**Weather station - Lucas Barclay**

# **Project rundown**

I am planning to build a weather forecasting site which will provide a prediction on how the weather will change throughout the next few days based on measurements from different arduino sensors and user submitted images of the sky. Most weather forecasts make use of expensive, inaccessible technology such as satellites. These satellites also require methods of reaching orbit which are extremely expensive and polluting to the environment. By using cheaper equipment and creating a community of users which submit images of their skies, we can forecast the skies with a fair amount of accuracy without the need for a number of satellites.

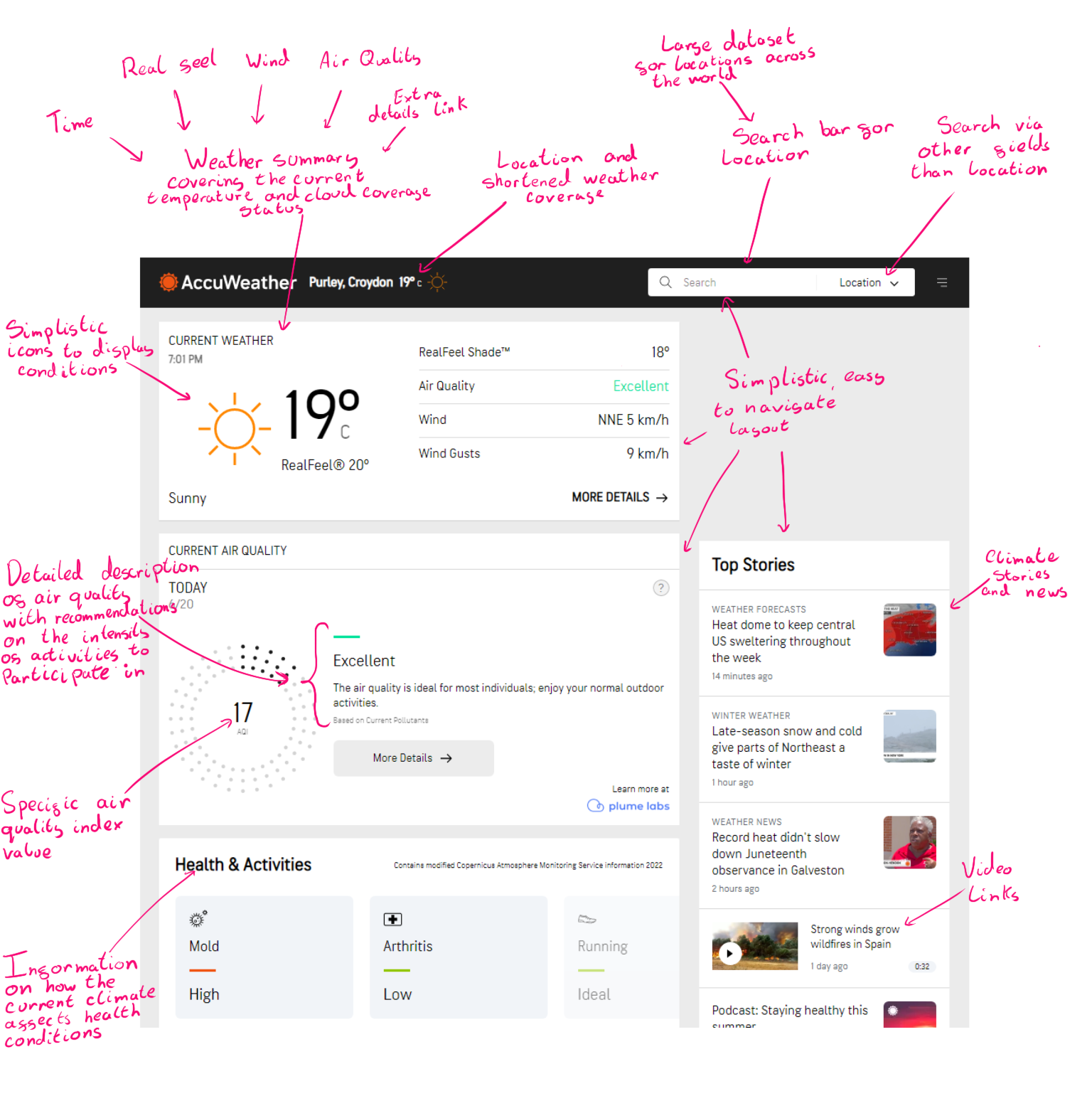
The website will aim to display hourly and daily predictions based on data gathered from the main weather station and user submitted images. A login system will be used so that users who are to submit images are more likely to provide valid images. This login system will only require the user to provide a username and password. The user’s details will be stored in a database and images which they have submitted will be associated with them. The station will not store the images it takes but rather will just store the calculated cloud cover % and exif data (in particular GPS coordinates and the timestamp).

To make the predictions, the program will use linear regression to find the line of best fit in each model (for example cloud cover against rainfall). It will also select the model which has the greatest correlation depending on the condition as a way to get a more accurate prediction. This may increase the time it takes to make a prediction with a large volume of data so the data may be split up into smaller blocks to be processed.

# **Project audience**

The UI will be aimed to be clean and simple so that users of all ages should find it easy to navigate and use. This project will be aimed at a large age group from ages 16-70, in particular commuters who need to be aware of weather conditions ahead of travel. The website will be aimed at the general audience who travel often and need to be aware of weather conditions before heading out. It will also be aimed at those who prefer to use a website or application over a TV weather forecast. This is because there will not be any kind of video forecast as part of the website, the website should be mostly autonomous apart from the aid from user submitted images.

# **Similar projects**

**Accuweather** - Accuweather is a weather forecasting application/site which provides a bountiful of features ranging from minutecast to suggesting activities for the current conditions. The minutecast is one of accuweather’s most powerful features as it provides a highly accurate prediction of the cloud coverage and rainfall within the next hour. However, after this hour its accuracy begins to decrease but can still hold up fairly strongly. It also monitors the air quality to suggest whether people should or shouldn’t participate in highly intensive activities, radar to see the cloud cover and rainfall across a wide area and summary for the weather for each day. 

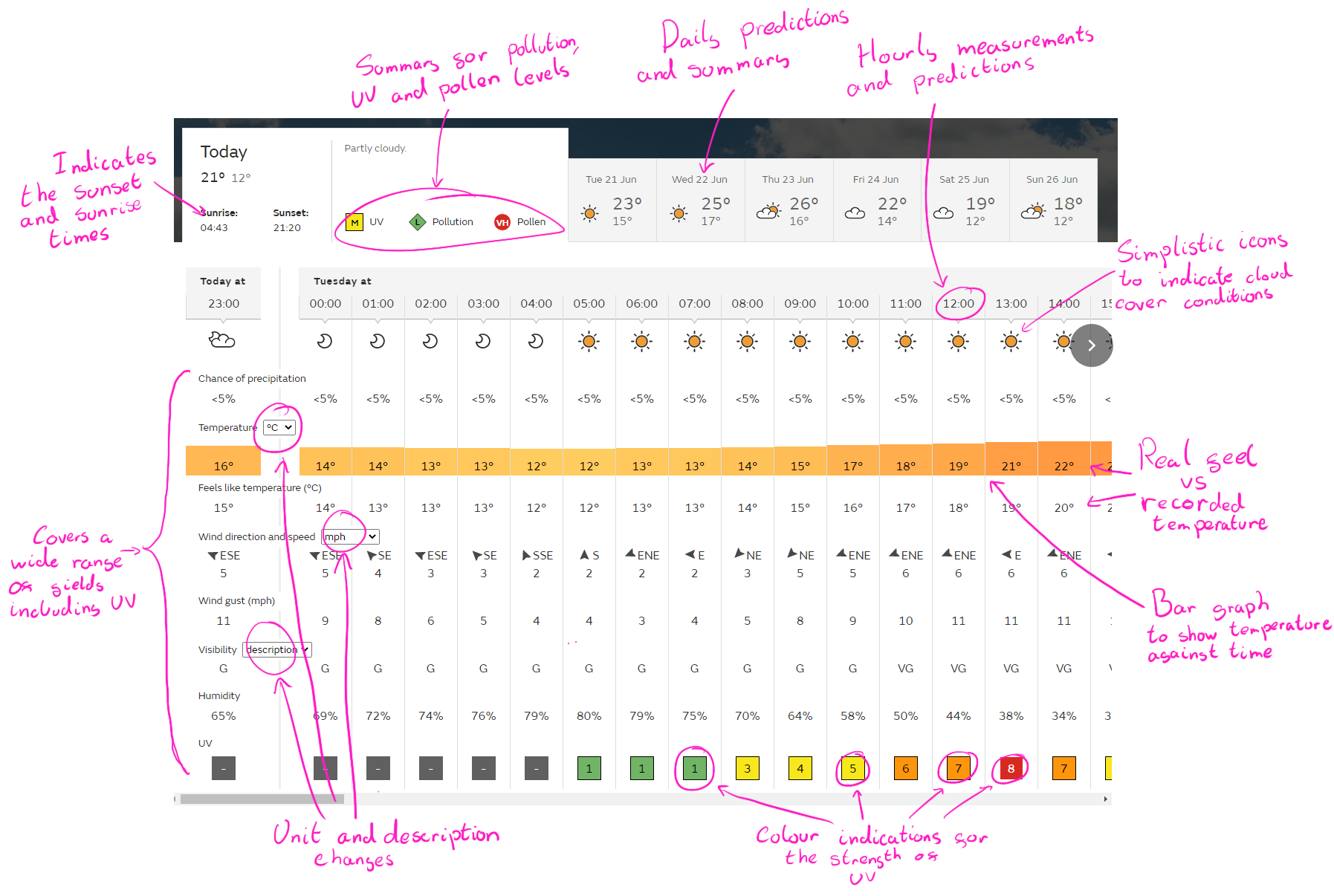
Accuweather appears to provide a wide range of features and measurements, ideal for users worldwide. The user interface of the website is also very friendly for a wide range of ages. However, the application version could be seen as a little cluttered and messy to older users.

