[Document title]

[Document subtitle]

Harin Vyas
EL GENERICO®

Contents

El Generico® Company

Section 1 – Analysis

Part 1 – Description of the problem

El Generico® strives to make high-quality applications for our wide range of clients including myself, myself and myself. The problem which we shall be addressing for our client is the problem of aviation weather. Normally, pilots will get the generic flight info before the flight but the live data is provided using a Meteorological Terminal Aviation Routine Weather Report (METAR). This is what mainly determines what runway shall be in use, how you would land your plane and if you will be visually landing the plane (VFR) for electronically (IFR) (and probably some other stuff too).

Traditionally, this is given via Radio using a Text to Speech (TTS) but the pilot still has to decode this information. This is where the problem is. It is inconvenient to get you METAR like this: EGCC 111420Z 01012KT 5000 RA SCT021 BKN030 10/08 Q1018 TEMPO -RA. When you can get it decoded which makes it easier to understand so making it less likely to have air accidents as well as lowering flight time. The solution must be a way to present METAR in a more readable format

Part 2 – Stakeholders

Geoffrey Windsor – Pilot for Virgin Atlantic

Using my interview template.

1) What would you expect from this type of app?
   I would expect it to tell me the weather.
   Search functionality to get ICAO codes for me (I am very forgetful).
   Little icons to show weather.
2) What would make this app different from the alternatives?
   Using a TTS system so I don't have to look at my screen.
3) Does [the problem] exist in your field of work and do you think that my app can solve it?
   The problem does exist, the normal METAR format is long and decoding it would make life much easier. Yes
4) What sort of customisation would you like?
   Colour theme (dark mode minimum) and preferably other themes as well.

Joseph Jones– Flight sim Enthusiast

Using my interview template.

1) What would you expect from this type of app?
   Accurate info and data in a correct pilot format. Concise + minimalistic
2) What would make this app different from the alternatives?
   Free of charge with no extra money for premium features.
3) Does [the problem] exist in your field of work and do you think that my app can solve it?
   Exists for new people and the younger people who come to 'flight simming' and find it hard to understand the terms
4) What sort of customisation would you like?

Dark mode. Enhanced brightness for night time use. Airline based themes. Real-time updates.

Summary

From the interviews, I can summarise that the users would like the app to be able to give an accurate decoded METAR report in a readable format. As well as having a search functionality for ICAO codes and for the data to be presented in a concise and minimalistic format that includes appropriate icons.

We shall be able to stand out from the competition as we have a completely free app (except maybe donations) and a TTS system which helps when your hands are not free.

The problem mentioned does exist in the stakeholder's fields of work/hobby so there is surely a need for the app. My app should be able to solve the problem by being able to just present decoded METAR.

The App should have at least dark mode and light mode. But it is preferred for other themes to be included for example themes for the airline you are flying for. It should also have enhanced brightness for dark conditions so the text is readable and the app should have real-time updates on the weather.

Part 3 – Similar apps and what I will use from them.

App 1: Uno: Avia Weather - METAR & TAF (Play Store)

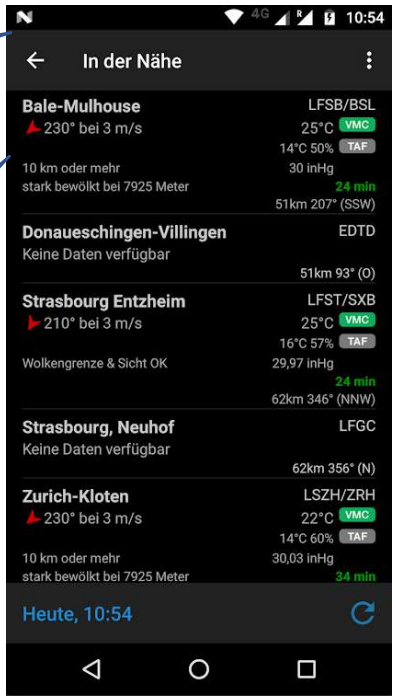
My App uses decoded METAR too. This makes it easier and quicker to read the data and still uses the same API.

This app gets runway data for each airstrip and uses it to help visualise the wind data for the airstrip so determines which runway shall be in use

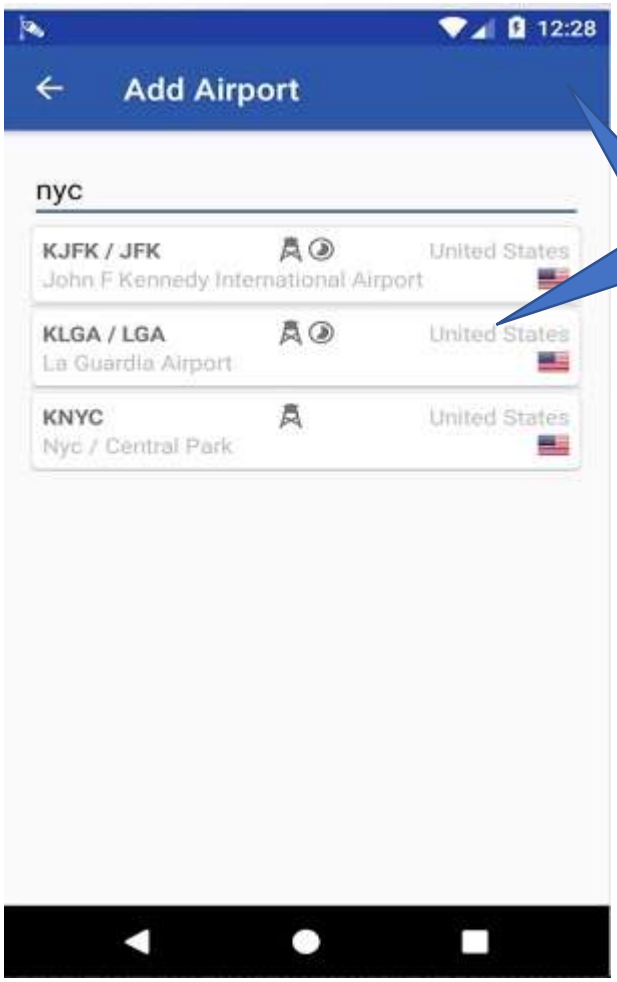
App 2: AeroWeather (Play Store)

This app uses dark mode to remove blue light so the users can use it at night time without disrupting the brains natural sleep cycle.

This app has multi-language support, something I could do with the google translate module to translate the default text to one which has been chosen in the user settings



App 3: Windsock - Automatic METAR/TAF (Play Store)



Here you can search for the ICAO code for the airport which is helpful if you forget it. It most likely searches through a csv file which has been decoded. It searches for the airport name then loads up the relevant ICAO code

This app has a action bar which helps the user navigate throughout the app. This will be implemented using the kivy.uix.Actionbar class. It has a back button which can send the user back to the menu. This should be on all the app 'pages'

Conclusion

From App 1, I will be using the decoded METAR as well as the runway data.

From App 2, I will be using the accessibly options of the translation and dark mode.

From App 3, I will be using the ICAO searcher and the navigation/action bar.

Part 4 – How the problem can be solved by computational methods

Thinking Abstractly and Visualisation

Abstraction can be used in the following ways:

- When the data is received from the API, it is in a complicated JSON format so the app should extract the stuff which is needed and present it in a clear format. The large JSON data is hidden from the user,

- Instead of worrying about class inheritance to move the user's data around the screens, we just set it as a global variable to make it easier to code and understand. The unnecessary need for class inheritance is cut out.
-

Thinking Ahead

- The App should be responsive. It should work on different screen sizes and operating systems as the users could be using a range of different devices to access the app. Without this, we would be narrowing the audience of the app to only Android users for example which would mean that there would be fewer users on the App so it wouldn't be solving the problem for everyone.
- The App should be accessible. It should have settings where users can change aspects of the app to suit them. Aspects include Dark/Light Mode, Language, Large text and a filter for color-blind people. This again gives a larger audience as well as hopefully standing out against any potential competition. Settings data will probably be stored in the database with numbers representing each option for the settings.

Thinking Procedurally & Decomposition

- There will be a login/ register system, a way to find an ICAO and a way to search for the weather.
  Decomposition is done in Section 2.

Thinking Logically

- Naturally, the app running state would be a loop in itself. Kivy (the framework which I am using) has this built into the 'App'. When the user does anything (types, presses a button, etc), the game state shall stop for the respective code to be run. The app running state will be a constant iteration that will end and close the app when the user exits the app.

Part 5 – Software and Hardware requirements

Hardware

- A Windows 7 or above PC with basic I/O devices
- At least 2GB of RAM
- 2GB free storage

Software

- A python interpreter
- Kivy module and dependences
- pyttsx3 module for TTS
- google translate module for translation

Other

- A stable internet connection, preferably 15 Mbps or above.

Part 6 – Success Criteria

The app must have at minimum:

Register System:

ID	Criteria	Why it's needed	How to check
1	A register system	So, we can add new users who can log in easily to the system. Also, requirement from clients	We are hoping for a working register system that can add new users and give error messages based on inputs.
2	Appropriate validation of the inputs	So, there are no overlaps in the user's data as well as no duplicate users with different passwords as this can mess up the login system.	Test it with all sorts of inputs e.g. missing inputs, existing users, etc.
3	The data to be kept secure e.g.: hashing passwords with salt.	This ensures that nobody who may have acquired access to the user's data cannot read their password off right away. Using salt ensures that hackers cannot use rainbow tables to get the passwords of users.	The passwords should be hashed with salt where the data is stored.
4	The user's data is set up in the correct format (TBD)	So, there is consistency in the data and to make it easier to parse and interpret the data.	Check where the data is stored that new user's data is in the correct format
5	Error messages if unsuccessful	So, the users know that their registration is unsuccessful and why it is unsuccessful.	Check by putting in data that should return an error message and check if the error message has appeared.
6	Success message	So, the user doesn't wonder, why isn't it done yet? They know that they can now log in.	^ but with data that would return a success message.

Login System

Id	Criteria	Why it's needed	How to check
7	A login system	Well, what good is a register system without a login system? Users need to be able to log in once they've registered of course.	If there is a working login system with good validation which takes the user to the main screen when successful.
8	Appropriate validation of the inputs	So, we don't waste processor time trying to log in on a user which doesn't exist	Fire in some inputs which should not make the user log in e.g. no password filled or the

			username not existing
9	Matching the hashes	This will hash the inputted password (with salt) to be able to match the hashed password as you cannot unhash a hashed password (unless you use PHP).	We need to check that when we input the correct password, it should log in which shows that the hashes have been matched.
10	Error messages if unsuccessful	So, the user knows that he messed up big time (and why).	Fire in some bad data, if we get an error message then we're cool.
11	On success, go to the main screen (TBD)	So, the user knows that he's logged in as he's now on the main screen. As well as to reassure them that they have done everything right.	Fire in some correct data (can be done at same time as matching the hashes) if the main screen loads then we're Gucci
12	On success also save their data as a global variable until the program shuts down.	So, the user's data can be used/updated in other parts of the program. For example, recent searches or settings preferences can be read and updated when necessary.	This we can only check once we have linked this data to something else e.g. a recent search system if they load properly then it works.

The metar search system.

Id	Criteria	Why it's needed	How to check
13	METAR search system	As it is the main point of the program innit.	A system which presents a decoded accurate METAR in a clear and concise format
14	A recent search system	So, there would be little point in saving the user's data then. It also allows for pilots who fly routes frequently.	Check if the recent searches have merged from the user's data and when you click it, it automatically goes in the search box.
15	Backend code which searches for the METAR using an API and returns the results	It is the main purpose of the program of course.	Check-in console with a print statement.
16	The results to be presented in a clear format	This will most likely be a list view with each section of the decoded metar on a separate line.	If when we show it to our shareholders, they find it sufficient.

17	A way for the user to search for the ICAO code	As this is a shareholder requirement because people can forget their ICAO codes. This will probably be implemented in another part of the app with buttons linking the 2.	Check that when a button linking the 2 screens is clicked, the screen changes.

The ICAO search system

ID	Criteria	Why it's needed	How to check
18	ICAO search system	Shareholder requirement from Geoffrey. As users can forget their ICAO code.	A system that accurately returns all results available based on the query in a clear format.
19	Search results to be clear and accurate	So, it makes it easiest for our users to understand.	Ask the shareholders if it is fine or not.
20	A way to navigate back to the METAR search system	It allows the app to be smooth and the user to use the newly discovered ICAO code for good use.	Check that when a button linking the 2 screens is clicked, the screen changes.

General stuff

	Criteria	Why it's needed	How to check
21	Dark mode and light mode	Dark mode to satisfy our shareholders and for users so be able to see better at night (as well as their eyes not dying from the sun god beaming down on them with light mode). Light mode is better for daytime (or normies) so that should also be an option.	It should be by default dark mode and when the option is set for light mode, it should change to what is generally considered light mode.
22	A TTS system	So, the users can hear the METAR when their hands are not free e.g. finishing off the checklists.	When the TTS button is clicked, you should be able to hear the METAR.
23	If the app is disturbed using windows store, app store, or play store the app must be kept free	To make sure that those who cannot afford such an app can still reap the benefits from it. It also lets us keep up with the competition which is generally paid apps.	Kinda obvious innit.

What isn't plausible

What isn't plausible	Why it isn't
Real-time updates	This involves having to keep the app open at all times as well as having to fiddle about with time in Kivy which can mess up the app state horribly so rather not take the risk tbh.

Section 2 – Design

Part 1 - Introduction

I will split this section by class. Each part will include the structure diagram, proposed screen designs, pseudocode, and any test data which we'll be firing in.

As my structure diagram is too big to fit in one piece, I will have each class diagram below. Note that the parent of each class is the same (the main app).

The structure diagram was hand-drawn on OneNote (with dark mode on, of course). Rectangles at the lowest generation shows a process and circles at the lowest level shows some text

I have decomposed the problem using a Top-down Module Design as it is an easier way to visualize the program showing the parent and child classes, and the widget inheritance so we know what Kivy styling will affect what parts. It also shows how the python code for each class will work. What processes are needed (in the procedures) after clicking certain buttons? The number hierarchy is used in order to match parts of the structure diagram with the screen designs.

All pseudo-code was typed in Notepad ++ then a screenshot of that was used to put the pseudo-code in.

Part 2 - How Kivy works

Kivy code (henceforth kv) works differently in most programming languages. It is best thought of as CSS as it is generally used to create 'widgets' (defined as "elements of a graphical user interface that form part of the User Experience. The kivy.uix module contains classes for creating and managing Widgets. "- https://kivy.org/doc/stable/api-kivy.uix.html). This includes things like the screen manager, UX widgets (e.g.: dropdown lists, buttons, text boxes, etc.) and layouts (how UX widgets are laid out on the screen e.g. boxlayout).

You would typically have a 'root' class (the parent class) that handles the screens (each of its children classes) e.g.: a login page and a register page would be screens which inherited by the root class. As the screens are children classes, they will also inherit from the root class (the parent) so if you added some code to change the background colour in the root class to pink (note that this does nothing to the root as the root is not a screen so it is not displayed), each of the screens will have a pink background colour.

In the python code, you would have what I will call a grandparent class which I believe can be best explained by the docs:

> "The App class is the base for creating Kivy applications. Think of it as your main entry point into the Kivy run loop. In most cases, you subclass this class and make your own app. You create an instance of your specific app class [which includes your instance of the root and screens] and then when you are ready to start the application's life cycle, you call your instance's App.run() [so if I called my instance WeatherApp, I call it using WeatherApp().run() (note the '()' at the end of WeatherApp. This is because we define the instance as a class in python)] method."
>
> (https://kivy.org/doc/stable/api-kivy.app.html)

So overall, it is best to think of it just like making a website. The root is the htdocs folder and each screen (or child class) is a webpage. The python code is like HTML scripting, you could use it do the jobs of JS and CSS (by the <script> and <style> respectively) but it is good practice to have separate .js and .css files and call them in (in Kivy, no code is needed for this). So, the kv file is the HTML and CSS which can create and style HTML elements (widgets in Kivy) and moves between the different webpages (screens for the sake of Kivy). The python code defines the screens, the root, and the app life cycle (through the grandparent class) and runs it all. The python code also performs most of the same code as JS can (even if modules are needed) which in my case will be done by adding methods and properties in the screens (as they are classes).

Notes

Initial Imports:

- App class from kivy.app to create the base for our application (as quoted before)
- BoxLayout class from kivy.uix.boxlayout for the layout system for our app
- DropDown class from kivy.uix.dropdown to be able to use the dropdown widget
- ObjectProperty class from kivy.properties so we can call for an object property from the kv code (explained below)
- UrlRequest from kivy.network.urlrequest to be able to make web requests for our weather
- Screen and ScreenManager classes from kivy.uix.screenmanager to be able to access and switch between different screens.
- JSON, hashlib, and os from standard library for data storage hashing passwords and for our data storage system respectively.

In the python/pseudocode for python code, when something like varname = ObjectProperty() is used, this means that it gets (inherits) varname from the kv code so in this case, it will get the 'varname' property from the kv code. This is like the getElementByID method in JS.

Part 3 – RegisterPage Class (1.1)

Breakdown of 1.1

The register page is split into a backend (with all the python code) and the frontend (with the Kivy code). The frontend will consist of a box layout that has been centred using padding on all sides (to make the page look better basically as centred layouts generally appeal to the user more). This layout will have 3 labels with 3 text-input boxes underneath. These are (in the vertical top to bottom order): Username, Password (with password formatting (*'s) for security) and email. Each of these inputs will have an id assigned to them (username, password, email respectively) so they can be used in the python code via ObjectProperty when they are assigned as an object property in the Kivy

code (e.g.: username_input: username with the same for new properties password_input and email_input).

Next, there will be an anchor layout (who's parent is the box layout) which has 2 buttons: submit which will trigger the backend code to run and log in (CHANGE ON SCREENSHOT) which takes the user to the login page (1.2) when clicked. Finally, there is the confirmation text which is a label with an id assigned to it ('confo' so at the top it is assigned to property 'validation' via 'validation: confo'). In the screen design, a box has been shown to show where the confirmation text would appear but in-fact it is invisible until the backend changes the confirmation text. The confirmation text will either report any errors in the python code (e.g. missing the input for email or username is already taken) or will report that the registration was successful.

The backend will firstly get all the object properties from the Kivy code by doing varname = ObjectProperty(), this is done for all the object properties. Next, there should be a procedure that does some validation on the inputs so it should check if the form is properly completed and if the username/email is already taken (exists in the JSON file/DB). To use the JSON file/DB, it should open it using the JSON module/the DB's module and extract each user's data into an embedded list (then the file closed) which then can be iterated over to check for the validation criteria. If any of these are true then the validation text (which has been brought over as an object property) should be changed to an appropriate message e.g. Form not completed. If it is fine then it moves to the next procedure.

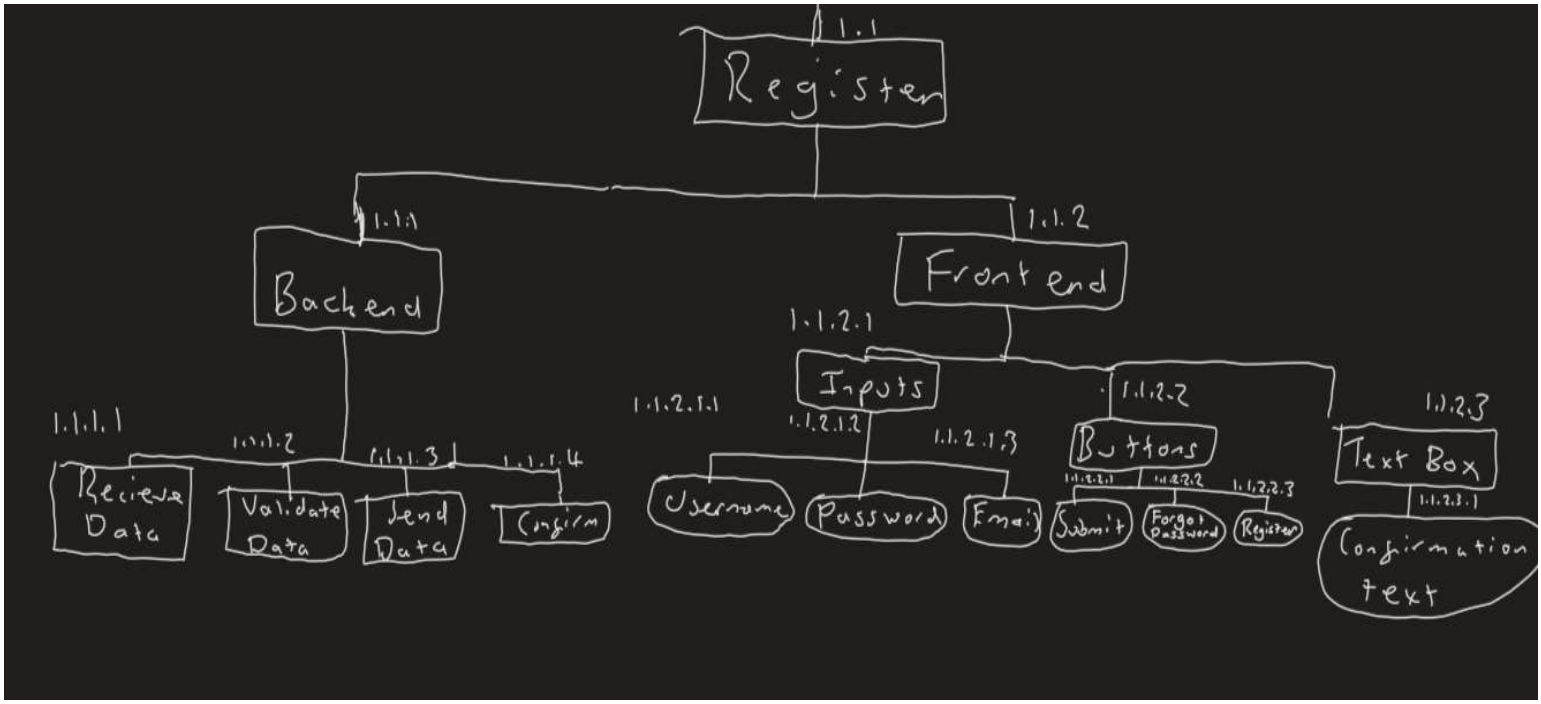
Next, a new procedure will be used for the actual registration. A variable called 'id' shall be the highest number id in the JSON file/DB + 1. Next, a salt should be created and stored under a variable. Then a variable with the password hash is created which is equal to an md5 hash (using hashlib module) of the salt + the password_input object property, this is encoded using utf-8. I am using hashing so a hacker cannot get the passwords and embedded salt to protect against rainbow tables. After this:

For JSON, update the extracted data with the new user's data (id, username, password, email, etc.), in the correct formatting. Then the JSON file will be closed, then this data will be dumped to a new temporary file, the old one deleted and the temporary file renamed to the old one's name using the os module.

