4.2 Development of the Front-end

After the foundations for the back-end had been developed, work on the front-end could begin.

4.2.1 Making Calls to the Back-end

In order to reduce code duplication and speed up development, the decision was made to create a single exported JavaScript module called 'API.js'. The benefit of this is that it allows for the implementation of the methods used to make the API calls before their full implementation is complete. All non-authentication API calls will use Axios, a promise-based HTTP library to make requests to the back-end and will use the following structure:

```
const createTask = async (payload) => {
    const auth = getAuth(firebaseApp);
    const token = await getIdToken(auth.currentUser);
    const url = host + '/tasks/createtask/';

    try {
        await axios.post(url, payload, {headers:
                { authorization: `Bearer ${token}` }}).then(async (r) => {
            await responseHandler(r, "Task Created")
        })
    }
    catch (e) {
        await errorHandler(e)
    }
}
```

Figure 37: Front-end Create Task Method

Another benefit to using a single API module, is that it allows for the creation of a response handler, which can be used to humanise the HTTP responses for the user in the form of error messages and status messages:

```
// Response handler used to display toast message
// if api call is successful / if there is an error
const responseHandler = async (r, okay) => {
    if (r.status === 200) {
        const toast = await toastController
            .create({
                message: okay,
                duration: 2000,
                color: 'dark',
                position: 'top'
            })
        return toast.present();
    } else {
        // If response has no status or the
        // response handler encounters an error
        const toast = await toastController
            .create({
                message: "Oops an error has occurred!",
                duration: 2000,
                color: 'dark',
                position: 'top'
            })
        return toast.present();
    }
}
```

Figure 38: Front-end HTTP Response Handler

4.2.2 User Authentication

Login Page

Authenticating users on the front-end requires more work than the back-end, as not only is a method of preventing unauthorised access to the methods required, but also a system to prevent users from accessing the main application without being signed in or registered. In order to keep the code clean and structured, a single API module was created which would contain the separate API calls the front-end will make to the back-end API. This also allows for reuse of any calls throughout the application if required, without code duplication.

The first step was to begin work on the user interface, making use of and customising Ionic's inbuilt components such as the input fields and buttons in order to create a login screen:

```
<template>
  <ion-page>
    <ion-content fullscreen class="ion-padding" scroll-x="false" scroll-y="false">
      <div class="container" style="top: 20%;">
        <img src="../../assets/Logo.png" class="logo">
      </div>
      <div class="container ion-padding-horizontal" style="top: 50%;">
        <ion-item fill="outline" mode="md" style="min-width: 80%; padding-bottom:
20px;">
          <ion-input label="email" label-placement="floating" placeholder="email" v-
model="payload.email"></ion-input>
        </ion-item>
        <ion-item fill="outline" mode="md" style="min-width: 80%; padding-bottom:
20px;">
          <ion-input label="password"
                     type="password"
                     label-placement="floating"
                     placeholder="password"
                     v-model="payload.password">
          </ion-input>
        </ion-item>
      </div>
    </ion-content>

    <ion-footer class="ion-no-border ion-padding-bottom" translucent="true">
      <div>
        <ion-button expand="block"
                    class="ion-padding-horizontal"
                    size="large"
                    style="color: white; --border-radius: 20px; font-size: 25px;"
                    @click="doLogin">login
        </ion-button>
        <br>
        <ion-button expand="block"
                    class="ion-padding-horizontal"
                    size="large"
                    style="color: white; --border-radius: 20px; font-size: 25px; --
background: #02C39A;"
                    @click="signUp()">create account
        </ion-button>
      </div>
    </ion-footer>
  </ion-page>
</template>
```

Figure 39: Snippet of the Front-end Login Page

This produces the following user interface:

