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 Ted Yuan. They provided an initial layout of features based on the initial examples from the original Notion Design Document.

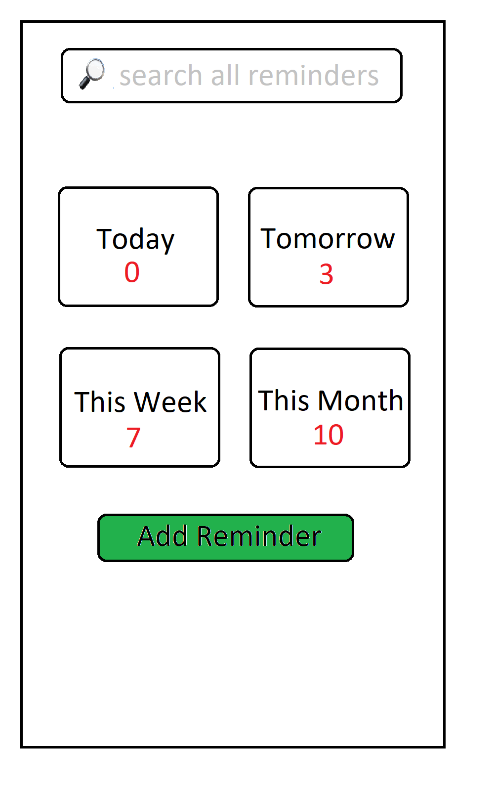
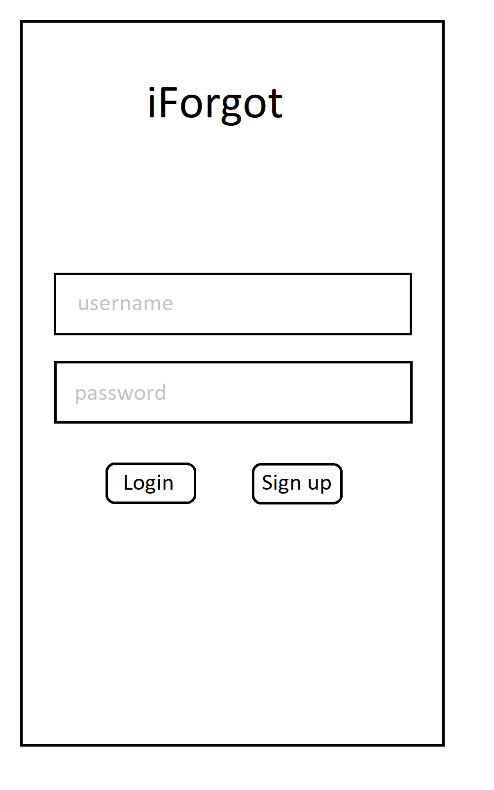


Figure 1: Original Basic Layout detailing key information

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.

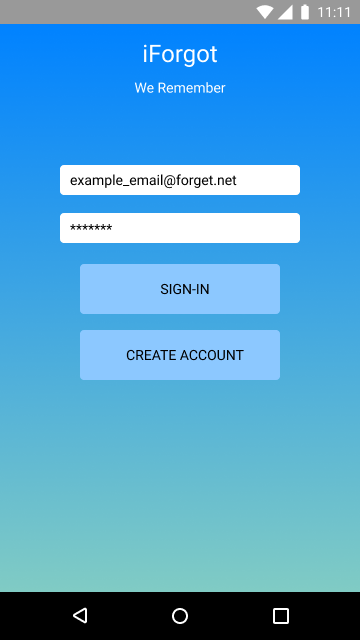
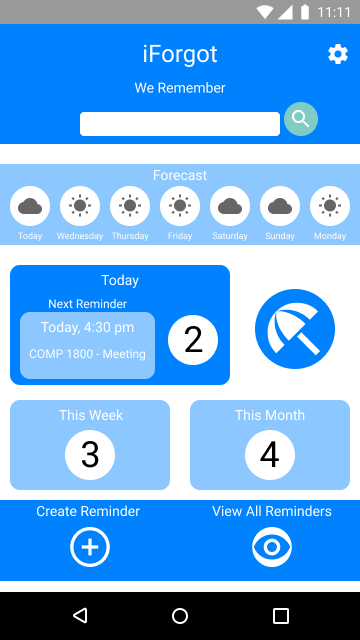
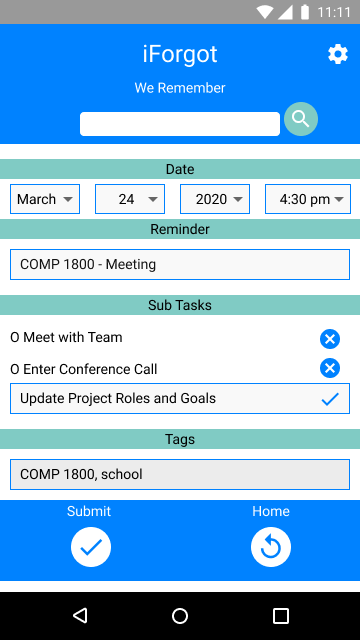
  

Figure 2: Prototypes for the sign-in, main application, and reminder creation pages.

