COMP 1800 – Term Project

Reminder Application Documentation

Simon Newell

Trevor Hobenshield

Eric Dong

Ted Yuan

Contents

[1. Landing Page and Application Design 2](#_Toc37113817)

[1.1 Prototyping and Design 2](#_Toc37113818)

[1.2 Current Progress 5](#_Toc37113819)

[1.3 Work to be done/In-Progress 6](#_Toc37113820)

[1.3.1 Mocking Server 6](#_Toc37113821)

[1.3.2 Merging HTML 6](#_Toc37113822)

[2. Creating/Editing Reminders 7](#_Toc37113823)

[2.1 Current Progress 7](#_Toc37113824)

[2.2 Work to be done/In-Progress 7](#_Toc37113825)

[2.2.1 Description, Tags, and Date Information 7](#_Toc37113826)

[2.2.2 Reformat Reminders to Objects and Database Integration 8](#_Toc37113827)

[2.2.3 Editing Reminders 8](#_Toc37113828)

[3. Reminder Persistence 9](#_Toc37113829)

[3.1 Current Progress 9](#_Toc37113830)

[3.2 Work to be done/In-Progress 11](#_Toc37113831)

[4. Authentication 11](#_Toc37113832)

[5. Weather Forecasting (Dark Sky API) 11](#_Toc37113833)

[5.1 Current Progress 11](#_Toc37113834)

[5.2 Work to be Done/In-Progress 12](#_Toc37113835)

[5.2.1 Weekly Forecast 12](#_Toc37113836)

[5.2.2 Reminder Forecast 12](#_Toc37113837)

# Landing Page and Application Design

## 1.1 Prototyping and Design

The original wireframes for the application were made using figma by Trevor Hobenshield. They provided an initial layout of features based on the initial examples from the original Notion Design Document.

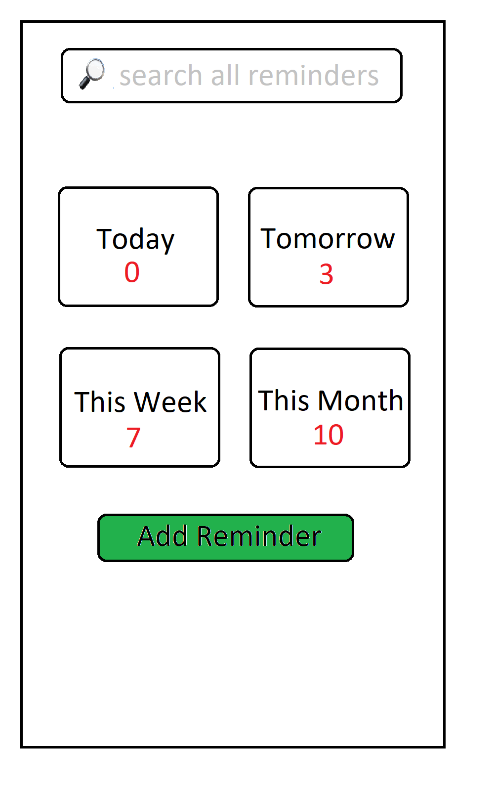
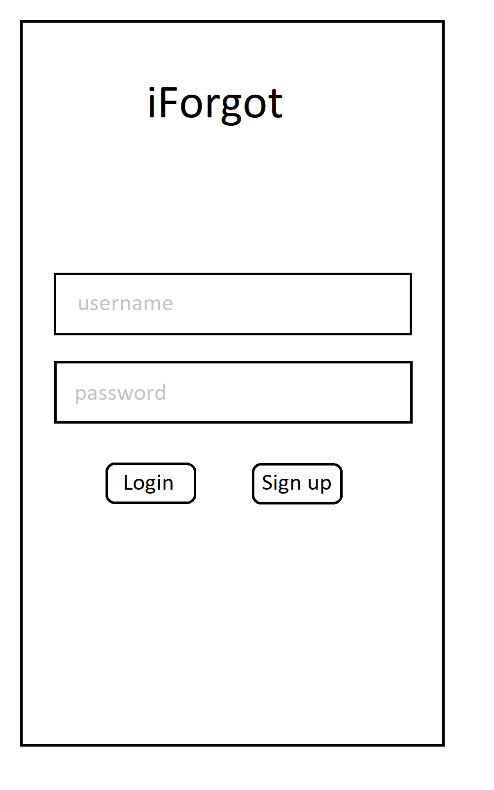


Figure 1: Original Basic Layout detailing key information

These wireframes were made as a guide line to provide context for page design and functionality while coding in javascript and ejs. These wireframes have now been replaced with the figma prototype pages.