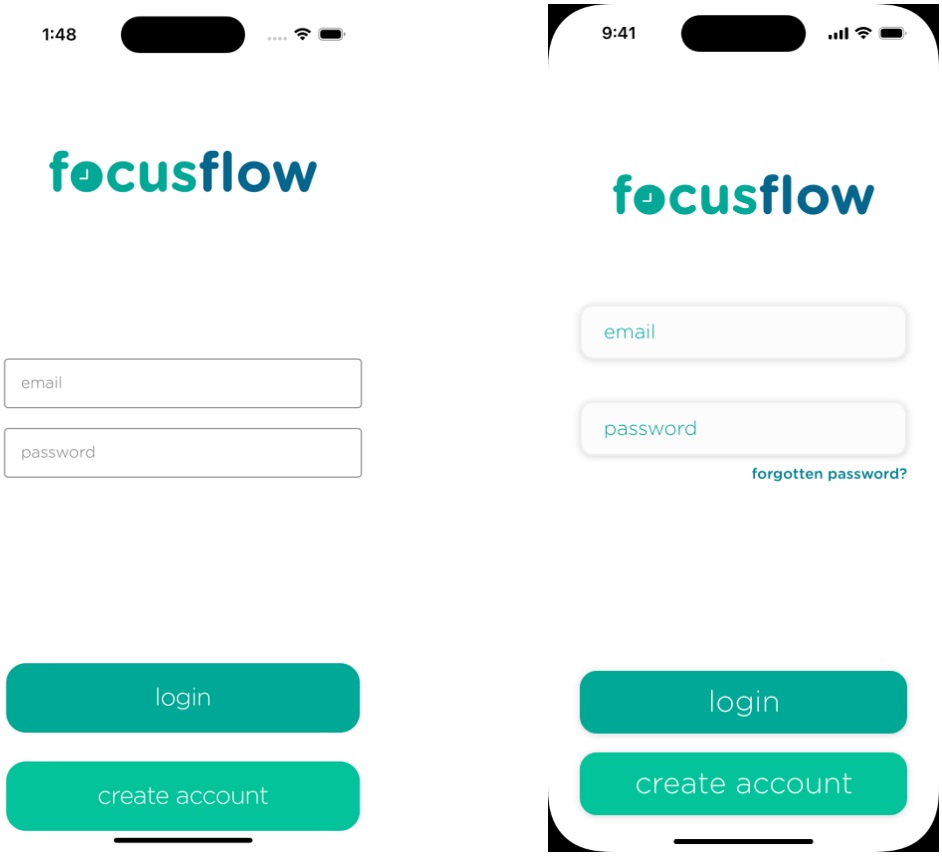

Figure 40: Login page prototype (left) compared with the mockup (right)

Next, the login method can be created allowing the users to login. The login method was created using built in Firebase authentication methods. As previously mentioned, all back-end API calls and authentication methods will be contained within a single exported module called 'API.js'. The reason for this is to improve maintainability and reduce code duplication, as there may be a need for a single method to be used by multiple components.

```
const login = async (payload) => {
   const auth = getAuth(firebaseApp);
    try {
        await signInWithEmailAndPassword(auth,
            payload.email,
            payload.password).then(async r => {
                r["status"] = 200
                await responseHandler(r,"Logged in successfully")
        })
    }
    catch (e) {
        e["status"] = 200
        await responseHandler(e, e.message)
    }
}
```

Figure 41: Front-end Login Method

Navigation Guard

In order to proceed, the application requires a Vue.js router. A router allows the application to be made up of multiple pages by mapping a URL endpoint to specific Vue.js components [22]. This allows for navigation through the app by changing the URL either manually or programmatically. The ability to change the URL programmatically allows for the use of a navigation guard. Navigation guards are used to guard navigations either by redirecting it or cancelling it [22]. This is required to prevent unauthorised users from accessing pages other than the login or registration page.

```
import { createRouter, createWebHistory } from '@ionic/vue-router';
import { RouteRecordRaw } from 'vue-router';
import api from "@/api/api";

const { getCurrentUser } = api();

const routes: Array<RouteRecordRaw> = [
  {
    path: '/',
    redirect: '/login'
  },
  {
    path: '/login',
    name: 'login',
    component: () => import('@/views/auth/LoginView.vue'),
  },
  {
    path: '/tasks',
    name: 'tasks',
    component: () => import('@/views/HomeView.vue'),
    meta: {
      requiresAuth: true
    }
  }
]

const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
  routes
})

// Before each route transition:
router.beforeEach(async (to) => {
  // Check if the target route requires authentication.
  const requiresAuth = to.matched.some((record) => record.meta.requiresAuth)

  // If it requires authentication and no user is currently logged in,
redirect to '/login'.
  if (requiresAuth && !(await getCurrentUser())) {
    return '/login'
  }
  // Otherwise, proceed as normal.
})

export default router
```

Figure 42: Front-end Vue.js router

4.2.3 Navigation

Currently after logging in, users directed to '/tasks' will be met with a blank screen due to HomeScreen.vue – the main component for the tasks URL endpoint containing no elements. Following the results from the design survey, the decision was made to attempt to combine swipe-based page navigation with a navigation bar. This allows for increased flexibility as not only will it serve as a standard navigation bar which can be clicked for navigation, users will be able to swipe between pages.