For a DB, the DB will just be updated using the data at the end then the DB closed.

Either way, at the end of the code, the input boxes will be cleared and the validation text object property updated with a success message. MAYBE IN GREEN COLOUR FONT.

Structure Diagram:

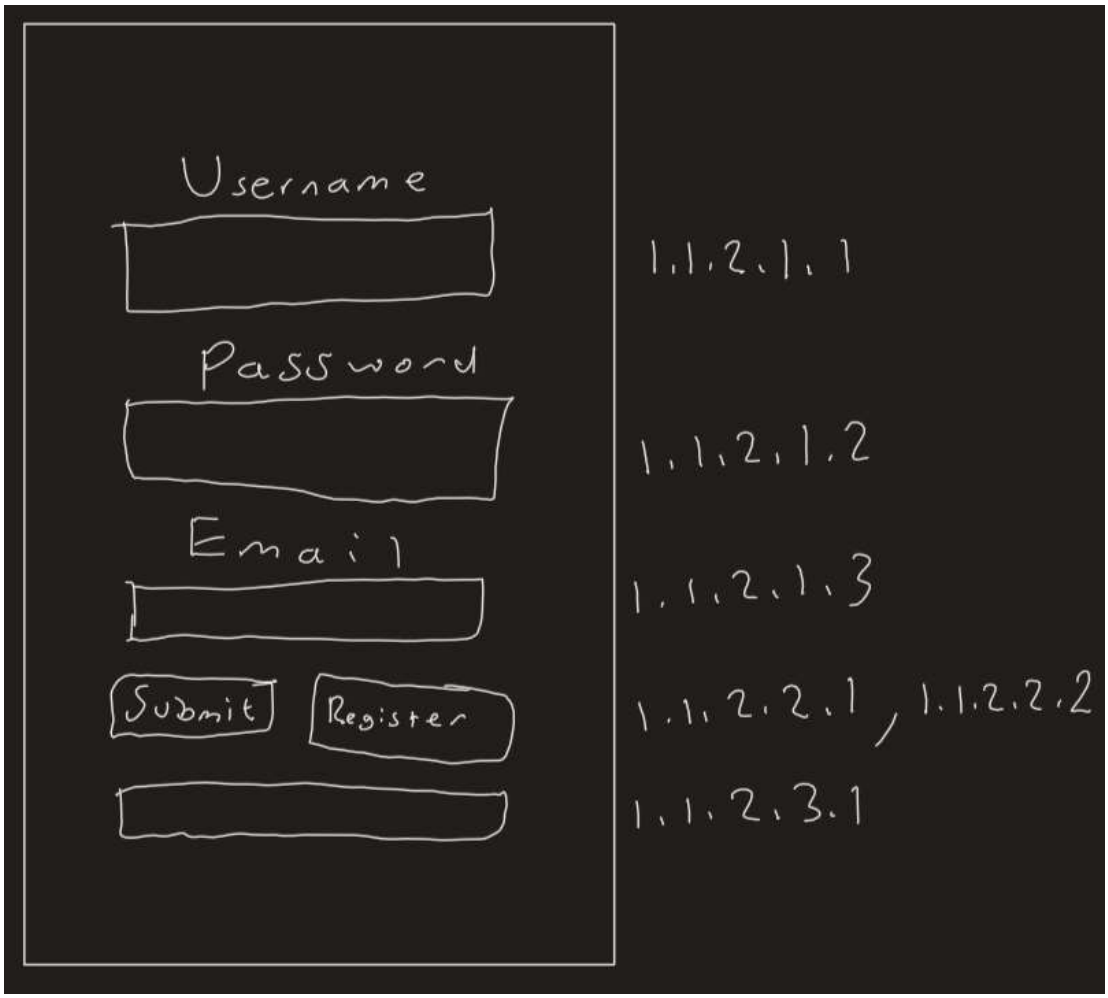


Key Variables:

Name	Type	Explanation/Usage	Link to success criteria
register_validation	procedure	The procedure for validating the details. Will return error message or will move to register procedure	1
register	procedure	Registers the user to the data source (adds the user details there).	1
user_input	ObjectProperty	Input from the form which will be added to the DB/JSON file. It is also used for validation	1,2
password_input	^	Input from the form which will be added to the DB/JSON file.	1
Email_input	^	^ as well as validation	1,2

validation	^	This is used to tell the user of success/failure messages	1,5,6
data	List	This is where the DB/JSON files data will be stored then will be edited with the new user's program then added to the data source. It will also be searched through	1, 4
idList	Dictionary	Will be iterated over for validation (so we iterate over all existing users)	1,2
passHash	String	The hash (with the salt) for the user.	1,3
f	JSON file	What we open our file as so we dump our data	1,4

What it should look like:



CHANGE SCREENSHOT ^ Pseudocode

```
RegisterPage Class

Python

//defines a class which inherits BoxLayout and Screen
CLASS RegisterPage(BoxLayout, Screen) {

    // gets the respective object properties from the kv code and sets them as variables
    username_input = ObjectProperty()
    password_input = ObjectProperty()
    email_input = ObjectProperty()
    validation = ObjectProperty()

    //defines a method which takes self
    PROCEDURE register_validation(self){
        // opens data.json to read and sets it to variable f
        f = OPENREAD("data.json")
        // uses json.load to decode the data into a python dictionary called data
        data = json.LOAD(f)
        // initialises a list which will store the id's of each user
        idLst = []
        // closes the file as all neccesary data has been extrapolated
        f.CLOSE()
        // a for loop which iterates over each user, gets their id and adds it to the list
        FOR users IN data['users'] {
            idLst.append(users)
        }
        ENDFOR
```

```
29
30          // a for loop to validate the user (iterates over each users id)
31          FOR id IN idLst{
32              // if any of the input fields are empty then it changes the validation text to tell the user that the form isn't complete
33              IF (self.username_input.text OR self.password_input.text OR self.email_input.text) == "" {
34                  self.validation.text = "Form not completed"
35              }
36              // if the username or email is already in use (in the list of users) then it will tell the user so by changing the validation text
37              ELSEIF (self.username_input.text == data['users'][id]['username']) OR (self.email_input.text == data['users'][id]['email']) {
38                  self.validation.text = "Username or Email is taken"
39              }
40              // else it will send the user to the register method, giving it the list of user ids and the whole decoded json dictionary
41              ELSE {
42                  self.REGISTER(idLst, data)
43              }
44              ENDIF
```

```
45          }
46          ENDFOR
47      }
48      ENDPROCEDURE
49
50      // defines a method which takes self, the list of users ids and the whole decoded json dictionary
51      PROCEDURE register(self, idLst, data){
52          // as every id is currently a string, this is converted to an integer so we can get the last id number and add 1 as the id's increment
53          // We convert this to a string as the formatting in the json file requires the id to be a string
54          id = TOSTRING(TOINTEGER(idLst[-1]) + 1)
55          // sets the hash salt (used to account for rainbow tables)
56          salt = "foo"
57          // this hashes the salt and the password (both are joined with the salt first) using md5 encryption
58          passHash = hashlib.MD5((salt & self.password_input.text).ENCODE("utf-8")).HEXDIGEST()
59          // this adds the new users data to the end of the decoded json dictionary
60          data['users'][id] = {"username": self.username_input.text, "password_hash": passHash,
61                               "email": self.email_input.text, "recent_searches_METAR": [], "recent_searches_ICAO": []}
62
63          // this creates a temporary json file and dumps the modified data there and closes it
64          f = OPENWRITE("temp.json")
65          json.DUMP(data, f, indent=2)
66          f.CLOSE()
67
68          // this deletes the old json file, and renames the temporary one to be the same name as the old one (basically replacing them)
69          os.REMOVE("Data/data.json")
70          os.RENAME("Data/temp.json", "Data/data.json")
71
72          // this tells the user that the registration is complete and empties the form incase another user wants to be registered.
73          self.validation.text = "Registration Complete"
74          self.password_input.text = ""
75          self.email_input.text = ""
76          self.username_input.text = ""
77      }
78      ENDPROCEDURE
79  }
80  ENDCLASS
81
82
```

```
83  Kivy
84
85  // child of WeatherRoot so inherits the background color
86
87  CLASS RegisterPage {
88
89      // sets the name so the screen manager can identify this screen
90      name = "Register"
91      // sets object properties for certain widgets given their id so they can be manipulated in the python code.
92      username_input = username
93      password_input = password
94      email_input = email
95      validation = confo
96
97      // sets a box layout for the widgets on the screen.
98      // box layout lays out the widgets on the screen horizontally so all of the screen is filled with equal space given to each widget
99      BoxLayout {
100         // tells the box layout to lay the widgets vertially instead.
101         orientation = "vertical"
102         // adds padding to either side (L, T, R, B) in pixels so the boxlayout is centered and doesn't take up the entire screen
103         padding = [100, 50, 100, 50]
104         // adds a spacing of 30 pixels between each widget
105         spacing = 30
106         // centers the widgets horizontally
107         center x = True
```

```
        // creates a label widget which has the text "Username" and who's font size is 25 pixels
        Label {
            text = "Username"
            font_size = 25
        }
        // creates an input widget which has been given an id, has a font size of 25 pixels. The widget is also stretched vertically so it is 175% of the original.
        TextInput {
            id = username
            font_size = 25
            size_hint_y = 1.75
        }
        // creates a label widget which has the text "Password" and who's font size is 25 pixels
        Label {
            text = "Password"
            font_size = 25
        }
        // creates an input widget which has been given an id, has a font size of 25 pixels. The input cannot take multiple lines and has been given password formatting
        // The widget is also stretched vertically so it is 175% of the original.
        TextInput {
            id = password
            password = True
            multiline = False
            font_size = 25
            size_hint_y = 1.75
        }
        // creates a label widget which has the text "Email" and who's font size is 25 pixels
        Label {
            text = "Email"
            font_size = 25
        }
        // creates an input widget which has been given an id, has a font size of 25 pixels. The widget is also stretched vertically so it is 175% of the original.
        TextInput {
            id = email
            font_size = 25
            size_hint_y = 1.75
        }
        // Creates a anchor layout inside the box layout. An anchor layout, lays out its widgets vertially and horizontally to a set place eg left, center or right
        AnchorLayout {
            // creates a button widget which has the text "Go", font size is 25 pixels and is stretched vertically to 60% of the original
            // and horizontally to 150% of the original. It is aligned so it is in the centre (horizontally) of the anchor layout
            Button {
                text = "Go"
                font_size = 25
                size_hint_x = 0.6
                size_hint_y = 1.5
                halign: 'center'
                // once the button is clicked, it runs the register_validation() method in the python code
                on_release {
                    root.register_validation()
                }
            }
        }
        // Creates a anchor layout inside the box layout
        AnchorLayout {
            // creates a button widget which has the text "Already got an account?", font size is 28 pixels and is stretched vertically to 60% of the original
            // and horizontally to 150% of the original. It is aligned so it is in the centre (horizontally) of the anchor layout
           Button {
               text = "Already got an account?"
               size_hint_x = 0.6
               size_hint_y = 1.5
               halign = 'center'
               font_size = 28
               // once the button is released, it calls the screen manager to move to the login screen by setting the current screen to "Login" with a transition to the left
               on_release{
                   app.root.current = "Login"
                   root.manager.transition.direction = "left"
               }
           }
       }
       // creates a label widget with no text, who's font size is 25 pixels and has the id "confo". This will be where the confirmation/error text will be.
       Label {
           id = confo
           text = ""
           font_size = 25
       }
   }
}
ENDCLASS
```

PUT IN TEST DATA

Test Data

Test number	What are we testing for	Expected result	Test data
1	Check usage of the text boxes and to make sure nothing is missing	When input text into the boxes, it should be presented clearly and be correctly sized.	Just some random strings
2	To see if the password formatting works	Whatever text we put in the password box; it should be replaced with asterisks.	Random string

Test number	What are we testing for	Expected result	Test data
3	Test validation process, no username	Expect to see “Form not completed” show up in the confirmation text area and for no registration to continue.	Random strings entered for password and email, nothing in username input box.
4	Test validation process, no password	^	Random strings entered for username and email, noting in password input box.
5	Test validation process, no email	^	Random strings entered for password and username, noting in email input box.
6	Test validation, already existing user	“Username or email is already taken” warning message.	Username = “f”, password and email have random strings
7	Testing validation, existing email	^	Email is “h@h.h”, password and username are random strings.
8	Test with legitimate data which hasn’t been used	It should register the user and there should be a confirmation message for the user.	Username = “re” Password = random string Email = “re@re.re”

Post Test Data

Part 4 - LoginPage Class (1.2)

Description

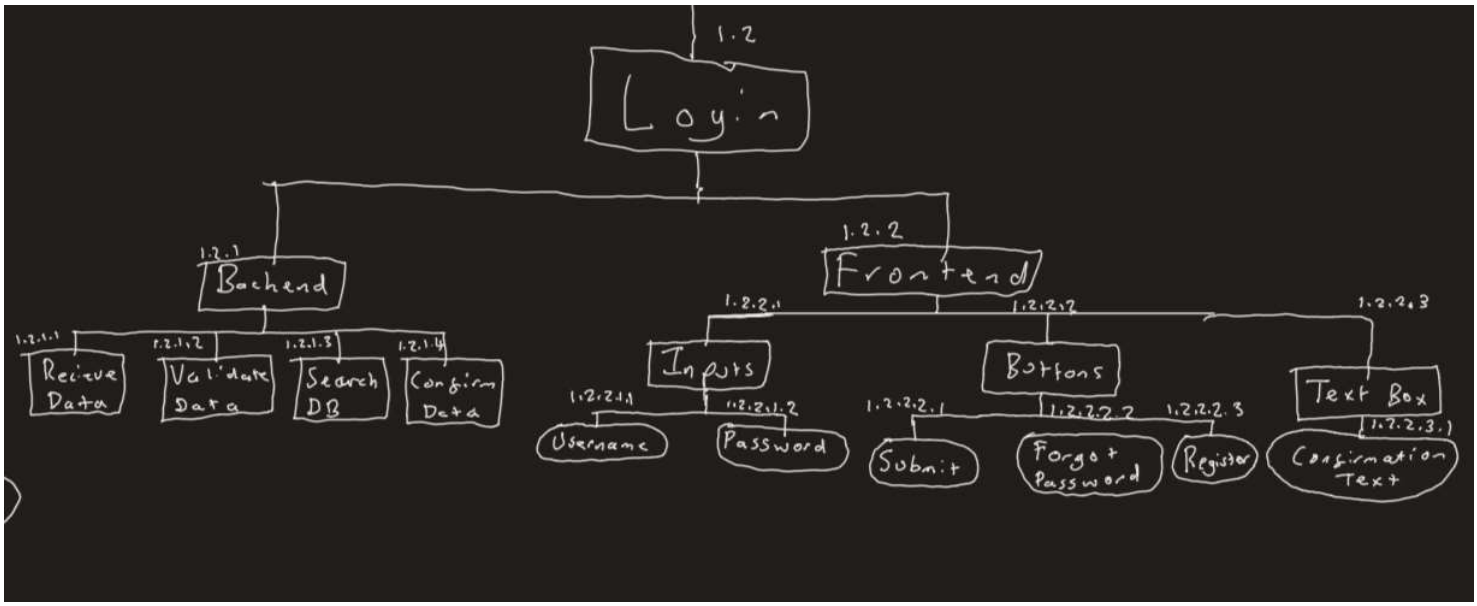
The login page class also consists of a frontend (with Kivy code) and a backend (with python code ofc). The frontend consists of a box layout also but this time its structure is a bit different. This time, there are only 2 label/ input boxes: for username and password, each with their own id’s and have been declared as object properties. Then there is an anchor layout for the Login/Go (PLS CHOOSE ONE OF THESE) button so it can be aligned in the centre using the halign property; once clicked, it will run the backend code. Next, there is an embedded box layout that contains the forgot password and register buttons this is used because we can stack these 2 buttons horizontally and box layout is the best for stacking widgets. The forgot password DOES THIS and the register button moves the

user to the register screen (1.1). Finally, there is the confirmation text which works the exact same as in 1.1.

The backend would firstly, get the object properties from the Kivy code using varname = ObjectProperty(), this is done for the username input, password input, and validation text. Next, there is a procedure that will extract the data from the JSON file/ DB into a dictionary. This is then iterated over to create a list of id's and users (which has the user info in it). Next, it would validate by checking if the form is complete and if the user exists in the list. If not then an appropriate message is set as the validation text object property. If it passes the tests then it is moved to the next procedure

There will then be another procedure which opens the user's info based on the username input and sets it to a dictionary. Next, we create a password hash using the same salt + password input method as in 1.1. Then we check this against the password hash in the database. If they don't match then the validation text is set saying that the password input is wrong. If they match then the user is moved to the AddLocationForm class (1.3).

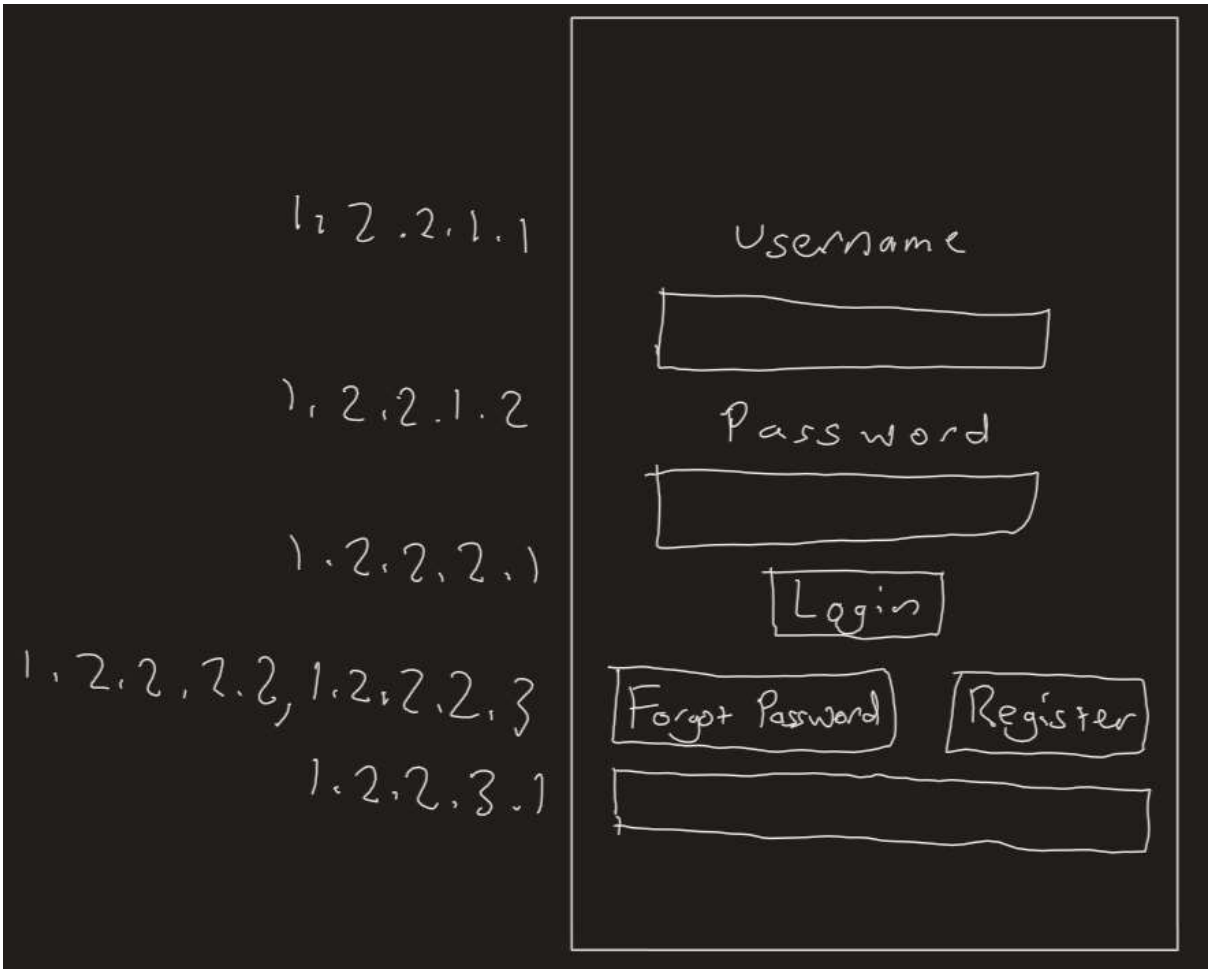
Structure Diagram



Name	Type	Explanation/Usage	Link to success criteria
validate	procedure	The procedure for validating the details. Will return error message or will move to register procedure	7
login	procedure	Registers the user to the data source (adds the user details there).	7

user_input	ObjectProperty	Input from the form which will be used for validation	7,8
password_input	^	Input from the form which will be used for validation	7,8,9
validation	^	This is used to tell the user of success/failure messages	7,8
Data	List	This will be searched through to validate the user's data.	7, 8
idList	Dictionary	Will be iterated over for validation (so we iterate over all existing users)	7,8
passHash	String	The hash (with the salt) to match with the data.	7,8,9
f	JSON file	What we open our file as so we dump our data	7
Usr_data	ObjectProperty	This is where the user's data is stored for future usage	7, 12