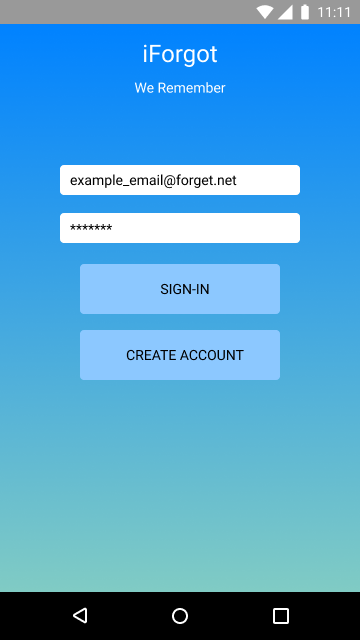
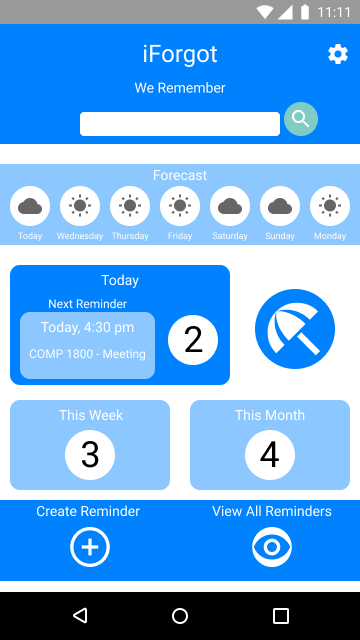
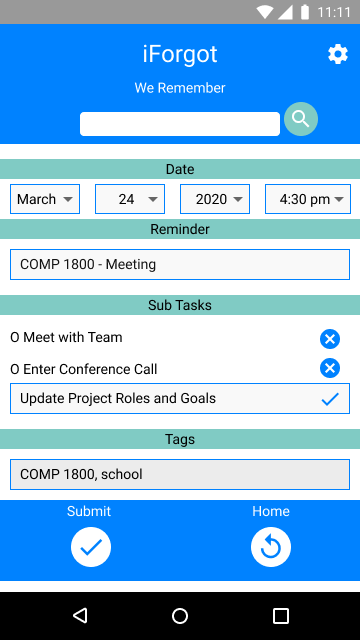
  

Figure 2: Prototypes for the sign-in, main application, and reminder creation pages.

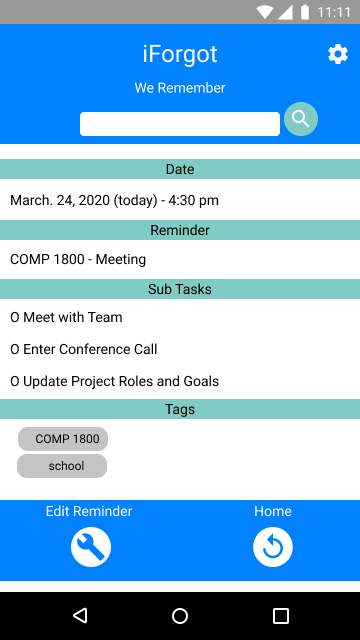
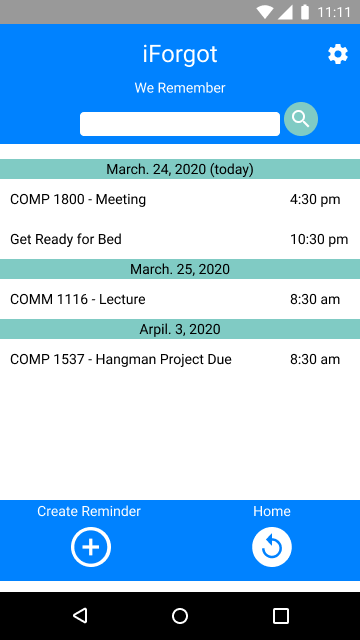
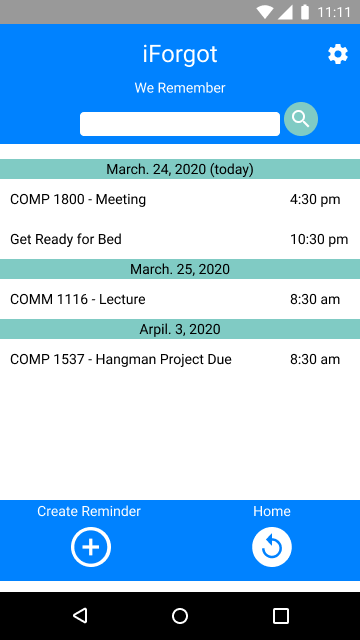
  