In order to implement this feature, a library called Swiper.js has been utilised. Swiper.js is a JavaScript framework used for creating a touch-based slides system [23]. The following placeholder components were also created: Progress tab, Planner tab, Tasks tab, Timer tab and Settings tab. This allows a Swiper.js 'slide' component to be created for each placeholder:

```
<template>
  <ion-page>
    <ion-content scroll-y="false">
      <swiper :modules="modules"
              @swiper="setSwiperInstance"
              :slides-per-view="1"
              :space-between="10"
              class="swiper"
              @slideChange="onSlideChange"
              @slideChangeTransitionStart="destroyPanes"
              @slideChangeTransitionEnd="notDestroyPanes"
              @afterInit="notDestroyPanes">
        <swiper-slide>
          <progress-tab></progress-tab>
        </swiper-slide>
        <swiper-slide>
          <planner-tab @lock-slide="lockSlide"
                       @unlock-slide="unlockSlide"
                       :destroy="destroy"
                       v-if="activeIndex === 1"
                       :tab="activeIndex === 1">
          </planner-tab>
        </swiper-slide>
        <swiper-slide>
          <tasks-tab @view-timer-parent="viewTimer"
                     v-if="activeIndex === 2"
                     :tab="activeIndex === 2">
          </tasks-tab>
        </swiper-slide>
        <swiper-slide>
          <timer-tab :task="data.task"></timer-tab>
        </swiper-slide>
        <swiper-slide>
          <settings-tab @lock-slide="lockSlide"
                        @unlock-slide="unlockSlide">
          </settings-tab>
        </swiper-slide>
      </swiper>
    </ion-content>
  </ion-page>
</template>
```

Figure 43: Front-end Navigation

Implementing the navigation bar proved much more challenging. In order to achieve this, tags were used in the header with custom CSS styling and classes to allow the name of the current page to be displayed at the top of the application. This CSS styling allowed for the implementation of the opacity and scrolling effect on the different page titles from the mock-ups. Finally, the @click attribute allows the user to click on each page title allowing for navigation bar style navigation:

```
<ion-header>
  <ion-toolbar style="min-height: 70px; --background: #ffffff; --border-
color: #ffffff;">
    <div class="swiper-pagination" slot="end">
      <span
          v-for="(slide, index) in tabs"
          :key="index"
          :class="[
          'custom-bullet',
          { 'swiper-pagination-bullet-active': index === activeIndex }
        ]"
          @click="handlePaginationClick(index)"
          :style="{
          transform: `translateX(${
            (index - activeIndex) * (
              index === activeIndex - 1 || index === activeIndex + 1
                ? 180
                : 180
            )
          }%)`,
          fontSize: index === activeIndex ? '2rem' : '1.2rem',
          opacity: index === activeIndex ? 1 : 0.5
        }"
      >
        {{ slide }}
      </span>
    </div>
  </ion-toolbar>
</ion-header>
```

Figure 44: Front-end Navigation Bar

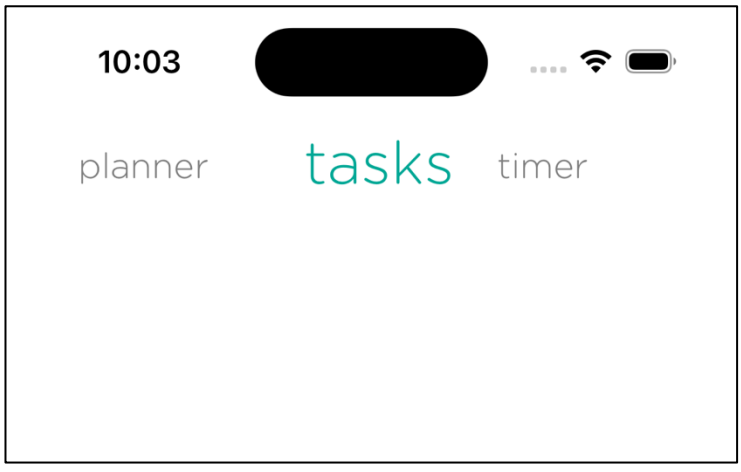

Figure 45: Navigation Bar Prototype

4.2.4 Tasks Page

Viewing Tasks

With navigation complete, development on implementing the user interface for viewing, managing and creating tasks could begin. In order to read the Firestore database, a library called Vuefire was used. Vuefire provides bindings for Vue.js allowing for real-time syncing between the front-end and the database [24]. This reduces the need to send GET requests to the back-end, reducing server load as it is not required to serve data. One benefit of using Vue.js is that it allows for components to be nested, also known as parent components and child components. This can reduce code duplication whilst also resulting in much more readable, tidy code. This was utilised for the tasks page to create a component called "TaskListItem". The task list item component is the template for an individual task as displayed on the tasks page. As the parent tasks page iterates through the tasks, each task will be displayed in accordance with the TaskListItem template:

```
<template>
  <ion-card>
    <ion-card-content>
      <ion-grid>
        <ion-row>
          <ion-col size="auto">
            <ion-avatar :class="priorityColour"></ion-avatar>
          </ion-col>
          <ion-col>
            <h1>{{ task.title }}</h1>
          </ion-col>
          <ion-col size="auto" class="right-align">
            <ion-card-title color="primary"
                            style="font-size: 25px">{{ duration }}</ion-card-title>
          </ion-col>
        </ion-row>
      </ion-grid>
    </ion-card-content>
  </ion-card>
</template>
```