Screen Design



Pseudocode

```
LoginPage Class

Python

CLASS LoginPage(BoxLayout, Screen) {

    // gets the respective object properties from the kv code and sets them as variables
    username_input = ObjectProperty()
    password_input = ObjectProperty()
    validation = ObjectProperty()

    PROCEDURE validate(self) {
        // opens data.json to read and sets it to variable f
        f = OPENREAD("data.json")
        // uses json.load to decode the data into a python dictionary called data
        data = json.LOAD(f)
        // initialises 2 lists, one for the id's and the other for the usernames
        idLst, usrLst = []

        // fills out these lists by iterating over the data from the json file
        FOR id IN data['users'] {
            idLst.APPEND(id)
            usrLst.APPEND(data['users'][id]['username'])
        }
```

```
        ENDFOR
        // tells the user if the form is not completed
        IF (self.username_input.text OR self.password_input.text) == "" {
            self.validation.text = "Form not completed"
        }
        // tells the user if the user dowsn't exist in the json file
        ELSEIF (self.username_input.text NOT IN usrlst) {
            self.validation.text = "User doesn't exist"
        }
        // if neither is true, validation is complete and the data is carried onto the next procedure
        ELSE {
            self.login(usrLst, idLst, data)
        }
        ENDIF
    }
    ENDPROCEDURE

    // the procedure for checking passwords and login taking the arguments from the previous procedure
    PROCEDURE login(self, usrLst, idLst, data) {
        // creates a dictionary from the usernames and id's with usernames as the keys
        users = DICT(ZIP(usrLst, idLst))

        // creates a dictionary which has all the specific users data in it
        usrinfo = data['users'][users[self.username_input.text]]
        // we save the user info for later
        self.usr_info = str(usrinfo)
        // the salt which was used in the RegisterPage Class
        salt = "foo"
        // we hash the input for the password using the same method as in the RegisterPage Class
        passHash = hashlib.MD5((salt + self.password_input.text).ENCODE("utf-8")).HEXDIGEST()
        // If the inputed password's hash matches the hash stored then the user is moved to the screen with id 'AddLocation'.
        IF usrinfo['password_hash'] == passHash {
            self.manager.current = 'AddLocation'
        }
        // if the inputed password's hash doesn't match then the user is told that their password input is wrong
        ELSE {
            self.validation.text = "Password is wrong"
        }
        ENDIF
    }
    ENDPROCEDURE
}
ENDCLASS
```

```
Kivy

// child of WeatherRoot so inherits the background color

CLASS LoginPage {

    // sets the name so the screen manager can identify this screen
    name = "Login"
    // sets object properties for certain widgets given their id so they can be manipulated in the python code.
    username_input = username
    password_input = password
    validation = confo

    // sets a box layout for the widgets on the screen.
    // box layout lays out the widgets on the screen horizontally so all of the screen is filled with equal space given to each widget
    BoxLayout {
        // tells the box layout to lay the widgets vertially instead.
        orientation = "vertical"
        // adds padding to either side (L, T, R, B) in pixels so the boxlayout is centered and doesn't take up the entire screen
        padding = [100, 50, 100, 50]
        // adds a spacing of 30 pixels between each widget
        spacing = 30
        // centers the widgets horizontally
        center_x = True

        // creates a label widget which has the text "Username" and who's font size is 25 pixels.
        Label {
```

```
            text = "Username"
            font_size = 25
            // sets text colour to blanched almond
            color = 1, 0.9, 0.8, 1
        }
        // creates an input widget which has been given an id, has a font size of 25 pixels.
        // The widget is also stretched vertically so it is 175% of the original.
        TextInput {
            id = username
            font_size = 25
            size_hint_y = 1.75
            color = 1, 0.9, 0.8, 1
        }
        // creates a label widget which has the text "Password" and who's font size is 25 pixels and with the blanched almond colour.
        Label {
            text = "Password"
            font_size = 25
            color = 1, 0.9, 0.8, 1
        }
```

```
    // creates an input widget which has been given an id, has a font size of 25 pixels.
    // The input cannot take multiple lines and has been given password formatting
    // same colour settings.
    TextInput {
        id = password
        password = True
        multiline = False
        font_size = 30
        color = 1, 0.9, 0.8, 1
    }
    // Creates a anchor layout inside the box layout
    AnchorLayout {
        // creates a button widget which has the text "Go", font size is 30 pixels.
        // It is aligned so it is in the centre (horizontally) of the anchor layout.
        Button {
            text = "Go"
            font_size = 30
            size_hint_x = 0.5
            halign = 'center'
            color = 1, 0.9, 0.8, 1
            // once the button has been pressed, it runs the validate method in the LoginPage class in the python code.
            on_release {
                root.validate()
            }
        }
    }
    // creates a box layout in the box layout
    BoxLayout {
        // creates a button with text "Register" with font size 28 for the text, blanched almiond text colour and size 50% of the original in the x axis.
        Button {
            text = "Register"
            size_hint_x = 0.5
            font_size = 28
            color = 1, 0.9, 0.8, 1
```

```
            // once the botton has been pressed, it moves to the register screen with an epic screen transition
            on_release {
                app.root.current = "Register"
                root.manager.transition.direction = "right"
            }
        }
        // creates a button with text "Forgot Password" which has been shrinked to 50% of the original in the x axis, font size is 28 and colour is blanched almond
        Button {
            text = "Forgot Password"
            size_hint_x = 0.5
            font_size = 28
            color = 1, 0.9, 0.8, 1
        }
    }
    // creates a label (and gives it an id) with font size 28, this is where the confirmation / error text will go.
    Label {
        id = confo
        text = ""
        font_size = 25
    }
}


}
```

Test Data

Test number	What are we testing for	Expected result	Test data
1	Check usage of the text boxes and to make sure nothing is missing	When input text into the boxes, it should be presented clearly and be correctly sized.	Just some random strings
2	To see if the password formatting works	Whatever text we put in the password box; it should be replaced with asterisks.	Random string
3	Checking if the validation which checks if the form is filled woks. (no username)	The validation text should change to “Form not completed”	Nothing as username as a random string as password
4	^ but with no password	^	Nothing as password but some random string as username.
5	^ but with no username or password	3	Nothing for either username or password
6	Test if the validation to check if the user exists	There should be an error message as we have not created the AddLocation class yet.	A user that exists for the username e.g. f and some random string for the password

Post Data

Part 5 - AddLocationForm (1.3)

This is the main part of the program., where we get the METAR reports. Just like the other classes, it will consist of a frontend and a backend. The frontend will consist of a title at the top which will be centred in the middle horizontally. Underneath that, there will be a horizontally orientated box layout. This will consist of:

- A search box, which will be id’s so its contents can be retrieved in the backend.

- A search button, to trigger the backend code for searching.
- A recent search button which will open a dropdown box directly below it. See below
- Forgot the ICAO button which takes the user to 1.4.

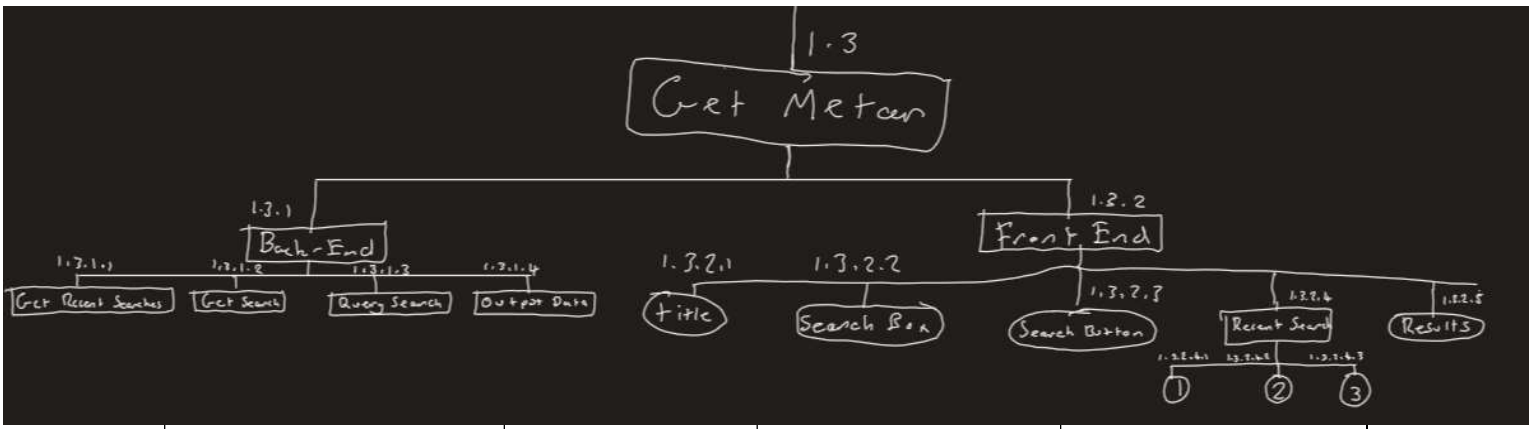
The dropdown is id'd and only opens when the recent search button is opened. If the screen elsewhere is clicked then it closes (so it doesn't stay opened permanently). It has 3 buttons inside. Once the recent search button is clicked, a backend script will run which collects the backend scripts from the user's data than will fit the 3 buttons with the recent searches as their text so when one of these 3 buttons are clicked, the search box contents changes to that of the button. The recent search system allows a pilot or enthusiast to quickly search for METAR for frequently flown routes hence helping our target market.

Finally, there is the big space where (for now) there will just be a ListView widget (which displays a list/ array in vertical order filling up the screen). This is where we will display the decoded METAR in the ListView way so each bit of the metar is stacked vertically like bullet points. In future versions, this can include such things as a map, runway information, etc.

For the backend, there will be 3 procedures, one to fetch the recent searches and 2 to handle the search (fetch it, query it and output the results). The first one will get the users data from their login (saved a global variable), format it into a dictionary format then fill them in for the dropdown boxes.

For the search, there will be 2 procedures, one to send the request and one which triggers the request once the request comes back (auto-triggered with Kivy's URL request class). The first will put the search request in an already defined template then fire it off with the URL request class. The second will decode the request data using JSON.loads into a python dictionary. An empty list is created for the data to add, this is populated with the summary of the metar for now. Once done, it will update the list view widget on the screen.

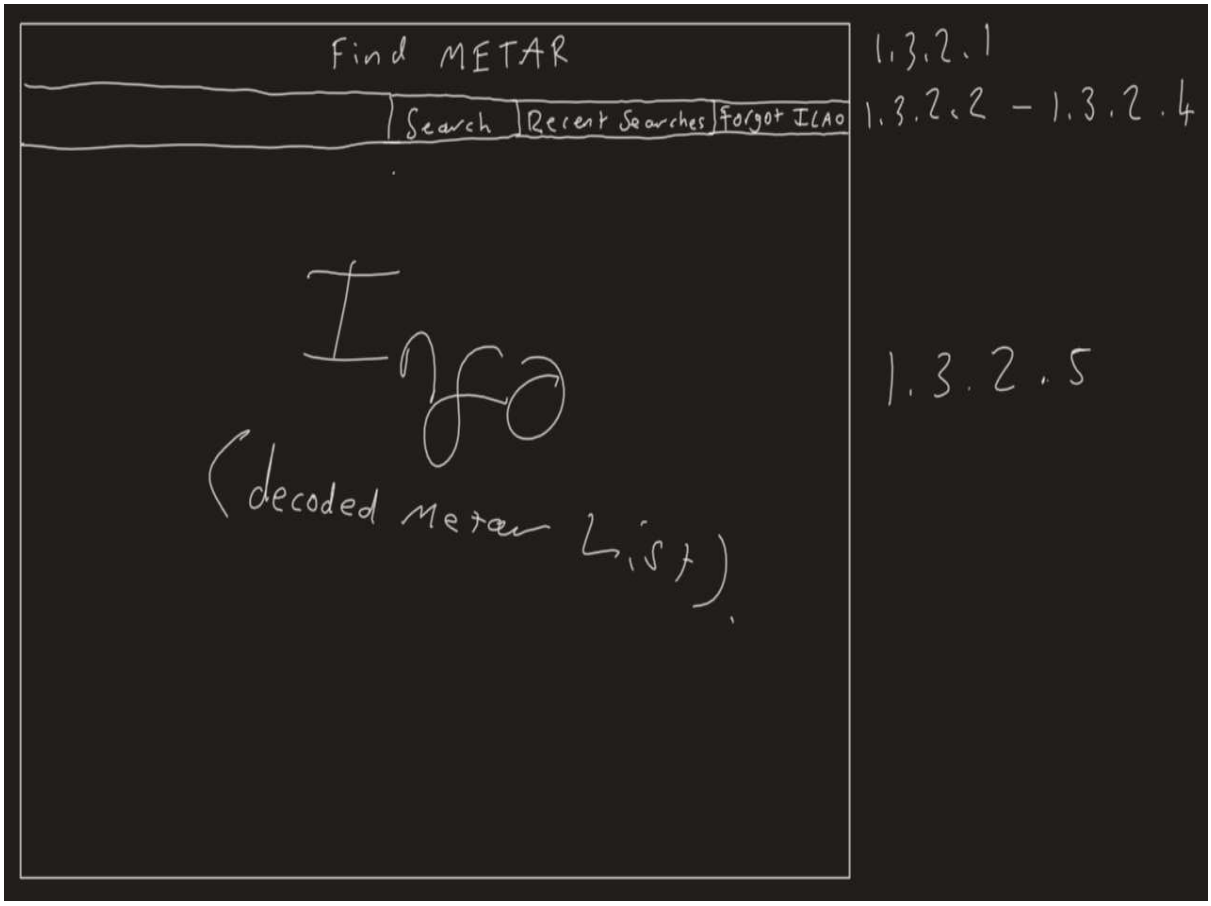
Structure Diagram



Name	Type	Explanation/Usage	Link to success criteria
search_location	procedure	This creates the template for the API request and moves to the next procedure when completed.	13, 15, 17
found_location	procedure	Gives the user the details from the request	13, 16

fill	procedure	Fills in the contents of the recent search dropdown.	14
search_input	ObjectProperty	The input from the search box used to search.	17
search_results	^	The results from the search which will be displayed to the user	16
recent_search_one, recent_search_two, recent_search_three	^	The recent searches. Used for convenience.	14
usr_details	StringProperty	The user's details	14
Search_url	String	The URL where we request our data from.	15
Data	Dictionary	The entire API request data	15,16
toAdd	List	What will be displayed to the user. By adding to the search_results ObjectProperty.	16
rs	list	The recent searches	14

Screen Design



Pseudocode

```
AddLocationForm class

Python

CLASS AddLocationForm {

    // gets the respective object properties from the kv code and sets them as variables
    search_input = ObjectProperty()
    search_results = ObjectProperty()
    recent_search_one = ObjectProperty()
    recent_search_two = ObjectProperty()
    recent_search_three = ObjectProperty()
    usr_details = StringProperty('')

    // creates a procedure called search_location
    PROCEDURE search_location(self){
        // creates a template for our api request the {} indicate where the ICAO code will be
        search_template = "https =//avwx.rest/api/metar/{}?options=summary&format=json&onfail=cache"
        // adds the ICAO code from the input box to the search template to make a usable url where we can get our request
        search_url = search_template.FORMAT(self.search_input.text)
        // this uses the kivy built in UrlRequest class to make a request and store the result in the variable.
        // it requests the data by asking for the url to request from then what method to go to once the request has come back (it sends 2 arguments)
        request = URLREQUEST(search_url, self.found_location)
    }
    ENDPROCEDURE

    PROCEDURE found_location(self, request, data){
        // sets variable data to a python decoded then json formatted version of the request data if the request data is not a python dictionary.
        data = json.LOADS(data.decode()) IF NOT ISINSTANCE(data, dict) else data
        // creates an empty list where everything to be added to the list will be stored.
        toAdd = []
        // this a list with each part of the raw METAR
```

Kivy

```
        raw = data['raw'].SPLIT()
        // this is a list with each part of the summary of the METAR (the summary is provided by the API.
        summary = data['summary'].SPLIT(',')
        // adds the list contents of summary to toAdd
        FOR i IN summary =
            toAdd.APPENND(i)
        ENDFOR
        // changes the list from the kivy code so it equals our list. This changes what is displayed.
        self.search_results.item_strings = toAdd
    }
    ENDPROCEDURE

    PROCEDURE fill(self):
        // get ready for this one it's a long boi
        // it gets the dictionary format of the user deatils which is then split and formated multiple times to get the recent searches we need.
        rs = ((((((self.usr_details).SPLIT(':'))[4].SPLIT(']'))[0].REPLACE(' ', '')).REPLACE("'", "")).REPLACE('[', '').SPLIT(','))
        // fills in the recent searches
        self.recent_search_one.text, self.recent_search_two.text, self.recent_search_three.text = rs[0], rs[1], rs[2]
}
ENDCLASS
```

```
Kivy

// creates a class
CLASS AddLocationForm {

    // sets the name so the screen manager can identify this screen
    name = "AddLocationForm"

    // sets object properties for certain widgets given their id so they can be manipulated in the python code.
    search_input = search_box
    search_results = search_results_list
    recent_search_one = rsone
    recent_search_two = rstwo
    recent_search_three = rsthree

    // sets stacking of widgets to be vertical
    orientation = "vertical"

    // Creates a box layout which is 40dp in height (size_hint_y has to be set to None for this to occur)
        height = "40dp"
        size_hint_y = None
        Label {
            text = "Enter ICAO Code"
        }
    }
    Creates a box layout which is 40dp in height. The orientation is not defaulted to horizontal.
    BoxLayout {
        height = "40dp"
        size_hint_y = None
        // An input box which is 40% of the width of the screen, has shaddow text and has no default text.
        TextInput {
            id = search_box
            size_hint_x = 40
            hint_text = "ICAO"
            text = ""
        }
        // creates a button which is 10% of the screen's width, has text on the button
        // when pressed it runs runs the backend procedure search_location().
        Button {
            text = "Search"
            size_hint_x = 15
            on_press {
                root.search_location()
            }
        }
        // A button which thakes up 20% of the screen's width
        Button {
            text = "Recent Searches"
            size_hint_x = 20
            // if the user clicks on the parent widget then the dropdown (defined below) closes
            on_parent {
                dropdown.dismiss()
            }
            on_release {
            // if the user clicks the button then the dropdown opens.
                dropdown.open(self)
            }
        }
```

```
        }
        // This is the dropdown menu mentioned before (it is a widget in itself)
        It has a id and initially handles the button text.
        Dropdown {
            id = dropdown
            on_select {
                btn.text = '{}'.format(args[1])
            }
            // There are 3 child button which have id's so the recent searhes can be set to them.
            // They have text and when clicked, they will change the text in the search box to that of the button
            Button {
                id = rsone
                text = 'First Item'
                size_hint_y = None
                height = 35
                on_release {
                    search_box.text = self.text
                }
            }

            Button {
                id = rstwo
                text = 'Second Item'
                size_hint_y = None
                height = 35
                on_release {
                    search_box.text = self.text
                }
            }

            Button {
                id = rsthree
                text = 'Third Item'
                size_hint_y = None
                height = 35
                on_release {
                    search_box.text = self.text
                }
            }
        }
```

```
        // This button takes up 20% of the screen's width and takes the user to 1.4 when clicked.
        Button {
            text = "Forgot your ICAO?"
            size_hint_x = 25
            on_release {
                app.root.current = "ICAO"
                root.manager.transition.direction = "left"
            }
        }
    }
    // This is the list where all the search reults will be put. It has an id for this reason.
    ListView {
        id = search_results_list
        item_strings = []
    }

}
```

TEST DATA

Part 5 – ICAOFinder

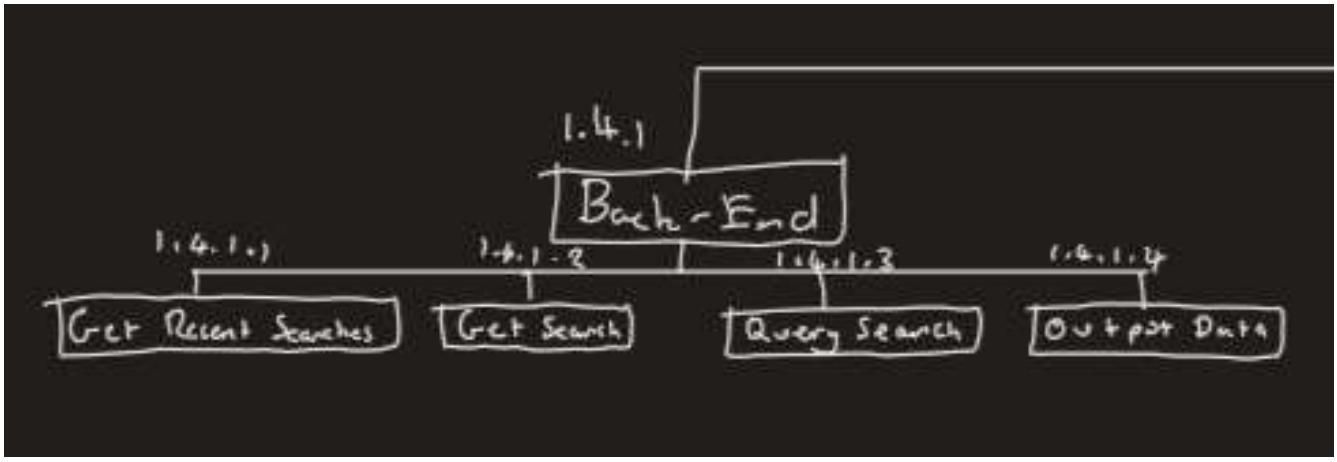
This part was added because of the demands of our clients as pilots and enthusiasts can forget their ICAO code so we thought of a system where the user could search by airport name and it would show the airports ICAO.

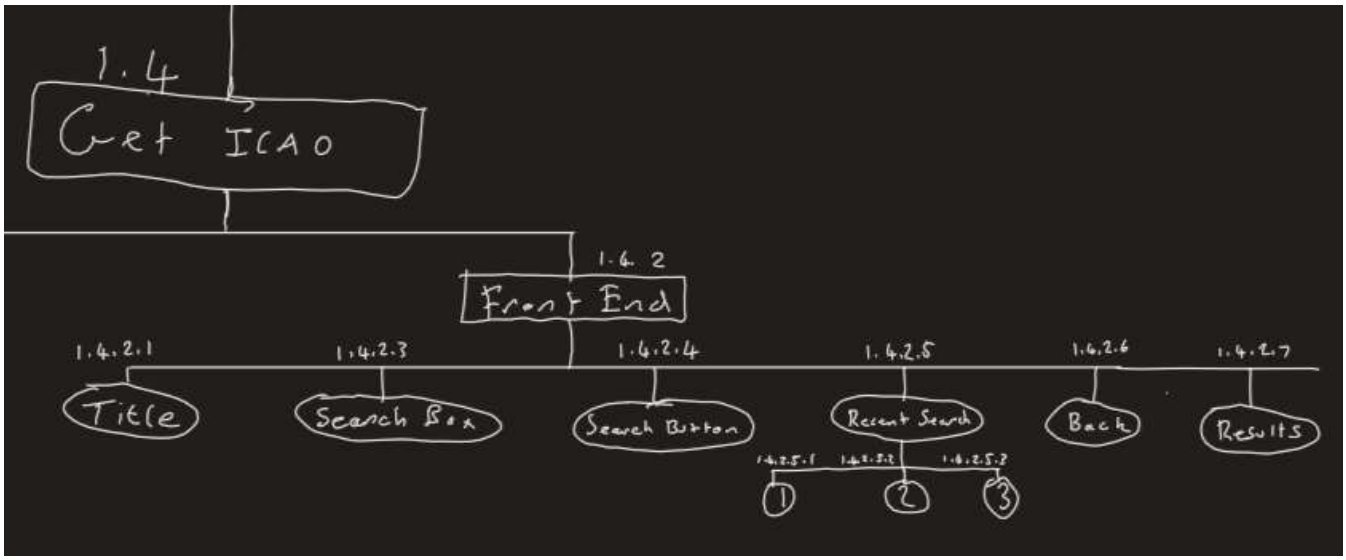
This app consists of a frontend and a backend (you've got the gist by now for sure). The frontend will consist of a title at the top which will be centred in the middle horizontally. Underneath that, there will be a horizontally orientated box layout. This will consist of:

- A search box, which will be id's so its contents can be retrieved in the backend.
- A search button, to trigger the backend code for searching.
- Back button which takes the user to 1.3.