**Advantages**

* Simplistic and easy to use website
* Covers a wide range of measurements
* Provides suggestions on the intensity of activities to participate in based on air quality
* Climate stories and news reports
* Accurate and precise minutecast within the first hour

**Disadvantages**

* Slightly confusing UI for the application for older users
* Decreased minutecast accuracy after an hour

**The met office** - The met office is another popular choice, similar to that of accuweather with a lot of popular features such as daily summaries, hourly summaries and a very accurate prediction system with 92.5% of their next day temperature forecasts being within 2oC. One of the more interesting aspects of the met office site which is rare to find is its specialist forecasts for coast and sea, mountains and even space weather. As someone who is interested in astronomy, I appreciate this feature and find it interesting to see how big of an impact space weather has on the Earth’s weather, as well as the chance of formation for an aurora. 

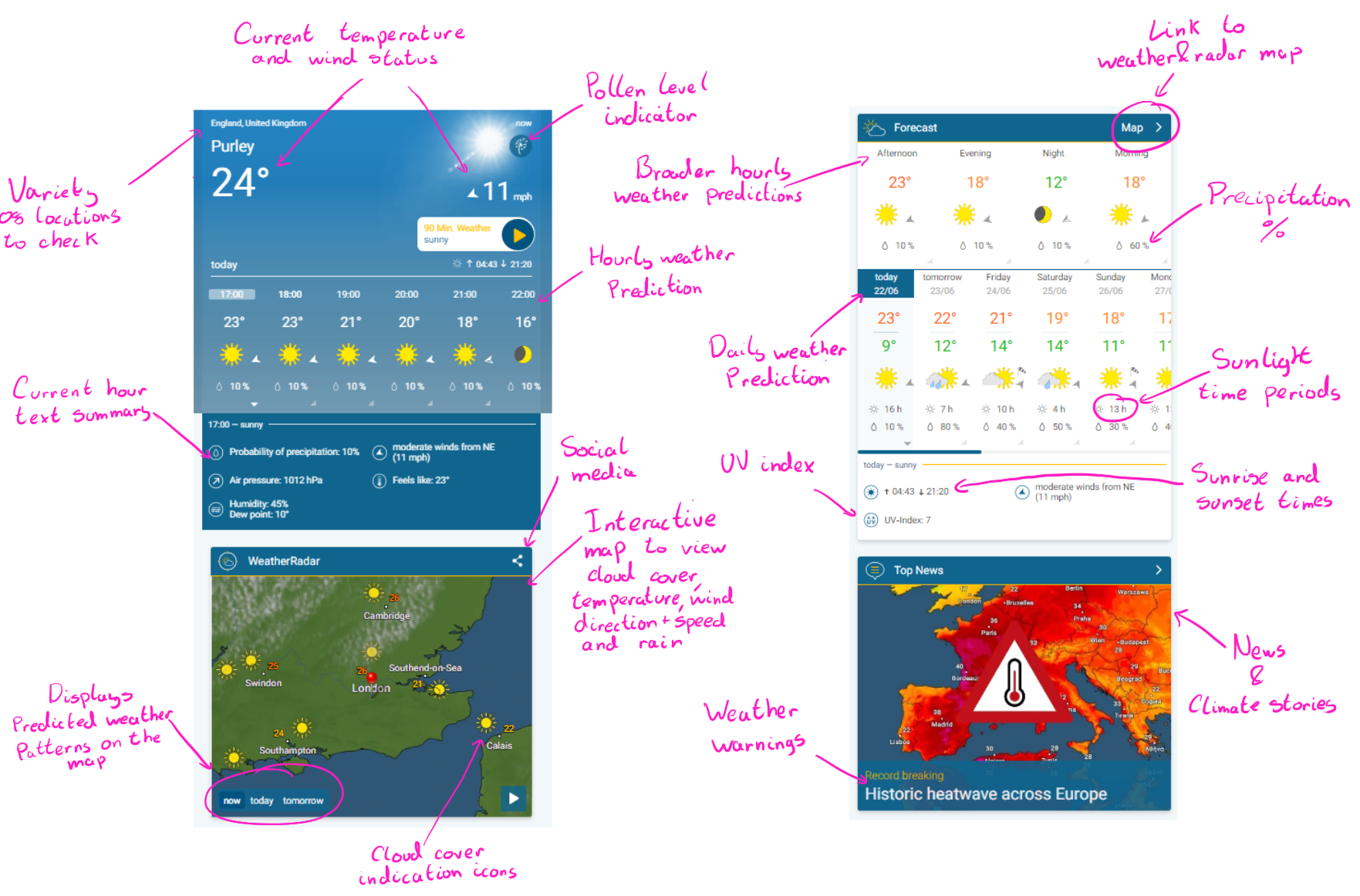
Conditions such as sunspots can greatly impact the Earth’s weather as more energy is dissipated by sunspots to the Earth’s atmosphere, thus increasing the global temperature. Access to this data makes predictions more accurate. Alternatively, to accuweather, the met office also has an hourly prediction on the UV intensity levels. UV can be very harmful and displaying this data can help people be more careful in choosing when to go outside.

**Advantages**

* Specialist forecasting
* Accurate daily temperature and wind predictions
* Hourly UV predictions
* Worldwide weather predictions

**Disadvantages**

* Slightly cluttered UI

**Weather&Radar** - Weather&Radar is a less well-known weather forecasting organization, less popular compared to that of the met office and accuweather. This is most likely due to its unoptimized website design which appears to be optimized for mobile phones but not for laptops, desktops or equivalent devices. Nonetheless it still has some strong elements such as its interactive map which displays cloud cover, wind speed and direction, temperature and rainfall. This feature is particularly useful for Astro photographers who are constantly monitoring cloud coverage conditions to see whether they have a chance to image that night or not. This visual aid is also helpful for people to assess which areas to avoid, what to wear, what time to go out, etc.. 

Similarly, to leading organizations, Weather&Radar distributes news and climate stories to make people aware of current climate issues and warnings put in effect. It also covers a wide range of areas so that most people can check how the weather is going to develop for them. However, as stated beforehand, the UI is quite unoptimized for PC’s and doesn’t look as professional compared to accuweather and the met office.

**Advantages**

* Useful interactive map for Astro photographers and the general public
* States the number of hours of sunlight predicted for each day
* Assesses the current pollen levels in the atmosphere
* Social media integration

**Disadvantages**

* Unoptimized for PC
* Less professional UI
* No specialist weather forecasting or additional features

# **Limitations**

**Limitations to public sites and applications**

Some of the current limitations with weather forecasting is the accuracy of weather predictions and the use of older models to predict the weather. The ever-changing climate makes it more challenging to predict future atmospheric conditions, particularly with global warming which is increasing global temperatures and resulting in changes in the distribution and intensity of extreme weather events. As a result, old models are not suitable in predicting future events and in some cases, new, more recent models are struggling to predict the weather.

More data is required to make more accurate predictions as old data is much less reliable. There are also limits on how far in the future weather can be predicted to a suitable level of accuracy. Furthermore, forecasts begin to get less accurate the further you are away from an observation station. These observation stations are expensive to construct and cannot be built in certain areas due to the geography of the location and human infrastructure.