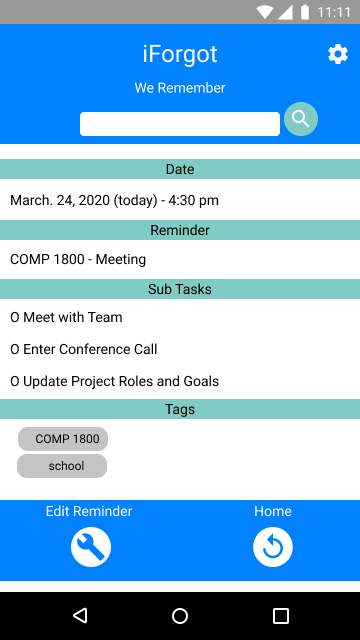
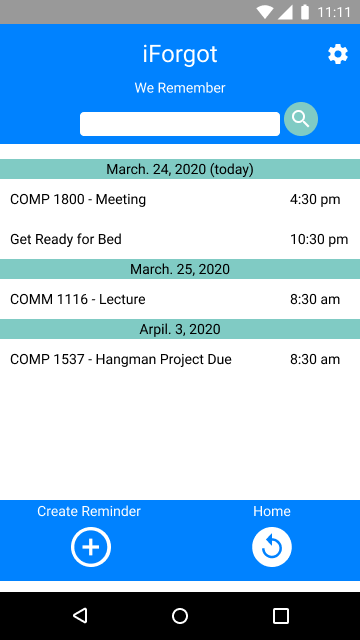
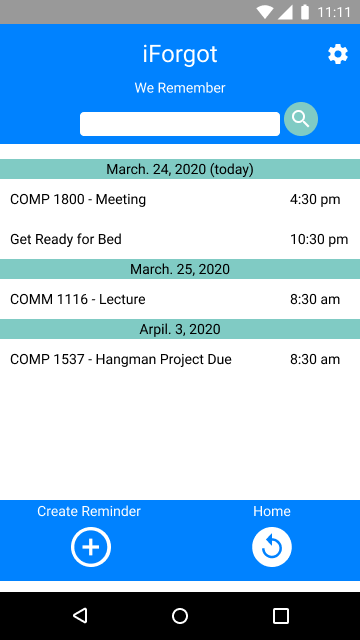
  

Figure 3: Prototypes of the individual, total, and monthly reminder pages

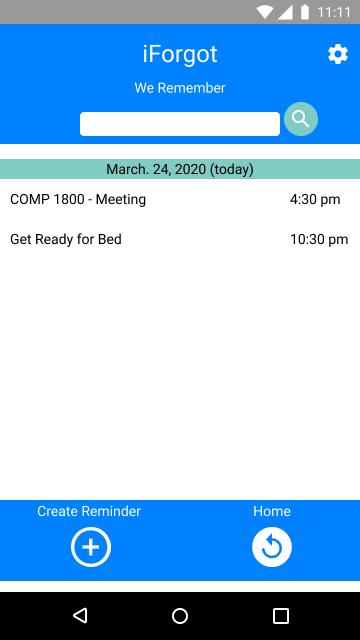
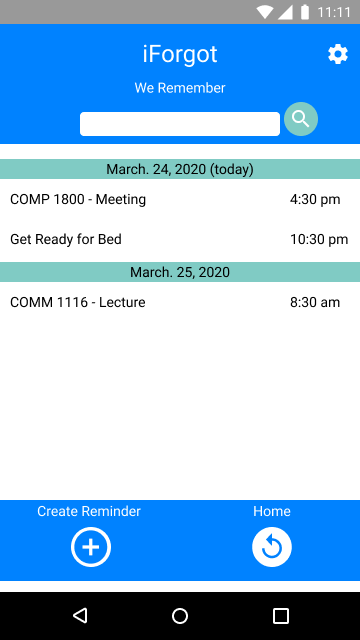
 

Figure 4: Prototypes for the daily and weekly reminders pages.

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.