The backend will consist of 2 procedures, the first will get the recent searches from the global variable, format it and add it into the dropdown buttons. The second shall be triggered from the search button and it will get the search, query it and update the list view based on the results. The search system would get its data from an external CSV file which will be formatted to get the results in a dictionary then it would see if the query is in the key or not. If it is then the key and value are added to a list which will then be used to update the list view list.

Structure diagram

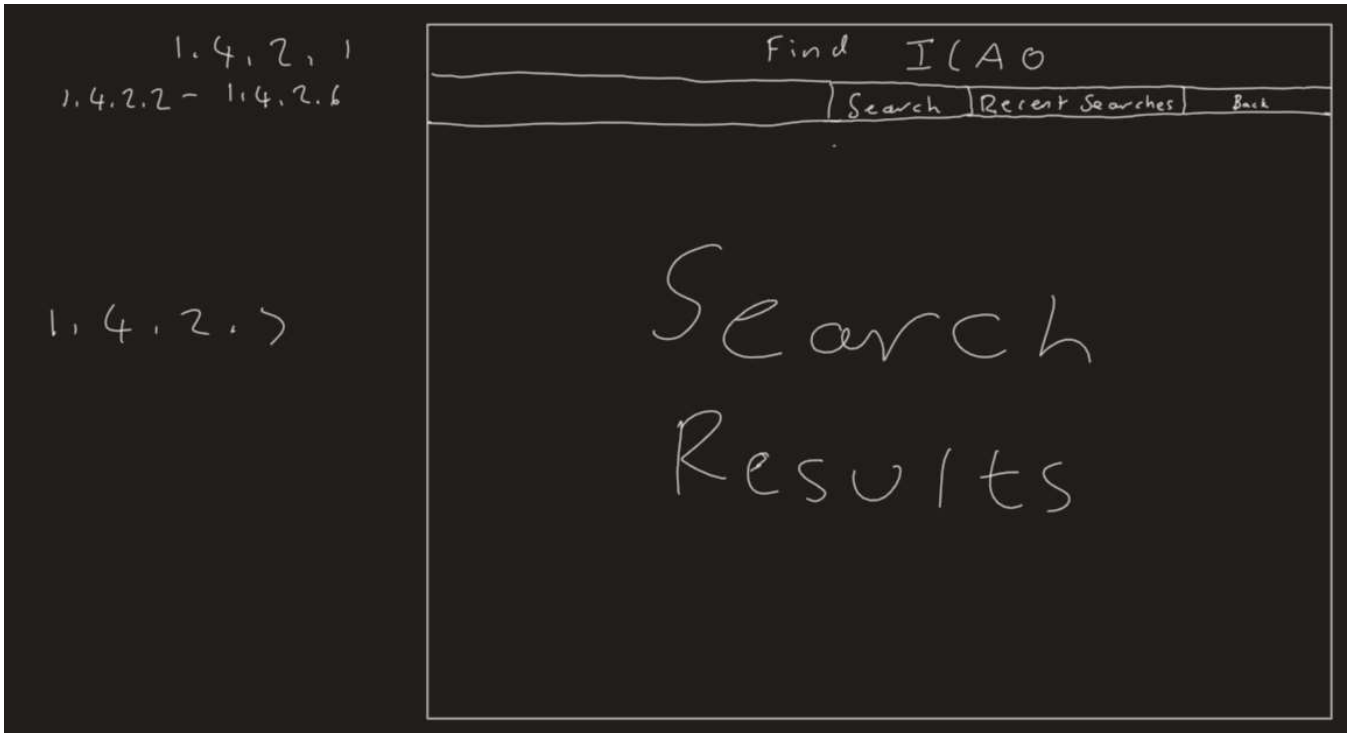




Name	Type	Explanation/Usage	Link to success criteria

findICAO	procedure	This creates the template for the API request and moves to the next procedure when completed.	13, 15, 17
search_input	ObjectProperty	The input from the search box used to search.	17
search_results	^	The results from the search which will be displayed to the user	16
Data	List	All the ICAO codes and their names	15,16
ICAOList	List	What will be displayed to the user. By adding to the search_results ObjectProperty.	16

Screen Design



Pseudocode

Python

```
ICAOFinder Class

Python

CLASS ICAOFinder(BoxLayout, Screen) {

    // gets the respective object properties from the kv code and sets them as variables
    search_input = ObjectProperty()
    search_results = ObjectProperty()

    // creates a procedure called findICAO
    PROCEDURE findICAO(self) {
        // gets the data from the csv file and sets it to f
        f = OPENREAD("Data/airports.csv")
        // creates an empty list
        data = []
        // a for loop which formats the data to make embedded lists with are then put in the 'data' list
        FOR line IN f {
            data_line = line.rstrip().split('\n')
            data_list = data_line[0].split(',')
            data.append(data_list)

        }
        // creates a list for the search results to go in
        ICAOList = ["Search results are: "]
        // a couner so we know how many search results we have got
        counter = 0
        // a for loop which searches the data for the search query, formats the results and then adds them to the results list.
        // this automatically updates the screen with the updated list each loop.
        FOR lst IN data {
            IF self.search_input.text IN lst[1] {
                counter +=1
                ICAOList.APPEND("{} ICAO is {}".FORMAT(lst[1, lst[0]))

            }
            ICAOList.INSERT(0, "Total number of results: {}".FORMAT(counter))
            self.search_results.item_strings = ICAOList
        }
    }
    ENDPROCEDURE

}
ENDCLASS
```

Kivy

```
Kivy

CLASS ICAOFinder {
    name: "ICAO"
    search_input: search_box
    search_results: search_results_list_ICAO
    recent_search_one: rsone
    recent_search_two: rstwo
    recent_search_three: rsthree
    orientation: "vertical"
    BoxLayout {
        height: "40dp"
        size_hint_y: None
        Label (
            text: "Enter Name of airport"
        }
    }
    BoxLayout {
        height: "40dp"
        size_hint_y: None
        TextInput {
            id: search_box
            size_hint_x: 40
        }
```

```
68              on_press {
69                  root.findICAO()
70              }
71          }
72          Button {
73              text: "Recent Searches"
74              size_hint_x: 20
75              on_parent (
76                  dropdown.dismiss()
77              }
78              on_release {
79                  dropdown.open(self)
80              }
81          }
82          Button {
83              size_hint_x: 10
84              text: "Clear"
85              on_release {
86                  search_results_list_ICAO.item_strings = []
87              }
88          }
89          Button {
90              text: "Back"
91              size_hint_x: 10
92              on_release {
93                  app.root.current = "AddLocation"
94                  root.manager.transition.direction = "right"
95              }
96          }
```

```
97      ListView {
98          id: search_results_list_ICAO
99          item_strings: []
100     }
101 }
102 ENDCLASS
103
```

Test Data

Post-Test Data

Part 6 – Meeting with shareholders

I decided to meet with them separately, showed them this document and asked for their opinion and any questions they may have.

Joseph

Geoffrey

Section 3 - Development

Introduction

I will be using an agile development method based on versioning the app with iterations of the development cycle. For each iteration, there will be:

- Why is this version being coded?

- Requirments

- Pseudocode

- Test data

- Actual programming (The commented code will be available in the appendix)

- Testing

- Any changes needed

- Final Report

Version 1.0

For version 1.0, everything up to actual programming is already in the report.

Day 1 – Making the base appstate

```
WeatherRoot:
<WeatherRoot>:
    id: screen_manager
```

The above sets up the screen manager for all our screens.

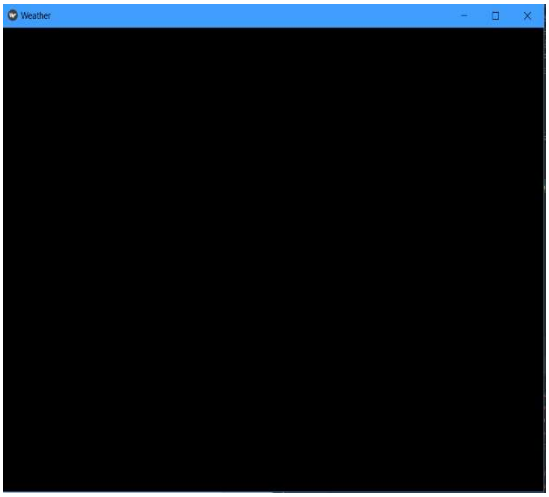
The classes have pass in them as we have not yet added any backend code into them. The classes also inherit from our different imports where needed for example, WeatherRoot inherits ScreenManager as it will be where the screen manager is so we can switch between different screens when needed.

```
from kivy.app import App
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.dropdown import DropDown
from kivy.properties import ObjectProperty
from kivy.network.urlrequest import UrlRequest
from kivy.uix.screenmanager import ScreenManager, Screen
from kivy.properties import StringProperty
import json
import hashlib
import os

class WeatherRoot(ScreenManager, BoxLayout):
    pass


class RegisterPage(Screen, BoxLayout):
    pass


class WeatherApp(App):
    pass


if __name__ == "__main__":
    WeatherApp().run()
```

Here I have made the base app as well as imported what I'll be needing for the app (can be seen in initial imports part).

It seems to pass our test so we will move o to our next part.

I have also set up the screen manager in the Kivy code for future use.

Test number	What are we testing for	Expected result	Test data	Success/failure + Proof	Notes
1	For a black window with Weather as the title		none	Success 	Continue onto the next part

Day 1, 2 and 3 – RegisterPage Class

Day 1, making the frontend

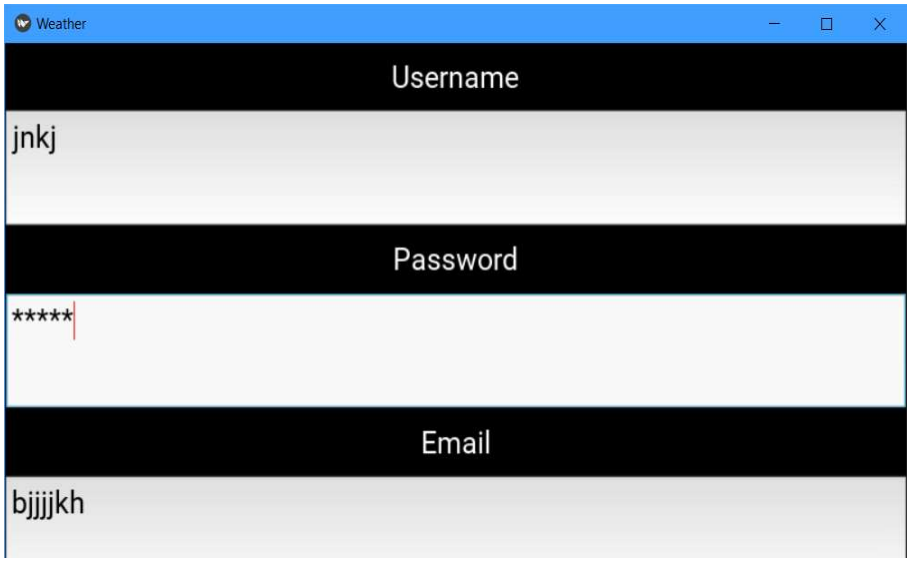
The rest of day 1 consisted of preparing how the class would look to the user. We start with the buttons and labels.

```
BoxLayout:
    orientation: "vertical"

    Label:
        text: "Username"
        font_size: 25
    TextInput:
        id: username
        font_size: 25
```

```
<RegisterPage>:
    name: "Register"
    username_input: username
    password_input: password
    email_input: email
    validation: confo
```

I have created a vertically aligned box layout with 3 labels and 3 input boxes. The password input box has the password formatting added in. I have chosen a font size of 25 for ow to make the text clear and readable.

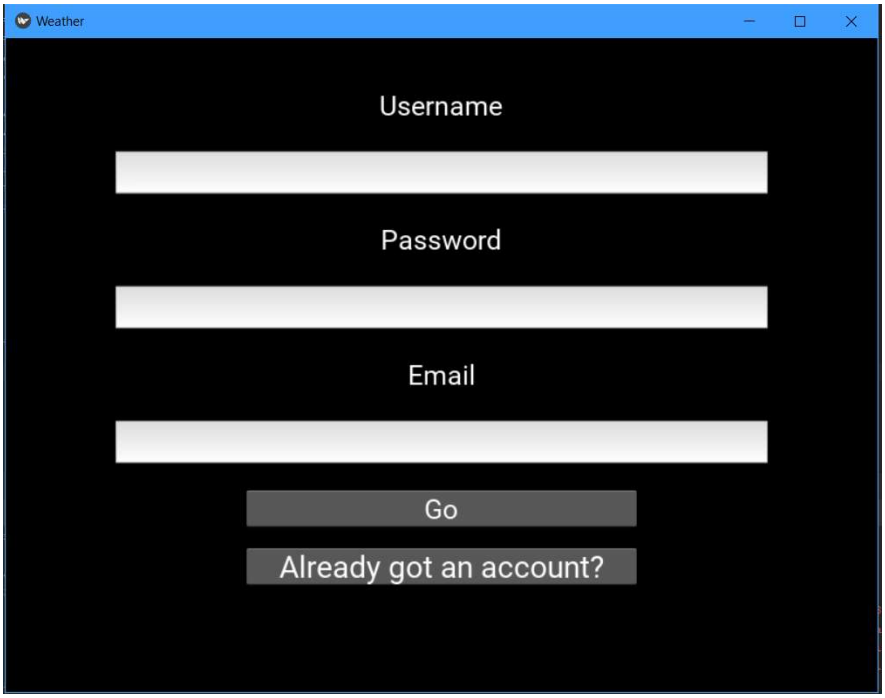
I have also prepared the object properties and the name for switching screens.




```
WeatherRoot:
<WeatherRoot>:
    id: screen_manager
    RegisterPage:
```

We ensure that this class is a child of the root/screen manager. This is so we can move between screens/classes in our app.

Next, I added padding and spacing to make sure the page looks presentable.

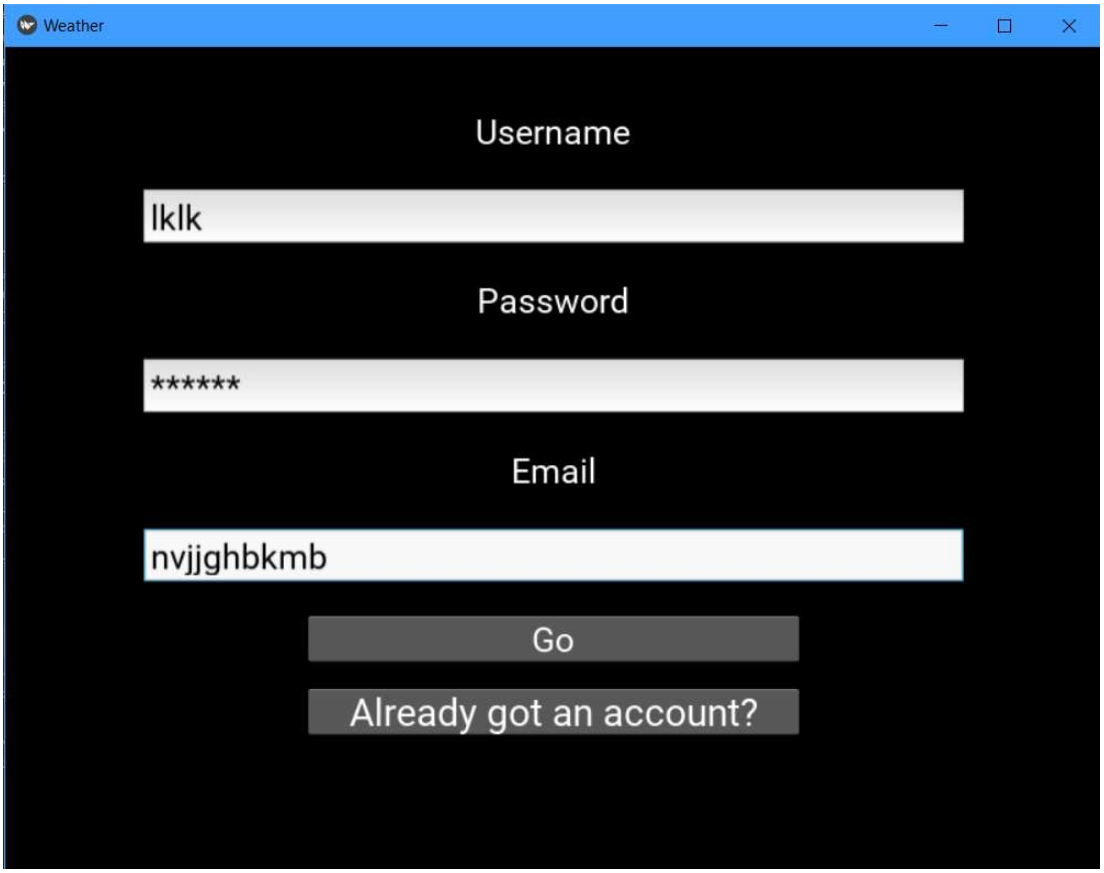



```
padding: [100, 50, 100, 50]
spacing: 30
center_x: True
```

I have also placed the buttons in to make sure that the screen can handle all of the widgets on an already padded screen. The buttons were put inside anchor layouts and has the size hint changed to 60% width and 150% height.

```
AnchorLayout:
    Button:
        text: "Go"
        font_size: 25
        size_hint_x: 0.6
        size_hint_y: 1.5
        halign: 'center'
```

2	Check usage of the text boxes and to make sure nothing is missing	When input text into the boxes, it should be presented clearly and be correctly sized.	Just some random strings	Success (see screenshot)	none
3	To see if the password formatting works	Whatever text we put in the password box; it should be replaced with asterisks.	Random string	Success (see screenshot)	none



As you can see, the password input box is formatting the input with asterisks and the other inputs are clearly readable.

Day 2 – making the backend.

To begin, I had to decide between using a JSON file or DB to store the data. After some contemplation, it was decided that a JSON file should be used now and a DB would be used depending on the success of the JSON file. So, following the pseudocode written, I began to write the code.

```
username_input = ObjectProperty()
password_input = ObjectProperty()
email_input = ObjectProperty()
validation = ObjectProperty()
```

Our first job was to get the Object properties from the Kivy code with the help of the Kivy.properties.ObjectProperty class. This is so we can access the input box text and the confirmation texts.

Next, I decided to write the code for validating the register form inputs. This first included getting then formatting the data in the JSON file so we can iterate over it. The file has been closed sooner than later to save system resources.

```
def register_validation(self):
    f = open("Data/data.json", "r")
    data = json.load(f)
    idLst = []
    f.close()
    for users in data['users']:
        idLst.append(users)
```

```
for id in idLst:
    if (self.username_input.text or self.password_input.text or self.email_input.text) == "":
        self.validation.text = "Form not completed"
    elif (self.username_input.text == data['users'][id]['username']) or (self.email_input.text == data['users'][id]['email']):
        self.validation.text = "Username or Email is taken"
    else:
        self.register(idLst, data)
```

Here we validate that the user has completed the form as well as iterating over the data to check if the user already exists in the JSON file. If they are then the confirmation text should change based on this.

```
def register(self, idLst, data):
    id = str(int(idLst[-1]) + 1)
```

A new id is created for the new user and hash for their password is made using md5 encryption and salt to protect against hackers and the salt to protect against rainbow tables.

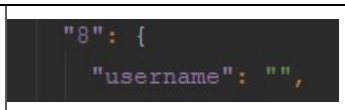
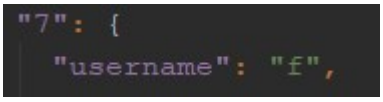
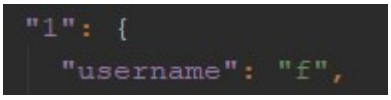
```
salt = "yousaltybro"
passHash = hashlib.md5((salt + self.password_input.text).encode("utf-8")).hexdigest()
```

```
data['users'][id] = {"username": self.username_input.text, "password_hash": passHash,
                     "email": self.email_input.text, "recent_searches_METAR": [], "recent_searches_ICAO": []}
with open("Data/temp.json", "w", encoding="utf-8") as f:
    json.dump(data, f, indent=2)
f.close()
os.remove("Data/data.json")
os.rename("Data/temp.json", "Data/data.json")
```

Lastly, the program updates the extracted data with the new user's data (id, username, password, email, etc.), in the correct formatting. Then the json file will be closed, then this data will be dumped to a new temporary file, the old one deleted and the temporary file renamed to the old one's name using the os module.

Finally, to conclude the day, I decided to run some tests. The results can be seen below.

Test number	What are we testing for	Expected result	Test data	Results	Notes
4	Test validation process, no username	Expect to see "Form not completed" show up in the confirmation text area	Random strings entered for password and email, nothing in the	Failure, even though it says form not complete: Form not completed The registration process still continues.	We will need an overhaul of the validation system. See below for how I

		and for no registration to continue.	username input box.		fixed it on day 3.
5	Test validation process, no password	^	Random strings entered for username and email, noting in the password input box.	^	^
6	Test validation process, no email	^	Random strings entered for password and username, noting in the email input box.	^	^
7	Test validation, already existing user	"Username or email is already taken" warning message.	Username = "f", password and email have random strings	Failure, it just sends the data for registration anyway, confirmation text says "Form not completed".   	Need to redo on day 3.
8	Testing validation, existing email	^	Email is "h@h.h", password and username are random strings.	^	^
9	Test with legitimate data which	It should register the user and there should	Username = "re"	Failure, it does send the data off perfectly but the confirmation text still says that the form is not complete.	^

	hasn't been used	be a confirmation message for the user.	Password = random string Email = "re@re.re"		

Day 3 – Fixing the validation system

To start off, I tried to diagnose where the problem was coming from so I had to add some print statement's in the code to see how it is running in the terminal. This didn't work as all I could get to print out were the inputs and existing data so I decided to look at the code again.