Figure 46: Front-end TaskListItem

The TaskListItem template was constructed using an Ionic card component – a basic container element that can be customised using a variety of properties and CSS, along with an avatar component, typically used to display circular profile pictures. This avatar component worked perfectly for displaying the task priority colour as it can be easily changed by altering the background colour, resulting in a coloured circle:

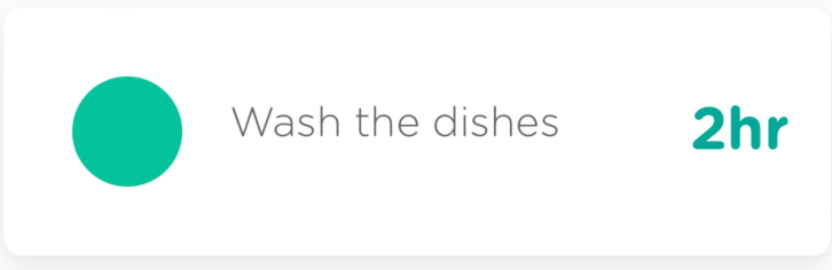

Figure 47: Front-end Individual Task

Then, to use TaskListItem on the tasks page, the tasks need to be iterated through with v-for, passing each task to the TaskListItem component:

```
<template>
  <ion-page>
    <ion-content :scroll-y="draggable">
      <div v-if="placeholder">
        <task-list-item v-for="task in tasks" :task="task"></task-list-item>
      </div>
      <ion-card v-else>
        <ion-card-content style="font-size: 1.4rem; padding-bottom: 30px">
          swipe up to create a task!
        </ion-card-content>
      </ion-card>
    </ion-content>
  </ion-page>
</template>
```

Figure 48: Front-end Tasks Page Template

In Figure 48, a basic placeholder card component was also created. This placeholder is displayed when a user has no tasks in order to avoid a blank screen which may be confusing to them:

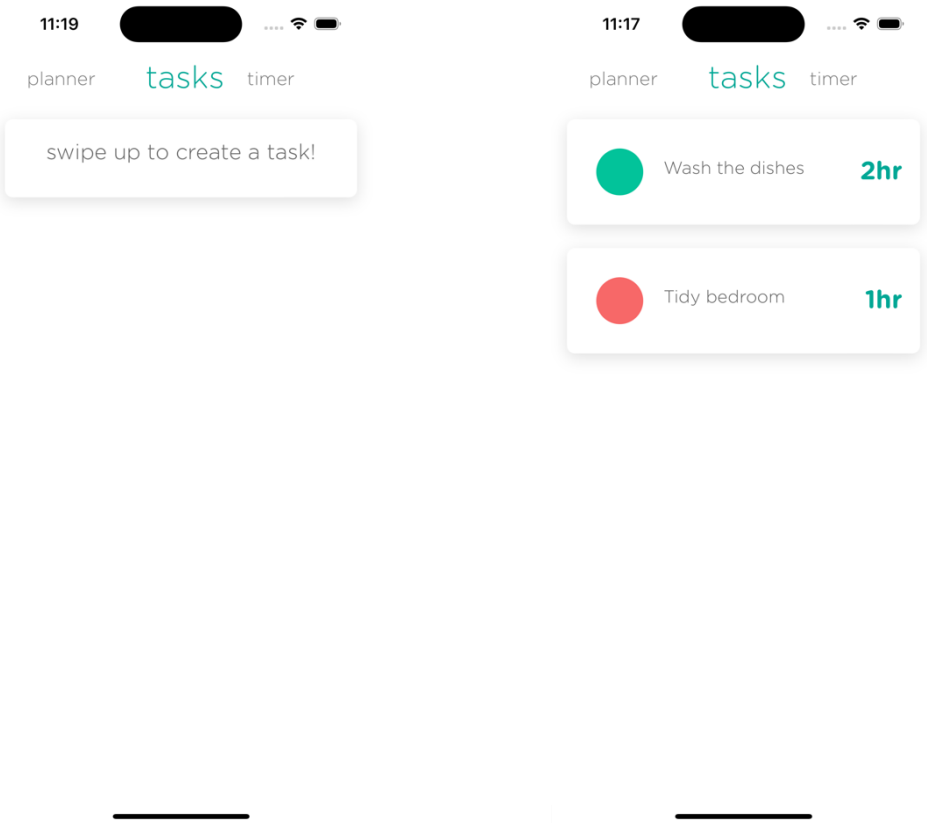

Figure 49: Front-end Tasks Page

Creating Tasks

In the mock-up designs in section 3.8, the tasks page utilises a draggable component which users can drag up from the bottom of the screen to reveal an input form to create a task. In order to implement this feature, a library called v-cupertino was used. V-cupertino provides a draggable pane component [25] that can be customised to the requirements of this project. The create task component also uses the input, button and picker Ionic components in order to create the input form:

```
<template>
  <v-cupertino :drawerOptions="options" ref="bottomSheet">
    <div>
      <h1 style="padding-bottom: 40px"> create a task </h1>
    </div>
    <div>
      <ion-list>
        <ion-item style="min-width: 80%; padding-bottom: 20px;">
          <ion-input label="title" v-model="payload.title"></ion-input>
        </ion-item>

        <ion-item style="min-width: 80%; padding-bottom: 20px;">
          <ion-input label="priority" v-model="payload.priority"></ion-input>
          <ion-picker
              :isOpen="pPicker"
              :columns="priorityOptions"
              :buttons="priorityButtons"
              @did-dismiss="pPicker = false"
          ></ion-picker>
        </ion-item>

        <ion-item style="min-width: 80%; padding-bottom: 20px;">
          <ion-input label="duration" v-model="displayedDuration"></ion-input>
          <ion-picker
              :isOpen="dPicker"
              :columns="durationOptions"
              :buttons="durationButtons"
              @did-dismiss="dPicker = false"
          ></ion-picker>
        </ion-item>

        <ion-button id="createtask" :disabled="valid" style="min-width: 80%;"
@click="callCreateTask">Submit</ion-button>
        <ion-loading :is-open="loading" trigger="createtask" message="Creating
task..."> </ion-loading>
      </ion-list>
    </div>
  </v-cupertino>
</template>
```

Figure 50: Front-end Create Task Component Snippet

When the Submit button is clicked, the method referenced in Figure 37 from API.js to create a task is called, submitting the payload from the form to the back-end where it is handled. The form also has validation methods implemented, to prevent invalid tasks from being submitted. The current validation implementation only checks for blank fields and a task duration of 0, but can be further expanded in the future if required. When added to the tasks page, the create task component can be seen in Figure 51.

