature is
implemented I will run the associated tests to check its working, if it
doesn't work then I'll fix it before moving on to the next iteration.

Login

No.	Description	Data	Expected
L1	Log in with a correct username and password	Valid	Login succeeds and returns a auth token
L2	Log in with a username that does not exist	Invalid	Error is raised and user is asked to sign up
L3	Log in with a correct username and invalid password	Invalid	Error is raised and user is asked to recheck their password
L4	Log in with either an empty username or password	Invalid	Error is raised and user is asked to fill username and password boxes
L5	Goes to log in page while already logged in	Boundary	User is redirected to overview page

Signup

No.	Description	Data	Expected
S1	Sign up with an unused username and password	Valid	Sign up succeeds and returns a auth token
S2	Sign up with a used username	Invalid	Error is raised and user is asked to choose a different username
S3	Sign up with either an empty username or password	Invalid	Error is raised and user is asked to fill username and password boxes

| S4 | Goes to sign up page while already logged in | Valid | User is redirected to overview page |

Overview

No.	Description	Data	Expected
01	Go to page while there is a current task	Valid	Page shows current task, remaining time, and other information
02	Go to page while there is not a current task	Valid	Page shows there is no task and shows the next upcoming task
03	Complete button pressed while there is a current task	Valid	Completes the task, regenerates schedule and updates page
04	Complete button pressed while there is no current task	Invalid	Errors saying there is no ongoing task
05	Snooze button pressed while there is a current task	Valid	Snoozes the task, regenerates the schedule and updates page
06	Snooze button pressed while there is no current task	Invalid	Errors saying there is no ongoing task
07	Break button pressed	Valid	Pop up prompting for break duration
08	Duration entered into break popup	Valid	Clears all task for the entered duration, regenerates schedule and updates the page
09	Task list button pressed	Valid	Goes to task list page
010	Schedule list button pressed	Valid	Goes to schedule view page
011	Overview button pressed	Valid	Nothing happens
012	Any task pressed	Valid	Task edit UI appears

Task List

No.	Description	Data	Expected
T1	Go to page	Valid	See a list of all tasks
T2	Click on any task	Valid	Opens task edit UI
T3	Overview button pressed	Valid	Goes to overview page
T4	Task list button pressed	Valid	Does nothing
T5	Schedule button pressed	Valid	Goes the schedule view page

Task Edit

No.	Description	Data	Expected
E1	Delete button pressed	Valid	Task deleted and schedule regenerated
E2	Title editted to a non-empty value	Valid	Task title changed

E3	Title editted to an empty value	Invalid	Error message appears saying tasks must have a title
E4	Deadline editted	Valid	Task deadline changed, schedule regenerated
E5	Duration editted to a non-zero value	Valid	Task duration changed, schedule regenerated
E6	Duration editted to a zero	Invalid	Error appears saying tasks must have a duration
E7	Priority changed	Valid	Task priority changed, schedule regenerated

Schedule Generation

No.	Description	Data	Expected
G1	Schedule generated with valid list of tasks	Valid	All tasks are scheduled to be completed before their deadline
G2	Schedule generated with valid list of tasks	Valid	All tasks are scheduled to take their duration
G3	Schedule generated with impossible list of tasks	Invalid	Error raised about impossible task list
G4	Schedule generated with valid list of tasks	Valid	Tasks with higher priorities are often schedules before tasks with lower priorities

Schedule View
Will come up with these once I've decided more about the schedule view

Implementation

Tech Stack
When making web apps there is a massive choice in frameworks and libraries to help with making reactive websites in JavaScript, such as React, Vue, Svelte, SolidJS, and even new WebAssembly libaries like Leptos and Yew which are both written in Rust. I need my app to load anywhere, including places with slow internet so I need my bundle sizes to be as small as possible, so that means frameworks like React, Vue, Leptos and Yew won't be a good fit as they come with large library bloat. Whereas Svelte and SolidJS both compile away when building the website meaning the bundle will be kept small, and only contain the code I use. I've used Svelte in the past and I want to learn something new so I've chosen to use SolidJS for my frontend library.

For the backend, I've also chosed SolidJS as I'm planning to use a fullstack framework based on SolidJS called SolidStart, this allows me to write the website frontend and the backend API in one codebase.

Authentication

Since most of the other features depend on authentication, for example the timetable needs an owner, I've chosen to start with it. Firstly, I designed a login and sign up page based on the UI mockups I created earlier:

TODO: SCREENSHOTS

The base HTML code for this is as follows, I've used some prebuilt components from a library called solid-ui and then modified them to my preference:

```html
<Card class="mx-auto my-16 max-w-[320px]">
  <CardHeader>
    <CardTitle>Sign Up</CardTitle>
  </CardHeader>
  <CardContent>
    <Label for="username">Username</Label>
    <Input onInput={(e) => set("username", e.target.value)} type="username" id="username"
placeholder="Username" autocomplete="username" />
    <br />
    <Label for="password">Password</Label>
    <Input onInput={(e) => set("password", e.target.value)} type="password" id="password"
placeholder="Password" autocomplete="new-password" />
    <br />
    <Label for="confirm">Confirm Password</Label>
    <Input onInput={(e) => set("confirm", e.target.value)} type="password" id="confirm"
placeholder="Confirm Password" autocomplete="new-password" />
  </CardContent>
  <CardFooter>
    <Button onClick={() => signup()} class="w-full" type="submit">Sign Up</Button>
  </CardFooter>
</Card>
```

The form inputs are all placed into a card element then gives the border around the outside, I've also registered the event functions that update the fields in the JavaScript, as well as the button on click event handler.

Next I had to call the server API for signup when the button was pressed, which I used the following JS:

```js
const [fields, setFields] = createStore({ username: "", password: "", confirm: "" });

const signup = async () => {
  // Send POST request to signup with username and password
  const res = await fetch("/api/signup", {
    method: "POST",
    body: JSON.stringify({ username: fields.username, password: fields.password })
  });

  if (res.status === 409) { // Username conflict
    setTaken(true);
    return;
  }

  if (!res.ok) return; // Return if other failure

  showToast({ title: `Welcome ${fields.username}!`, description: "Going to overview..." }); // Pop up

  const token = await res.text();
  localStorage.setItem("token", token); // Save token for later
}
```

```
// Clear errors
const set = (field: "username" | "password" | "confirm", value: string) => {
  setFields(field, value);
};
```

This doesn't do any data validation at the moment, but I just want to get something working. The input data is stored in a SolidJS store, which will update any code that uses it when something changes. Next I moved onto the server-side to build up the api route:

For the database, I've decided to use MongoDB as its simple and I can host it online for free, to interface with it I've chosen to use a library called mongoose which allows me to create schemas and easily manage a connection. Before I started on the API route I needed to connect to the database:

```
import mongoose, { Schema } from "mongoose"
import dotenv from "dotenv"

//get variables from .env
dotenv.config()

//connect to local mongoose server
mongoose.connect(process.env.MONGO_URI)

mongoose.connection
.on("open", () => console.log("Connected to Mongoose"))
.on("error", (error) => console.log(error))

const userSchema = new Schema({
    username: String,
    password: String
})

export const User = mongoose.model("User", userSchema)
```

This loads a URI to connect to from my .env file, this means I don't have to hardcode the database URI which will help keep my database secret and allow me to quickly change it later. Once connected it logs a message to the console and creates a User schema which for now just has a username and a password.

Now I have a database to work with, I started to work on the API route:

```
const SALT_ROUNDS = 10;

export async function POST({ request }: APIEvent) {
    const { username, password } = await request.json();
    if (username == null || password == null) {
        return new Response("Missing field", { status: 400 });
    }

    // TODO: Check username is free
    const existing = await User.findOne({ username });
    if (existing != null) {
        console.log(existing);
        return new Response("Username taken", { status: 409 });
    }

    const hashed = await bcrypt.hash(password, SALT_ROUNDS);
```

```
    const user = new User();
    user.username = username;
    user.password = hashed;
    await user.save();

    const token = jwt.sign({ username }, process.env.JWT_SECRET);

    return new Response(token);
}
```

This code gets the username and password out of the incoming request from
the frontend, it also checks these both are present before continuing, else
it responds with an error code. Then the route queries the database to
check there aren't any users with the same username, if there are the route
returns a 409 Conflict code and an error will be displayed on the website.
Next the password is hashed using bcrypt to ensure no passwords will be
leaked in the case of a data breach, this is more secure than standard
encryption because hashing is a one way function its very hard to get the
password back from the hashed form, even if you have the key used to hash
it. This is not true for standard encryption algorithms, as if you have the
key most encrypted data can be easily decrypted to get back the original
passwords.