Figure 3: Prototypes of the individual, total, and monthly reminder pages

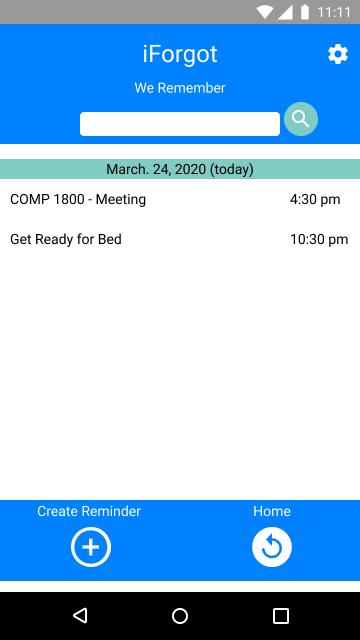
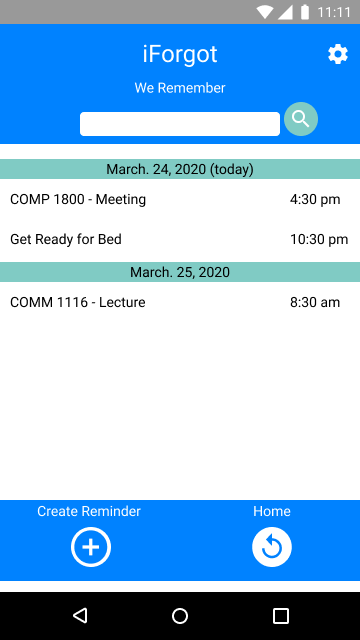
 

Figure 4: Prototypes for the daily and weekly reminders pages.

These prototypes were also designed in figma and contained interactive links to demonstrate the movement between pages in addition to their overall layout and design. These prototypes were created by Simon Newell.

The next prototype designed was the landing page, which was designed both for desktops and phone screen sizes, unlike the reminders which were based on the phone screen for readability and use.

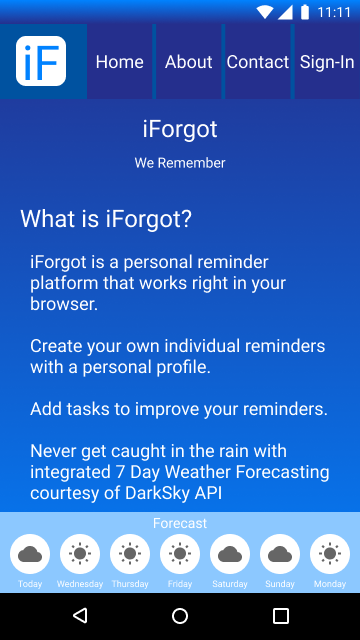


Figure 5: Prototype of the collapsed view of the landing page for phones and small screens.

The prototype landing page, as seen above, also featured use of the forecasting function of the main application, using the current geographical location of the user on page load. Due to difficulty mocking the server and subsequently fetching weather data this feature was not implemented in the current iteration of the landing page.

Other differences in the design were the removal of the gradient background in favor of single block colours for the colour scheme. This was partly to assist in readability and partly to reduce difficulty in implementation of style using Bootstrap.

The navbar would also be made into a collapsible ‘burger’ button for small screens, rather than reducing the size of individual buttons. Once again this was primarily a consideration of ease of use as well as readability, as shrinking buttons would make them difficult to read.