Some weather sites such as BBC weather get their data from transatlantic flights and make their forecasts based on this data. However, when there are fewer flights in effect, less data can be gathered by the BBC and the accuracy of these forecasts may be much lower. During the Covid-19 pandemic, there were much fewer flights which made the BBC’s weather predictions less accurate as a result.

**Limitations to my project**

Due to time constraints, I have decided that I will be building a website, optimized for PC users as trying to build multiple optimized designs for a number of devices takes up a lot of time. Furthermore, I have settled on building a website rather than an application since it is easier to design the UI for a website using CSS than it is for an application (for example, creating applications using tkinter in python has little flexibility). Some conditions such as pollen and air quality I will be unable to measure directly using a raspberry pi or arduino as the sensors for pollen are expensive compared to the others and aren’t very accurate.

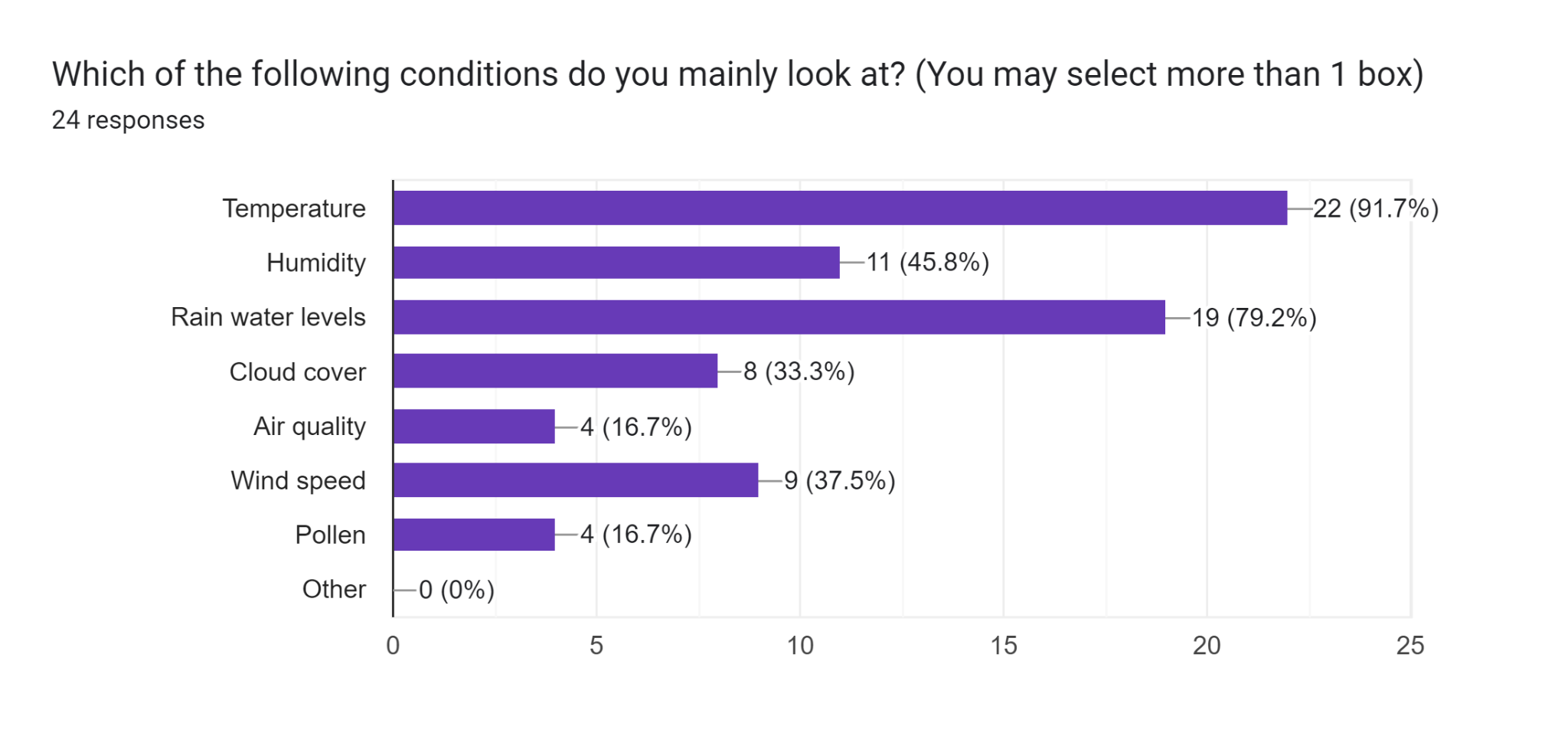
The accuracy of the model from the sensors will be fairly weak compared to other models as the database will be small (readings taken every minute). Most public datasets include records ranging across many years whereas the maximum number of records I will be able to take will only range across a couple months. As a result, the strength of the predictions may be fairly weak.

# **Analysis on survey results**

Before looking at the results of the survey I had a few hypotheses. My first thought was that those who commuted the most were more likely to check the weather as they are more likely to head outside, and travel compared to other people. After receiving the results from the google forms, it showed that 76.5% of everyday commuters checked the weather either once or twice a week or every day. From this I can assume that my program will be of use to a fairly large percentage of commuters. Of this 76.5%, 84.6% were of the ages 16-17, 7.70% were of the ages 22-30 and another 7.70% were of the ages 30 and above. It is hard to make an accurate judgment from this data as the sample size for ages 18-30+ was very small of only 6 people compared to those of the ages 16-17 which was 18.

One of the unexpected results I noticed from the survey was the percentage of people who used google weather as a preferred weather forecast given how few features it provided in comparison to accuweather and the met office. From this I can infer that most people choose the forecasting sites with the more simplistic UI’s and features over the more complicated, featureful sites and applications. To adapt to this knowledge, I will try to organize my site so that its UI is simple and easy to navigate, asking for people’s opinions during development on its design.

An overwhelmingly large percentage of the sample population said that they used a website/app over watching a forecast, a percentage of 95.8% against 4.2%. Whilst the sample size is small, it does suggest that a website appeals to a much larger population rather than a television report. The age range which was part of the 4.2% was of ages 22-30, following no correlation between increasing age and chance that a person of age would watch a television forecast. By using the data gathered, we can safely assume that a website should appeal to people of all ages.

The top 5 conditions which people tend to pay attention to when looking at a weather forecast, in descending order, were temperature, rainwater levels, humidity, wind speed and cloud cover. Fewer people were likely to use forecasts to monitor the air quality and pollen levels in the atmosphere. To try and maximize the best use of my time I will be tackling the monitoring and prediction of the top 5 conditions in order of their descending popularity (starting with temperature).

When answering the question “why do you check the weather”, most people responded with “to know whether they need an umbrella” or “to know what to wear depending on the weather”. This applied to 76.2% of the sample population who answered the question. From this, a system which suggests what kind of clothing to wear or what equipment should be taken could be implemented for ease of use and simplicity. Interestingly, one of the answers was “for my plants”, showing that some people even use weather forecasts to attend to nature according to what events were predicted to follow that day or hour.

# **Entity Relationship Diagram**

The entity relationship diagram below shows how each of the tables (Registered Users, Weather data, Spring model, summer model, autumn model, winter model) are all connected to one another. The registered users table is connected via a one-to-many relationship with weather data. Each record in weather data will contain a UserID column so that registered users can return a catalog of the data they have submitted and see a count of how many records they have associated with them. The admin weather station will be registered as a user with a user ID of 0. Ideally, additional weather stations can be set up across the country, all pushing their observational data to the database.

The data will be split up depending on their timestamp so that the prediction software can deal with smaller datasets which should be more suitable for the season at the time. A modelID will be associated with the DataID.



RegisteredUsers(UserID, Username, Password)

WeatherData(DataID, UserID, Timestamp, Temperature, Humidity, RainWater, CloudCover, GPSCoords)

Models(ModelID, DataID)

***# Temperature, Humidity, RainWater can be null***

# **Data flow**

The data flow diagram below shows how the data will be passed from weather stations and users to the database, used in prediction for the next hours and week and then displayed to the user. Images submitted by users and weather stations will undergo some processing to extract the exif data (timestamp and GPS coordinates in particular) as well as calculate the cloud cover %. This data is then appended to the database. Data from sensors such as temperature sensors, humidity, etc.. will also be appended to the database. All this data in the database will then be processed by a prediction program to work out how the weather is meant to change over time. Once calculated it will transfer this data to be displayed on the website.