Next the user record is created and filled out using the username and the
hashed password, and then saved to the database. Finally the username is
encoded into a JSON Web Token, which is a nice why of encrypting some data
into a token, with an encrypted secret that can be used to confirm the
token was issued by the server. The JWT will be used to authenticate the
user in other API routes.

Signup Testing
Now that the signup UI has been made and the API route was written, I
decided to do a small test to check that the user account is created in the
database and the JWT issued is valid.

Test Id	Test Title	Expected	Outcome	Fix
S1	Sign up with an unused username and password	Sign up succeeds and returns a auth token	Record is created in database and token is issued	
S2	Sign up with a used username	Error is raised and user is asked to choose a different username	API route fails but no error is visibly raised	Display error in UI
S3	Sign up with either an empty username or password	Error is raised and user is asked to fill username and password boxes	API route fails but no error is visibly raised	Display error in UI

S4	Goes to sign up page while already logged in	User is redirected to overview page	N/A	

The API route was working well but the UI was not raising any errors about empty fields or the username being taken, so I decided to add some red labels next to the fields to explain the issue.

```
<Label for="username">Username</Label>
<Input onInput={(e) => set("username", e.target.value)} type="username" id="username"
placeholder="Username" autocomplete="username" />
{empty.username ? <p class="text-red-500">Cannot be empty</p> : <></>}
{taken() ? <p class="text-red-500">Username taken</p> : <></>}
<br />
<Label for="password">Password</Label>
<Input onInput={(e) => set("password", e.target.value)} type="password" id="password"
placeholder="Password" autocomplete="new-password" />
{empty.password ? <p class="text-red-500">Cannot be empty</p> : <></>}
<br />
<Label for="confirm">Confirm Password</Label>
<Input onInput={(e) => set("confirm", e.target.value)} type="password" id="confirm"
placeholder="Confirm Password" autocomplete="new-password" />
{empty.confirm ? <p class="text-red-500">Cannot be empty</p> : <></>}
{mismatch() ? <p class="text-red-500">Passwords do not match</p> : <></>}
```

This UI toggles error labels if different values are set when processing the submit request. Next I needed to write the JS code to toggle the values depending on the issue with the users input:

```
const [empty, setEmpty] = createStore({ username: false, password: false, confirm: false });
const [mismatch, setMismatch] = createSignal(false);
const [taken, setTaken] = createSignal(false);

...

    let error = false;

    // Username empty
    if (fields.username == "") {
      setEmpty("username", true);
      error = true;
    }

    // Password empty
    if (fields.password == "") {
      setEmpty("password", true);
      error = true;
    }

    // Confirm password empty
    if (fields.confirm == "") {
      setEmpty("confirm", true);
      error = true;
    }

    // Mismatch
    if (fields.password != fields.confirm) {
      setMismatch(true);
      error = true;
    }
```

```
    if (error) return;

    // Send POST request to signup with username and password
    const res = await fetch("/api/signup", {
      method: "POST",
      body: JSON.stringify({ username: fields.username, password: fields.password })
    });

    if (res.status === 409) { // Username conflict
      setTaken(true);
      return;
    }

    ...
```

This code raises errors if any of the fields are empty, or if the passwords
don't match, or if the username is taken and shows a red label in the UI.

TODO: SCREENSHOTS

Signup Error Testing

Test Id	Test Title	Expected	Outcome	Fix
S1	Sign up with an unused username and password	Sign up succeeds and returns a auth token	Record is created in database and token is issued	
S2	Sign up with a used username	Error is raised and user is asked to choose a different username	API route fails and error appears in UI	
S3	Sign up with either an empty username or password	Error is raised and user is asked to fill username and password boxes	API route fails and error appears in UI	
S4	Goes to sign up page while already logged in	User is redirected to overview page	N/A	

With the sign in page success criteria fully met, except for S4 which
depends on the overview page, I decided to move onto the login page

Login
Firstly I took the signup UI and duplicated it as I thought it would be
easier to modify it into a login view:

```
<Card class="mx-auto my-16 max-w-[320px]">
  <CardHeader>
    <CardTitle>Login</CardTitle>
  </CardHeader>
  <CardContent>
    <Label for="username">Username</Label>
    <Input onInput={(e) => set("username", e.target.value)} type="username" id="username"
placeholder="Username" autocomplete="username" />
    {empty.username ? <p class="text-red-500">Cannot be empty</p> : <></>}
    {incorrect() ? <p class="text-red-500">Incorrect username or password</p> : <></>}
    <br />
```

```
    <Label for="password">Password</Label>
    <Input onInput={(e) => set("password", e.target.value)} type="password" id="password"
placeholder="Password" autocomplete="new-password" />
    {empty.password ? <p class="text-red-500">Cannot be empty</p> : <></>}
    {incorrect() ? <p class="text-red-500">Incorrect username or password</p> : <></>}
  </CardContent>
  <CardFooter>
    <Button onClick={() => login()} class="w-full" type="submit">Login</Button>
  </CardFooter>
</Card>
```

This is very similar to the signup UI with the confirm password input
removed and the error messages changed slightly, next I moved onto the
verification login and sending the request to the server:

```
const [empty, setEmpty] = createStore({ username: false, password: false });
const [fields, setFields] = createStore({ username: "", password: "" });
const [incorrect, setIncorrect] = createSignal(false);

const login = async () => {
  let error = false;

  if (fields.username == "") {
    setEmpty("username", true);
    error = true;
  }

  if (fields.password == "") {
    setEmpty("password", true);
    error = true;
  }

  if (error) return;

  const res = await fetch("/api/login", {
    method: "POST",
    body: JSON.stringify({ username: fields.username, password: fields.password })
  });

  if (res.status === 401) {
    setIncorrect(true);
    return;
  }

  if (!res.ok) return;

  showToast({ title: `Welcome ${fields.username}!`, description: "Going to overview..." });
  const token = await res.text();
  localStorage.setItem("token", token);
}

const set = (field: "username" | "password", value: string) => {
  setFields(field, value);
  setEmpty(field, false);
};
```

With the frontend page finished, I moved onto the API route. It needs to
get the user record from the database using the username from the request,
and then compare the hashed password with the password from the request. If
the username and the password is correct, then a JWT is generated similar
to the signup API and returned to the frontend:

```
export async function POST({ request }: APIEvent) {
    const {username, password} = await request.json();

    const user = await User.findOne({username});
    if (!await bcrypt.compare(password, user.password)) {
        return new Response("Incorrect username or password", {status: 401})
    }

    const token = jwt.sign({username}, process.env.JWT_SECRET);
    return new Response(token)
}
```

With login route finished I decided to test it:

Login Testing

Test Id	Test Title	Expected	Outcome	Fix
L1	Log in with a correct username and password	Login succeeds and returns a auth token	As expected	
L2	Log in with a username that does not exist	Error is raised and user is asked to sign up	As expected	
L3	Log in with a correct username and invalid password	Error is raised and user is asked to recheck their password	As expected	
L4	Log in with either an empty username or password	Error is raised and user is asked to fill username and password boxes	As expected	
L5	Goes to log in page while already logged in	User is redirected to overview page	N/A	

All tests passed except for the one that requires the overview page, which is what I decided to work on next.

Move to SvelteKit

While developing the overview page, I kept running into strange issues where API routes would return HTML of other pages instead of running the API route, for instance /api/signup would sometimes return the HTML of the page for sign up. After researching a bit I found that this was a bug within solid-start, the framework I was using, and that it was in early access and very unstable, I tried to find a way to work around this most of them meant iterating would take too long as I'd have to run a release build every time or I'd have to move out the server-side into a different program, which defeated the purpose of a full-stack framework. With this is mind I decided to switch to SvelteKit, another full-stack framework, but this time using Svelte instead of SolidJS. I already had some experience with SvelteKit so it was quite quick to rebuild authentication in it:

I decided to use a library called superforms which made the verifcation of forms and handling all the accessibility and error messages for me, which made the code a lot shorter and simpler:

login/+page.ts

```
<script lang="ts">
// Imports omitted
export let data: SuperValidated<Infer<Schema>>;

const form = superForm(data, {
validators: zodClient(schema)
});

const { form: formData, enhance, message } = form;

message.subscribe((token) => {
toast('Logged in');
});
</script>

<Card class="mx-[40vw] my-16 p-16">
<form method="POST" use:enhance>
<Form.Field {form} name="username">
<Form.Control let:attrs>
<Form.Label>Username</Form.Label>
<Input {...attrs} bind:value={$formData.username} />
</Form.Control>
<Form.FieldErrors />
</Form.Field>
<Form.Field {form} name="password">
<Form.Control let:attrs>
<Form.Label>Password</Form.Label>
<Input {...attrs} bind:value={$formData.password} type="password" />
</Form.Control>
<Form.FieldErrors />
</Form.Field>
<br />
<Form.Button>Log In</Form.Button>
</form>
</Card>
<Toaster />
```

Sign up was nearly exactly the same but with some different labels, so I won't show the code. When the form is submitted it calls a server action, which can be handled in the routes `+page.server.ts` file.