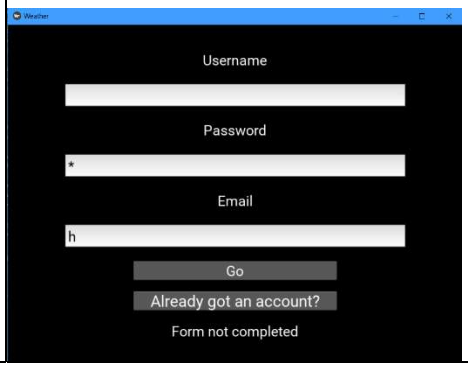
Here I realised that the bit which checks if the form is full is stuck in a loop even though there is no need for this. There is also no way to break out of the procedure in case of an error. So, I devised a plan where there could be a Boolean which is set to false, whenever there is an error, it is set to true. At the end of the procedure, if the Boolean is false then it will go to the register procedure, otherwise, the backend will stop.

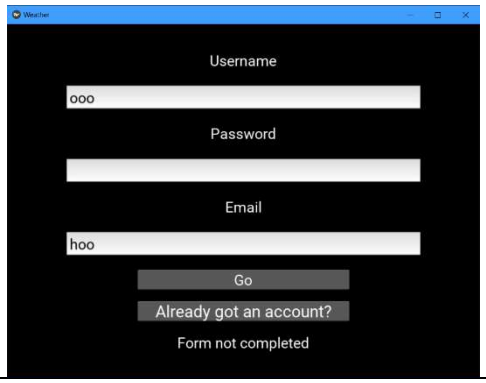
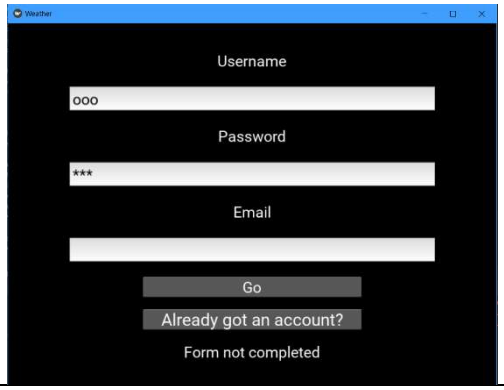
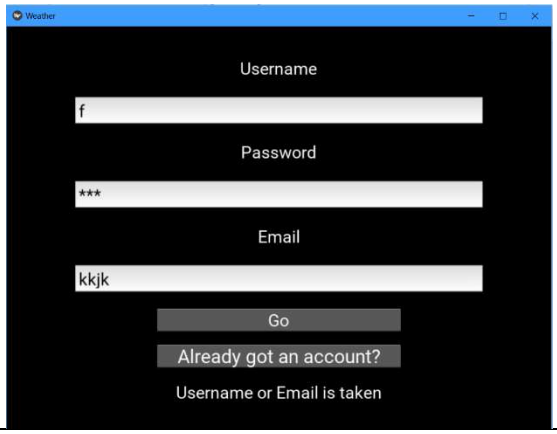
```
data = json.load(f)
err = False
idLst = []
```

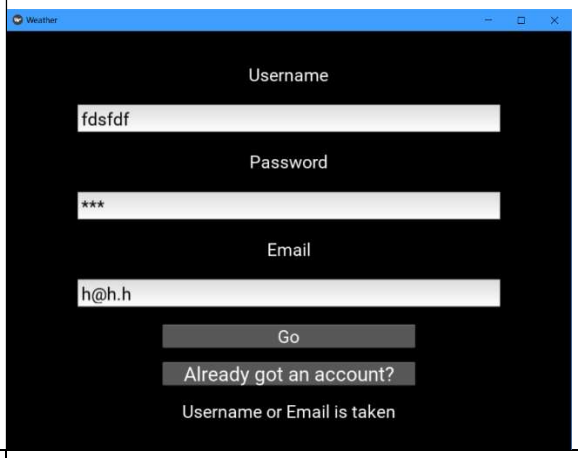
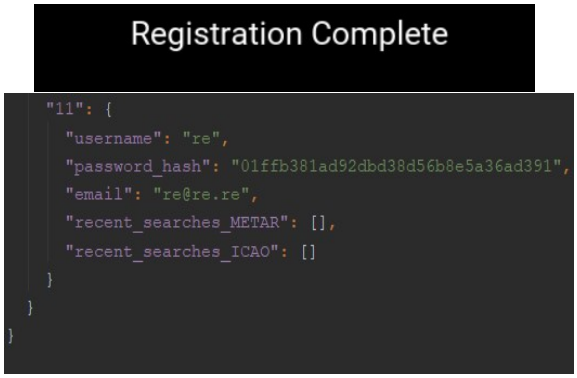
For the code, I have called the Boolean err.

err is changed to false when a validation error occurs and is validated at the end asking if it is false.

```
if (not self.username_input.text) or (not self.password_input.text) or (not self.email_input.text):
    self.validation.text = "Form not completed"
    err = True
for id in idLst:
    if (self.username_input.text == data['users'][id]['username']) or (self.email_input.text == data['users'][id]['email']):
        self.validation.text = "Username or Email is taken"
        err = True
if err == False:
    self.register(idLst, data)
```

10	Test validation, no username	Form not completed message and no updated JSON file	Random strings entered for password and email, nothing in the username input box.	Success 	

11	Test validation, no password	^	Random strings entered for username and email, nothing in the password input box.	Success 	
12	Test validation, no email	Username or email is taken message and no updated JSON file.	Random strings entered for password and username, noting in the email input box.	Success 	
13	Test validation, already existing user	^	Username = “f”, password and email have random strings	Success 	

14	Test validation, already existing email	^	Email is "h@h.h", password and username are random strings.	Success 	
15	Test with legitimate data.	Should confirm that registration is complete	Username = "re" Password = random string Email = "re@re.re"	Success 	

Checking with our criteria

Criteria	How to check	Completed with tests successful (Y/N)	Notes
A register system	We are hoping for a working register system that can add new users and give error messages based on inputs.	Y	
Appropriate validation of the inputs	Test it with all sorts of inputs e.g. missing inputs, existing users, etc.	Y	
The data to be kept secure e.g.: hashing passwords with salt.	The passwords should be hashed with salt where the data is stored.	Y	
The user's data is set up in the correct format (TBD)	Check where the data is stored that new user's data is in the correct format	Y	
Error messages if unsuccessful	Check by putting in data that should	Y	

	return an error message and check if the error message has appeared.		
Success message	^ but with data that would return a success message.	Y	

Now onto the next class.

Day 4, 5 and 6 LoginPage Class

Day 4 - Making the frontend

After the relief from yesterday's work, I set out on making the frontend for 1.2.

```
<LoginPage>:
    name: "LoginPage"
    username_input: username
    password_input: password
    validation: confo
    BoxLayout:
        orientation: "vertical"
        padding: [100, 50, 100, 50]
        spacing: 30
        center_x: True
```

I started with the basic design for the boxlayout with the padding which had worked with 1.1

Of course, this would not display anything just yet, but it would help with structuring our widgets in a reasonable manner on the screen.

We also initialise the class in the python script (otherwise, the root would raise an error). It inherits from boxlayout so we can use it in the backend and Screen so we can connect it to the screen manager system

```
class LoginPage(BoxLayout, Screen):
```

With the login page, I decided that I wanted to add a bit of colour to the font so I researched how the colour system works and got this.

color

Text color, in the format (r, g, b, a).

color is a ListProperty and defaults to [1, 1, 1, 1].

Colour in Kivy, is between 0 and 1 (with allowance for 1 decimal place) for all the rgba values so the default (1,1,1,1) is black with no transparency. As I wanted to add my favourite colour for text on a black background, blanched almond, I had to think of a way to turn the 0 – 255 system into the Kivy system.

Blanched almond in the 0-255 system is

RGB Decimal	255, 235, 205

So how about we divide these values by 255 and round to 1 decimal place. Then we get:

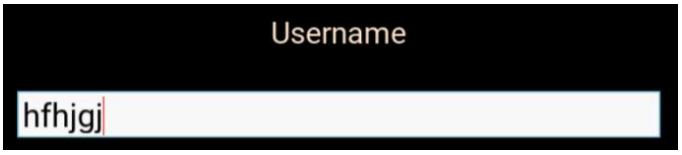
1, 0.9, 0.8, 1 (1 at the end as we want no transparency).

Let's try it out then:

```
Label:
    text: "Username"
    font_size: 28
    color: 1, 0.9, 0.8, 1
```

Well that went well, how about we try with a text input

```
TextInput:
    id: username
    font_size: 30
    color: 1, 0.9, 0.8, 1
```



I guess it didn't work. How about we try colouring a button's text?

Before we do that, I think we should add our other input and label in – for the password.

```
Label:
    text: "Password"
    font_size: 28
    color: 1, 0.9, 0.8, 1
TextInput:
    id: password
    password: True
    multiline: False
    font_size: 30
    color: 1, 0.9, 0.8, 1
```

Now that that works, we can go back to experimenting with the colouring as we will add the submit button next.

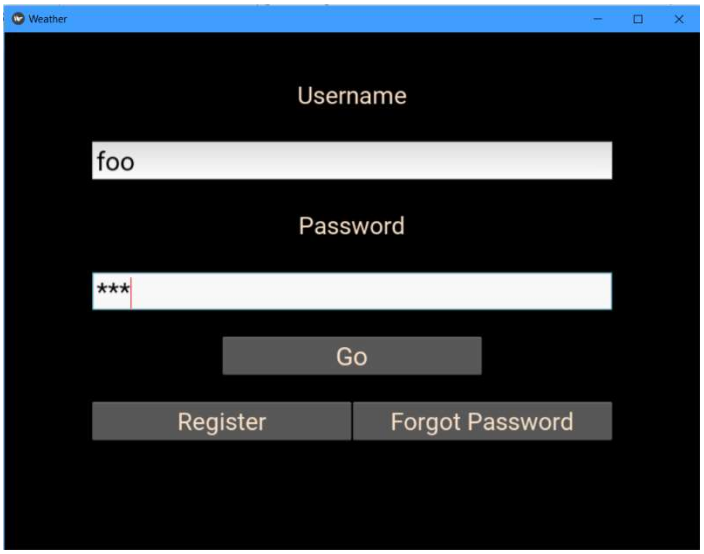
```
AnchorLayout:
    Button:
        text: "Go"
        font_size: 30
        size_hint_x: 0.5
        halign: 'center'
        color: 1, 0.9, 0.8, 1
        on_release:
            root.validate()
```

Great. Now we might as well finish off the frontend with the last 2 buttons in a box layout and the confirmation text.

```
BoxLayout:
    Button:
        text: "Register"
        size_hint_x: 0.5
        font_size: 28
        color: 1, 0.9, 0.8, 1
        on_release:
            app.root.current = "Register"
            root.manager.transition.direction = "right"
    Button:
        text: "Forgot Password"
        size_hint_x: 0.5
        font_size: 28
        color: 1, 0.9, 0.8, 1
Label:
    id: confo
    text: ""
    font_size: 25
    color: 1, 0.9, 0.8, 1
```

Now that those work, we can run our tests just for procedure's sake

Test number	What are we testing for	Expected result	Test data	Results	Notes
16	Check usage of the text boxes and to make sure nothing is missing	When input text into the boxes, it should be presented clearly and be correctly sized.	Just some random strings	Success (see screenshot)	none
17	To see if the password formatting works	Whatever text we put in the password box; it should be replaced with asterisks.	Random string	Success (see screenshot)	none



Now that that's that, let's move on to the backend of 1.2. I'm sure you realise why blanched almond was a great choice now.

```
WeatherRoot:
<WeatherRoot>:
    id: screen_manager
    RegisterPage:
    LoginPage:
        id: login_page
        name: "LoginPage"
        manager: screen_manager
```

We make this class a child of the root/screen manager, we identify this clearly in the last line.

Day 5 and 6 – The Backend

Now it's time for the backend code. Hopefully, I don't have any issues this time.

```
class LoginPage(BoxLayout, Screen):
    usr_info = StringProperty('')
    username_input = ObjectProperty()
    password_input = ObjectProperty()
    validation = ObjectProperty()
```

We begin by bringing the object properties (well more like establishing a link between the 2) from the frontend and saving them as variables.

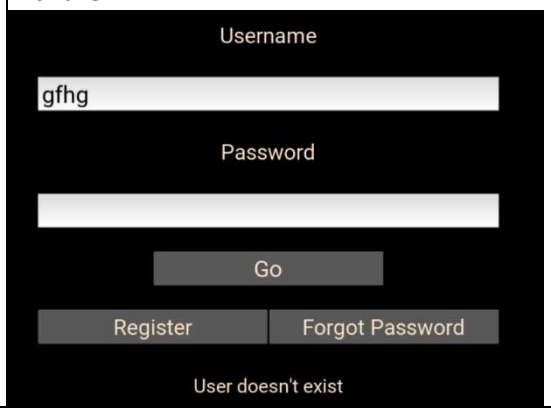
Now we should begin with the validation procedure. For this, I loaded up the JSON file, formatted it and set the user's data and usernames as separate lists. This is so we can check if their username exists and we keep the user's data to confirm their password in the next procedure.

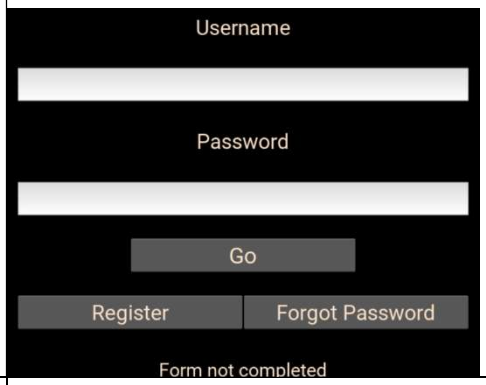
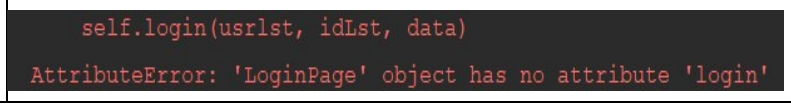
```
def validate(self):
    f = open("Data/data.json", "r", encoding="utf-8")
    data = json.load(f)
    idLst, usrlst = [], []
    for id in data['users']:
        idLst.append(id)
        usrlst.append(data['users'][id]['username'])
```

```
if (self.username_input.text or self.password_input.text) == "":
    self.validation.text = "Form not completed"
elif (self.username_input.text not in usrlst):
    self.validation.text = "User doesn't exist"
else:
    self.login(usrlst, idLst, data)
```

This validates for a completed form and if the user exists. If it fails then an error message is set as the validation text. If it passes then it moves to the login procedure with the data and the lists as arguments.

Testing:

Test number	What are we testing for	Expected result	Test data	Results	Notes
18	Checking if the validation which checks if the form is filled woks. (no username)	The validation text should change to “Form not completed”	Nothing as username as a random string as password	Failure 	This will be fixed after the tests are done. Also, the validation text looks good in blanched almond, right?
19	^ but with no password	^	Nothing as password but some random string as username.	Failure 	^

Test number	What are we testing for	Expected result	Test data	Results	Notes
20	^ but with no username or password	Same as 19 and 20	Nothing for either username or password	Success 	none
21	Test if the validation to check if the	A surprise	A user that exists for the	Success 	Add a login method pls

	user exists		username e.g. f and some random string for the password	Well we were expecting that error, right? As we haven't created the login method yet.	

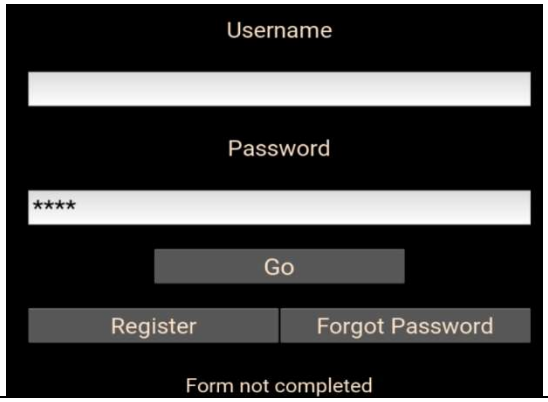
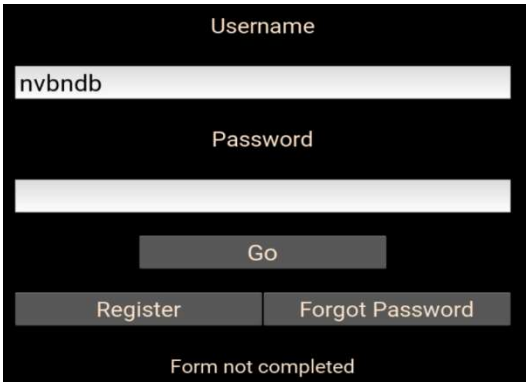
To end day 5, let's try to fix the errors.

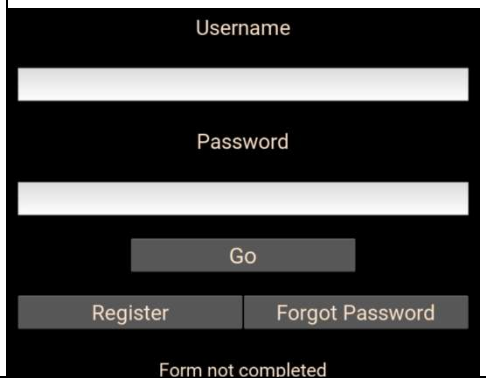
After a quick consolidation session with Quackius V of Mancunium (my rubber duck), we found out that I was dumb and it was quite an easy fix. We replace this:

```
elif (self.username_input.text not in usrlst):
```

With this:

```
if (self.username_input.text == "") or (self.password_input.text == ""):
```

Test number	What are we testing for	Expected result	Test data	Results	Notes
22	Checking if the validation which checks if the form is filled woks. (no username)	The validation text should change to "Form not completed"	Nothing as username as a random string as password	Success 	
23	^ but with no password	^	Nothing as password but some random string as username.	Success 	^

24	^ but with no username or password	Same as 22 and 23	Nothing for either username or password	Success 	none

That took up all of day 5.

Day 6

```
def login(self, usrLst, idLst, data):
    users = dict(zip(usrLst, idLst))
    usrinfo = data['users'][users[self.username_input.text]]
    self.usr_info = str(usrinfo)
```

We begin the next procedure creating a dictionary from the 2 lists carried along so it is easier to find the user's info. Then we load up the specific user's data with the help of dictionary querying and the newly created dictionary. Finally, we set this as our global variable so we can use it later.

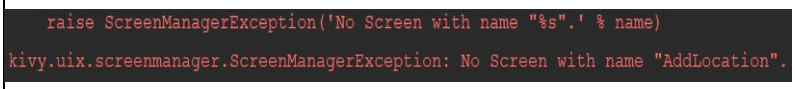
```
salt = "yousaltybro"
passHash = hashlib.md5((salt + self.password_input.text).encode("utf-8")).hexdigest()
```

Here we set a salt and we create out hashed password using our salt (and encoding it for integrity).

```
if usrinfo['password_hash'] == passHash:
    self.manager.current = 'AddLocation'
else:
    self.validation.text = "Password is wrong"
```

Finally, we check if the hashed input password = the one stored in the database. If it matches then the user is moved to the main class (which is the metar searcher – poorly named as 'AddLocation' using Kivy' s screen manager. If it fails then an error message is shown to the user via the validation text.

Time for some testing

Test number	What are we testing for	Expected result	Test data	Results	Notes
18	Checking if the validation works	The validation text should change to “Password is wrong”	Nothing as username and a random string as the password	Success 	The password is actually f
19	^	There should be an error message as we have not created the AddLocation class yet.	Nothing as password but some random string as username.	Success 	Gr8 success, we can move on to the next class now.

Checking with our criteria

Criteria	How to check	Successful with testing done (Y/N)	Notes
A login system	If there is a working login system with good validation which takes the user to the main screen when successful.	Y	
Appropriate validation of the inputs	Fire in some inputs which should not make the user log in e.g. no password filled or the username not existing	Y	
Matching the hashes	We need to check that when we input the correct password, it should log in which shows that the hashes have been matched.	Y	

Error messages if unsuccessful	Fire in some bad data, if we get an error message then we're cool.	Y	
On success, go to the main screen (TBD)	Fire in some correct data (can be done at same time as matching the hashes) if the main screen loads then we're Gucci	Y	
On success also save their data as a global variable until the program shuts down.	This we can only check once we have linked this data to something else e.g. a recent search system if they load properly then it works.	Y	

Now onto 1.3

Day 7, 8 and 9: AddLocationForm class

Day 7 – The Frontend

```
WeatherRoot:
<WeatherRoot>:
    id: screen_manager
    RegisterPage:
    LoginPage:
        id: login_page
        name: "LoginPage"
        manager: screen_manager
    AddLocationForm:
        id: add_location_form
        manager: screen_manager
        usr_details: login_page.usr_info
```

← Firstly, we make our class a child of the root/ screen manager so we can move along classes smoothly. We also make sure that we can access the user info by setting it as an object property of our new class.

Next, we set up the class itself and its object properties and its root settings →

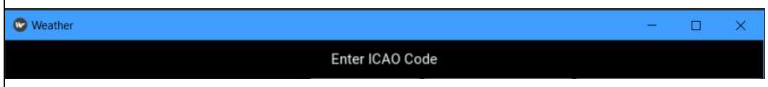
```
<AddLocationForm>:
    name: "AddLocation"

    search_input: search_box
    search_results: search_results_list
    recent_search_one: rsone
    recent_search_two: rstwo
    recent_search_three: rsthree
    orientation: "vertical"
```

Then we set up the class in the python code remembering to inherit from Boxlayout and Screen.

```
class AddLocationForm(BoxLayout, Screen):
```

```
BoxLayout:
    height: "40dp"
    size_hint_y: None
    Label:
        text: "Enter ICAO Code"
```

We set up the title text (following our pseudocode properties of the widget) and run a test to make sure everything is working

Test number	What are we testing for	Expected result	Test data	Results	Notes
20	If we can access the class directly after login	The class should load and we should be able to see the title text.	Just putting in an existing user's info e.g. Username = h, Password = h.	Success 	
21	If the title text appears and is the correct size	The title text should appear at the top and be of font size 40	^	Success (see above)	Surprisingly, the text auto-aligns to the centre. Guess that's less work for the next versions.

Here, I create a new box layout for the rest of the widgets (except the ListView) which conveniently auto aligns the widgets horizontally so we can use size_hint_x to specify the width of each of these widgets (as long as they all add up to 100).

The id is added to the input box so we can capture the search. We link the button to our currently non-existent backend code.

The search box has to be larger than the button as it needs more space for inputs and naturally is larger in most designs.

```
BoxLayout:
    height: "40dp"
    size_hint_y: None
    TextInput:
        id: search_box
        size_hint_x: 40
        hint_text: "ICAO"
        text: ""
    Button:
        text: "Search"
        size_hint_x: 15
        on_press:
            root.search_location()
```

On to the recent search system. First, we create the button which is set to fill the dropdown buttons then to open the dropdown when clicked. These dropdown buttons can then be clicked to make the search box have the same value.