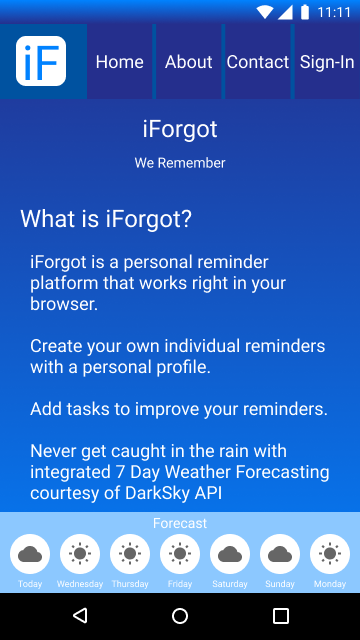


Figure 5: Prototype of the collapsed view of the landing page for phones and small screens.

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.

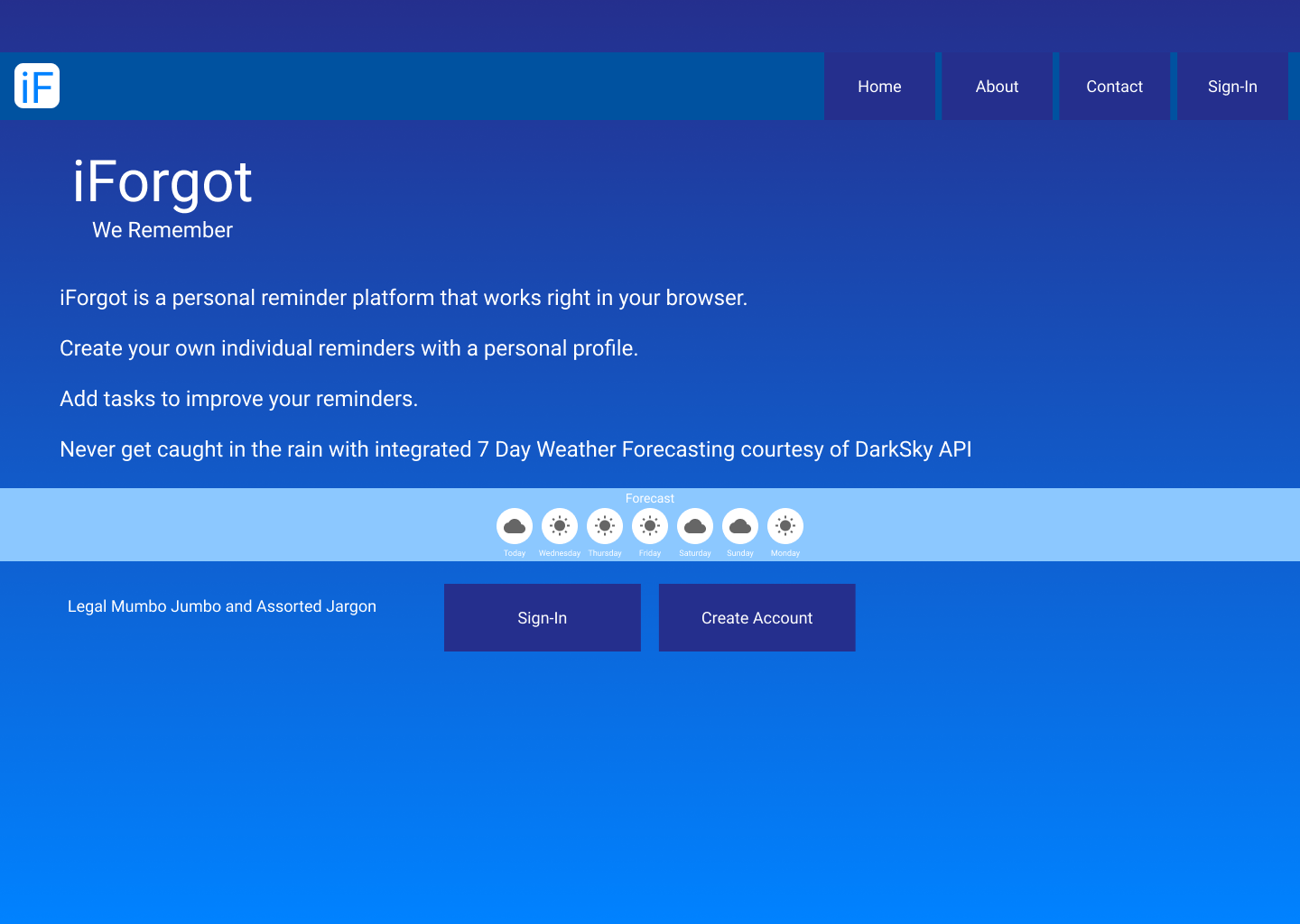


Figure 6: Prototype of the expanded landing page for desktop and larger screens.

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 ad layout completed and exist as individual html pages and are currently found in Ralph’s working branch of the project repository. 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.

## 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, the mocking of the server using node.js and express, and the merging of the regular html designs into a navigable series of linked web pages.