Here's the handling code for login:

login/+page.server.ts

```
// Import omitted

export const actions: Actions = {
    default: async (event) => {
        const form = await superValidate(event, zod(schema));
        if (!form.valid) {
            return fail(400, {
                form,
            });
        }

        let user = await db.query.users.findFirst({ where: eq(users.username,
form.data.username) });
        if (user == null) {
            return setError(form, "username", "User not found");
        }

        if (!await compare(form.data.password, user.password)) {
```

```
                return setError(form, "password", "Incorrect password");
        }

        let token = jwt.sign(form.data.username, JWT_SECRET);

        return message(form, token);
    },
};
```

This is nearly identical to the previous version, the data is validated and the passwords are compared, if they match then a JWT is returned to the client. The only big change here is that I decided to swap my database manager from mongoose which uses MongoDB, to drizzle-orm which uses SQL, mainly because the query syntax is very similar to SQL and I wanted to get some practice in for my exam, but also because its a lot faster and lighter than mongoose. It also has the added benefit of being fully type safe which means I can catch database errors early.

The handling for signup is also very similar:

+page.server.ts

```
// Imports omitted

const SALT_ROUNDS = 10;

export const actions: Actions = {
    default: async (event) => {
        const form = await superValidate(event, zod(schema));
        if (!form.valid) {
            return fail(400, {
                form,
            });
        }

        let hashed = await hash(form.data.password, SALT_ROUNDS);

        if (await db.query.users.findFirst({
            where: eq(users.username, form.data.username)
        })) {
            return setError(form, "username", "Username taken");
        }

        await db
            .insert(users)
            .values({ username: form.data.username, password: hashed });

        console.log("Created user");
        let token = jwt.sign(form.data.username, JWT_SECRET);

        return message(form, token);
    },
};
```

Now that I've moved to a more stable framework, I could finally move onto the tasks view:

Signup Error Testing

Test Id	Test Title	Expected	Outcome	Fix

S1	Sign up with an unused username and password	Sign up succeeds and returns a auth token	Record is created in database and token is issued	
S2	Sign up with a used username	Error is raised and user is asked to choose a different username	API route fails and error appears in UI	
S3	Sign up with either an empty username or password	Error is raised and user is asked to fill username and password boxes	API route fails and error appears in UI	
S4	Goes to sign up page while already logged in	User is redirected to overview page	N/A	

Login Testing

Test Id	Test Title	Expected	Outcome	Fix
L1	Log in with a correct username and password	Login succeeds and returns a auth token	As expected	
L2	Log in with a username that does not exist	Error is raised and user is asked to sign up	As expected	
L3	Log in with a correct username and invalid password	Error is raised and user is asked to recheck their password	As expected	
L4	Log in with either an empty username or password	Error is raised and user is asked to fill username and password boxes	As expected	
L5	Goes to log in page while already logged in	User is redirected to overview page	N/A	

Tasks

This time I decided to start with the API routes instead of the frontend, as I knew I was going to need all CRUD (Create Read Update Delete) methods on tasks. I started with the GET route:

api/tasks/+server.ts

```ts
export async function GET({ cookies }) {
    const token = cookies.get("token");
    if (token == null) {
        return error(401);
    }

    let tasks = getTasks(token);

    if (typeof tasks == "number") {
        return error(tasks);
    }
```

```
        return new Response(JSON.stringify(tasks));
    }
```

I decided to split out the majority of the logic into a separate getTasks function as I knew that was going to be used in different places such as scheduling and the GET route:

```
export async function getTasks(token: string): Promise<Array<Task> | number> {
    let username = await jwt.verify(token, JWT_SECRET);
    if (typeof username != "string") { return 400; }
    let user = await db.query.users.findFirst({
        with: {
            tasks: true
        },
        where: eq(users.username, username)
    });

    if (user == null) {
        return 404;
    }

    return user.tasks;
}
```

This function verifies the JWT to get the username, which is then uses to run a query for all tasks belonging to the user, if the username is invalid the function returns a 404, if the JWT is invalid the function returns a 400 client error, if everything is successful the function returns the list of tasks belonging to the user.

Next I decided to implement the POST route which will be used to create the tasks, it takes a task structure from the requests body, as well as the token located in the users cookies, to create the task in the database:

```
export async function POST({ request, cookies }) {
    let { deadline, duration, title } = await request.json();
    let deadlineDate = new Date(deadline);
    const token = cookies.get("token");
    if (token == null) {
        return error(401);
    }

    let username = await jwt.verify(token, JWT_SECRET);
    if (typeof username != "string") { return error(400, "Bad token"); }
    const user = await db.query.users.findFirst({ where: eq(users.username, username) });
    if (user == null) {
        return error(404);
    }

    const task = await db.insert(tasks).values({ deadline: deadlineDate, duration, title,
user_id: user.id }).returning({ id: tasks.id });

    return new Response(JSON.stringify({ id: task[0].id }));
}
```

If the token is not set then a 401 unauthorised code is returned, if the username is invalid then a 404 not found is returned, else the primary key of the task is returned, this is so the user can manipulate the task later using that primary key.

Next I made the PATCH route, which updates a task:

```
export async function PATCH({ request, cookies }) {
    const { id, deadline, duration, title } = await request.json();
    let deadlineDate = new Date(deadline);

    const token = cookies.get("token");
    if (token == null) {
        return error(401);
    }

    let username = await jwt.verify(token, JWT_SECRET);
    if (typeof username != "string") { return error(400, "Bad token"); }
    const user = await db.query.users.findFirst({ where: eq(users.username, username) });
    if (user == null) {
        return error(404);
    }

    const task = await db.query.tasks.findFirst({ where: eq(tasks.id, id) });
    if (task == null) {
        return error(404);
    }

    if (task.user_id != user.id) {
        return error(401);
    }

    await db.update(tasks).set({ deadline: deadlineDate, duration, title }).where(eq(tasks.id,
id));

    return new Response();
}
```

This route takes a the primary key of the task the user wants to edit, as
well as the new data to update the task to. The task is found from the
database to verify that it belongs to the authenticating user, if it does
then the task is updated and the route returns successfully.

Finally was the DELETE route, which deletes a task using its primary key:

```
export async function DELETE({ request, cookies }) {
    const { id } = await request.json();
    const token = cookies.get("token");
    if (token == null) {
        return error(401);
    }

    let username = await jwt.verify(token, JWT_SECRET);
    if (typeof username != "string") { return error(400, "Bad token"); }
    const user = await db.query.users.findFirst({ where: eq(users.username, username) });
    if (user == null) {
        return error(404);
    }

    const task = await db.query.tasks.findFirst({ where: eq(tasks.id, id) });
    if (task == null) {
        return error(404);
    }

    if (task.user_id != user.id) {
        return error(401);
    }

    await db.delete(tasks).where(eq(tasks.id, id));
```

```
    return new Response();
}
```

As with the PATCH route, the task is fetched before the operating to verify
the task belongs to the authenticated user, if it does then the task is
deleted.

With the task manipulation API routes written I decided to move onto the
website UI. As I was still using the svelte-shadcn component library, I had
a look at the easiest way to display a grid of information, and I came
across the table component, I thought it looked really nice and did
everything I wanted to do so I made the decision to center my UI around it.
I started by mocking up a basic table with to display information how I
wanted:

```ts
<script lang="ts">
export let data;
let tasks = writable(data.tasks);

let staging: Task = { deadline: new Date(Date.now()), duration: 0, title: '', id: 0 };

function duration(duration: number): string {
function pad(data: string): string {
if (data.length == 1) {
return data + '0';
} else {
return data;
}
}

let hours = Math.floor(duration / 3600);
let minutes = Math.floor((duration % 3600) / 60);

return `${hours}:${pad(minutes.toString())}`;
}

function deadline(deadline: Date): string {
console.log(deadline);
if (deadline.toDateString() == new Date(Date.now()).toDateString()) {
return `${deadline.getHours()}:${deadline.getMinutes()}`;
} else {
return `${deadline.getDate()}-${deadline.getMonth()}-${deadline.getFullYear()}`;
}
}
</script>

<Card.Root class="m-16 px-16 pb-16 pt-4">
<Card.Header>
<Card.Title class="cols-span-9 text-center text-5xl">Tasks</Card.Title>
</Card.Header>
<Card.Root>
<Table.Root>
<Table.Header>
<Table.Row>
<Table.Head>Title</Table.Head>
<Table.Head>Duration</Table.Head>
<Table.Head>Deadline</Table.Head>
</Table.Row>
</Table.Header>
<Table.Body>
{#each $tasks as task}
```

```
<Table.Row>
<Table.Cell>{task.title}</Table.Cell>
<Table.Cell>{duration(task.duration)}</Table.Cell>
<Table.Cell>{deadline(task.deadline)}</Table.Cell>
</Table.Row>
{/each}
</Table.Body>
</Table.Root>
</Card.Root>
</Card.Root>
```

The array of tasks is stored in one of svelte's writable stores, they allow me to control the reactivity of the tasks array so I can force the UI to update when I come to updating the information. I can do this by calling the `update` function on the store, which rerenders all the UI using the store after changing the value.