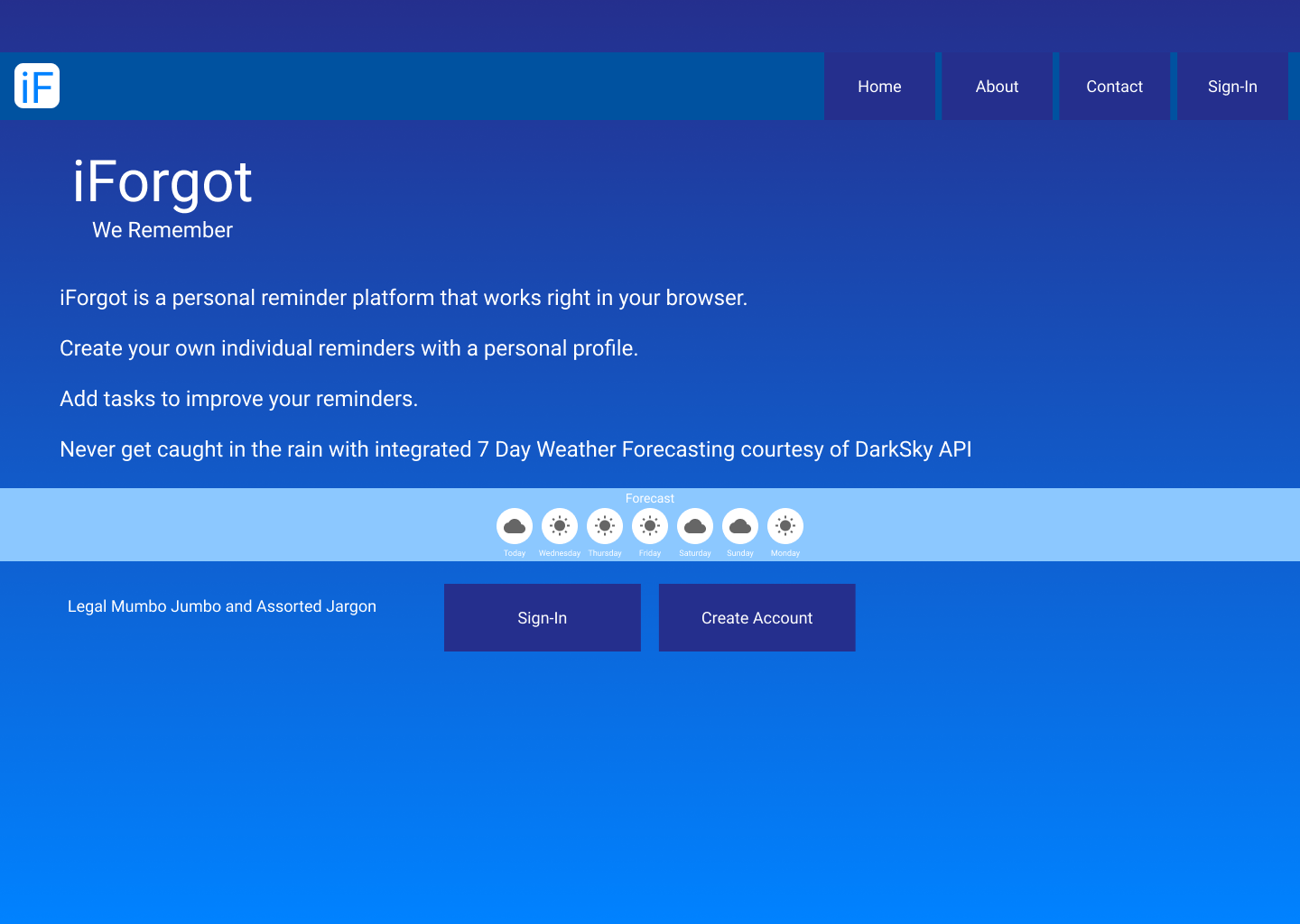


Figure 6: Prototype of the expanded landing page for desktop and larger screens.

The larger landing page showcases the lower buttons. Currently application functionality has account creation and account sign-in both using the common sign in screen. This makes the second ‘Create Account’ button somewhat erroneous, and it may be removed in future iterations.

No pages have been designed for about or contact pages. These would normally link to appropriate (essentially static) pages.

## 1.2 Current Progress

Currently the landing page, main reminder app page, reminder creation, editing, daily reminders, and single reminder view pages have basic style and layout completed and exist as individual html and ejs, pages and are currently found in the reminder app directory. All these pages make use of Bootstrap for stylistic elements and layout format and attempt to adhere to the main design from the figma prototypes.

Our server is currently mocked using node js. As the index exists in the reminder app directory, it must be accessed first in the terminal in order to be initialized. This process is detailed in the README file in the main repository.

Using ejs layouts and a control script, pages are linked through a series of functions which render the appropriate ejs pages.