Diagram

Description automatically generated

# **Data Dictionary**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Field Name** | **Field purpose** | **Field type** | **Field Size** | **Example** | **Validation** |
| Timestamp | Store datetime when data was recorded. | Date/Time | DD/MM/YY HH:MM:SS | 25/06/22 23:53:46 | Check that it is in the correct format and before the current datetime. |
| GPS coordinates | Stores the position at which the data was recorded. | Lat/Long | 10 | 047.60/-122.32 | Latitude bigger than -90, less than 90. Longitude bigger than -180, lower than 180. |
| Temperature | Stores the temperature value in oC. | Float | 4 | 19.72 | Above -273oC. |
| Humidity | Stores the percentage of humidity. | Integer | 2 | 45 | Bigger than 0, less than 100. |
| Rainwater levels | Stores the levels of rainwater in mm. | Float | 4 | 00.05 | Greater than or equal to 0. |
| Cloud cover | Stores percentage of cloud cover. | Float | 5 | 47.425 | Bigger than 0, less than 100. |
| Correlation constant | Used to check the strength of correlation in a model. | Float | 5 | 0.9576 | Bigger than 0, less than 1. |
| Blue constant threshold | Used to check whether the percentage of blue in a pixel is high enough. | Float | 3 | 0.45 | Bigger than 0, less than 1. |

# **Objectives**

## 1 Sensing the environment and transfer of data

- 1.1 Measure temperature, Humidity, Wind speed and Rainwater through arduino sensors

- 1.2 Fetch data from the sensors in python using libraries and programs such as BMP180.py

- 1.3 Store this data in JSON to be transferred to the web server

- 1.4 Store the names of the types of samples to be transferred in the same JSON (e.g temperature, humidity, etc..)

- 1.5 Store a timestamp in the same JSON which indicates when the samples were made with the format YYYY-MM-DD HH:MM:SS

- 1.6 Attempt to send the JSON to the web server through a HTTP POST

- 1.7 If the JSON is unable to be transferred it should be stored in a buffer.txt file

- 1.8 When JSON is able to be transferred, send all contents in the buffer.txt file

- 2 Receiving data from weather stations

- 2.1 Check if the station’s details are registered in the database and deny the store of the data if it is not registered

- 2.1.1 Use an SQL SELECT statement to COUNT the number of stations in the RegisteredDevices table with the details transferred in the JSON

- 2.1.2 If this number is equal to 1, then allow for the samples to be stored in the database

- 2.2 Loop through the types of samples transferred in the JSON and add each sample to the Samples table

- 2.2.1 Get the ID of the sample type from the SampleType table with a SELECT sqlite statement

- 2.2.2 Get the ID of the location from the RegisteredDevices table with a SELECT sqlite statement where the Name is equal to the stationName transferred in the JSON

- 2.2.3 Add the sample to the Samples table with an INSERT sqlite statement, coving the stationID, typeID, locationID, timestamp and the data itself

- 2.3 Check if the sample type is ‘media’ and if so, use an algorithm for measuring the % cloud cover in the image

- 3 Image analysis for % cloud cover

- 3.1 Using the pillow library, convert the image into an array of pixels, of which each element in the 2D array contains a tuple of the red, green and blue values in the pixel

- 3.2 Scan across the image in a linear fashion (row by row) and classify each pixel as either a cloudy or clear pixel

- 3.2.1 On each pixel calculate the percentage blue in the image by dividing the blue value of the pixel by the total value of all colors

- 3.2.2 If this percentage of blue is above a certain threshold, it can be classed as a clear pixel which adds a count of one to the image’s clear pixel counter

- 3.2.3 If this percentage of blue is below the threshold, it is classed as a cloudy pixel, adding a count of one to the image’s cloudy pixel counter

- 3.3 Once scanned through all pixels in the image, the total % cloud cover in the image can be calculated by dividing the clear pixel counter by the total number of pixels in the image

- 3.4 The condition can then be determined as cloudy, slightly cloudy or overcast depending on what the % cloud cover is

## 4 Preparing the dataset for prediction algorithms

* 4.1 Using the SELECT sqlite statement, get all samples with the corresponding sample type ID and location ID for which location the user is checking and the type ID currently selected in the loop
* 4.2 Loop through each of the records which have been fetched using the sqlite statement and, using the datetime library, work out the difference between the time the user accesses the page and the timestamp on the record currently being compared in the loop and store this as delta time
* 4.3 Check if the delta time is below the specified period and store the data and timestamp of the record in two separate arrays if this is true
* 4.4 Check if there are any records with the same timestamp and if so, work out the average value of the two
* 5 Linear regression for minutecast
* 5.1 Prepare the dataset using the algorithm outlined in stage 4 using data from the last hour
* 5.2 If there is only one value in the dataset, set the gradient to 0 and the y intercept should be equal to that one value
* 5.3 Otherwise, calculate the mean of the data and the mean of the difference in time between the current time and the timestamp on the data
* 5.4 Use the equation sigma (data \* delta time) – n \* mean data \* mean delta time where n is the length of the array of data and store the value in a variable named XY
* 5.5 Use the same equation but change sigma (data \* delta time) to (data \* data) and store the value in a separate variable called XX
* 5.6 Calculate the gradient which best fits the data by dividing XY by XX, storing it in a variable
* 5.7 Calculate the y intercept of the line by subtracting the gradient \* mean data from mean time
* 5.8 Loop through from values 1-60 and substitute i\*60 into the equation y=mx+c where m is the calculated gradient and c is the calculated y intercept, storing the results in an array
* 5.9 Create a timestamp for each predicted value in the array
* 6 Time series forecasting for hourly and daily predictions
* 6.1 If a model exists for the sample type and location, use that model in producing the forecast
* 6.2 Load the .pkl model using the statsmodels library where its name equals model-sampleTypeId-LocationID.pkl
* 6.3 Using the statsmodels library, use the forecast function on the loaded model to produce a prediction of the next 24 hours (hourly) or 7 days (daily)
* 6.4 If no model is present, prepare the dataset using the algorithm outline in stage 4, using data from the last month
* 6.5 If there are gaps in the data, substitute in the last value using the pandas library (e.g 1.7,..,..,..,2 would be changed to 1.7,1.7,1.7,1.7,2)
* 6.6 Train the SARIMAX model to the data fetched from the last month using the statsmodels library, ensuring to use the best order and seasonal order values
* 6.7 Save the model as a .pkl file named model-sampleTypeId-LocationID.pkl
* 6.8 Make a prediction of the next 24 hours (hourly) or 7 days (daily) using the forecast function
* 6.9 Store the results of the forecast function in an array

## 7 Website

* 7.1 Forecast site home page
* 7.1.1 Create a list of all locations (locationID and locationName) using a SELECT sqlite statement and store them in a list ordered by the number of samples per location.
* 7.1.2 Store these values in JSON to be used when the site is loaded
* 7.1.3 Using the google maps API, display all the locations registered as points on a heatmap where the intensity depends on the cloud cover % in the region
* 7.2 Login page
* 7.2.1 When the user clicks the submit key, all details should be fetched from the form and stored in appropriately named variables
* 7.2.2 Register – Duplicates of the username and email should be checked for on the database using a SELECT COUNT statement; if this value is 0 then the user can be added to the database
* 7.2.3 Register – The password of the user should be salted and hashed using the SHA256 hashing algorithm for added security
* 7.2.4 Register – Add the user to the database through an INSERT sqlite statement
* 7.2.5 Register – A verification email should be sent to the user’s email address using the SMTP protocol, containing a link which updates the user’s verification state from 0 to 1 using an UPDATE sqlite statement
* 7.2.6 Login – Check that the username or password field isn’t empty when the submit button is clicked
* 7.2.7 Login – Check that the user is registered using a SELECT COUNT sqlite statement; if this value is 1 then the user may proceed with the login
* 7.2.8 Login – Check that the user’s email has been verified by getting their validation state through a SELECT sqlite statement
* 7.2.9 – Redirect the user to their user page which contains their image count (number of images they have submitted
* 7.3 User page
* 7.3.1 Allow for the user to upload images through a file upload input, storing the image with its timestamp and passing it through the % cloud cover algorithm in stage 3
* 7.3.2 Add a sample to the database containing the cloud cover % of the image using an INSERT sqlite statement
* 7.3.3 Allow the user to add a new location by giving the name of the location and its latitude/longitude in a form
* 7.3.4 Store this new location in the database using an INSERT sqlite statement
* 7.3.5 Provide an option to create a station where its name depends on the location of which the user is measuring for and the time at which the station is registered
* 7.3.6 Create a station API key, a hash of the station name
* 7.3.7 Add this station and it’s details to the RegisteredDevices table through an INSERT sqlite statement
* 7.4 Location pages/hourly and daily predictions
* 7.4.1 Display the location name, a table and graph of the last 60 minutes of data and the next 60 minutes of data and a summary of the data
* 7.4.2 Display the data in the JSON on the graph using the google graphs API
* 7.4.3 Display daily/hourly predictions on graphs on separate pages