Next I needed to add some buttons in order to interact with the task table, I started with an edit and delete button which I placed in a column to the right.

```
<Card.Root class="m-16 px-16 pb-16 pt-4">
<Card.Header>
<Card.Title class="cols-span-9 text-center text-5xl">Tasks</Card.Title>
</Card.Header>
<Card.Root>
<Table.Root>
<Table.Header>
<Table.Row>
<Table.Head>Title</Table.Head>
<Table.Head>Duration</Table.Head>
<Table.Head>Deadline</Table.Head>
        <Table.Head class="px-8 text-right">Actions</Table.Head>
</Table.Row>
</Table.Header>
<Table.Body>
{#each $tasks as task}
<Table.Row>
<Table.Cell>{task.title}</Table.Cell>
<Table.Cell>{duration(task.duration)}</Table.Cell>
<Table.Cell>{deadline(task.deadline)}</Table.Cell>
        <Table.Cell class="text-right">
<Button variant="outline" size="icon">
<Pencil class="h-4 w-4" />
</Button>
<Button variant="destructive" size="icon">
<Trash class="h-4 w-4" />
</Button>
</Table.Cell>
</Table.Row>
{/each}
</Table.Body>
</Table.Root>
</Card.Root>
</Card.Root>
```

With the buttons in place, I now needed to make them do something, I decided the delete button would be the easiest to implement first as it didn't need a form.

```
<script lang="ts">
  ...
```

```
  async function deleteTask(task: Task) {
await fetch('/api/tasks', { method: 'DELETE', body: JSON.stringify({ id: task.id }) });
tasks.update((tasks) => {
tasks.splice(data.tasks.indexOf(task), 1);
return tasks;
});
}
</script>

...
<Button variant="destructive" size="icon" on:click={() => deleteTask(task)}>
<Trash class="h-4 w-4" />
</Button>
...
```

Next was the task editting/creation UI, I decided to put it all in a dialog
box that pops up when you press the edit/create buttons. Because this was
going to be a standalone component, I decided to break it up into its own
file:

```
<script lang="ts">
export let task: Task;

let title: string;
let hours: number, minutes: number;
let deadlineDate: DateValue, deadlineHours: number, deadlineMinutes: number;
let open: bool = false;

function start() {
title = task.title;
hours = task.duration / 3600;
minutes = (task.duration % 3600) / 60;
deadlineDate = fromDate(task.deadline, 'Europe/London');
deadlineHours = task.deadline.getHours();
deadlineMinutes = task.deadline.getMinutes();
}

async function submit() {
let duration = hours * 3600 + minutes * 60;
let deadline = deadlineDate.toDate('Europe/London');
deadline.setHours(deadlineHours);
deadline.setMinutes(deadlineMinutes);
task = { title, duration, deadline, id: task.id };
await fetch('/api/tasks', { method: 'PATCH', body: JSON.stringify(task) });
open = false;
}
</script>

<Dialog.Root bind:open>
<Dialog.Trigger class={buttonVariants({ variant: 'outline' })} on:click={start}>
<slot />
</Dialog.Trigger>
<Dialog.Content>
<Dialog.Header>
<Dialog.Title class="text-center">Edit task</Dialog.Title>
</Dialog.Header>
<div class="grid gap-4 py-4">
<div class="grid grid-cols-5 items-center gap-4">
<Label for="title" class="text-right">Title</Label>
<Input id="title" bind:value={title} class="col-span-4" />
</div>
```

```
<div class="grid grid-cols-5 items-center gap-4">
<Label for="duration" class="text-right">Duration</Label>
<Input id="hours" bind:value={hours} class="col-span-2" />
<Input id="minutes" bind:value={minutes} class="col-span-2" />
</div>
<div class="grid grid-cols-5 items-center gap-4">
<Label for="duration" class="text-right">Deadline</Label>
<DatePicker bind:value={deadlineDate} class="col-span-4" />
</div>
<div class="grid grid-cols-5 items-center gap-4">
<p />
<Input id="hours" bind:value={deadlineHours} class="col-span-2" />
<Input id="minutes" bind:value={deadlineMinutes} class="col-span-2" />
</div>
</div>
<Dialog.Footer>
<Button type="submit" on:click={submit}>Save</Button>
</Dialog.Footer>
</Dialog.Content>
</Dialog.Root>
```

This Svelte code creates a dialog popup and a trigger button, the popup is a full form including a date picker for the deadline, and several inputs for title, duration minutes and hours, and deadline minutes and hours. I used HTML grids in order to align and structure the form to keep it easy to use and aesthetic.

Another thing to note is that instead of directly modifying the task passed into the component, I instead read out the task's values and copy them to other variables, which are modified instead, this is so the original task isn't modified until the request is send, after which the values are copied back into the task. Originally, I tried to modify the original and keep a backup to restore to if the request failed, but I switched to this method as it was simpler.

I did a quick test at this point to check deletion and editting were working correctly, to do this I filled the database with mock data and used the UI to edit them.

Task Editing and Deletion Testing

Test Id	Test Title	Expected	Outcome
E1	Delete button pressed	Task deleted and schedule regenerated	Task deleted and UI updated
E2	Title edited to a non-empty value	Task title changed	Failed due to server error
E3	Title edited to an empty value	Error message appears saying tasks must have a title	Failed due to server error
E4	Deadline edited	Task deadline changed, schedule regenerated	Failed due to server error
E5	Duration edited to a non-zero value	Task duration changed, schedule regenerated	Failed due to server error

E6	Duration editted to a zero	Error appears saying tasks must have a duration	Failed due to server error
E7	Priority changed	Task priority changed, schedule regenerated	N/A as there is no priority yet

All of the tests other than deletion failed due to a server error: "value.toUTCString is not a function" being raised from within my PostgreSQL driver. This meant I had an issue with handling the deadline Date object on the edit API route, taking a look at the request body, the deadline was being represented as a UTC string rather than an actual Date object, which is what the server was expecting. To fix this I needed to either make the server parse the UTC string, or use a different format which the server could easily parse. Parsing a UTC timedate string is very slow and not in the JS standard library, so I decided to search for a exchange format, I found the `Date.getTime()` function which returns the number of seconds since the 1970 epoch, similar to the UNIX timestamp used in UNIX derivates. This meant that it was just a simple number I could send in the request body, and it could easily be turned back into a Date object, in fact its constructor takes one of these values. So with a bit of modification I should be able to fix the issue:

api/tasks/+server.ts

```
export async function PATCH({ request, cookies }) {
    const { id, deadline, duration, title } = await request.json();
    let deadlineDate = new Date(deadline);

    ...

    await db.update(tasks).set({ deadline: deadlineDate, duration, title }).where(eq(tasks.id,
id));

    return new Response();
}
```

tasks/Edit.svelte

```
async function submit() {
let duration = hours * 3600 + minutes * 60;
let deadline = deadlineDate.toDate('Europe/London');
deadline.setHours(deadlineHours);
deadline.setMinutes(deadlineMinutes);
task = { title, duration, deadline, id: task.id };
  const body = {
deadline: task.deadline.getTime(), // Changed
duration: task.duration,
title: task.title,
id: task.id
};
await fetch('/api/tasks', { method: 'PATCH', body: JSON.stringify(body) }); // Changed
open = false;
}
```

With this done I reran the test suite:

Test Id	Test Title	Expected	Outcome

E1	Delete button pressed	Task deleted and schedule regenerated	Task deleted and UI updated
E2	Title edited to a non-empty value	Task title changed	Task title changed
E3	Title edited to an empty value	Error message appears saying tasks must have a title	Changed to empty title
E4	Deadline editted	Task deadline changed, schedule regenerated	Task deadline changed
E5	Duration editted to a non-zero value	Task duration changed, schedule regenerated	Task duration changed
E6	Duration editted to a zero	Error appears saying tasks must have a duration	Client error
E7	Priority changed	Task priority changed, schedule regenerated	N/A as there is no priority yet