## 1.3 Work to be done/In-Progress

The primary issues surrounding the completion of the app and its functional logic are in two distinct areas. Firstly, linking the landing page properly to the existing ejs pages, as this is the only remaining html static page. Secondly, full integrating the remaining ejs pages, such as the sign in/authentication page and the main reminder app.

The current application has ejs pages for the main application page, the creation and editing of reminders, viewing all reminders, viewing a single reminder, and a login page. Currently however, the viewing of weekly and monthly reminders is not available, as was initially planned in the design. These features are tied into the search functionality and the sorting of the reminders. This is detailed in section 6.

The monthly and weekly pages will need to be added and will stylistically match the regular total reminder view in form. However, they will need to display each reminder in order of date, for the current week or month.

Additionally, the current static landing page will need to be revised and added as an ejs, with appropriate links to the login page. While the landing page itself contains static content, adding it to the ejs files and networking it in the controller will prevent confusion and easily allow for editing or the application as a whole. To achieve this not only does the ejs itself need to be made, but additional script added to the index.js and reminder\_controller.js to render the page.

# Creating/Editing Reminders

## 2.1 Current Progress

Current progress on creating and editing reminders is near complete. Multiple reminders can be made and stored. Reminders have individual names, and stored dates, easily generated using an integrated calendar widget. Subtasks can be individually added to each reminder with unique text. Reminders can be edited, with current information persisting on the edit page.

Reminder name and substasks are entered directly by the user as typed strings. The sub tasks are added one by one. The date for the reminder is given by the output from the calendar widget. Upon submission using the save button, the new reminder is stored as an object in the database, and is given it’s own unique id number for reference in the url.

A button is attached to single reminder view. When pushed, this redirects to a variation on the reminder creation screen but with information filled with the appropriate reminder’s data. This prepopulated information can now be fully edited and saved, with the changes persisting afterword.

## 2.2 Work to be done/In-Progress

As the creation of the reminders is the principle task of the application, this feature is a high priority. The main areas that must be added or fixed are as follows:

* Addition of Reminder Tags
* Organization of Reminders by Date
* Check Box/Check Mark for complete sub-tasks (Optional/Low Priority)
* Full Integration of Database (see section 3. Reminder Persistence)

### 2.2.1 Tags

The tags may be handled in one of two ways. First, tags may be submitted one at a time with their own dedicated submission button to add them to the reminder object. This is considered non ideal, partly because it requires the separation of the submissions for the tags and the reminder as a whole. More importantly, it will severely impact the methodology by which a reminder is stored and edited. To deal with this a second method is proposed.

Have tags have a dedicated text box. All tags are to be comma separated, then stripped of leading and trailing whitespace. Each tag will be added in succession to an array/list which will be the value of the tags key of the reminder object.

### 2.2.2 Database Integration

This particular part of our implementation should be carried out in tandem with section 3. Reminder Persistence. Each reminder may be stored as a dictionary of which itself is stored as an item in a list. This list of reminders can then be held in the individual dictionary belonging to a single user. Please see section 3 for an illustrative data structure.

This structure is already partially implemented using a mock database. To progress the application to full functionality however additional changes must be made. The reminders are already stored in the object format described above (see Section 3 for an example). This is for a single mock user. To be a fully functional system sign-in/authentication must be integrated so specific users can have their own individual reminders which are stored in the database.

When the user creates or edits a reminder the object should then be appended to the user’s object in the database. This can be done using a server.post() command in the server/index.js and an associated control script, or directly using a mock database, which will need to be imported/required. From this point it is a simple matter of retrieving the reminder(s) from the database and displaying it/them to the screen.

# Reminder Persistence

## 3.1 Current Progress

The current functional iteration of our reminder application uses a mock database in the form of an exported dictionary from the database.js file. This mock database uses a modified structure to the proposed data structure of our planned data-structure found in the database\_exampke.json.

let Database = {

cindy: {

reminders: [

{title: "Example Reminder", datetime: "2020-04-08 12:00", tasks: ["task 1", "task 2"], id: 1},

{title: "Reminder 2", datetime: "2020-04-08 12:00", tasks: ["Tsk 1", "Tsk 2"], id: 2}],

username: "cindy@gmail.com",

password: 12345

}

}

This structure is simplfied. It does not have any tags and does not specifiy a first or last name. It also holds the date and time as a single value. However, much of the remaining structure remains the same, such as storing username and passwords, storing (sub)tasks as strings in a list, and so on. Please consider below, our proposed expanded database structure.