**Programming language key**

* ***Python***
* ***Python + HTML + Javascript + CSS***
* ***Sqlite + Python***
* ***Python + Arduino***

**Design**

**Database design**

This database design aims to improve extensibility by identifying the different sample types via ID’s and creating a new record for each sample type uploaded, rather than storing each sample type in a column. This allows for other sample types to be added easily to the “SampleType” table where sample types are identified by a “TypeID”. This “TypeID” will be used in the “Samples” table to identify what the sample values are referring to. The “Locations” table is used to store a list of location names and their GPS co-ordinates to be used for geofencing and identifying which sample belongs to which location for analysis. It is also used to identify the home location of users and their own weather stations. The “RegisteredDevices” table will store the name, password and “LocationID” of all users and weather stations. The “DeviceID” is used in the samples table, this allows users to view a count of how many submissions they have made as well as their weather station. The “Type” column identifies whether the record contains information of a user or weather station. Finally, the “Samples” table, being where all samples are stored and read from to be used in forecasting. I decided to make a compromise so that I can improve extensibility. There will be many more records made in the “Samples” table, however, it allows for new data types to be added with ease. May a weather station add a new sensor such as a UV sensor and wish to upload the data, a new sample type can be added to the “SampleType” table to be used in the “Samples” table. The following shows the structure of each table:

Locations(*LocationID INTEGER,* LocationName TEXT, Latitude FLOAT, Longitude FLOAT)

RegisteredDevices(*DeviceID INTEGER*, LocationID INTEGER, Type TEXT (user or station), Name TEXT, Password TEXT)

SampleType(*TypeID INTEGER*, TypeName TEXT, Units TEXT)

Samples(*SampleID INTEGER*, DeviceID INTEGER, TypeID INTEGER, LocationID INTEGER, Timestamp TEXT, Value FLOAT)

Diagram

Description automatically generated

**Overall system design**

The overall system will take inputs both from weather stations (raspberry pi’s sending sensory data via the request library) and users who are registering themselves, setting their home location or submitting images for analysis. There will also be data transferred between the backend web server and the front-end website in HTML and JavaScript.

|  |  |  |  |
| --- | --- | --- | --- |
| **Inputs** | **Processes** | **Storage** | **Outputs** |
| User details  User image submission | Image count per location.  Cloud cover analysis.  Greatest contributor.  Set home location. | Database tables:  RegisteredDevices  Samples  Locations | User profile  Cloud cover %  Contributor leaderboard  Samples map |
| Hourly/Daily forecast request | Grab training data from table.  Create time series weather forecast using machine learning with statsmodels library.  Display forecast in google charts (JSS). | Database tables:  Samples  Locations  SampleType | Google charts time series graph |
| MinuteCast request | Linear regression.  Create timestamps for forecast.  Create a dictionary in JSON. | Database tables:  Samples  SampleType  Locations | Google charts time series graph  Adlibs forecast summary |
| Location details | Post location details using a form.  Insert location into Locations table. | Database tables:  Locations | Confirmed location insert |
| Station details  Check station connection  Send samples | Create a hash code for the name and pass.  Insert the new station name and pass into Registered devices.  Create a backlog of samples when web server is offline.  Request to send samples when web server is online every minute/hour.  Check that the station is registered. | Database tables:  Locations  RegisteredDevices  Samples  SampleType | Confirm connection  Add sample to table  Add station to registered devices |
|  |  |  |  |

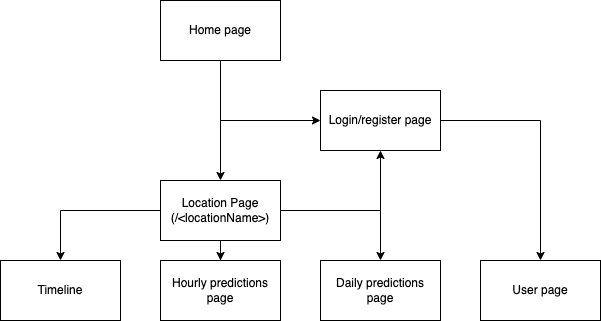
**Modular design**

The following diagram shows how the main modules and functions in forecasting site communicate with one another and process data to be displayed on the site. It also shows how each webpage links with one another and the functions associated with the pages.

Diagram

Description automatically generated

**Page navigation**



**Validation**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Validation check** | **Description** | **Applicable fields** | **Valid data** | **Erroneous data** |
| Duplicate | Check that there are no duplicate values. | Usernames | “Bob”, “Dave” | “Bob”, “Bob” |
| Datatype | Makes sure that the data being input is only of one type, such as an integer or string. | Temperature | 12.5 | “1oo” |
| Range | Checks that data is within a range of two values, for example, temperature should be above –273 and below 1000 (this covers extremes, with a lot of leeway). | Temperature | 25.0 | -800 |
| Length | Checks that the length of values is valid for the task, for example, when carrying out linear regression, more than 1 data point is needed. | Temperature(list) | [10.1,11.5,12.3,13.5,14.1] | [] |
| Connection | Checks that data can be sent to the web server. | Temperature | “Post request received (Temperature = 25.5)” | Could not send to server |
| Is empty | Checks whether the data is null/empty and therefore has no use. | Sqlite SELECT statement results | ([1,2],) | ([]) |

**Validation checks on data**

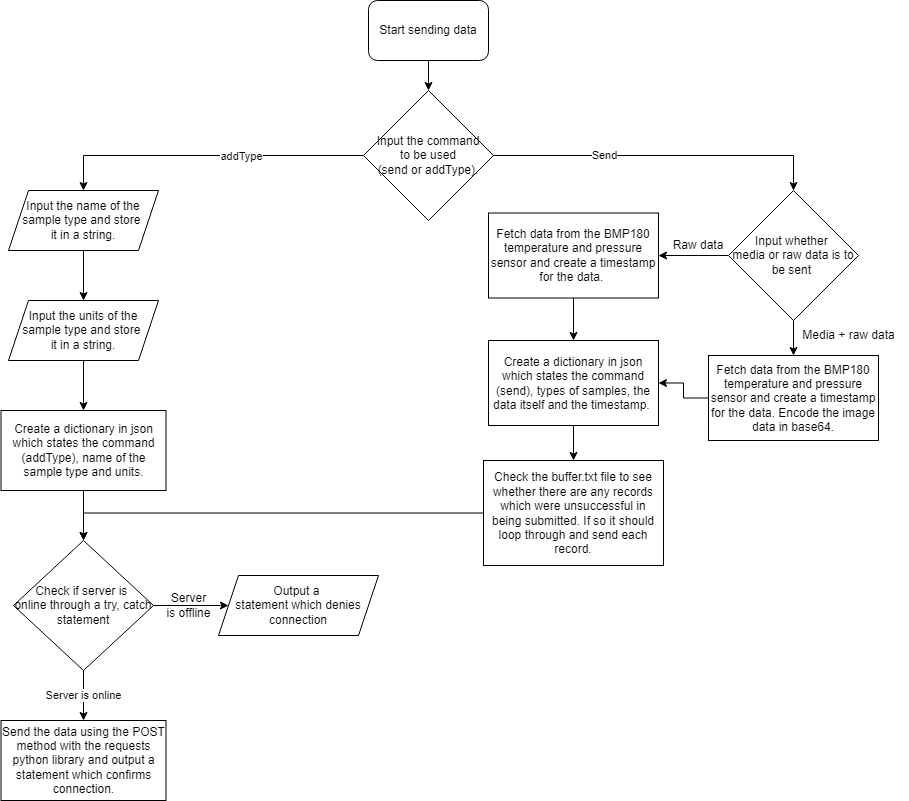
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Field name** | **Validation checks** | **Description** | **Error message** | **Data** | **Caught** |
| Temperature | Range, Datatype | Only allow floats, doubles within –273 to 1000 degrees. | Extreme value detected, ignore value. | -99999 | Yes |
| Pressureh | Range,Datatype | Only allow floats, doubles within 500 to 2000hPa. | Extreme value detected, ignore value | 100 | Yes |
| Sqlite SELECT statement results |  |  |  |  | ([]) |