Next, we create the buttons and id them so we can access the value.

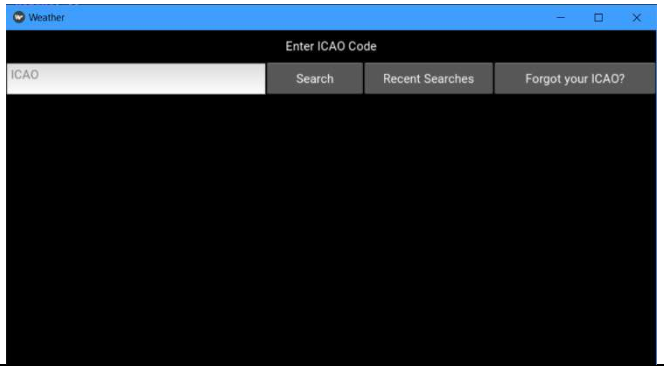
Finally, we see them up and add the on_release code.

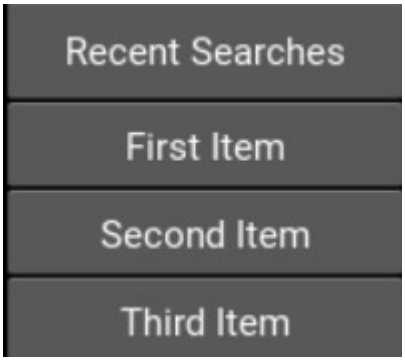
It is basically the same thing for all 3 buttons apart from differences in id name and text.

```
Button:
    text: "Recent Searches"
    size_hint_x: 20
    on_parent: dropdown.dismiss()
    on_release:
        root.fill()
        dropdown.open(self)
DropDown:
    id: dropdown
    on_select: btn.text = '{}'.format(args[1])

    Button:
        id: rsone
        text: "First Item"
        size_hint_y: None
        height: 35
        on_release: search_box.text = self.text

    Button:
        id: rstwo
        text: 'Second Item'
        size_hint_y: None
        height: 35
        on_release: search_box.text = self.text

    Button:
        id: rsthree
        text: 'Third Item'
        size_hint_y: None
        height: 35
        on_release: search_box.text = self.text
```

```
    Button:
        text: "Forgot your ICAO?"
        size_hint_x: 25
        on_release:
            app.root.current = "ICAO"
            root.manager.transition.direction = "left"

ListView:
    id: search_results_list
    item_strings: []
```

(Above left), we finish off the backend with the button which links to the ICAOFinder class and the list view for the search results. Now for some testing.

Test number	What are we testing for	Expected result	Test data	Results	Notes
22	If the buttons load up	The widgets should be loaded horizontally, with correct sizing	none	Success 	

23	If the dropdown works	It should open up, revealing 3 buttons that should be stacked vertically underneath the dropdown button.	none	Success 	

We will test the buttons in conjunction with the backend code later.

Day 8 – The Backend

```
class AddLocationForm(BoxLayout, Screen):
    search_input = ObjectProperty()
    search_results = ObjectProperty()
    recent_search_one = ObjectProperty()
    recent_search_two = ObjectProperty()
    recent_search_three = ObjectProperty()
    usr_details = ObjectProperty()
```

We start the day by initialising the object properties (usr_details is the one which we are porting from the login page).

I have changed the usr_details call to an ObjectProperty instead of a StringProperty here as it makes no difference as it calls in the same data and fits in with the other calls.

Next, we set up the search location class which prepares the API request and sends it off. When it returns successfully it should move onto our next class.

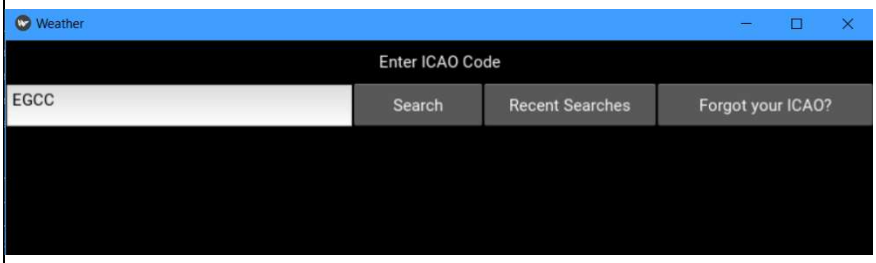
```
def search_location(self):
    search_template = "https://avwx.rest/api/metar/{}?options=summary&format=json&onfail=cache"
    search_url = search_template.format(self.search_input.text)
    request = UrlRequest(search_url, self.found_location)
```

```
def found_location(self, request, data):
    print("jsdfhk")
```

For now, I'll have a little message show in the console to confirm that the request has been completed successfully.

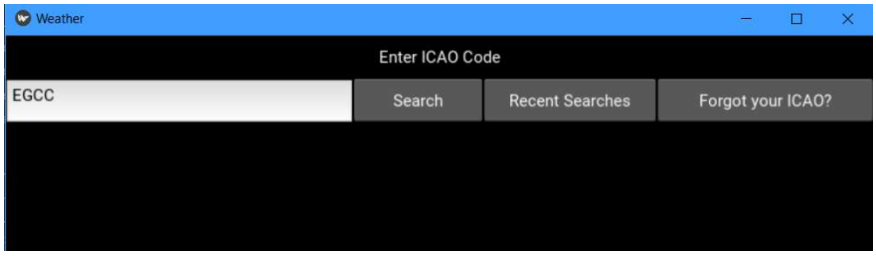
Now to test it

Test number	What are we testing for	Expected result	Test data	Results	Notes

24	Check if we can access the API successfully.	We should get a message on the console.	EGLL as the ICAO	Failure  There seems to be no message in the console after waiting for quite a while… 	Guess I'll have to diagnose the bug.

<u>Finding the problem</u>

Firstly, I should reset my WIFI and check if it works



Guess not…

Let's check the docs again and see if there is anything that can help us there.

The syntax to create a request:

```
from kivy.network.urlrequest import UrlRequest
req = UrlRequest(url, on_success, on_redirect, on_failure, on_error,
                 on_progress, req_body, req_headers, chunk_size,
                 timeout, method, decode, debug, file_path, ca_file,
                 verify)
```

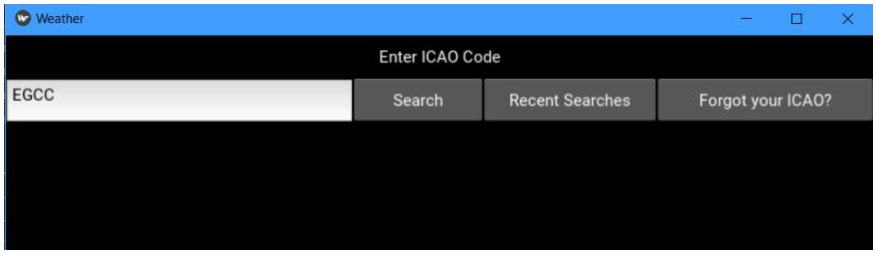
It seems like the ide may have not understood the parameters properly so how about we add the URL and on_success parameters in? So, it changes from this:

```
request = UrlRequest(search_url, self.found_location)
```

To this:

```
request = UrlRequest(url=search_url, on_success=self.found_location)
```

It seems not to fix the error:



I guess we can try the on_error parameter. We'll make it print something if there is an error. So, we should change it from:

```
request = UrlRequest(url=search_url, on_success=self.found_location)
```

To:

```
request = UrlRequest(url=search_url, on_success=self.found_location, on_error=print("error"))
```

`error` So, it seems that there is an error. We can now try to see what the error is. After googling how to print the error, there was a stack overflow answer which says to add the on_failure tag and make this and the on_error print the error by changing it to:

```
request = UrlRequest(url=search_url, on_success=self.found_location, on_error=print, on_failure=print)
```

At last, we got an error to deal with at least.

```
'meta': {'validation_error': 'You are missing the "Authorization" header or "token" parameter.
```

Authorisation error's usually come in API requests where you need an API key. When I checked the website for the pseudocode and research, it said that no API key was needed. Perhaps, it has been updated.

After looking at the docs, it seems like the API now needs authentication via an API token (aka API key)

Authentication

AVWX uses bearer tokens for user authentication. Create an account at account.avwx.rest to generate your token and include it in your requests.

The API looks for tokens in two places:

- `"Authorization"` header. The token value can have any prefix (`"TOKEN abcde"` , `"BEARER abcde"` , etc) or just the token value itself
- `"?token=abcde"` URL parameter

From further investigation it seems like on November 1st, the system was implemented. I remember writing my pseudocode and researching the API before this and today it is after November 1st.

> The header value will supercede the URL parameter if both are found. All examples in the documentation use the header method.
>
> Some features require paid plans to be associated with the supplied token. Feature availability is listed on the Account Page.
>
> Tokens will be required for all endpoints starting November 1st, 2019 with daily rate limits put in place starting on January 1st, 2020. For more information, see this post on the dev blog.

After creating an account and getting my key, I added the token header in the request template and put my API key in. So, it goes from:

```
def search_location(self):
    search_template = "https://avwx.rest/api/metar/{}?options=summary&format=json&onfail=cache"
    search_url = search_template.format(self.search_input.text)
    request = UrlRequest(url=search_url, on_success=self.found_location, on_error=print, on_failure=print)
```

To:

```
def search_location(self):
    search_template = "https://avwx.rest/api/metar/{}?options=summary&format=json&onfail=cache&token={}"
    search_url = search_template.format(self.search_input.text, "[redacted]")
    request = UrlRequest(url=search_url, on_success=self.found_location, on_error=print, on_failure=print)
```

With the API key put inside the red rectangle (I'm hiding it for security reasons of course).

Time to test it again:

Test number	What are we testing for	Expected result	Test data	Results	Notes
25	Check if we can access the API successfully.	We should get a message on the console.	EGLL as the ICAO	Success `jsdfhk` We got our message in the console.	

That ended up taking the rest of day 8.

Day 9 – Finishing off the backend

To start the day, I decoded the data using json.loads and prepared the list which will be displayed in the listView

```
def found_location(self, request, data):
    print("jsdfhk")
    data = json.loads(data.decode()) if not isinstance(data, dict) else data
    toAdd = []
```

```
raw = data['raw'].split()
summary = data['summary'].split(',')
for i in summary:
    toAdd.append(i)
```

Here I create a list for the raw data containing the raw data split by comma.

The same is applied to the summary of the METAR. Next, each part of the summary is added to the toAdd list.

Finally, we update the list view with the toAdd list so the METAR summary will be displayed to the user.

```
self.search_results.item_strings = toAdd
```

I have not decided to put in a system to update the user's details with the new recent search in this version as version 1 is for the fundamental parts of the app and I don't believe that said system is fundamental to the app.

We move onto our final procedure to fill the recent search dropdown with data. The first line of the procedure may look complicated (because it is). As the user's data was a string and json.loads() is

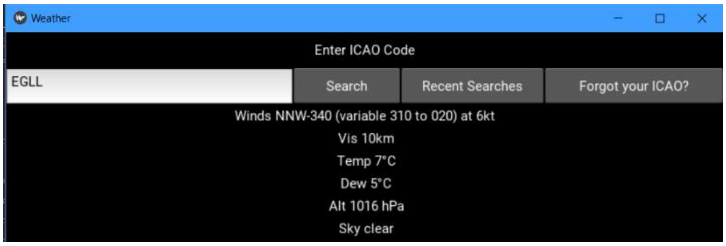
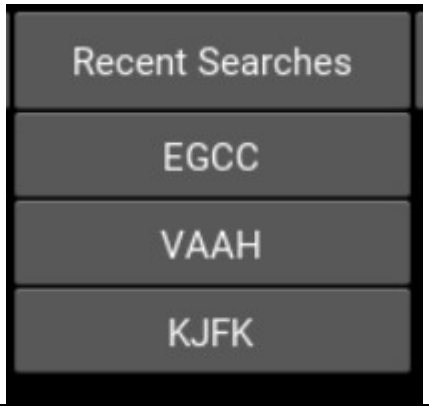
```
def fill(self):
    rs = (((((self.usr_details.split(':'))[4].split(']'))[0].replace(' ', '')).replace("'", "")).replace('[', '').split(','))
```

quite buggy with this sort of stuff, I have decided that doing it manually would be more reliable for now (hence the painful formatting). It basically gets the recent searches from the user details and puts into a list.

Lastly, I update the recent search object properties with the contents of rs to make sure that the recent searches are correct and are visible and functional to the user.

```
self.recent_search_one.text, self.recent_search_two.text, self.recent_search_three.text = rs[0], rs[1], rs[2]
```

Time to test it:

Test number	What are we testing for	Expected result	Test data	Results	Notes
26	Check if the METAR is displayed properly on the screen	We should get the details stacked vertically so each part is under each other	EGLL as the ICAO	Success  This is the format which I expected.	
27	Checking if the recent searches dropdown is filled correctly	We should get EGCC, VAAH and KJFK respectively	We are using the account with the username 'h'	Success  	

28	Checking if the button to the ICAOFinder works	We should get an error as we have not created that class yet	n/a	Success
				`raise ScreenManagerException('No Screen with name "%s".' % name)`
				`kivy.uix.screenmanager.ScreenManagerException: No Screen with name "ICAO".`
				We got the desired error	

Time to check with our criteria.

Checking with our criteria

Criteria	How to check	Completed (Y/N)	Notes
METAR search system	A system which presents a decoded accurate METAR in a clear and concise format	Y	
A recent search system	Check if the recent searches have merged from the user's data and when you click it, it automatically goes in the search box.	Y/N	We have got the fundamentals working but the rest will be in another version as explained.

Backend code which searches for the METAR using an API and returns the results	Check-in console with a print statement. It should also show the results on the screen.	Y	
The results to be presented in a clear format	If when we show it to our shareholders, they find it sufficient.	N	Again, I have displayed the fundamentals and the formatting will be done in another version
A way for the user to search for the ICAO code	Check that when a button linking the 2 screens is clicked, the screen changes.	Y/N	There is a button which does this, but we need the create the next class for it to work

On to 1.4!

Day 10 and 11 – ICAOFinder Class

Day 10 – The Frontend

This class will act quite similarly to 1.4. Except for the recent searches and the data which we'll be using.

```
WeatherRoot:
<WeatherRoot>:
    id: screen_manager
    RegisterPage:
    LoginPage:
        id: login_page
        name: "LoginPage"
        manager: screen_manager
    AddLocationForm:
        id: add_location_form
        manager: screen_manager
        usr_details: login_page.usr_info
    ICAOFinder:
```

Firstly, we update the root in the kv code so we can navigate to our new class.

As we'll not be needing the user details here (for now), there is little point in adding the rest of the details like we did with 1.4.

We also initialise the class in the python code.

```
class ICAOFinder(BoxLayout, Screen):
```

Next, we create the class and set its name, the ObjectProperties which we'll be manipulating in the backend and setting the classes orientation to vertical so the widgets stack properly.

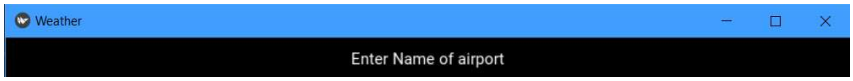
```
<ICAOFinder>:
    name: "ICAO"
    search_input: search_box
    search_results: search_results_list_ICAO
    orientation: "vertical"
```

```
BoxLayout:
    height: "40dp"
    size_hint_y: None
    Label:
        text: "Enter Name of airport"
```

Just like 1.4, we create a title at the top of the screen, this time asking the user for the name of the airport. The height remains the same and the size_hint_x is used to make sure that the height can be set.

Now, we should test if this is working.

Now that we know it works, we can now finish off the rest of the backend.

Test number	What are we testing for	Expected result	Test data	Results	Notes
29	Check if the class loads and to check if the title text is displayed to the user	The title text will be displayed to the user. If this is displayed then clearly the class has loaded	N/A	Success 	

Here, I have created the BoxLayout which will house the rest of the widgets (apart from the ListView). Its height has been set and the input box and the search button has been created.

Both have been id'd so we can call it in the backend and the button has been programmed to run one of the backend procedures once clicked.

```
BoxLayout:
    height: "40dp"
    size_hint_y: None
    TextInput:
        id: search_box
        size_hint_x: 40
    Button:
        text: "Search"
        size_hint_x: 10
        on_press: root.findICAO()
```

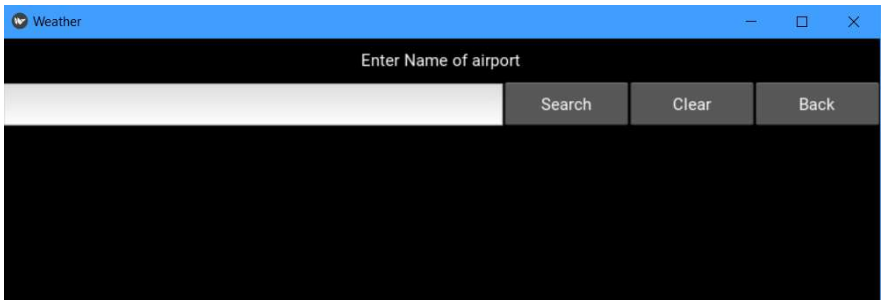
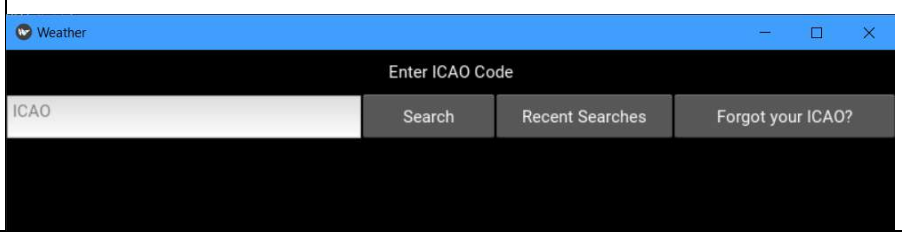
Next, we finish off the boxlayout by creating 2 final buttons. One which will clear the contents of the ListView and another which moves the user back to the AddLocation Class when clicked.

There is no need to fiddle with the size hint's here as if they don't add up to 100, Kivy will automatically make it do so.

```
Button:
    size_hint_x: 10
    text: "Clear"
    on_release: search_results_list_ICAO.item_strings = []
Button:
    text: "Back"
    size_hint_x: 10
    on_release:
        app.root.current = "AddLocation"
        root.manager.transition.direction = "right"
```

Finally, I create the ListView. This is outside of the boxlayout as if it were inside, it would be next to the back button (as that boxlayout is horizontally orientated) and not underneath (as the widgets in the class are aligned vertically. The horizontal alignment inside the boxlayout overrides the parent inside the boxlayout but outside it remains the same.

```
ListView:
    id: search_results_list_ICAO
    item_strings: []
```

Test number	What are we testing for	Expected result	Test data	Results	Notes
30	Check if the widgets have loaded in properly and are displayed correctly to the user.	The title text will be displayed at the top. With the boxlayout underneath (with its widgets aligned horizontally) and the ListView underneath this.	N/A	Success 	
31	If the Back button sends you back to 1.4	The screen should be changed to 1.4	N/A	Success 	

Now we can test the frontend.

The ListView is there, we won't be able to see it until we put something in it. A bit like the confirmation text from 1.1 and 1.2.

Now on to the backend

Day 11 – The Backend

```
class ICAOFinder(BoxLayout, Screen):
    search_input = ObjectProperty()
    search_results = ObjectProperty()
```

First, we initialise the object properties from the kv code so we can manipulate them here.

Next, we create our main procedure and open our CSV file (see note below) so we can search in there (we ensure that its encoding is correct to avoid any errors). Then we initialise a list and iterate over the CSV file, putting each line into a list then splitting each line into its individual components (stored as a list) which will be put into our initialised list.

```
def findICAO(self):
    f = open("Data/airports.csv", encoding="utf-8")
    data = []
    for line in f:
        data_line = line.rstrip().split('\n')
        data_list = data_line[0].split(',')
        data.append(data_list)
```

Note – The CSV file has been retrieved from https://ourairports.com/data/ which has a note to encode its contents in utf-8 for usage. Any other CSV files used in this app will also be from this website unless specified otherwise.

```
ICAOList = ["Search results are: "]
counter = 0
```

We prepare the start of our list which will be displayed to the user (via the ListView) and a counter which will be used to specify the total number of results.

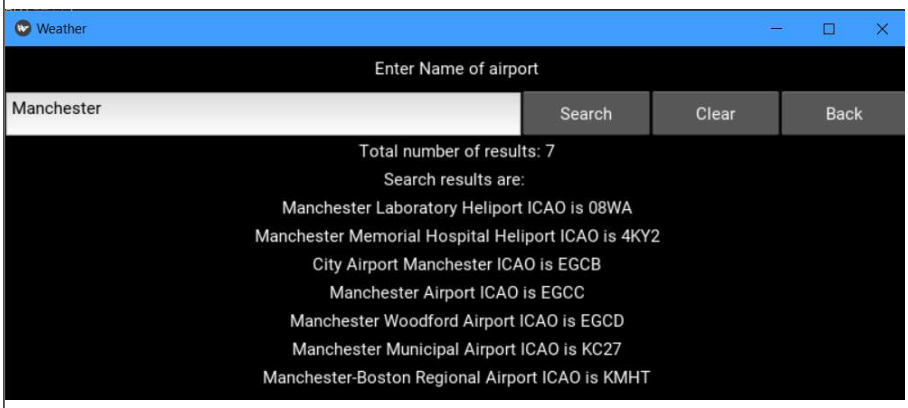
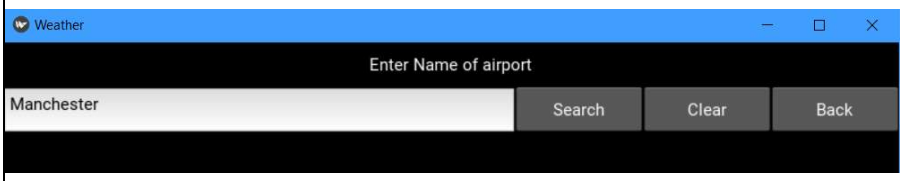
After that, we create a loop. This iterates over the list made earlier and checks if our search query is part of the airport name (stored in index 1 of said list). If it is then the counter is incremented and these details are added to the ICAOList formatted so it is clear to the user.

```
for lst in data:
    if self.search_input.text in lst[1]:
        counter += 1
        ICAOList.append("{} ICAO is {}".format(lst[1], lst[0]))
```

Finally, we add the total number of results to the start of the list and update the ListView with the list so the results can be displayed to the user.

```
ICAOList.insert(0, "Total number of results: {}".format(counter))
self.search_results.item_strings = ICAOList
```

Time to test it out.