"users": [

{

"user\_name": "snewell",

"sign\_in\_password": "River Lethe 77",

"first\_name": "Simon",

"last\_name": "Newell",

"reminders": [{

"reminder\_name": "Work on 1800",

"date": "YYYY-MM-DD",

"time": "hh:mm",

"sub\_tasks": ["Review help videos", "Complete Landing Page", "Get forecaster.js working"],

"tags":["school", "1800", "project", "reminderapp"]},

{

"reminder\_name": "Get to Bed",

"date": "2020-04-06",

"time": "00:00",

"sub\_tasks": ["Get in bed", "Shut eyes", "Stay still for 6 hours"],

"tags":["school", "1800", "project", "reminderapp", "health"]

} ] } ]

Labelled Example:

}

"user\_name": "this\_is\_where\_their\_sign\_in\_goes",

"sign\_in\_password": "pass\_words\_go\_here",

"first\_name": "We can also store their names",

"last\_name": "First and last",

"reminders": [{

"reminder\_name": "The name of a specific reminder, or task.",

"date": "YYYY-MM-DD standard date form for ease and compatibility",

"time": "hh:mm hours then minutes on a military/24 hour clock",

"sub\_tasks": ["Each subtask as it is added", "Is assigned a new index", "."],

"tags":["we can search", "within a user's reminders", "for the reminder name", "but also tags"]

}]

}

As can be read above, the database at its’ highest level contains a key (users) which holds an array as its’ value. Each index of that array contains a dictionary belonging to a single user. The dictionary has the following keys for reference:

* “user\_name”:
* “sign\_in\_password”:
* “first\_name”:
* “last\_name”:
* “reminders”:

All of these keys, except ‘reminders’, contain strings containing the users personal identifying information. The ‘reminders’ key contains an array/list. Each index of the list contains a dictionary which is reminder object generated by the app. The common keys for all reminders are as follows:

* “reminder\_name”:
* “date”:
* “time”:
* “sub\_tasks”:
* “tags”:

Reminder\_name, date, and time are all strings representing the correct attributes of the particular reminder. Date and time are stored separately so that weather forecasts can easily process them. Sub\_tasks and tags contain arrays/lists holding strings. In the case of sub\_tasks these are descriptions written by the user. Tags are to be words or short collections of words which can be used by the yet to be implemented ‘search’ function of the application, as detailed in section 6.

Tags is a simple array with each index being a single tag, which was originally a comma separated item during the reminder’s creation. The tags are stored this way so that a search function is able to quickly iterate through them to find relevant reminders.

## 3.2 Work to be done/In-Progress

The main issues with data persistence are properly mocking the server, and the addition of the functions (discussed in section 2.2) which generate and update reminders. As the application is not currently designed to be hosted, edits to the mock database may only persist on a local machine. However, being able to add, edit, and remove reminders will be considered a proof of concept for persistent reminders.

To this end, the server.get() and server.post() functions, the reminder controller, and the mock database must all be properly established. The above example may already be used to mock database content and structure reminders. The development of the functions for interacting with it will require the functionality described in sections 1.3 and 2.2. Please refer to these sections for information on implanting specific functions.

# Authentication

(To be added after consultation with Authentication project lead, Trevor Hobenshield)

# Weather Forecasting (Dark Sky API)

## 5.1 Current Progress

Current progress on the use of the Dark Sky API includes the functional acquisition of relevant geolocational data of the app client. Code exists to attempt to fetch the API json to the application has been written as well as additional code to autogenerate images for the weekly forecast. This code is found in the public directory in the script.js file as a commented section.

However, repeated attempts to fetch the API were met with failure. Despite being in a node environment, the ‘require’ function continuously produces an error in console as being undefined. This error persists even in a mocked server environment. As the json response cannot be properly fetched further testing and implementation of the forecaster portion of the application has been put on hiatus.

Images for the various weather patterns that would be used by the forecasting functions of the application have been made, and static versions of these images have been added to the main reminder page in order to illustrate how the forecast was supposed to be displayed.

## 5.2 Work to be Done/In-Progress

The completion of the forecasting script in the main script.js, and its integration into the main page as an active element. The intention of this code is to provide accurate weather information in two areas. Firstly, a seven day forecast, beginning with the current day. Secondly, a weather reminder for the current day and additional warning if any reminders are present on days of inclement weather.