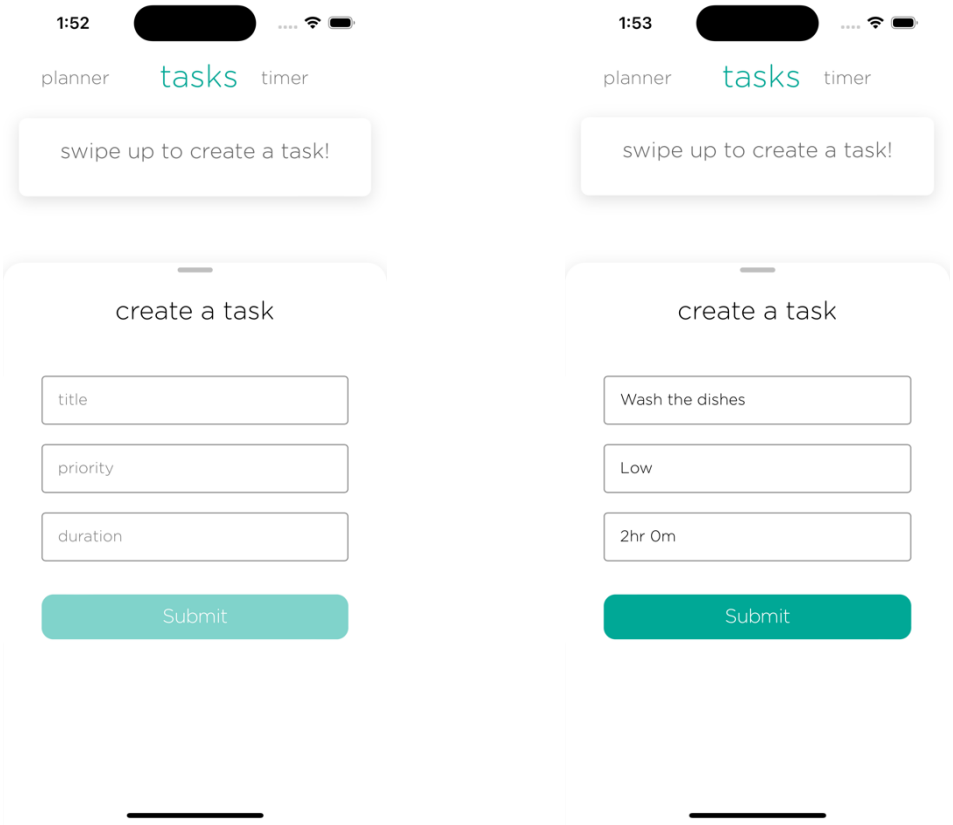

Figure 51: Front-end Create Task Dialogue

Deleting Tasks

In order to implement the functionality to delete a task, a UI component suitable for this needed to be used. This was done with an Ionic component known as a popover, which can be used to reveal a context menu to the user when tapping on a task. This menu can be used to include access to other functions or features should it become necessary. The code below for the popover component was added to the code for TaskListItem (Figure 46):

```
<ion-popover :trigger="task.id" trigger-action="click" :dismiss-on-select="true">
  <ion-list>
    <ion-item :button="true" @click="viewTimer(task)">Start Timer</ion-item>
    <ion-item :button="true" @click="callDeleteTask(task)">Delete Task</ion-item>
  </ion-list>
</ion-popover>
```

Figure 52: Front-end Context Menu Snippet

Upon implementation, the decision was made to also add the option to view the associated timer for that task. The method 'callDeleteTask()' is also contained within the API.js module and is identical in structure to the method used (Figure 37) to call the create task function on the back-end.

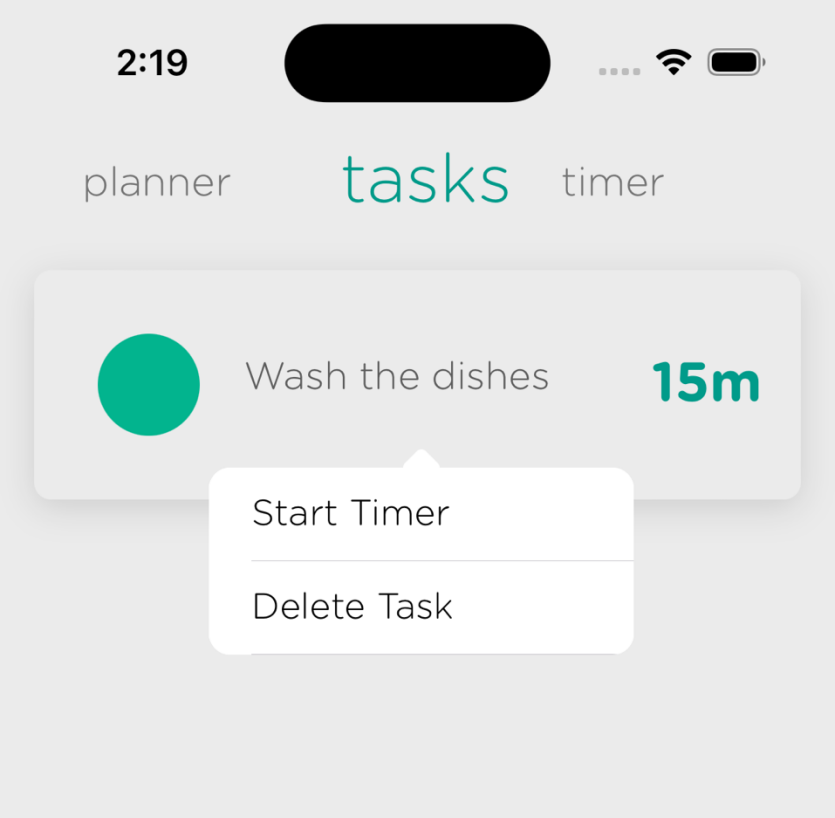

Figure 53: Front-end Context Menu

Completing a Task

The intention is for the user to be able to mark a task as completed quickly and easily, without having to access any menu. This was achieved by adding click functionality to the avatar circle, which then calls the API method from API.js, in order for the back-end mark a task as completed. Functionality was also added to include an animation with CSS when a task is marked as complete for improved user feedback and experience:

```
<ion-avatar :class="[priorityColour, { 'spin-once': spin }]"
            @click="markTaskCompleted(task)"
            ref="completedCircleRef"
            @mousedown="press"
            @mouseup="release"
            @onmouseleave="release">
  <ion-icon v-if="spin" style="color: #ffffff; font-size: 1.6rem"
:icon="checkmark"></ion-icon>
</ion-avatar>
```

Figure 54: Complete Task HTML Snippet

```
@keyframes spin {
  0% {
    transform: rotate(0deg);
  }
  100% {
    transform: rotate(360deg);
  }
}

.spin-once {
  animation: spin 0.5s ease-in-out;
}

@keyframes slide-off {
  0% {
    transform: translateY(0);
    opacity: 1;
  }
  100% {
    transform: translateY(-100%);
    opacity: 0;
  }
}

.slide-off {
  animation: slide-off 0.5s ease-in-out forwards;
}
```

Figure 55: Complete Task CSS Animation Snippet

The animation causes a white checkmark to appear within the priority circle of a task, followed by a spin animation after which the task slides upwards and out of view.