Test number	What are we testing for	Expected result	Test data	Results	Notes
32	Check if the search system works	7 results should appear with Manchester in the name.	Manchester in the search box.	Success 	
33	Check if the clear button works	The ListView section should be empty	N/A	Success 	

Checking with our criteria

Criteria	How to check	Completed (Y/N)	Notes
ICAO search system	A system that accurately returns all results available based on the query in a clear format.	Y	
Search results to be clear and accurate	Ask the shareholders if it is fine or not.	Y	
A way to navigate back to the METAR search system	Check that when a button linking the 2 screens is clicked, the screen changes.	Y	

End of version review (Day 12)

What has been done

- A working register/login system has been made which meets our requirements.
- There are buttons linking the 2
- We have tried out home text colour which can be used later
- The user can search for METAR, getting the latest accurate METAR available.
- The user can access the recent searches currently stored on the system.
- There is a button which links to the ICAO Finder
- The ICAO finder provides offline and accurate searching for ICAO codes based on input.
- The search can be cleared
- The user can return to the METAR searches with the click of a button.

What I need to do

- Create a settings system which has different colour schemes and can be accessed by the user.
- There is some form of TTS system available for users
- The recent search info is updated each search
- The output of the METAR search is formatted clearly and has extra info.
- For the app to be free if it is to be distributed.

How will I do it

On day 12 of the project, I started to create a plan about the project's future and what the next versions will have in them (a sort of roadmap).

<u>V1.1 – Cleaning up after myself.</u>

This version will have:

- A fully working recent search system which updates the user's info after each search

- A formatted version of the METAR search system

<u>V1.2 – Personalisation</u>

This version will have:

- A working settings system where settings can be set by each user

- A light and dark theme for colours

<u>V1.3 – Speak to me</u>

This version will have a working TTS system.

<u>V1.4 – Optional Stuff</u>

Optional features could be:

- Translation

- Voice recognition

- Airline based themes

Version 1.1 (Day 13, 14 and 15)

Introduction

Our goal with version 1.1 is to have a fully working recent search system and the METAR results to be formatted so they are clear and concise.

This version will be split into 2 parts:

Version 1.0.5 – The recent search system

Version 1.1 – Formatted METAR results

Version 1.0.5 (Day 13)

- Why is this version being coded? – The recent search system needs to be completed for it to be fully functional. It is also a requirement of our criteria.

- Requirments – Once a METAR search is complete, the recent searches must be updated both on the user's info as well as the JSON file.

- Pseudocode

After some research, I have discovered that instead of manually decoding our string dictionary (the user's data), we can use the ast module and its literal_eval method to convert it to a dictionary that can be easily handled. So, our first point of order is to change this up.

Note – the Pseudocode is now done in pycharm directly.

```
Updating the python code:
    Updating the AddLocationForm Class:

        Updating the fill method:

            Update the fist line of this method to:

            // This is a much easier and more efficient way of getting the recent searches.
            rs = (ast.literal_eval(self.usr_details))["recent_searches_METAR"]

        Updating the found_location method:

            Adding the following to the end of the method:
                // We get a dictionary form of the user details
                usrdata = ast.LITERAL_EVAL(self.usr_details)
                // This updates the recent searches for the new search
                usrdata["recent_searches_METAR"] = [self.recent_search_one.text, self.recent_search_two.text, self.recent_search_three.text]
                // Then we update the usr_details, making sure that it is formatted to a string to avoid a type error.
                self.usr_details = STRING(usrdata)
                // we call the method to update the JSON file, adding the appropriate parameters in.
                update_JSON(usrdata['recent_searches_METAR'], 'recent_searches_METAR', usrdata['username'])
```

Our next port of call is to update the JSON file. As we'll be doing this quite often, we should make a method that is not attached to any class. This class should take a parameter of the data which you'll be updating, where we want to update this and for what user. e.g. parameter's being: ['EGCC', 'EGLL', 'KJFK'], 'recent_searches_METAR', 'h'.

The great thing about this method is that it can be called from anywhere and it can also be used to update any part of the user's data - not just the recent searches. This means we could potentially use this with the theme settings too.

```
Updating the python code:

    PROCEDURE update_JSON(update_data, location, user):
        // opens data.json to read and sets it to variable f
        f = OPENREAD("Data/data.json", encoding="utf-8")
        // uses json.load to decode the data into a python dictionary called data
        data = json.LOAD(f)
        // initialises a list which will store the id's of each user
        idLst = []
        // closes the file as all necessary data has been extrapolated
        f.CLOSE()
        // a for loop which iterates over each user, gets their id and adds it to the list
        FOR users IN data['users'] {
            idLst.append(users)
        }

        // We now want to find the users data so we iterate over the id's and update the data based on the parameters
        // given as a parameter
        FOR id IN idList {
            IF id['username'] == user {
                data['users'][id][location] = update_data
                pass
            }
        }

        // we open a temporary file where we dump the updated file. Then, we close the file.
        f = OPENWRITE("Data/temp.json", encoding="utf-8")
        json.DUMP(data, f, indent=2)
        f.CLOSE()
```

This continues from the f.CLOSE()

```
    // this deletes the old json file, and renames the temporary one to be the same name as the old one (basically replacing them)
    os.REMOVE("Data/data.json")
    os.RENAME("Data/temp.json", "Data/data.json")
ENDPROCEDURE
```

- Test data

- Actual programming (The commented code will be available in the appendix)

Firstly, I update the fill method as in the pseudocode

```
def fill(self):
    rs = (ast.literal_eval(self.usr_details))["recent_searches_METAR"]
    self.recent_search_one.text, self.recent_search_two.text, self.recent_search_three.text = rs[0], rs[1], rs[2]
```

After that, I tested to make sure that the last module works.

Test number	What are we testing for	Expected result	Test data	Results	Notes
34	Check if the recent search still loads	3 results should load: EGCC, VAAH, and KJFK	'h' as the user	Success 	

Next, I moved onto the found_location method

```
usrdata = ast.literal_eval(self.usr_details)
usrdata["recent_searches_METAR"] = [self.recent_search_one.text, self.recent_search_two.text,
                                    self.recent_search_three.text]
self.usr_details = str(usrdata)
print(self.usr_details)
```

I also print the usr_details in the console for testing purposes.

Test number	What are we testing for	Expected result	Test data	Results	Notes
35	Check if the recent search is updated	3 results should load: EGLL, EGCC, and VAAH	'h' as the user. And we search for EGLL.	Success 	

The console also confirms this:

```
{'username': 'h', 'password_hash': '251ae1a4ffe473bf2ac4b8970f4da26d', 'email': 'h@h.h', 'recent_searches_METAR': ['EGLL', 'EGCC', 'VAAH']}
```

As this works, we finish the rest of this method:

```
update_JSON(usrdata['recent_searches_METAR'], 'recent_searches_METAR', usrdata['username'])
```

The red line means we haven't created the procedure yet, but it will be done.

Next, we set up the new procedure by loading up the JSON file and preparing it for iteration.

```
def updateJSON(update_data, location, user):
    f = open("Data/data.json", "r", encoding="utf-8")
    data = json.load(f)
    idLst = []
    f.close()
    for users in data['users']:
        idLst.append(users)
```

Here, I realise that we should call the data parameter something else as it is overridden in the procedure.

So, I change it. First in the pseudocode:

```
PROCEDURE update_JSON(update_data, location, user):
```

```
    data['users'][id][location] = update_data
```

And then the python code:

```
def updateJSON(update_data, location, user):
```

Then we update the data with the parameters

```
for id in idLst:
    if id['username'] == user:
        data['users'][id][location] = update_data
        pass
```

Then we finish off the class by creating the temporary JSON file then updating the old one.

```
with open("Data/temp.json", "w", encoding="utf-8") as f:
    json.dump(data, f, indent=2)
f.close()
os.remove("Data/data.json")
os.rename("Data/temp.json", "Data/data.json")
```

We should now test the program:

Test number	What are we testing for	Expected result	Test data	Results	Notes
36	Checking if the recent search updates the JSON file.	3 recent searches should be in the user's data in the JSON file: EGLL, EGCC, and VAAH	'h' as the user. And we search for EGLL.	Failure: NameError: name 'update_JSON' is not defined	

It is because I forgot the underscore in the actual procedure. It can be fixed easily:

```
def update_JSON(update_data, location, user):
```

Time to test it again.

Test number	What are we testing for	Expected result	Test data	Results	Notes

37	Checking if the recent search updates the JSON file.	3 recent searches should be in the user's data in the JSON file: EGLL, EGCC, and VAAH	'h' as the user. And we search for EGLL.	Failure:
				`if id["username"] == user:`
				`TypeError: string indices must be integers`	

This is because id is a number, we should search the data for the id and so on like so:

```
if data['users'][id]["username"] == user:
```

Hopefully it will work this time:

Test number	What are we testing for	Expected result	Test data	Results	Notes
38	Checking if the	3 recent searches should be	'h' as the user. And	Success:	

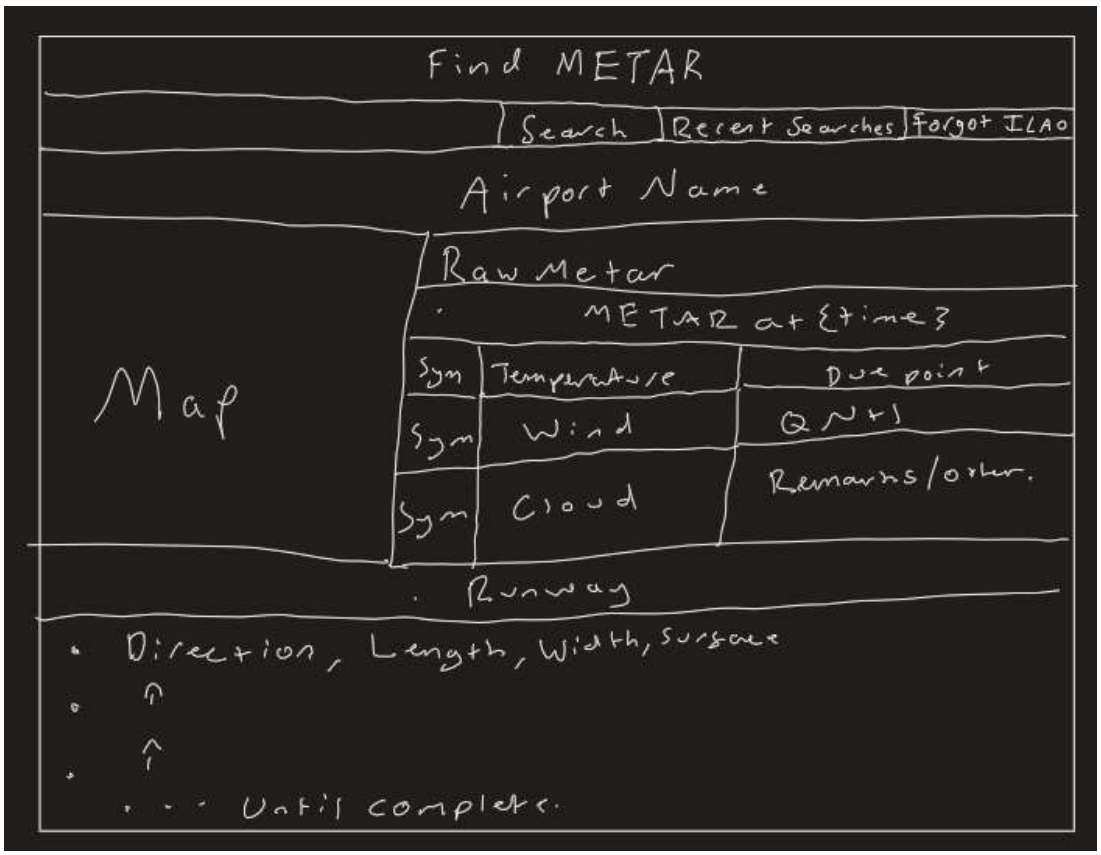
	recent search updates the JSON file.	in the user's data in the JSON file: EGLL, EGCC, and VAAH	we search for EGLL.	"3": { "username": "h", "password_hash": "251ae1a4ffe473bf2ac4b8970f4da26d", "email": "h@h.h", "recent_searches_METAR": ["EGLL", "EGCC", "VAAH"] },	

Well that's 1.05 complete. It is now time to move onto version 1.1

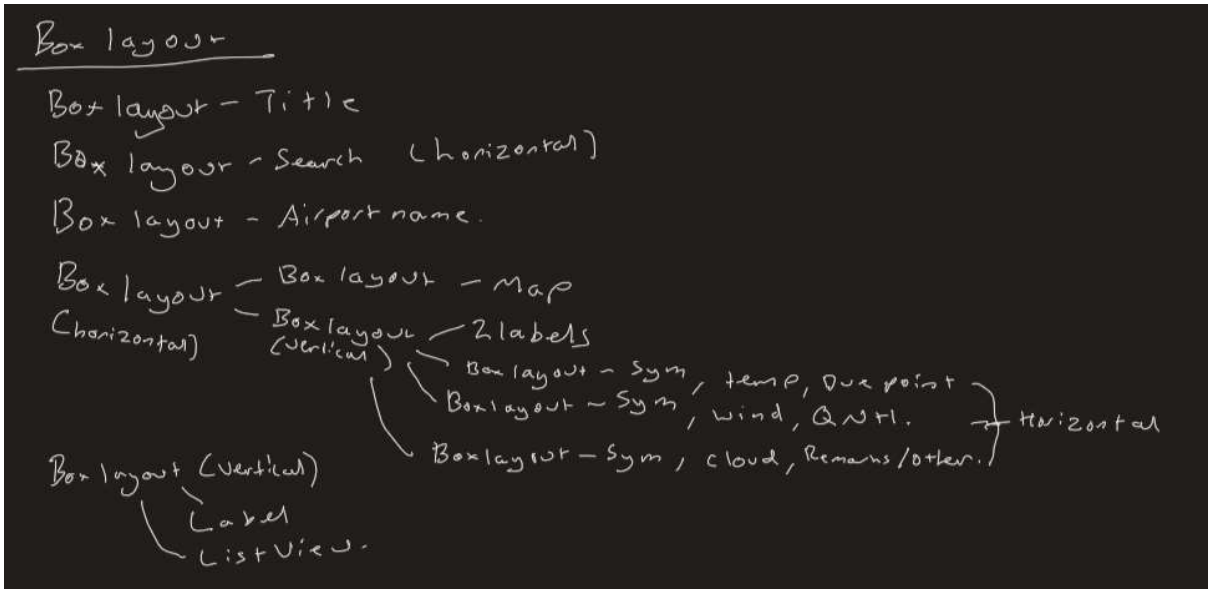
Version 1.1.0 (Day 14 and 15)

For this version, we wish to reform the metar search results.

Firstly, I created a little diagram with how I wanted the end result to look like:



Here I made a little map of how this should be layed out using box layouts.




```
airport_info: airport
raw: raw_METAR
time: time
temp: temp
dew: dew
wind: wind
alt: alt
cloud: cloud
other: other
runway_data: runway_data
airport: airport

map: map
```

Day 14 - Changes to the frontend

Firstly, I created the object properties for the backend so we can access them in the python code:

After that we create a label for the airport name and id it. We add a size hint to enforce our height limit.

```
Label:
    id: airport
    text: "Airport Name"
    height: "20dp"
    size_hint_y: None
```

We then create the first BoxLayout, and have a background colour of red (for now, just to make it stand out). The background colour is set by drawing a rectangle behind the boxlayout.

```
BoxLayout:
    canvas.before:
        Color:
            rgba: (1,0,0,1)
        Rectangle:
            pos: self.pos
            size: self.size
    orientation: "vertical"
```

```
orientation: "vertical"
BoxLayout:
    orientation: "vertical"
    BoxLayout:
        height: "100dp"
        orientation: "horizontal"
        Map:
            id: map
            zoom: 12
            lat: 40.63980103
            lon: -73.77890015
        BoxLayout:
            orientation: "vertical"
            Label:
                id: raw_METAR
                text: "Raw METAR"
            Label:
                id: time
                text: "METAR at Time"
            BoxLayout:
                orientation: "horizontal"
                Label:
                    id: temp
                    text: "1"
                Label:
                    id: dew
                    text: "2"
```

We create our next boxlayout and its children as shown in the planning. We remember to keep the orientation correct and id anything which is needed.

The map widget is a bit different (it is not a base widget so it has not got the same colouring as the rest. It requires a default zoom, latitude and longitude. I have set the default to JFK airport with the zoom fitting correctly in its borders. The map widget uses OpenStreetMap as a default and doesn't require an API key on my behalf.

```
            BoxLayout:
                orientation: "horizontal"
                Label:
                    id: wind
                    text: "1"
                Label:
                    id: alt
                    text: "2"
            BoxLayout:
                orientation: "horizontal"
                Label:
                    id: cloud
                    text: "1"
                Label:
                    id: other
                    text: "2"
```

```
BoxLayout:
    height: "100dp"
    orientation: "vertical"
    Label:
        text: "Runway Data"
    ListView:
        id: runway_data
        item_strings: []
```

The rest of the current box layout is finished off, adding default text where needed.

The rest of the main boxlayout is completed with the runway data. I have decided to use a listview as there can be multiple runways in a airport and trying to guess a number would not be efficient or reliable.

For this version, we'll be testing once the backend is completed so the map doesn't crash.

Onto the backend!

Day 15 - Updating the backend

Before we do anything, we'll have to import the map module.

```
from mapview import *
```

The Kivy mapview module is the most up to date one with the best docs available.

But we have to create the class for the map where we will create an instance for when we search.

```
class Map(MapView):
    def build(self):
        mapview = MapView(zoom=AddLocationForm.zoom, lat=AddLocationForm.lat, lon=AddLocationForm.long)
        return mapview
```

```
airport_info = ObjectProperty(
raw = ObjectProperty()
time = ObjectProperty()
temp = ObjectProperty()
dew = ObjectProperty()
wind = ObjectProperty()
alt = ObjectProperty()
cloud = ObjectProperty()
other = ObjectProperty()
runway_data = ObjectProperty()
airport = ObjectProperty()
map = ObjectProperty()
```

Then we bring over the object properties in the class so we can use and manipulate them.

Then we create a method which is essentially the same as the one for getting the METAR but instead

```
def get_info(self, ICAO):
    search_template = "https://avwx.rest/api/station/{}?options=format=json&onfail=cache&token={}"
    search_url = search_template.format(self.search_input.text, self.token)
    request = UrlRequest(url=search_url, on_success=self.update_info, on_error=print, on_failure=print)
```

of a METAR, we request information from the station which the API also provides in a JSON format.

The token is set as a separate string here for convenience.

Before we move onto creating the update_info method, we need to make some changes to the found_location class.

Firstly, we need to make sure the raw metar bit is being displayed:

```
self.raw.text = data['raw']
```

Then we delete the bit which refers to our old ListView as it no longer exists (I deleted it previously implicitly).

Then we parse the relevant info from the api request and update the screen with this data.

```
self.wind.text = summary[0]
self.other.text = summary[1].replace('Vis', 'Visibility -')
self.temp.text, self.dew.text = summary[2].replace('Temp', 'Temperature -'), summary[3].replace('Dew', 'Dew Point -')
self.alt.text, self.cloud.text = summary[4].replace('Alt', 'Altimeter'), summary[5]
```

We add some text around the data to make it presentable for the user.

Then we use both of our methods, we get the info, using the search input as an ICAO the we directly

```
if self.get_info(self.search_input.text) == True:
    update_JSON(usrdata['recent_searches_METAR'], 'recent_searches_METAR', usrdata['username'])
```

call the update_JSON method to update the recent searches in the JSON file.

This works because the get_info method calls another method which has a return statement as it is a function, not a procedure.

Then we create the update_info method, taking the relevant data from the get_info class. Westart

```
def update_info(self, request, data):
    data = json.loads(data.decode()) if not isinstance(data, dict) else data
```

this class just like found_location class, by getting our json data and turning it into a python dictionary.

Then we format the airport name so it is clearer to the user with extra info and centre our map on the airport my editing it's longitude and latitude.

```
name = '{}, {} ({})'.format(data['name'], data['country'], data['icao'])
self.map.center_on(float(data['latitude']), float(data['longitude']))
```

The rest of this class is focused on the runway data, first we start by preparing the listview and a counter for iteration.

```
toAdd = []
counter = 1
```

Then we move onto the actual iteration.

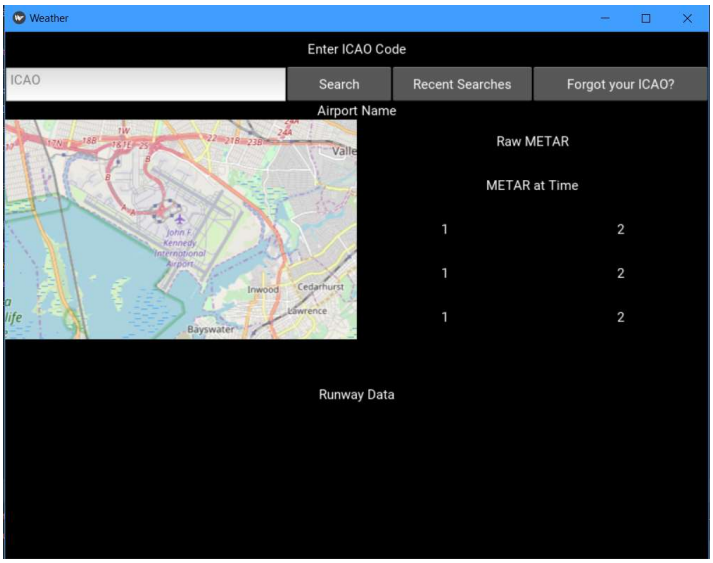
```
for runway in data['runways']:
    d = "{}) {}/{}, Length = {}ft, Width = {}ft".format(counter, runway['ident1'], runway['ident2']
                                                        , runway['length_ft'], runway['width_ft'])
    toAdd.append(d)
```

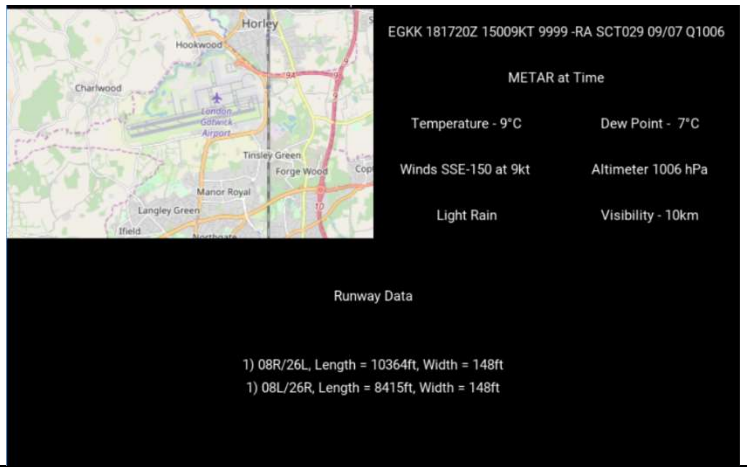
Here, we iterate over all the runways then create a string which has the runway number, its directions, length and width. All of this is extremely important for a pilot as they need to position themselves with the runway when landing. After that we add the string to the toAdd list.