This shows the core functionality is working, and all I had left was some validation, to do this I used zod again, but without superforms this time:

```ts
let schema = z.object({
hours: z.number().min(0),
minutes: z.number().min(0).max(60),
deadlineHours: z.number().min(0),
deadlineMinutes: z.number().min(0).max(60),
deadlineDate: z.date(),
title: z.string().min(1)
});

const result = schema.safeParse({ hours, minutes, deadlineHours, deadlineMinutes,
deadlineDate: deadlineDate.toDate('Europe/London'), title });
if (!result.success) {
  console.log(result.error.issues)
  issues = result.error.issues.map(issue => `${issue.path[0]}: ${issue.message}`);
  return
}
```

The errors are not the best since I'm not using superforms, at somepoint I'll probably go back and fix this but for now it works well. As part of this I also made a NumericInput component, which is like an input but it only lets you input digits:

NumericInput.svelte

```ts
<script lang="ts">
import { Input } from '$lib/components/ui/input';

let className: string | null | undefined = undefined;
export let value: number;
  let text: string
export { className as class };

function check() {
```

```
let c = text[text.length - 1];
if (c < '0' || c > '9') {
text = text.substring(0, text.length - 2);
} else {
        value = parseInt(text);
    }
}
</script>
```

```
<Input class={className} bind:value={text} on:input={check} />
```

I replaced the standard Input component with these, meaning the hours and minutes fields were already guarenteed to be numeric. With validation done I decided to rerun the test suite:

Test Id	Test Title	Expected	Outcome
E1	Delete button pressed	Task deleted and schedule regenerated	Task deleted and UI updated
E2	Title editted to a non-empty value	Task title changed	Task title changed
E3	Title editted to an empty value	Error message appears saying tasks must have a title	Error message appears saying tasks must have a title
E4	Deadline editted	Task deadline changed, schedule regenerated	Task deadline changed
E5	Duration editted to a non-zero value	Task duration changed, schedule regenerated	Task duration changed
E6	Duration editted to a zero	Error appears saying tasks must have a duration	Error appears saying tasks must have a duration
E7	Priority changed	Task priority changed, schedule regenerated	N/A as there is no priority yet

With all test passing I was happy with the editting UI, so I decided to work on generalising it to work with both the creation form and the editting form, firstly I made the form title an input, as well as taking an effect callback which takes the task from the form and does something with it:

```
<script lang="ts">
  ...

  export let dialogTitle: string;
export let callback: (task: Task) => void;

  async function submit() {
let schema = z.object({
hours: z.number().min(0),
minutes: z.number().min(0).max(60),
deadlineHours: z.number().min(0),
deadlineMinutes: z.number().min(0).max(60),
deadlineDate: z.date(),
```

```
      title: z.string().min(1)
});

const result = schema.safeParse({ hours, minutes, deadlineHours, deadlineMinutes,
deadlineDate: deadlineDate.toDate('Europe/London'), title });
if (!result.success) {
            console.log(result.error.issues)
            issues = result.error.issues.map(issue => `${issue.path[0]}: ${issue.message}`);
            return
}

let duration = hours * 3600 + minutes * 60;
let deadline = deadlineDate.toDate('Europe/London');
deadline.setHours(deadlineHours);
deadline.setMinutes(deadlineMinutes);
task = { title, duration, deadline, id: task.id };
callback(task);
open = false;
}

  ...
</script>

    ...

    <Dialog.Header>
<Dialog.Title class="text-center">{dialogTitle}</Dialog.Title>
</Dialog.Header>

    ...
```

With that I added the create button at the top of the page, created a
patchTask and createTask function, and wired up the new generic UI:

```ts
<script lang="ts">
export let data;
let tasks = writable(data.tasks);

let staging: Task = { deadline: new Date(Date.now()), duration: 0, title: '', id: 0 };

function duration(duration: number): string {
function pad(data: string): string {
if (data.length == 1) {
return data + '0';
} else {
return data;
}
}

let hours = Math.floor(duration / 3600);
let minutes = Math.floor((duration % 3600) / 60);

return `${hours}:${pad(minutes.toString())}`;
}

function deadline(deadline: Date): string {
console.log(deadline);
if (deadline.toDateString() == new Date(Date.now()).toDateString()) {
return `${deadline.getHours()}:${deadline.getMinutes()}`;
} else {
return `${deadline.getDate()}-${deadline.getMonth()}-${deadline.getFullYear()}`;
}
}
```

```
async function deleteTask(task: Task) {
await fetch('/api/tasks', { method: 'DELETE', body: JSON.stringify({ id: task.id }) });
tasks.update((tasks) => {
tasks.splice(data.tasks.indexOf(task), 1);
return tasks;
});
}

async function patchTask(task: Task) {
const body = {
deadline: task.deadline.getTime(),
duration: task.duration,
title: task.title,
id: task.id
};
await fetch('/api/tasks', { method: 'PATCH', body: JSON.stringify(body) });
}

async function createTask(task: Task) {
const body = { deadline: task.deadline.getTime(), duration: task.duration, title:
task.title };
let res = await fetch('/api/tasks', { method: 'POST', body: JSON.stringify(body) });
task.id = (await res.json()).id;
tasks.update((tasks) => {
tasks.push(task);
return tasks;
});
}
</script>

<Card.Root class="m-16 px-16 pb-16 pt-4">
<Card.Header>
<Card.Title class="cols-span-9 text-center text-5xl">Tasks</Card.Title>
<br />
<Edit callback={createTask} bind:task={staging} dialogTitle="Create task">
<Plus class="h-4 w-4" />
</Edit>
</Card.Header>
<Card.Root>
<Table.Root>
<Table.Header>
<Table.Row>
<Table.Head class="text-center">Title</Table.Head>
<Table.Head class="text-center">Duration</Table.Head>
<Table.Head class="text-center">Deadline</Table.Head>
<Table.Head class="px-8 text-right">Actions</Table.Head>
</Table.Row>
</Table.Header>
<Table.Body>
{#each $tasks as task}
<Table.Row>
<Table.Cell class="text-center">{task.title}</Table.Cell>
<Table.Cell class="text-center">{duration(task.duration)}</Table.Cell>
<Table.Cell class="text-center">{deadline(task.deadline)}</Table.Cell>
<Table.Cell class="text-right">
<Edit callback={patchTask} dialogTitle="Edit task" bind:task>
<Pencil class="h-4 w-4" />
</Edit>
<Button variant="destructive" size="icon" on:click={() => deleteTask(task)}>
<Trash class="h-4 w-4" />
</Button>
```

```
</Table.Cell>
</Table.Row>
{/each}
</Table.Body>
</Table.Root>
</Card.Root>
</Card.Root>
```

With this done I reran the test suite for the whole page:

No.	Description	Expected	Outcome
T1	Go to page	Valid	See a list of all tasks
See a list of all tasks	T2	Click on any task	Valid
Opens task edit UI	Opens task edit UI	T3	Overview button pressed
Valid	Goes to overview page	N/A	T4
Task list button pressed	Valid	Does nothing	N/A
T5	Schedule button pressed	Valid	Goes the schedule view page
N/A			

Test Id	Test Title	Expected	Outcome
E1	Delete button pressed	Task deleted and schedule regenerated	Task deleted and UI updated
E2	Title editted to a non-empty value	Task title changed	Task title changed
E3	Title editted to an empty value	Error message appears saying tasks must have a title	Error message appears saying tasks must have a title
E4	Deadline editted	Task deadline changed, schedule regenerated	Task deadline changed
E5	Duration editted to a non-zero value	Task duration changed, schedule regenerated	Task duration changed
E6	Duration editted to a zero	Error appears saying tasks must have a duration	Error appears saying tasks must have a duration
E7	Priority changed	Task priority changed, schedule regenerated	N/A as there is no priority yet

All passed as expected, so next I moved onto the schedule generation and overview page:

Schedule Generation

To start I decided to implement the basic schedule generation algorithm from the algorithms planning section. It sorts the tasks based on deadlines and then schedules them between the user's start and end time in that order.

```
export type ScheduledTask = {
    task: Task,
    start: Date,
}

function end(time: string, day: Date): Date {
    let day_copy = new Date(day.getTime()); // Create copy to avoid modifying argument
    let hours = parseInt(time.substring(0, 2));
    let minutes = parseInt(time.substring(3, 5));
    day_copy.setHours(hours);
    day_copy.setMinutes(minutes);
    day_copy.setSeconds(0);
    return day_copy;
}

function nextDay(time: string, day: Date): Date {
    let day_copy = new Date(day.getTime()); // Create copy to avoid modifying argument
    let hours = parseInt(time.substring(0, 2));
    let minutes = parseInt(time.substring(3, 5));
    day_copy.setHours(hours);
    day_copy.setMinutes(minutes);
    day_copy.setSeconds(0);
    day_copy.setMilliseconds(0);
    return new Date(day_copy.getTime() + 24 * 3600);
}

export function generateSchedule(user: User, tasks: Array<Task>): Array<ScheduledTask> {
    const sorted: Array<Task> = tasks.sort((a, b) => a.deadline.getTime() -
b.deadline.getTime());
    let start: Date = new Date(Date.now());
    let schedule: Array<ScheduledTask> = [];
    sorted.forEach(task => {
        const next_day = nextDay(user.start, start);
        const day_end = end(user.end, start);
        if (day_end.getTime() < (start.getTime() + task.duration)) { // If the task would
finish after the day's ended, move it to the beginning of the next day
            start = next_day;
        }

        schedule.push({ start, task }); // Add the task to the schedule
        start =  new Date(start.getTime() + task.duration); // Advance the start time by the
task's duration
    });
    return schedule;
}
```

I decided to put the start and end times in the user database record, this means that eventually I could implement a user configuration page where the user can choose when their scheduled day starts and ends. However this mean I had to modify my databse schema for the user table:

```
export const users = pgTable('users', {
    id: serial('id').primaryKey(),
    username: text('username').notNull(),
    password: text('password').notNull(),
```

```
    start: time('start').notNull(),
    end: time('end').notNull()
});
```

I chose the `time` field type as it only needs to store the time of day, not the date or a specific time stamp, this also meant the exact time of day could be represented in a much cheaper way. I decided to make the defaults for these values 9am till 5pm, inline with the standard working hours in the UK. Since I had modified table columns I had to regenerate migrations and apply them to the database, after that was done I filled the database with some mock data and decided to generate a test schedule.

I created 3 tasks, a 15 minutes task due first, then a 40 minute task, then a 1 hour long task. It was 4pm at the time so I only had an hour left for the scheduler to place tasks, the expected output was that the both the 15 and 40 minute tasks would be scheduled for today, and the 1 hour one would be scheduled for the beginning of tomorrow. After a quick test I could see this wasn't working as expected, all 3 tasks were scheduled for today with odd durations and start times. After digging through the `generateSchedule` function, I found that the `Date.getTime` function actually returns the number of milliseconds since the UNIX epoch, rather than seconds like UNIX timestamps, this meant I was adding duration in milliseconds, despite the fact the value represented seconds, this meant each task took 0.1% of the time it should, and to fix it I'd need to multiply duration by 1000 to convert it to milliseconds:

Bugs: Turns out JS Date.getTime() returns milliseconds instead of seconds like the UNIX timestamp

```
export function generateSchedule(user: User, tasks: Array<Task>): Array<ScheduledTask> {
    const sorted: Array<Task> = tasks.sort((a, b) => a.deadline.getTime() -
b.deadline.getTime());
    let start: Date = new Date(Date.now());
    let schedule: Array<ScheduledTask> = [];
    sorted.forEach(task => {
        const next_day = nextDay(user.start, start);
        const day_end = end(user.end, start);
        if (day_end.getTime() < (start.getTime() + task.duration * 1000)) { // If the task
would finish after the day's ended, move it to the beginning of the next day
            start = next_day;
        }

        schedule.push({ start, task }); // Add the task to the schedule
        start =  new Date(start.getTime() + task.duration * 1000); // Advance the start time
by the task's duration
    });
    return schedule;
}
```

With this fixed I reran the function and this time it the correct schedule, so I moved onto the overview page. Referring back to the design I made earlier, I decided the easiest way to make this UI would be to use a mixture of CSS flex and grid, two things I wasn't very familiar with at the time, so I did a bit of research on how to use them with TailwindCSS. With that I made the overview page, which started with the server side function fetching the schedule.

```
export async function load({ cookies }) {
    const token = cookies.get("token");
```

```ts
    if (token == null) {
        return error(401);
    }

    const user = await getUser(token);
    if (typeof user == "number") {
        return error(user);
    }

    const tasks = await getTasks(user);
    if (typeof tasks == "number") {
        return error(tasks);
    }

    const schedule = await generateSchedule(user, tasks);

    return { schedule };
}
```

This function will be run on the server when the page is accessed, it gets the username out of the token and authenticates the user, the user id is then used to get the list of tasks, which is then used to generate the schedule. Finally the schedule is returned to be used by the Svelte component:

```ts
<script lang="ts">
export let data; // Return value of server-side function will be stored here

let schedule: Array<ScheduledTask> = data.schedule;
  $: schedule = data.schedule;
let current: ScheduledTask | undefined;
let next: Array<ScheduledTask> = [];

  // Constantly check whether current or next tasks need changing
onMount(() => {
setInterval(setCurrent, 1000);
setInterval(setNext, 1000);
});

function setCurrent() {
current = schedule.find((task, a, b) => {
let now = Date.now();
let start = task.start.getTime();
let end = task.start.getTime() + task.task.duration * 1000;
return start <= now && now <= end;
});
}

function setNext() {
next = schedule
.sort((a, b) => a.start.getTime() - b.start.getTime())
.filter((task) => task.start.getTime() > Date.now());
}

function timeUntil(task: ScheduledTask): string {
    let between = task.start.getTime() - Date.now();
    return formatDuration(between / 1000);
  }
</script>

<div class="p-8 md:p-16 lg:mx-[15vw] lg:grid lg:grid-cols-3 lg:gap-4">
<div class="flex flex-col items-stretch gap-4 lg:col-span-2">
```

```
{#if current}
<Task description="Current" task={current} />
{:else}
<Card.Root class="p-16">
<Card.Header>
<Card.Title class="text-center text-5xl">Nothing to do!</Card.Title>
</Card.Header>
</Card.Root>
{/if}
<div class="grid gap-4 max-lg:grid-rows-2 lg:grid-cols-2">
<Button.Root class="p-8 text-xl lg:p-16 lg:text-3xl">Complete</Button.Root>
<Button.Root class="p-8 text-xl lg:p-16 lg:text-3xl">Snooze</Button.Root>
</div>
{#if next[0]}
<Task description={`Next in ${timeUntil(next[0])}`} task={next[0]} />
{:else}
<Card.Root class="p-16">
<Card.Header>
<Card.Title class="text-center text-5xl">Nothing upcoming</Card.Title>
</Card.Header>
</Card.Root>
{/if}
<Button.Root class="p-16 text-3xl">Take a break</Button.Root>
</div>
<div class="hidden grid-rows-4 gap-4 lg:grid">
    {#each next.slice(1, 4) as task}
  <Task description={`Next in ${timeUntil(task)}`} task={task} />
    {/each}
</div>
</div>
```

This component creates the UI from the design section successfully, and it also scales correctly for mobile usage:

TODO: PC Overview UI TODO: Mobile Overview UI

One thing I encountered while making the page is that displaying task information was something needed in many places, so I decided to abstract it into a separate component:

```
<script lang="ts">
    export let task: ScheduledTask;
    export let description: string;

    function duration(): string {
        return formatDuration(task.task.duration);
    }
    function deadline(): string {
        const value = task.task.deadline;
        return `${value.getDate()}/${value.getMonth()}/${value.getFullYear()}`
    }
</script>

<Card.Root>
    <Card.Header>
        <Card.Title class="text-center text-5xl">{task.task.title}</Card.Title>
        <Card.Description class="text-center">{description}</Card.Description>
    </Card.Header>
    <Card.Footer class="lg:px-16">
        <div class="flex grow justify-between">
            <div class="flex grow justify-start gap-2">
                <Clock />
                {duration()}
```

```
            </div>
            <div class="flex grow justify-end gap-2">
                <Calendar />
                {deadline()}
            </div>
        </div>
    </Card.Footer>
</Card.Root>
```

The Task component takes a task and description as input, and displays the task's title, duration and deadline. The usage is shown in the overview page.

With the UI done, the next step was to add some functionality to the buttons, snooze and take a break both required features I haven't added to the scheduling algorithm yet, so I started with the complete button.