4.2.5 Timer Page

The timer page is designed to act as a simple focus timer, which can be expanded and worked upon in the future. Based on the design from Section 3.8, the timer component would be made up of a circular progress bar, text to display the remaining duration and buttons to start and pause the timer as well as add an extra five minutes if required. To create the circular progress bar, a Vue.js compatible library called vue-ellipse-progress [26], which contains customisable, animated circle progress bars was used.

```
<template>
  <div>
    <ve-progress v-if="mounted"
                 thickness="10%"
                 color="#00A896"
                 :progress="progress"
                 :size="300"
                 animation="loop"
                 font-size="2rem">
      <span slot="legend-value">{{ remainingDuration }}</span>
    </ve-progress>

    <ion-alert
        :is-open="alert"
        header="Alert"
        message="Timer Complete!"
        :buttons="alertButtons"
    ></ion-alert>
  </div>

  <div class="button">
    <ion-button size="large" @click="addFive">add 5m</ion-button>
  </div>

  <div class="floatingbutton">
    <ion-fab>
      <ion-fab-button>
        <ion-icon :src="playPause" @click="toggleTimer"></ion-icon>
      </ion-fab-button>
    </ion-fab>
  </div>
</template>
```

Figure 56: Front-end Timer Page Snippet

During development, various techniques were experimented with to manage the timer's state and update the UI accordingly. Initially, an attempt was made to use the built-in JavaScript setInterval and clearInterval functions, but it was discovered that the "vue-timer-hook" library provided a more efficient and simple approach to handling timers in Vue.js.

Also included is the ability for an alert to appear when the timer is complete:

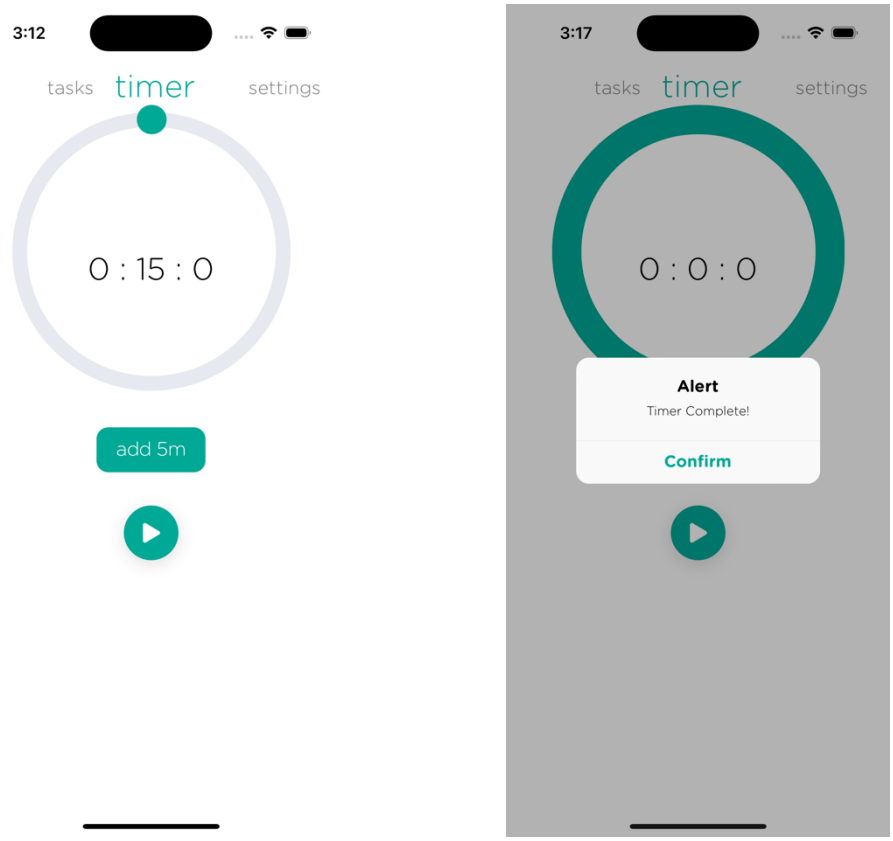

Figure 57: Front-end Timer Page

4.2.6 Planner Page

The planner page is intended to be used as a schedule / timetable to allow students with ADHD to plan their day. Based on the designs from Section 3.8, the planner page will utilise a draggable pane component – similar to the one used to create a task, from which users can drag any tasks that they have created onto the timetable to schedule them. Development first started with creating the draggable pane component using v-cupertino. In order to add the drag and drop functionality, a library called vue.draggable [27] was used. This adds a draggable container element which allows components within to be made draggable. The reason this library was chosen is because it is well supported due to being based upon Sortable.js and compatible with touch screens.