### 1.3.1 Mocking Server

Mocking the server properly is currently a high priority and a necessary component for progressing the project overall. Attempts by several team members to create a functional server environment have been unsuccessful. This is partly due to a misunderstanding of how to properly initialize npm and the subsequent modules that follow it. However, Trevor has successfully mocked the server for use in his authentication script.

Our main strategy for getting the server properly mocked is to use the 1800 reminder app development lectures and files as a reference. Once the localhost is properly initiated further development of the webpages and linking them will be able to be achieved.

### 1.3.2 Merging HTML

The current html pages for the application are found in Ralph’s working branch. This poses two issues. First, merging and rebasing the main branch to include the html files so that they can be properly accessed and linked to one another in addition to the other files and scripts. Secondly, the conversion of the html to ejs.

For the app to properly function as non static webpage, data will need to be exchanged between client and the (mock) server and database. The reminder pages will need to load in accordance to the correct user profile. This necessitates the use of ejs, in order to custom load the correct reminders and display them.

The method to achieve this will be to use a main control script and the express-ejs-layouts module from npm. This will allow individual links between pages to be designed as well as to allow information between database and webpage to occur.

The main server page (index) will allow individual pages to be linked using server.get() references to the controller.js. The controller.js will house the appropriate functions to create and render the proposed webpages, after they have been converted to ejs format. The converted ejs pages will be located in the views directory of the working application. This will allow each to be tracked and edited, as well as be accessed by the application.

# Creating/Editing Reminders

## 2.1 Current Progress

The current progress on editing reminders is highly limited and was generated by Ralph as a modified version of a list generating script. Current functionality is reserved to creating a named reminder, being able to mark an existing reminder as checked, and the deletion of a visible reminder. As with the current issues with mocking the server and linking of webpages, this script also must be integrated into the application’s greater framework.

## 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 to fix. The main areas that must be added or fixed are as follows:

* Addition of Reminder Descriptions
* Addition of Reminder Tags
* Addition of Date Information (Tied to .ejs as a menu rather than as a text box)
* Reformat of Reminders to java script objects
* Editing Reminders
* Integration of Database (see section 3. Reminder Persistence)

### 2.2.1 Description, Tags, and Date Information

The description and tags are easy additions to the current script in terms of input. Each may have its own dedicated text box as input within the ejs. The description can simply be given as a single text block submitted as a string.

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.

Date information is far easier to process, but requires greater attention in .ejs design. Dates will be a needed requirement for searching and displaying reminders, which are to be organized by date. They must be properly formatted such that they can be used without breaking backend java script and be used with forecast data from DarkSky.

The easiest way to implement the date is to use bootstrap. Once again there are two methods to consider. First, design out own form using bootstrap dropdowns and out own logic to generate Date and time for the reminder. This is reliable but will require considerable effort as we will need a total of 5 drop downs (year, month, day, hour, and minute) each with their own logic, however simple.

The second method is to use an existing opensource calendar/widget. This of course will require our investigation into the widget’s use and any required licencing issues.

### 2.2.2 Reformat Reminders to Objects and Database Integration

This particular part of our implementation should be carried out in tandem with 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.

In order to functionally implement this key feature the existing reminder creation must be modified. As above in 2.2.1, the description, tags, and date must be added to the form using text-boxes and drop downs. Upon submission, rather than immediately displaying to screen, the reminder object (dictionary) should be made using a java script object generator.

Ie:

function generateReminder(name, description, tags, date) { … }

let newReminder = new generateReminder(name, description, tags, date)

This 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.

## 2.2.3 Editing Reminders

To edit a reminder a dedicated page and script will be required. A button may be added to each displayed reminder, that when pushed initiates a script. This script should do two things: 1. Relocate the user to the editing screen, and 2. ‘select’ the current reminder and acquire its data.

The editing ejs page (currently html) will need to acquire and prepopulate the various reminder fields (name, description, tags, date). The user may then change, remove, or otherwise edit what is in the reminder. When the submit this page, the old reminder should be edited and updated in database.

# Reminder Persistence

## 3.1 Current Progress

Currently we have an object design structure for storing reminders in the database, which can be found as an example json file. Please see the below example of our current object format.

"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"]

}

]

}

]

}

"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. In the case of sub\_tasks this is where we can store the description. It will still be stored as an array, with a single entry, so that in future versions of the app the description can be broken into individual tasks or steps.

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 application. Unfortunately, due to multiple failures in attempting to create a mock server environment, this was not achieved. The use of the ‘require()’ function necessitates an active node.js/express server, so that the module can be retrieved. However, once this is done, affecting and improving the code should be viable.

## 5.2 Work to be Done/In-Progress

The completion of the application’s forecaster.js and the use of the Dark Sky API will need to address two issues. First, the 7-day forecast and display. Second, the daily forecast and reminder warning.

### 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.