**Data dictionary**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Field Name** | **Field purpose** | **Field type** | **Field Size** | **Example** | **Validation** |
| Timestamp | Store datetime when data was recorded. | Date/Time | DD/MM/YY HH:MM:SS | 25/06/22 23:53:46 | Check that it is in the correct format and before the current datetime. |
| GPS coordinates | Stores the position at which the data was recorded. | Lat/Long | 10 | 047.60/-122.32 | Latitude bigger than -90, less than 90. Longitude bigger than -180, lower than 180, should both be floats. |
| Temperature | Stores the temperature value in oC. | Float | 4 | 19.72 | Above –273oC and below 1000 oC, should be a float. |
| Humidity | Stores the percentage of humidity. | Integer | 2 | 45 | Bigger than 0, less than 100. |
| Pressure | Stores the pressure in kPa. | Float | 6 | 1006.34 | Bigger than half an atmosphere (500hPa) and less than double an atmosphere (2000hPa), should be a float. |
| Rainwater levels | Stores the levels of rainwater in mm. | Float | 4 | 00.05 | Greater than or equal to 0, should be a float. |
| Cloud cover | Stores percentage of cloud cover. | Float | 5 | 47.425 | Bigger than 0, less than 100, should be a float. |
| Correlation constant | Used to check the strength of correlation in a model. | Float | 5 | 0.9576 | Bigger than 0, less than 1, should be a float. |
| Blue constant threshold | Used to check whether the percentage of blue in a pixel is high enough. | Float | 3 | 0.45 | Bigger than 0, less than 1, should be a float |
| Cur | Used to make edits and fetch data from the sqlite3 database. | Sqlite cursor | Null | Sqlite.cursor object | Make sure that the same cursor isn’t trying to be used simultaneously. |
| Sample count | Stores the number of samples submitted by the user. | Integer | Infinite | 20 | Check that the user is registered, should |
| Condition | Determines the condition depending on the percentage cloud cover. | String | 15 | “Slightly cloudy” | Check that the input percentage isn’t too extreme such as 100%. |
| R, G, B | Holds the value for the red, green and blue value for a specific value. | Tuple (split into integers) | (3,3,3) | (187,50,90) | Check the colourspace to determine the range of values for red, green and blue (naturally this will range from 0 to 255). |
| Reference image | A pillow.image object used to store the image. | Pillow.image object | Null | Pillow.image object | Check that the route to the image is valid. |
| Image loader | A pillow.image.load object used for manipulating the image. | Pillow.image.load object | Null | Pillow.image.load object | No validation required. |
| Total cloud | An integer used to store the number of “cloudy” pixels. | Integer | Infinite | 568 | Check that the value is above 0. |
| Total clear | An integer used to store the number of “clear” pixels (predominantly blue pixels). | Integer | Infiinite | 1070 | Check that the value is above 0. |
| Cloud cover percentage | A float used to store the percentage of cloudy pixels in the image. | Float | 4 | 78.89 | Check that the value is below 100 and above 0. |
| X and Y maximum | Integers used to store the resolution of the image (length in pixels in the x and y axis). | Integer | Infinite | 1920,1080 | Add constraints on resolution so that the device running the web server isn’t put under stress when searching through the image. |
| Total pixel value | Stores the total value of the red, green and blue in one variable. | Integer | Commonly up to 3 | 407 | Check that value is above 0. |
| X and Y temporary value | Stores the current x and y coordinates of the pixel to be analyzed. | Integer | Infinite | 90,50 | Check that the x and y temporary values are below that of the resolution and above 0. |
| Condition | Stores the condition in a string depending on the % cloud cover. | enum | Varchar(15) | Slightly Cloudy | Check that the % is below 100 and above 0. |
|  |  |  |  |  |  |

**Data transfer program (raspberry pi to web server)**

The following pseudocode is for the program I made for sending data from the raspberry pi weather station to the web server. It handles creating the packets of data to be sent whether it be sensor data or encoded image data stored in a string. It also handles storing unsent data in a buffer.txt file to be sent when a successful connection is established. I made use of the requests, json, time, datetime, os and base64 libraries, as well as a separate program called bmp180.py. The bmp180.py program is used to get the data from the bmp180 pressure and temperature sensor. I had troubles trying to use the adafruit library for this and decided to make use of this program found on github.



Import following libraries

Requests, json, bmp180, time, datetime, os, base64

Url = “http://192.168.1.72:5000/DataReceiver”

Location = “PURLEY”

StationName = “STATION\_NAME”

StationKey = “STATION\_KEY”

Function get media

Os.system(‘libcamera-jpeg –o cloudImage.jpg --shutter 2000 –width 640 –height 480’)

# Line above runs the command in the parameter in command line. It takes an image across 2000ms with a resolution of 640\*480.

Img = openRead(‘cloudImage.jpg’, ‘rb’)

ImgBytes = img.read()

ImgEncoded = base64.b64encode(imgBytes).decode(‘utf8’)

# Lines above are used to read the image file and encode it in base 64 before decoding again in ‘utf8’ so that the image data can be stored in a string and processed on receiving.

Endfunction

Function write to buffer(pass in packet)

Buffer = openRead(‘buffer.txt’, ‘a’)

Buffer.write(packet)

# Writes the unsent data to a buffer so that when it can ensure a successful connection, it can resend the unsent data.

Endfunction

Function send data(pass in packet, url)

Try send data to the web server

jsonData = requests.post(url, json=packet)

# Tries to send a bit of json (the data) using the requests library. It uses the post method as we are sending data.

Ouput jsonData.text

return True

# Outputs the confirmed response from the web server and returns True from the function.

Otherwise if data cannot be sent

Output “Server is not online”

write to buffer(packet)

return False

# Gives indication that the server is offline and returns False from the function.

Endfunction

Function send buffer(pass in url)

Buffer = openRead(‘buffer.txt’, ‘r+’)

Packets = read all lines from buffer

# Reads all lines of the buffer.txt file and puts it in a list

If length of queries is greater than 0

count = 0

For i=0 To length of array queries

packet = replace all ‘ with “ and replace () with [] in querries[i][0:length of queries-1]

# Puts the packet into a readable format for json (string to json)

if sendData(json.loads(packet), url) or packets[i] is empty then

# Will attempt to send data to the web server, if it is successful count increases by 1

count += 1

Endif

time.sleep(0.0001)

# Adds a slight delay between sending the packets in case any issues arise

Endfor

If count = length of packets array then

# Checks to see that all packets were send before clearing out the buffer

clear buffer

Endif

Output “Sent (count) packets and missed (length of packets - count) packets”

Endif

Close buffer.txt

Endfunction

Command = input which command to be used (send or addType)

If command = “send” then

SampleTypes = input which samples are being sent (floats or both media and floats)

If sampleTypes = “both” then

timestamp = dt.datetime.now().strftime(“%Y-%m-%d, %H:%M:%S”)

# Creates the timestamp at which the data has been recorded in a string

temperature, pressure = bmp180.readBmp180()

# Gets the data from the bmp180 pressure and temperature sensor using a seperate program (the bmp180.py program was not written by me)

img = function getMedia()

packet = {"command": "send", "types": ("temperature", "pressure", "media"), "data": ({"temperature": temperature, "pressure": pressure, "media": (img)}), "login": ({"stationName": stationName, "stationKey": stationKey}), "timestamp": timestamp}

# Creates the packet in a dictionary to be received as json at the web server

if sendData(packet, url) then

sendBuffer(url)

# Tries to send the current packet and if succesful, will try to send the buffer

Endif

time.sleep(60)

Else if sampleTypes = “floats” then

timestamp = dt.datetime.now().strftime(“%Y-%m-%d, %H:%M:%S”)

temperature, pressure = bmp180.readBmp180()

packet = {"command": "send", "types": ("temperature", "pressure"), "data": ({"temperature": temperature, "pressure": pressure}), "login": ({"stationName": stationName, "stationKey": stationKey}), "timestamp": timestamp}

if sendData(packet, url) then

sendBuffer(url)

Endif

time.sleep(60)

Endif

Else if command = “addType”

Name = input name of the sample type

Units = input the units of the sample type

Packet = {"command": "addType", "name": name, "units": units}

Try post to the web server

json = requests.post(url, json=packet)

Output json.text

Else

Output “Server currently unavailable”

Else

Output “Invalid command”

Endif

**Image analysis algorithm for % cloud cover**

- Most weather stations use models which cover data points such as temperature, humidity, cloud cover, air pressure, wind speed, etc.. Whilst data points such as temperature, humidity, air pressure and wind speed tend to be measured from the ground, cloud cover is often monitored using satellite imagery. This makes it easy to predict which areas will have a high % of cloud cover and where rain is likely to hit. Alternatively, image analysis can be done on photos of the sky from different locations to estimate cloud cover and see how these conditions develop overtime. Diagram

Description automatically generated

These images would be taken from users across the world. The cloud cover % will be calculated via an algorithm in python. This algorithm does a linear scan (row upon row) of the image and checks the blue value in each pixel, as well as the total RGB values. The blue value is calculated as a percentage of the total RGB value. If the percentage of blue is above 40%, the program will record that pixel as a clear-sky pixel and add it to a total. Otherwise, the program records the pixel as a cloudy pixel and adds it to another total.