Creating the planner page proved more challenging than expected and a number of issues were encountered. The first limitation is that due to the way in which the timeline is calculated when a task is added, tasks can only be created in increments of 15 minutes. It is possible that this can be resolved, and more flexibility added in the future, but this will be discussed later in the evaluation. The draggable tasks feature was another area that required several iterations. Initially, the drag and drop feature experienced issues with tasks getting stuck or overlapping. To resolve this, a function called checkFull was implemented to check if the position that a task is being dropped on to already has a scheduled task or not. This function will return either true or false depending on this condition and will either allow or reject the drag operation.

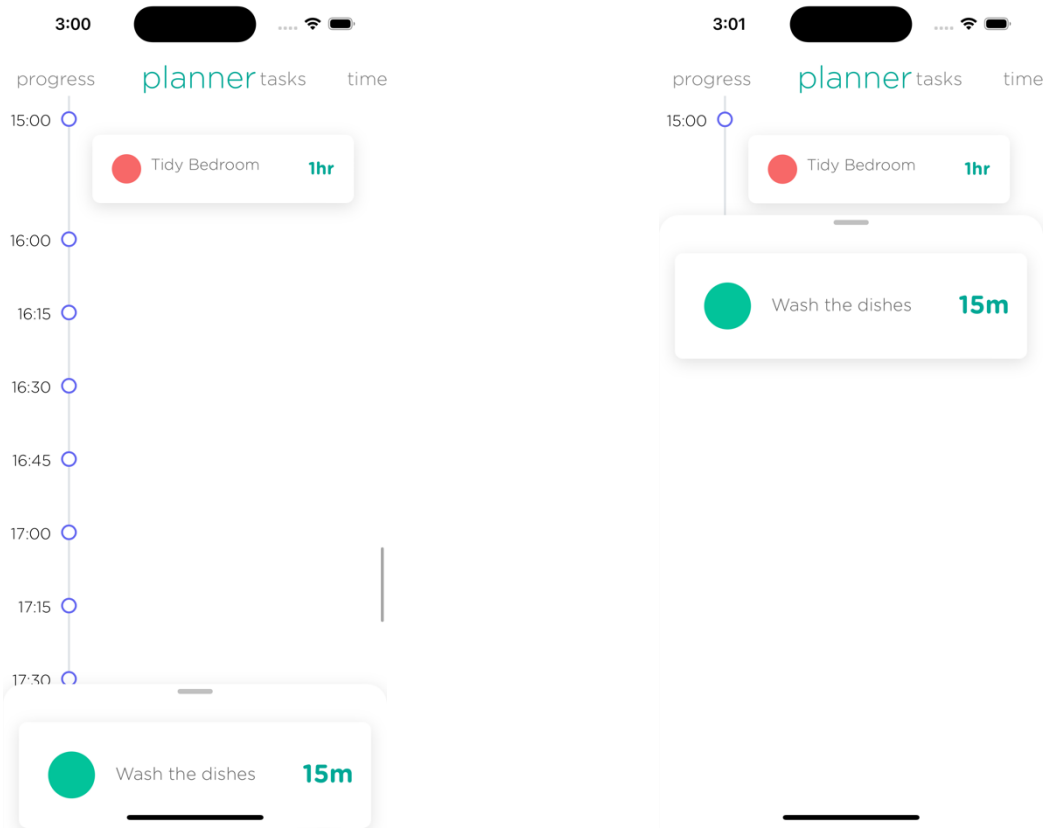

Figure 58: Front-end Planner Page

4.2.7 Progress Page

In order to implement FR06 and using the mock-ups from Section 3.8, the progress page would be made up of two main components: a chart displaying the daily number of tasks completed in the current week, and a circular progress bar to display the number of hours spent completing tasks against a target set by the user. This feature is intended to provide the user with a breakdown of their progress and has room for expansion in the future.

The tasks chart proves an intuitive and visual representation of the user's number of tasks completed over a week. The component was developed using vue-chartjs, a library based on Chart.js, which provide versatile capabilities for creating attractive and responsive charts [28]. Each column represents a day of the week, and its height corresponds to the number of tasks completed on that day. This allows users to instantly assess their productivity levels and patterns throughout the week.

In addition to the tasks chart, a circular progress bar, commonly known as a radial progress indicator is used to display the number of hours worked along with a goal that can be set by the user. This indicator provides a user-friendly and visually compelling way of displaying progress towards a goal. The circular design is particularly effective for representing a ratio or percentage, as it intuitively implies a 'whole' or 'complete' state when the bar is fully filled.

Figure 59: Front-end Progress Page