We finish off the class by updating the ListView and returning true for the previous method as well as making sure the airport name is updated.

```
self.runway_data.item_strings = toAdd
self.airport.text = data['name']
return True
```

Now it's time to test our code.

Test number	What are we testing for	Expected result	Test data	Results	Notes
39	Cheeking if the new metar search results layout works	The layout to look just like we planned it.		Success: 	This is basically testing the backend.

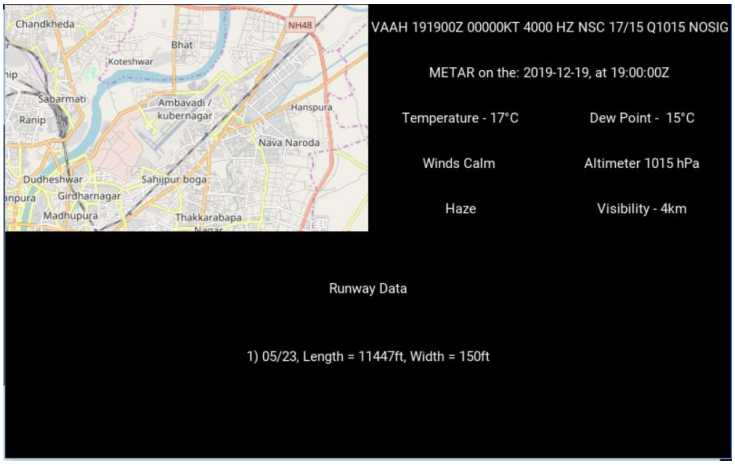
40	Checking if the search results show and that the map is updated	The map changes to Gatwick and the METAR loads up with the correct time.	EGKK as the search query	Failure 	It seems like we have forgotten to format the date/time section.

I shall add the date/time section with only 2 lines of code:

```
time = data['time']['dt'].split('T')
self.time.text = "METAR on the: {}, at {}".format(time[0], time[1].split('+')[0])
```

As the METAR search already has a timestamp for when the METAR was updated, we should use that to update our time ObjectProperty.

Time to test it again:

Test number	What are we testing for	Expected result	Test data	Results	Notes
41	Checking if the search results show and that the map is updated	The map changes to Gatwick and the METAR loads up with the correct time.	VAAH as the search query	Success: 	

End of version review (Day 16)

What has been done

- A working register/login system has been made which meets our requirements.
- There are buttons linking the 2
- We have tried out home text colour which can be used later
- The user can search for METAR, getting the latest accurate METAR available.
- The user can access the recent searches currently stored on the system.
- There is a button which links to the ICAO Finder
- The ICAO finder provides offline and accurate searching for ICAO codes based on input.
- The search can be cleared
- The user can return to the METAR searches with the click of a button.
- The recent search info is updated each search
- The output of the METAR search is formatted clearly and has extra info.

What I need to do

- Create a settings system which has different colour schemes and can be accessed by the user.
- There is some form of TTS system available for users
- For the app to be free if it is to be distributed.

How will I do it

V1.2 – Personalisation

This version will have:

- A working settings system where settings can be set by each user

- A light and dark theme for colours

V1.3 – Speak to me

This version will have a working TTS system.

V1.4 – Optional Stuff

Optional features could be:

- Translation

- Voice recognition

- Airline based themes

Time for the next version

Version 1.2.0

Introduction

This version will have:

- A working settings system where settings can be set by each user

- A light and dark theme for colours

Planning

The plan in a bit more detail:

For this version, we would like to add a popup box in 1.3 which lets the user pick between dark and light mode. Their option should be saved in a text file which is then opened when the app starts which edits the base parameters of widgets in line with the mode.

I have chosen a popup over a separate class as it is more lightweight which will reduce file size. A popup is also easier and faster to code which means our shareholders will get the product faster.

I have chosen to use a text file system instead of using our json file as we would like the theme to load as soon as the app is opened, not after login.

We should also consider a neutral theme, which would work well with popups and the metar search results

Themes

For a light theme we should have:

Background Colour – White

Text Input box colour – Black

Text Colour – Black

Button colours – 1.1 and 1.2 = blanched almond, 1.3 and 1.4 = black

For a dark theme, we should have:

Background Colour – Black

Text input box colour – Default

Text Colour – White

Button colours – 1.1 and 1.2 = blanched almond, 1.3 and 1.4 = black

For the neutral theme we should have:

Background colour – Indigo (RGBA – 75,0,130,0.5), the alpha value should mean that the grey will blend in with the main theme.

Text Colour – Blanched Almond (RGB - 255, 235, 205), this colour naturally looks good on grey

Other

The popup should have a box layout with 3 buttons vertically aligned. 2 of the buttons should be toggle buttons so we can toggle between dark and light and the last one should be a normal button which closes the popup.

The button to open the popup should be to the right of the title text in 1.3 with it taking 30% of the screen.

The font size should be 30 and the spacing set at 20.

The toggle buttons shall call a procedure not associated with any class which updates the text file so when the program next loads, the theme should change.

We'll use Kivy's clock to time a procedure (again not associated with any class) which will:

1) Open the text file and determine which theme is in use:
2) If it is dark theme, end the procedure
3) If it is light theme then it will find the relevant parts of the program and change the colours to the ones listed above.

Changes to the Frontend

Our first point of call is to change all the text colours so they are blanched almond in the metar results in 1.3 and blanched almond for all buttons in the program. For this, we simply add this to the end of all of the said areas. This is the line we add:

```
color: 1, 0.9, 0.8, 1
```

Then we need to change the background colour of the metar results boxlayout. We shall do this using canvas.before, creating a rectangle behind the layout then changing the colour of the rectangle. The colour settings are calculated by (RGB value / 256 rounded to 2dp) and the alpha value is the one set out in the planning stage.

```
canvas.before:
    Color:
        rgba: (0.29, 0, 0.50, 0.5)
    Rectangle:
        pos: self.pos
        size: self.size
```

Next, we import all the new classes which we'll need for our backend in the python code.

```
from kivy.uix.popup import Popup
from kivy.uix.togglebutton import ToggleButton
```

Next, we add the button to load the popup and change the size_hint_x's so the boxlayout looks like this:

```
BoxLayout:
    height: "40dp"
    size_hint_y: None
    Label:
        text: "Enter ICAO Code"
        size_hint_x: 70
    Button:
        size_hint_x: 30
        color: 1, 0.9, 0.8, 1
        text: 'Settings'
        on_release: Factory.MyPopup().open()
```

Here, we are using the Kivy.uix.factory class. This allows us to create a popup object from the popup class in the Kivy code and be able to call to it from other classes.

So, we'll need to import factory:

```
from kivy.factory import Factory
```

Next, we need to import factory into our Kivy code for it to work as well as creating the popup object with some base attributes:

```
#:import Factory kivy.factory.Factory
<MyPopup@Popup>:
    auto_dismiss: False
    title: "Settings"
```

We add a boxlayout in the popup, editing the colour to be our neutral theme in the same way as we did before:

```
BoxLayout:
    orientation: "vertical"
    spacing: 20
    canvas.before:
        Color:
            rgba: (0.29, 0, 0.50, 0.5)
        Rectangle:
            pos: self.pos
            size: self.size
```

Then we add our 2 toggle buttons with the settings as planned:

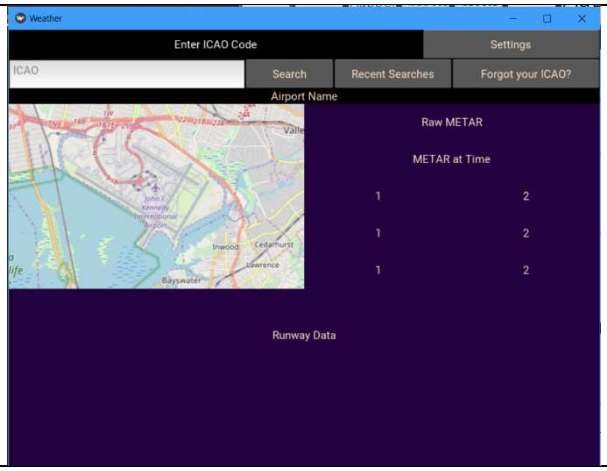
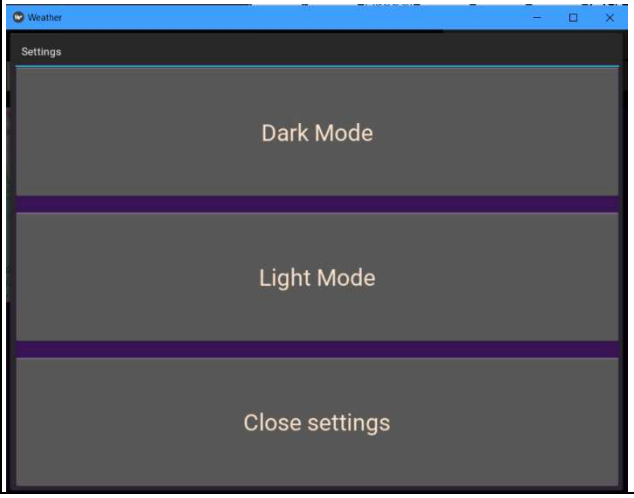
```
ToggleButton:
    font_size: 30
    text: "Dark Mode"
    group: "Themes"
    color: 1, 0.9, 0.8, 1
ToggleButton:
    font_size: 30
    text: "Light Mode"
    group: "Themes"
    color: 1, 0.9, 0.8, 1
```

We finish off the popup by adding the button to close the popup.

```
Button:
    font_size: 30
    text: 'Close settings'
    color: 1, 0.9, 0.8, 1
    on_release: root.dismiss()
```

Time to test:

Test number	What are we testing for	Expected result	Test data	Results	Notes
42	Checking if the colours have changed	The metar results background should be purple and its text		Success:	

		blanched almond			
43	Checking if the popup loads and is formatted properly	A button for the popup should be next to the tile text in 1.3 and when pressed should open a popup with 3 buttons. The popup should have the neutral theme.		Success: 	

Onto the Backend

Changes to the Backend

I have decided that I'll put the method which runs after the toggle button is used in 1.3 so I'll have to route the buttons to go there, using Kivy factory as so:

```
ToggleButton:
    font_size: 30
    text: "Dark Mode"
    group: "Themes"
    color: 1, 0.9, 0.8, 1
    on_release: Factory.AddLocationForm.change_theme('d')
ToggleButton:
    font_size: 30
    text: "Light Mode"
    group: "Themes"
    color: 1, 0.9, 0.8, 1
    on_release: Factory.AddLocationForm.change_theme('l')
```

This is the only code which will be added to the Kivy file here, so I have decided to put it in this section instead of the previous.

```
def change_theme(s):
    os.remove("Data/theme.txt")
    f = open("Data/theme.txt", "w+")
    f.write(s)
    f.close()
```

Next, we create the actual change theme function:

We delete the file and re-write it with the changed theme as it is easier to do and it doesn't require extra modules or enumeration.

After that, we edit the code which loads the program to run a procedure before it using Kivy.clock:

```
if __name__ == "__main__":
    Clock.schedule_once(load_theme)
    WeatherApp().run()
```

```
def load_theme(time):
    f = open("Data/theme.txt", "r+")
    d = f.read()
    f.close()
```

Then we create the procedure. It has to take time as a parameter so we can use Kivy.clock with it. The first lines of the procedure open the theme file and get the data from it so we know what theme is saved.

If the theme saved is light then we change the background colour to be white.

```
if d == 'l':
    Window.clearcolor = (1, 1, 1, 1)
    root = App.get_running_app().root  # WeatherRoot instance
```

Just like this bit, I will comment the rest of the code for the marker to understand it easier. But basically what we are doing is searching for the root instance then iterating over our classes, looking for a boxlayout, iterating over its contents. Then for every text_input or label in them, we change their colour to the one specified.

```
    root = App.get_running_app().root  # WeatherRoot instance
    for i in range(0, 4):
        screen = root.screens[i]  # RegisterPage instance
        box_layout = screen.children[0]  # BoxLayout instance
        for child in box_layout.children:  # children of box_layout
            if isinstance(child, TextInput):  # verify that the child is a TextInput
                child.background_color = (0, 0, 0, 1)
                child.foreground_color = (1, 1, 1, 1)
            elif isinstance(child, Label):
                child.color = (0,0,0,1)
```

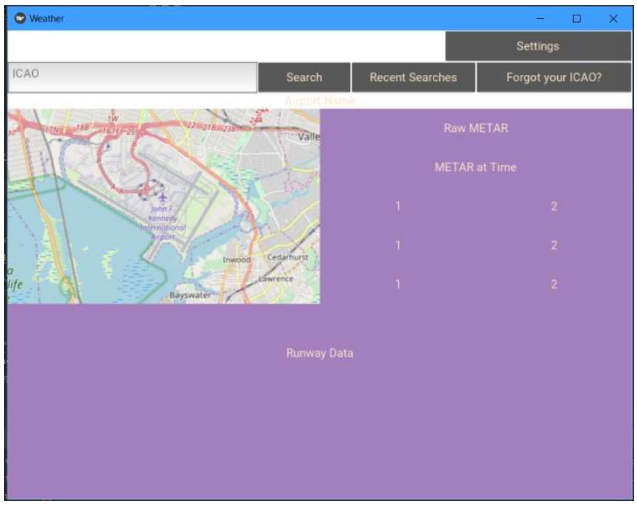
We do a try except loop with second boxlayouts as they may not exist in all of our classes e.g. they exist in 1.3 and 1.4 but not 1.1 or 1.2.

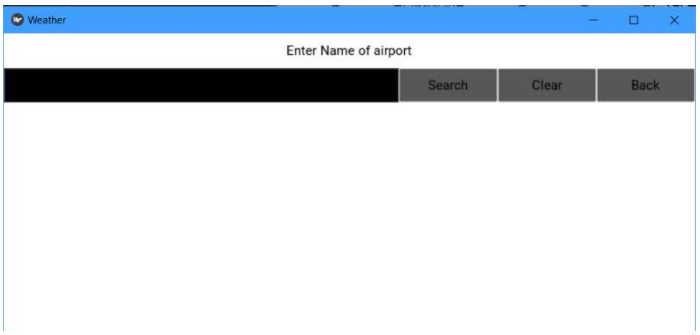
And at the end of the procedure's we pass so the procedure ends and the program can load.

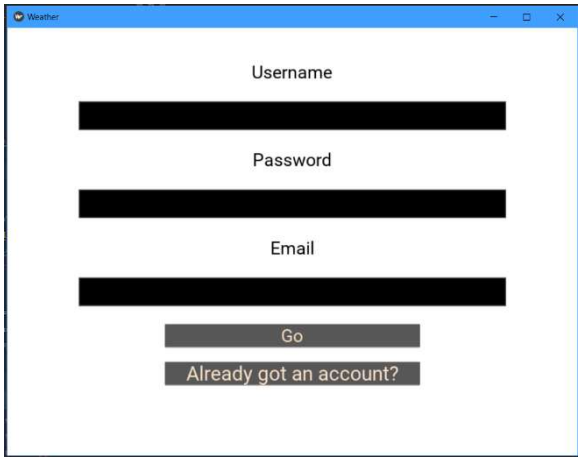
```
            try:
                box_layout = screen.children[1]  # BoxLayout instance
                for child in box_layout.children:  # children of box_layout
                    if isinstance(child, TextInput):  # verify that the child is a TextInput
                        child.background_color = (0, 0, 0, 1)
                        child.foreground_color = (1, 1, 1, 1)
                    elif isinstance(child, Label):
                        child.color = (0, 0, 0, 1)
            except:
                pass
    pass
```

Time to test it:

Test number	What are we testing for	Expected result	Test data	Results	Notes
44	Checking if the colours have changed for light mode	The colours should be changed to the ones set out in the plan		Failure See screenshots below	The airport name in 1.3 should be changed as it is not visible. The title text in 1.3 is not visible and the button colours in 1.3 are not correct







Fixing this is quite easy. We add another boxlayout in our try except loop as so:

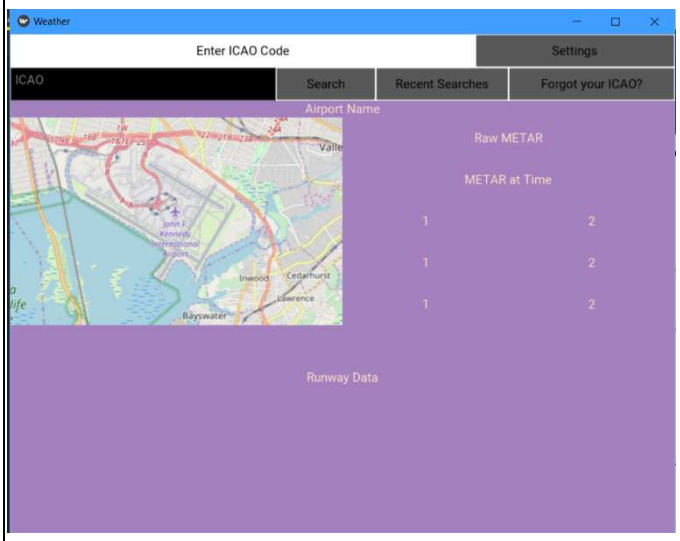
```
box_layout = screen.children[2]  # BoxLayout instance
for child in box_layout.children:  # children of box_layout
    if isinstance(child, TextInput):  # verify that the child is a TextInput
        child.background_color = (0, 0, 0, 1)
        child.foreground_color = (1, 1, 1, 1)
    elif isinstance(child, Label):
        child.color = (0, 0, 0, 1)
```

```
BoxLayout:
    orientation: "vertical"
    Label:
        id: airport
        text: "Airport Name"
        color: 1, 0.9, 0.8, 1
        height: "20dp"
        size_hint_y: None
```

Then we manually change the button colours to blanched almond

Finally we incorporate the airport name into the boxlayout as so:

Time to test it again

Test number	What are we testing for	Expected result	Test data	Results	Notes
44	Checking if the colours have changed for light mode	The colours should be changed to the ones set out in the plan		Success: 	

That's the end of the backend and so the end of the version.

End of Version Review

What has been done

- A working register/login system has been made which meets our requirements.
- There are buttons linking the 2
- We have tried out home text colour which can be used later
- The user can search for METAR, getting the latest accurate METAR available.
- The user can access the recent searches currently stored on the system.
- There is a button which links to the ICAO Finder
- The ICAO finder provides offline and accurate searching for ICAO codes based on input.
- The search can be cleared
- The user can return to the METAR searches with the click of a button.
- The recent search info is updated each search
- The output of the METAR search is formatted clearly and has extra info.
- Create a settings system which has different colour schemes and can be accessed by the user.

What I need to do

- There is some form of TTS system available for users
- For the app to be free if it is to be distributed.

Meeting with shareholders

When I met with the shareholders, I asked them the following questions:

1) What do you think of the app so far?
2) Is there a need for a TTS system, if no then is the development completed now?
3) What about distribution?

Where I got the following answers

Q1

Geoffrey – It seems pretty good to me, I like how you have fiddled about with the opacity to make the purple available for both themes.

Joseph – A agree with Geoffrey here, this app seems like it can beat the competition ok. Maybe I could add a donation system in the app once I get it.

Q2

Geoffrey – Not really, I wouldn't quite think that there are blind pilots out there even enthusiast, so the development seem alright to me. But perhaps we could add the TTS in the future sometime.

Joseph – The app seems done as well, I have tested it and it seems to work perfectly for me, I woun.t think that there would be any blind pilots but I do agree with adding it to future versions too. The optional goals were not really needed and if they are in the future then I'll add them in myself.

Q3

Geoffrey – I can handle the distribution side myself so the app is done, I guess. I have decided that we should make it paid as the app seems to beat all the paid alternatives, however I will discuss it with Joseph separately.

Joseph – I will also handle the handle the distribution, I know that Kivy has great distribution mechanisms. However, I will try to keep it free as I would certainly want such an app to be free. But we should have some donation system to help us out. Hopefully, me and Geoffrey can come to a compromise.

Section 4 – Evaluation

Criteria Met

Register System:

ID	Criteria	Met (Y/N)
1	A register system	Y
2	Appropriate validation of the inputs	Y

3	The data to be kept secure e.g.: hashing passwords with salt.	Y
4	The user's data is set up in the correct format (TBD)	Y
5	Error messages if unsuccessful	Y
6	Success message	Y

Login System

Id	Criteria	Met (Y/N)
7	A login system	Y
8	Appropriate validation of the inputs	Y
9	Matching the hashes	Y
10	Error messages if unsuccessful	Y
11	On success, go to the main screen (TBD)	Y
12	On success also save their data as a global variable until the program shuts down.	Y

The metar search system.

Id	Criteria	Met (Y/N)
13	METAR search system	Y
14	A recent search system	Y
15	Backend code which searches for the METAR using an API and returns the results	Y
16	The results to be presented in a clear format	Y
17	A way for the user to search for the ICAO code	Y

The ICAO search system

ID	Criteria	Met (Y/N)
18	ICAO search system	Y
19	Search results to be clear and accurate	Y
20	A way to navigate back to the METAR search system	Y

General stuff

ID	Criteria	Met (Y/N)
21	Dark mode and light mode	Y
22	A TTS system	N

23	If the app is disturbed using windows store, app store, or play store the app must be kept free	N

Usability Features

1) The user can search for METAR, getting the main data, a map which shows them the airport and runway data.
2) Recent search system saved according to user
3) Themes can be changed easily
4) The user can search for the ICAO by airport name.

   All evidence of this has been provided in the dev section.

Limitations

The biggest limitation is the API. From January, it will only allow a certain number of requests/day which means that the shareholders will probably have to get an upgraded API which has to be paid for so they may need to seem revenue for it from somewhere e.g. donations.

The app doesn't have a TTS system, this is because the shareholders have decided that it will not benefit many users is any at all so it is not worth the time for developing a TTS system.

Next, additional features such as airline based themes have been left out for the same reasons listed above.

To deal with the last 2 limitations, the shareholders could add it in post dev or inform users why they have not added such features.

Maintenance

Maintenance will include:

1) Ensuring the API is working correctly and that there are no problems with it
2) Making sure the users info is stored safely on the JSON file and making sure no hackers can access it.
3) If any unknown bugs do show up then it is the role of the shareholders to do something about it.