Finally, the algorithm calculates a percentage of clear-sky pixels against the total number of pixels to find the cloud cover in the image. The user should make sure to image nothing but the sky to get the best result out of this algorithm. All the other conditions (temperature, humidity and rainwater levels) will be monitored by a raspberry pi using the appropriate sensors. Below is an example of the cloud detection algorithm.

Chart

Description automatically generatedA blue sky with white clouds

Description automatically generated with medium confidence

**Testing**

**Testing table**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Test ID** | **Description** | **Data type** | **Test data** | **Expected results** | **Pass/fail** | **Cross reference/results** |
| 01 | Check that the user doesnt already exist | Typical | Username : Admin  Password : 12345  Check password : 12345 | - Checks if the username is a duplicate  - Returns that the username is a duplicate and user remains on login page |  |  |
| 02 | Check whether the password and check password match | Typical | Username : Bob  Password : helloWrld  Check password : hallowrld | - Compares the password against the check password  - Sees that they do not match and does not add user to the database  - User remains on login page |  |  |
| 03 | Check that the station attempting to send data is registered | Typical | Station name : | - Receives the request from the station  - Checks the station details against the database  - Accepts the data as it has been registered and more likely to be trusted |  |  |
| 04 | Check if there is a model for the location and |  |  |  |  |  |
| 05 | Check that the temperature values submitted are valid | Erroneous | Temperature : -274 | - Checks if the temperature is above –273 degrees and below 150 degrees  - Does not add the data to the database  - Sends an error message back to the station |  |  |
| 06 | Check that there are no outliers in the data | Typical | Temperature : [11,12,13,-10,11] | - Checks whether the value falls out of range of one standard deviation from the mean  - If so it won’t use that value in prediction and will continue to predict without it |  |  |
| 07 | Display a list of location pages to be accessed from the search bar | Typical | Search query : “Purley” | - Redirects user to a locations list page  - Lists all locations beginning with “Purley” |  |  |
| 08 | Redirect user to a page unique to the location selected when pressing a button on the card | Typical | Button is clicked,  Location name :  “Purley” | - Redirects user to the page for purley |  |  |
| 09 | Get all data from the last 60 minutes to be used in the linear regression algorithm | Typical | Temperature : [25.1,11.1,11.2,11.5,11.4,11.7]  Datetime :  [12:51, 00:01, 00:02, 00:03, 00:04, 00:05] | - Works out the difference in the timestamp of the data and the current time  - Checks that this difference is below that of 60 minutes  - Ignores any data which is above this difference of 60 minutes  - Adds the temperature and datetime values to two separate arrays |  |  |
| 10 | Use the data from the last 60 minutes in a linear regression algorithm which creates an equation for a linear graph | Typical | Temperature : [11.1,11.2,11.5,11.4,11.7]  Datetime :  [00:01, 00:02, 00:03, 00:04, 00:05] | - Uses this data and their timestamps to calculate a gradient and y intercept for the linear graph  - Stores the gradient of the graph and y intercept in two variables |  |  |
| 11 | Create an array for the next 60 minutes of data using the gradient and y intercept of the equation | Typical | Datetime : [00:06, 00:07, 00:08...] | - y = mx+c  - Use the gradient and y intercept values calculated beforehand and substitute them into the equation  - Substitute the minute for the next 60 minutes of data (0,1,2...)  - Calculate the y value (predicted value) for each minute in the next 60 minute  - Append the predicted data to an array |  |  |
| 12 | 3.1 Check that the image exists and, using the pillow library, convert the image into an array of pixels, of which each element in the 2D array contains a tuple of the red, green and blue values in the pixel | Typical | Image : ‘PURLEY-2022-11-07-200849.jpg’ | - Checks that the image exists  - Converts the image to a tuple using pillow  - Stores the image data to an array in the CloudCover class | Pass | Image reference No – 3.1.1 |
| 13 | 3.1 Check that the image exists and, using the pillow library, convert the image into an array of pixels, of which each element in the 2D array contains a tuple of the red, green and blue values in the pixel | Erroneous | Image : ‘ ’ | - Notices that the filename is invalid as it is empty, and no file has an identical name  - Does not allow the image to be processed  - Returns ‘None’ type for cloud cover % and condition  - Output ‘image is invalid’ | Pass | Image reference No – 3.1.2 |
| 14 | 3.2 Scan across the image in a linear fashion (row by row) and classify each pixel as either a cloudy or clear pixel | Typical | Image : ‘PURLEY-2022-11-07-200849.jpg’ <PixelAccess Object> | -Using incrementation in a for loop selects each and every pixel in the image  - Stores the total RGB values in a variable  - Divides the blue value by the total RGB value  - Checks that it is above the threshold of 0.4  - Adds 1 to either the total cloudy pixels (if below the threshold) or to the total clear pixels (if above the threshold) | Pass | Image reference No – 3.2 |
| 15 | 3.3 Once scanned through all pixels in the image, the total % cloud cover in the image can be calculated by dividing the clear pixel counter by the total number of pixels in the image | Typical | Total Clear : 809084  Total Cloudy : 660916 | - Stores the total number of pixels which is calculated from the total clear pixels + the total cloudy pixels  - Divides the total cloudy pixels by the total pixels  - Multiplies this value by 100 to calculate a percentage cloud cover for the image | Pass | Image reference No – 3.3 |
| 16 | 3.4 The condition can then be determined as cloudy, slightly cloudy or overcast depending on what the % cloud cover is | Typical | Cloud cover % : 44.96% | -Stores all different types of conditions in an enum to represent a set of conditions  - Outputs the correct enum depending on the % cloud cover (within a range of values) | Pass | Image reference No – 3.4 |
| 17 | 4.1 Using the SELECT sqlite statement, get all samples with the corresponding sample type ID and location ID for which location the user is checking and the type ID currently selected in the loop | Typical | http://127.0.0.1:5000/locationName=PURLEY | - Fetches the location name from the URL  - Loops through all typeID’s and fetches the data which matches the current typeID and locationID with a SELECT statement | Pass | Image reference No – 4.1 |
| 18 | 4.2 Loop through each of the records which have been fetched using the sqlite statement and, using the datetime library, work out the difference between the time the user accesses the page and the timestamp on the record currently being compared in the loop and store this as delta time | Typical | Current time : 2022-11-14 11:58:52.439677  Sample time : 2022-09-26 22:16:44 | - Find the timestamp of the currently selected record  - Get the current time using the datetime library, storing it in a string  - Work out the difference in time between the two different times in seconds  - Store this value as delta time | Pass | Image reference No – 4.2 |
| 19 | 4.3 Check if the delta time is below the specified period and store the data and timestamp of the record in two separate arrays if this is true | Typical | Period: 3600  Current timestamp: ‘2022-11-23 09:58:58’  Record timestamp: ‘2022-11-23, 09:24:34’  Data: 993.48 | - | Pass | Image reference No – 4.3 |
| 20 | 4.4 Check if there are any records with the same timestamp and if so, work out the average value of the two | Typical | Period: 3600  Record 1 timestamp: ‘2022-11-28, 11:24:34’  Record 2 timestamp: ‘2022-11-28, 11:24:34’  Value 1: 993.48  Value 2: 993.39 |  | Pass | Image reference No – 4.4 |
| 21 | 5.1 Prepare the dataset using the algorithm outlined in stage 4 using data from the last hour |  |  |  |  |  |
| 22 | 7.1.1 Create a list of all locations (locationID and locationName) using a SELECT sqlite statement and store them in a list | Typical |  |  | Pass | Image reference No – 7.1.1 |
| 23 | 7.1.2 Store these values in JSON to be used when the site is loaded | Typical | Location IDs: [1,3,2]  Location Name: [PURLEY, CHAMONIX, CROYDON] |  | Pass | Image reference No – 7.1.2 |
| 24 | 7.1.3 Using the google maps API, display all the locations registered as points on a heatmap where the intensity depends on the cloud cover % in the region | Typical | Image reference No – 7.1.3.1 |  | Fail | Image reference No – 7.1.3.2 |
| 25 | 7.1.3 Using the google maps API, display all the locations registered as points on a heatmap where the intensity depends on the cloud cover % in the region | Typical | Image reference No – 7.1.3.1 |  | Pass | Image reference No – 7.1.3.3 |
| 26 | 7.2.1 When the user clicks the submit key, all details should be fetched from the form and stored in appropriately named variables | Typical | Username: Bob  Password: Hello;123  Re-enter password: Hello;123  Email: lucasdbarclay@gmail.com |  | Pass | Image reference No – 7.2.1.1 |
| 27 | 7.2.1 When the user clicks the submit key, all details should be fetched from the form and stored in appropriately named variables | Typical | Username: Bob  Password: Hello;123 |  | Pass | Image reference No – 7.2.1.2 |
| 28 | 7.2.2 Register – Duplicates of the username and email should be checked for on the database using a SELECT COUNT statement; if this value is 0 then the user can be added to the database | Erroneous | Username: Bob  Password: Hello;123  Re-enter password: Hello;123  Email: lucasdbarclay@gmail.com |  | Pass | Image reference No – 7.2.2.1 |
| 29 | 7.2.2 Register – Duplicates of the username and email should be checked for on the database using a SELECT COUNT statement; if this value is 0 then the user can be added to the database | Typical | Username: obo  Password: olleh321  Re-enter password: olleh321  Email: 16BarclayL@rcmail.org.uk |  | Pass | Image reference No – 7.2.2.2 |
| 30 | 7.2.3 Register – The password of the user should be salted and hashed using the SHA256 hashing algorithm for added security  7.2.4 Register – Add the user to the database through an INSERT sqlite statement | Typical | Username: Bob  Password: Hello;123  Re-enter password: Hello;123  Email: lucasdbarclay@gmail.com |  | Pass | Image reference No – 7.2.3/7.2.4 |
| 31 | 7.2.5 Register – A verification email should be sent to the user’s email address using the SMTP protocol, containing a link which updates the user’s verification state from 0 to 1 using an UPDATE sqlite statement | Typical | Username: Bob  Password: Hello;123  Re-enter password: Hello;123  Email: lucasdbarclay@gmail.com |  | Pass | Image reference No – 7.2.5 |
| 32 | 7.2.6 Login – Check that the username or password field isn’t empty when the submit button is clicked | Typical | Username: Bob  Password: Hello;123 |  | Pass | Image reference No – 7.2.6.1 |
| 33 | 7.2.6 Login – Check that the username or password field isn’t empty when the submit button is clicked | Erroneous | Username:  Password: |  | Pass | Image reference No – 7.2.6.2 |
| 34 | 7.2.7 Login – Check that the user is registered using a SELECT COUNT sqlite statement; if this value is 1 then the user may proceed with the login | Typical | Username: Bob  Password: Hello;123 |  | Pass | Image reference No – 7.2.7.1 |
| 35 | 7.2.7 Login – Check that the user is registered using a SELECT COUNT sqlite statement; if this value is 1 then the user may proceed with the login | Typical | Username: asd  Password: dsa |  | Pass | Image reference No – 7.2.7.2 |
| 36 | 7.2.8 Login – Check that the user’s email has been verified by getting their validation state through a SELECT sqlite statement | Typical | Username: Bob  Password: Hello;123 |  | Pass | Image reference No – 7.2.8.1 |
| 37 | 7.2.8 Login – Check that the user’s email has been verified by getting their validation state through a SELECT sqlite statement | Typical | Username: Bob  Password: Hello;123  (After follwing the link in the verification email) |  | Pass | Image reference No – 7.2.8.2 |
| 38 | 7.2.9 – Redirect the user to their user page which contains their image count (number of images they have submitted | Typical | Username: Bob  Password: Hello;123  (After follwing the link in the verification email) |  | Pass | Image reference No – 7.2.9 |
| 39 | 7.3.1 Allow for the user to upload images through a file upload input, storing the image with its timestamp and passing it through the % cloud cover algorithm in stage 3 | Typical |  |  | Pass | Image reference No – 7.3.1 |
| 40 | 7.3.2 Add a sample to the database containing the cloud cover % of the image using an INSERT sqlite statement | Typical |  |  | Pass | Image reference No – 7.3.2 |
| 41 | 7.3.3 Allow the user to add a new location by giving the name of the location and its latitude/longitude in a form  (Store the name, longitude and latitude in the database) | Typical | Location name: Coulsdon  Latitude: 51.319721  Longitude: -0.140720 |  | Fail | Image reference No – 7.3.3.1 |
| 42 | 7.3.3 Allow the user to add a new location by giving the name of the location and its latitude/longitude in a form  (Store the name, longitude and latitude in the database) | Typical | Location name: Coulsdon  Latitude: 51.319721  Longitude: -0.140720 |  | Pass | Image reference No – 7.3.3.2 |
| 43 | 7.3.4 Store this new location in the database using an INSERT sqlite statement | Typical | Location name: Coulsdon  Latitude: 51.319721  Longitude: -0.140720 |  | Pass | Image reference No – 7.3.4 |
| 44 | 7.3.5 Provide an option to create a station where its name depends on the location of which the user is measuring for and the time at which the station is registered | Typical | Location name: Purley |  | Pass | Image reference No – 7.3.5 |
| 45 | 7.3.6 Create a station API key, a hash of the station name | Typical | Location name: Purley |  | Pass | Image reference No – 7.3.6 |
| 46 | 7.3.7 Add this station and its details to the RegisteredDevices table through an INSERT sqlite statement | Typical | Location name: Purley |  | Pass | Image reference No – 7.3.7 |