Several of these functions have already been written as a basic script, but due to errors in the fetch command are unable to be fully implemented. In order to add these functions to the application, the ‘require’ of node-fetch must be fixed. Following this, the forecast functions must be tested and additional <div> sections added to the appropriate ejs pages to provide the weather data.

As the, aforementioned, forecasting functions have not been tested, they are still considered In-progress. The specific requirements of the weekly forecast and daily forecast are detailed below in sections 5.2.1 and 5.2.2.

### 5.2.1 Weekly Forecast

The Dark Sky API json can retrieve a forecast from a specified future date, if the date/time is given in the url when called. In order to generate a weekly forecast the function must do the following steps in order:

1. Acquire the client’s coordinates and store them as latitude and longitude.
2. Create and store a variable with the current date
3. Create an iterative loop to increment the date at the end of each iteration for 7 days.
4. Within the loop fetch a json using the coordinates and the date
5. The supplied json has a key (daily) which stores a sub dictionary.
6. In the daily dictionary a key (data) hold a single item length list, which contains another dictionary.
7. This dictionary contains the relevant forecast for the given date in the API call.
8. Display this data for the correct day.

### 5.2.2 Reminder Forecast

The process for the reminder warning forecast is much the same as above. However, it does not require iteration through the days of the week with API calls. Rather, the user’s reminders are iterated through, while the dates are checked. If a reminder shares the current days date, then that reminder will be displayed as normal on the main reminder page.

Process an API call for the current day to Dark Sky. Display an appropriate weather icon next to the day’s reminder(s) on the main page.

The process for doing this is fundamentally the same as above. The display simply needs to associate the weather data returned with the correct icon. To do this a sub function should be created which cycles through possible returns from the weather data, and set the display img src to the correct image when a matching string is found.

# Searching and Organizing Reminders

## 6.1 Current Progress

The final version of the reminder application is to have two main methods of searching for reminders which have yet to be implemented. First, organizing reminders by date and time. This is already partly achieved through the reminder object’s data structure. However, the actual organization of the reminders needs to be coded.

Second, a search bar function which will display all reminders whose name or tags contain the searched term. This function has not been started.

## 6.2 Work to be Done/In-Progress

### 6.2.1 Sorting by Date

To sort by date, either weekly or monthly, each reminder will need to be retrieved. This already occurs during the page load for the view of all reminders. However, a new interstitial step must now occur. Rather than directly adding each reminder to the app page, they must now be sorted first.

This can be done one of two ways. Firstly, the reminders can be sorted during the page load in a separate function. This will require loading the entire list, sorting it, then displaying it. This is non ideal, as it is a laborious process which must be performed at each page load.

The ideal method to sort by date is to have the reminders pre-sorted, such that the existing code for loading reminders does not need to be changed. This is best done during the creation and editing of reminders. When input or saved, a reminder’s date information should be used to find the index it should appear at. Rather than appending to the end of the list of reminders, it should be inserted at this position. While this may momentarily increase the time to create a new reminder, each page load will not need to sort the reminders again.

Once sorted, two new ejs pages will need to be made. These are the Weekly and Monthly reminders pages. Both should resemble, and have the same functions, as the total reminders view. In the case of the weekly view, reminders should be checked as they are appended to the page. If a reminder is more than seven days away in terms of date (discounting time in hours/minutes) it should be excluded. As the list is pre-sorted, this is just a case of excluding any reminder that meets this condition.

The monthly view is the same but should only deal with reminders that share the current date’s month, rather than a pre-defined length of time. If the reminder has a matching month it is appended, while all others are excluded.

### 6.2.2 Searching for Reminders

Searching for reminders will require the reminders list to be searched in two different categories. First, the tags list and second the reminder name. These are the two areas to be referenced by the search function, in its most simple form.

During the search the user should input a string in the search bar and press the search button. Automatically a background script will run to render the ‘search’ page, which will display all reminders that are found by the search. The reminders themselves will be searched one by one.

In order to make the search condition as broad as possible, the input search term should be stripped of leading and trailing whitespace. Then it should be made all lowercase. All comparisons to the reminder name and tags should automatically also be made in the lowercase. This will prevent missing data due to strange or inexact case.

If a reminder name or tag contains the specified string it is to be appended to the search ejs page. Once again, pre-sorting the list at creation of a reminder ensures that all of these search items will be given in order.

As an additional precaution, search terms should most likely have a minimum length. This will prevent single character searches.