The complete button when pressed needs to check if there is a current task, if there is it should be deleted from the database and the schedule should be regenerated:

```
async function complete() {
    if (current == undefined) return;
    await fetch("/api/tasks", { method: "DELETE", body: JSON.stringify({ id:
current.task.id }) });
    await invalidateAll();
}
```

This uses the tasks API I made earlier to delete the task from its ID, and the invalidateAll function refetched the data calculated during the server-side load function, which since that is how the page fetched the schedule, it will update it. At this point I decided to do a run of the overview page's tests:

Overview Testing

No.	Description	Expected	Outcome
01	Go to page while there is a current task	Page shows current task, remaining time, and other information	Page shows information correctly
02	Go to page while there is not a current task	Page shows there is no task and shows the next upcoming task	Page shows information correctly
03	Complete button pressed while there is a current task	Completes the task, regenerates schedule and updates page	Task is deleted and schedule is updated
04	Complete button pressed while there is no current task	Errors saying there is no ongoing task	No effect
05	Snooze button pressed while there is a current task	Snoozes the task, regenerates the schedule and updates page	No effect
06	Snooze button pressed while there is no current task	Errors saying there is no ongoing task	No effect

07	Break button pressed	Pop up prompting for break duration	No effect
08	Duration entered into break popup	Clears all task for the entered duration, regenerates schedule and updates the page	No effect
09	Task list button pressed	Goes to task list page	N/A
010	Schedule list button pressed	Goes to schedule view page	N/A
011	Overview button pressed	Nothing happens	N/A
012	Any task pressed	Task edit UI appears	Removing

O1 through O3 were successful, however O4 failed as I have not yet implemented an error message, O5-O11 fail expectedly as the buttons either have no functionality, or don't exist. When speaking to stakeholders, we decided the ability to click on any task to edit it was not helpful, and would make the website hard to use on mobile platforms, so I have decided to remove O12 from the tests.

With testing completed, everything implemented so far was working correctly, however I still had to implement the snooze and take a break feature. However while testing, despite everything working correctly, there was some odd behaviour. Since the schedule was generated at page load, it would change every time a user would reload the page, or access it on another device. This meant that a users schedule would be inconsistent across devices, which isn't especially easy to follow. To address this I decided to make schedules persistent and only regenerate them when the user's task list is modified. To start I had to modify the task schema:

```
export const tasks = pgTable('tasks', {
    id: serial('id').primaryKey(),
    title: text('title').notNull(),
    duration: integer('duration').notNull(),
    scheduled: timestamp('scheduled'), // New
    deadline: timestamp('deadline', { withTimezone: true }).notNull(),
    user_id: integer('user_id').references(() => users.id)
});
```

I added an optional column, the scheduled time. This column would be used to store the scheduled time when the schedule was generated, meaning the generateSchedule function had to be modified:

```
export async function generateSchedule(user: User, tasks: Array<Task>) {
    const sorted: Array<Task> = tasks.sort((a, b) => a.deadline.getTime() - b.deadline.getTime());
    let start: Date = new Date(Date.now());
    sorted.forEach(task => {
        const next_day = nextDay(user.start, start);
        const day_end = end(user.end, start);
        if (day_end.getTime() < (start.getTime() + task.duration * 1000)) { // If the task would finish after the day's ended, move it to the beginning of the next day
            start = next_day;
        }
```

```
            task.scheduled = start; // New, schedule in this task
            (async () => await db.update(schema.tasks).set({ scheduled:
start }).where(eq(schema.tasks.id, task.id)))(); // New, update database to include scheduled
time
            start = new Date(start.getTime() + task.duration * 1000); // Advance the start time by
the task's duration
        });
    }
```

Now the schedule was persistent I had to update the overview page load
function to fetch the schedule from the database instead of generating it
at load time, this meant returning the task list, since tasks now contained
the time they were scheduled for:

```
export async function load({ cookies }) {
    const token = cookies.get("token");
    if (token == null) {
        return error(401);
    }

    const user = await getUser(token);
    if (typeof user == "number") {
        return error(user);
    }

    const tasks = await getTasks(user);
    if (typeof tasks == "number") {
        return error(tasks);
    }

    return { tasks };
}
```

Finally I needed to move the schedule regeneration to whenever the task
list was modified, which currently was only in the /api/tasks API:

```
export async function POST({ request, cookies }) {
  ...

  const taskList = await getTasks(user);
  if (typeof taskList === "number") {
      return error(taskList);
  }
  await generateSchedule(user, taskList);

  ...
}

export async function DELETE({ request, cookies }) {
  ...

  const taskList = await getTasks(user);
  if (typeof taskList === "number") {
      return error(taskList);
  }
  await generateSchedule(user, taskList);

  ...
}

export async function PATCH({ request, cookies }) {
  ...
```

```
const taskList = await getTasks(user);
if (typeof taskList === "number") {
    return error(taskList);
}
await generateSchedule(user, taskList);

...
}
```
With this done, I decided to retest the overview page just to double check
I didn't break anything:

No.	Description	Expected	Outcome
01	Go to page while there is a current task	Page shows current task, remaining time, and other information	Page shows information correctly
02	Go to page while there is not a current task	Page shows there is no task and shows the next upcoming task	Page shows information correctly
03	Complete button pressed while there is a current task	Completes the task, regenerates schedule and updates page	Task is deleted and schedule is updated
04	Complete button pressed while there is no current task	Errors saying there is no ongoing task	No effect
05	Snooze button pressed while there is a current task	Snoozes the task, regenerates the schedule and updates page	No effect
06	Snooze button pressed while there is no current task	Errors saying there is no ongoing task	No effect
07	Break button pressed	Pop up prompting for break duration	No effect
08	Duration entered into break popup	Clears all task for the entered duration, regenerates schedule and updates the page	No effect
09	Task list button pressed	Goes to task list page	N/A
010	Schedule list button pressed	Goes to schedule view page	N/A
011	Overview button pressed	Nothing happens	N/A
012	Any task pressed	Task edit UI appears	Removing

Everything seemed to be working as before, except with persistent schedules
so I could finally move onto the take a break feature.

Take a Break

Currently since I didn't have a way of forcing tasks to start after a certain time, I decided to just create a task with a high deadline priority when making a break. The task will have the duration specified and a deadline such that it will be scheduled for the first task:

```ts
<script lang="ts">
  let breakHours: number = 0;
  let breakMinutes: number = 0;

  async function takeBreak() {
    let duration = breakHours * 3600 + breakMinutes * 60;
    let deadline = new Date(Date.now() + duration * 1000);
    let title = "Break";
    await fetch("/api/tasks", { method: "POST", body: JSON.stringify({ duration, deadline, title }) });
    await invalidateAll();
  }
</script>

...

<NumericInput bind:value={breakHours} />
<NumericInput bind:value={breakMinutes} />
<Button.Root class="text-xl" on:click={takeBreak}>Take a break</Button.Root>

...
```

I decided to put the break hours and minutes inputs directly on the page rather than behind a popup, mainly to make it easier and more obvious how to enter the information, but also to simplify the UI. With the take a break feature working, there was only one other feature that didn't need start times, which was repeated tasks.

Repeat Tasks

To start with I had to add another column to the `tasks` table, the repeat column, which would specify the interval between repeats in days:

```
export const tasks = pgTable('tasks', {
    id: serial('id').primaryKey(),
    title: text('title').notNull(),
    duration: integer('duration').notNull(),
    scheduled: timestamp('scheduled'),
    repeat: integer('repeat'),
    deadline: timestamp('deadline', { withTimezone: true }).notNull(),
    user_id: integer('user_id').references(() => users.id)
});
```

The column is nullable, as a null value would mean the task is unrepeated. A value of 0 days is also treated as a one-time task. With the field in the database, I next moved onto changing the task edit UI to allow the user to enter a repeat interval:

```
<div class="grid grid-cols-5 items-center gap-4">
<Label for="duration" class="text-right">Repeat</Label>
<NumericInput bind:value={repeat} class="col-span-2" />
<Label for="duration" class="col-span-2 text-left">Days</Label>
</div>
```

This added another row to the form grid, I decided to display the units of the interval to make it clear to the user that the unit used is days. TODO: Screenshot

Now tasks were able to be set to repeat, I needed to reschedule repeat tasks when they are completed, rather than deleting them like I am currently. Since this meant I could no longer just use DELETE /api/tasks I decided to make a new API route /api/complete which would handle the completion of a task:

api/complete/+server.ts

```ts
export async function POST({ request, cookies }) {
    const { id } = await request.json();
    const token = cookies.get("token");
    if (token == null) {
        return error(401);
    }

    let username = await jwt.verify(token, JWT_SECRET);
    if (typeof username != "string") { return error(400, "Bad token"); }
    const user = await db.query.users.findFirst({ where: eq(users.username, username) });
    if (user == null) {
        return error(404);
    }

    const task = await db.query.tasks.findFirst({ where: eq(tasks.id, id) });
    if (task == null) {
        return error(404);
    }

    if (task.user_id != user.id) {
        return error(401);
    }

    if (task.repeat) {
        await db.update(tasks).set({ deadline: new Date(task.deadline.getTime() + task.repeat
* (24 * 3600 * 1000)) }).where(eq(tasks.id, task.id));
    } else {
        await db.delete(tasks).where(eq(tasks.id, task.id));
    }

    const taskList = await getTasks(user);
    if (typeof taskList === "number") {
        return error(taskList);
    }
    await generateSchedule(user, taskList);

    return new Response()
}
```

With this I changed the complete button to use the new API route instead of the DELETE method:

```ts
async function complete() {
if (current == undefined) return;
await fetch('/api/complete', { method: 'POST', body: JSON.stringify({ id: current.id }) });
  await sleep(500);
await invalidateAll();
}
```

I also added a sleep to the function, as I found the schedule wasn't being regenerated before it was being refetched so the change wasn't observed.

While a 500ms delay does make the refresh take longer it was a lot more reliable.

Start After

I finally had to bite the bullet and implement the start after feature. I needed a way of specifying that tasks couldn't be started until after a certain time, for example I wouldn't be able to do my maths lesson until it was time for it. To start I modified the `tasks` table again to add the start after field:

```ts
export const tasks = pgTable('tasks', {
    id: serial('id').primaryKey(),
    title: text('title').notNull(),
    duration: integer('duration').notNull(),
    scheduled: timestamp('scheduled'),
    repeat: integer('repeat'),
    startAfter: timestamp('start_after').notNull(), // New
    deadline: timestamp('deadline', { withTimezone: true }).notNull(),
    user_id: integer('user_id').references(() => users.id)
});
```

Like the repeat feature, I needed to update the task edit UI to add the new field:

```ts
<script lang="ts">
  let startDate: DateValue, startHours: number, startMinutes: number;

  ...
</script>

<div class="grid grid-cols-5 items-center gap-4">
<Label for="duration" class="text-right">Start After</Label>
<DatePicker bind:value={startDate} class="col-span-4" />
</div>
<div class="grid grid-cols-5 items-center gap-4">
<p />
<NumericInput bind:value={startHours} class="col-span-2" />
<NumericInput bind:value={startMinutes} class="col-span-2" />
</div>
```

I chose an identical design to the deadline input in order to be consistent with my timestamp inputs. After tweaking some parts to make sure it was being send to and recieved by the server, I was able to create tasks with an earliest start time, next I needed to update the schedule generation algorithm to take these times into account, for now I went with the simplest approach of just setting the current time to the start after time if the current time was too early:

```ts
export async function generateSchedule(user: User, tasks: Array<Task>) {
    const sorted: Array<Task> = tasks.sort((a, b) => a.deadline.getTime() - b.deadline.getTime());
    let start: Date = new Date(Date.now());
    sorted.forEach(task => {
        const next_day = nextDay(user.start, start);
        const day_end = end(user.end, start);
        if (day_end.getTime() < (start.getTime() + task.duration * 1000)) { // If the task would finish after the day's ended, move it to the beginning of the next day
            start = next_day;
        }

        // New
```

```
        if (start.getTime() < task.startAfter.getTime()) {
            start = task.startAfter;
        }

        task.scheduled = start; // Schedule in this task
        (async () => await db.update(schema.tasks).set({ scheduled:
start }).where(eq(schema.tasks.id, task.id)))();
        start = new Date(start.getTime() + task.duration * 1000); // Advance the start time by
the task's duration
    });
}
```

While this meant there were a lot more holes in the timetable, it did work correctly and allowed me to schedule tasks such as lessons which had to be in a fixed time. Finally I needed to update the complete API to update the start after time for repeated tasks, for instance if I have a maths lesson at 10am every week on Mondays, after its finished I want the start after time to become the start of next week's lesson:

```
  ...

  if (task.repeat) {
      await db.update(tasks).set({
          startAfter: new Date(task.startAfter.getTime() + task.repeat * (24 * 3600 * 1000)),
          deadline: new Date(task.deadline.getTime() + task.repeat * (24 * 3600 * 1000))
      }).where(eq(tasks.id, task.id));
  } else {
      await db.delete(tasks).where(eq(tasks.id, task.id));
  }


  ...
```

With this feature implement, not only was I able to schedule fixed tasks, I was finally able to implement the snooze feature!

Snoozing

I decided to implement snoozing by changing the current task's start after time to one hour after the current time, meaning it would be rescheduled for an hours time at the earliest. This meant snoozing was as simple as:

```
<script lang="ts">
  async function snooze() {
    if (!current) { return; }
    current.startAfter = new Date(current.startAfter.getTime() + (3600 * 1000));

    const body = {
      startAfter: current.startAfter.getTime(),
  deadline: current.deadline.getTime(),
  duration: current.duration,
  title: current.title,
  id: current.id,
  repeat: current.repeat
  };

    await fetch('/api/tasks', { method: 'PATCH', body: JSON.stringify(body) });
    await sleep(500);
  await invalidateAll();
  }
</script>

...
```

```
<Button.Root class="p-8 text-xl lg:p-16 lg:text-3xl" on:click={snooze}>Snooze</Button.Root>
```

...

With all of the buttons working in theory, I decided to do another full
test of the overview page:

No.	Description	Expected	Outcome
01	Go to page while there is a current task	Page shows current task, remaining time, and other information	Page shows information correctly
02	Go to page while there is not a current task	Page shows there is no task and shows the next upcoming task	Page shows information correctly
03	Complete button pressed while there is a current task	Completes the task, regenerates schedule and updates page	Task is deleted and schedule is updated
04	Complete button pressed while there is no current task	Errors saying there is no ongoing task	No effect
05	Snooze button pressed while there is a current task	Snoozes the task, regenerates the schedule and updates page	Snoozes the task and schedule is updates
06	Snooze button pressed while there is no current task	Errors saying there is no ongoing task	No effect
07	Break button pressed	Pop up prompting for break duration	Inputs work correctly
08	Duration entered into break popup	Clears all task for the entered duration, regenerates schedule and updates the page	Clears all tasks and updates schedule
09	Task list button pressed	Goes to task list page	N/A
010	Schedule list button pressed	Goes to schedule view page	N/A
011	Overview button pressed	Nothing happens	N/A

All features were working except for error messages, the app was nearly
finished! To do a proper test and get some feedback I deployed the app and
asked my stakeholders to sign up for an account and test it out. I left
them for a few hours to wait for feedback and these were the results:

Feedback	Decisions
Difficult to navigate	Create navigation bar with buttons that go to all important pages

Signup/Login pages don't go anywhere if successful	Create redirects on success that sends the user to the overview page
Have to go to tasks page to add task	Introduce add task button onto new navbar

Navigation

The majority of the feedback I recieved was that navigating around the pages was too difficult. To fix this I've decided to add a navigation bar at the top of all pages with buttons to all of the major pages on the site:

```
<div class="flex justify-between border-b p-4">
<div class="flex items--start gap-4">
<Button href="/">Overview</Button>
<Button href="/tasks">Tasks</Button>
</div>
{#if authed}
<Button on:click={logout}>Log Out</Button>
{:else}
<div class="flex items-end gap-4">
<Button href="/login">Log In</Button>
<Button href="/signup">Sign Up</Button>
</div>
{/if}
</div>
```

TODO: Screenshot

I decided to give the navbar two model, authenticated and unauthenticated, which will change the right hand buttons from Logout to Login+Signup respectively. The navbar's variant will be set on each page, the pages that require the user to be authenticated will show the navbar in the authenticated variant.

As well as the navbar I decided to implement different automatic redirects to try and guide the user to the correct page:

If logged in:

From	To
Signup	Overview
Login	Overview

If logged out:

From	To
Overview	Login
Tasks	Login

The login and signup forms also redirect to the overview page if they were successful.

This meant it was much harder to get lost on a page because the user wasn't authenticated. Finally I also added a new API route, /api/logout that allowed the user to log out:

```
export async function GET({ cookies }) {
    cookies.delete("token", { path: "/" });
```

```
    return new Response();
}
```

All the route does is delete the user's `token` cookie, meaning the client will look like it hasn't logged in, meaning it will be redirected to the login page.

With those changes made I updated the deployment and asked the stakeholders for feedback, they said the website was much easier to use than before. 2 of the 3 pieces of feedback I was given had been addressed, so I moved onto adding the new add task button onto the navbar.

Extra Add Task Button

Reminder Notifications

PWAs

Testing Phase