**Test case image reference**

|  |  |
| --- | --- |
| **3.1.1** |  |
| **3.1.2** | Text  Description automatically generated with medium confidence  Text  Description automatically generated |
| **3.2** | Text  Description automatically generated |
| **3.3** |  |
| **3.4** |  |
| **4.1** | Text  Description automatically generated |
| **4.2** | Text  Description automatically generated with medium confidence |
| **4.3** | Text  Description automatically generated with medium confidence |
| **4.4** |  |
| **7.1.1** |  |
| **7.1.2** | Graphical user interface, text, application  Description automatically generated |
| **7.1.3.1** | **Text  Description automatically generated** |
| **7.1.3.2** | **Text  Description automatically generated** |
| **7.1.3.3** | **Map  Description automatically generated** |
| **7.2.1.1** |  |
| **7.2.1.2** |  |
| **7.2.2.1** |  |
| **7.2.2.2** |  |
| **7.2.3/7.2.4** |  |
| **7.2.5** |  |
| **7.2.6.1** |  |
| **7.2.6.2** |  |
| **7.2.7.1** |  |
| **7.2.7.2** |  |
| **7.2.8.1** |  |
| **7.2.8.2** |  |
| **7.2.9** |  |
| **7.3.1** |  |
| **7.3.2** |  |
| **7.3.3.1** |  |
| **7.3.3.2** |  |
| **7.3.4** |  |
| **7.3.5** |  |
| **7.3.6** |  |
| **7.3.7** |  |

**Technical solution**

**Web server**

from Utility.Prediction import \*  
import Utility.WebServerFunctions as wf  
from flask import \*  
import os, base64, io  
import hashlib  
  
conn = sql.connect('Users.db', check\_same\_thread=False)  
cur = conn.cursor()  
app = Flask(\_\_name\_\_)  
  
# Setting up the home page on the web server  
@app.route('/')  
def home():  
 locationIDList, locationNameList = wf.getLocationIDandLocationName(conn)  
 sampleCountList = wf.getSampleCountByLocation(conn, locationIDList)  
 jsonPost = {'locationNames' : tuple(locationNameList), 'sampleCounts' : tuple(sampleCountList)}  
 return render\_template('ForecastSite.html', post=jsonPost)  
  
@app.route('/home?locationName=<locationName>')  
def locationPage(locationName):  
 locationInfo = wf.getLocationInfobyLocationName(conn, locationName)  
 stormWarning = checkStormWarning(cur, locationInfo[0][0], 3600)  
 jsonPost = {"locationName": locationName, "sampleTypes": tuple(typeNames), "stormWarning": stormWarning}  
 return render\_template('LocationTemplate.html', post=jsonPost)  
  
@app.route('/fetchData')  
def locationForecast():  
 locationName = request.args.get('locationName')  
 locationInfo = wf.getLocationInfobyLocationName(conn, locationName)  
 availableSampleTypeIds, availableSampleTypeNames = wf.getSampleInfo(conn)  
 data, time, latestValues, trendInfoList = minuteCast(locationID=locationInfo[0][0], cur=cur, sampleTypes=availableSampleTypeIds)  
 sensorDict = {}  
 for i in range(len(availableSampleTypeNames)):  
 sensorDict.update({availableSampleTypeNames[i]: {"data": data[i], "latestValue": latestValues[i],  
 "trend": trendInfoList[i], "time": time[i]}})  
 jsonData = {"data": sensorDict,  
 "location": {'locationName': locationName, 'latitude': locationInfo[1][0], 'longitude': locationInfo[1][0]}}  
 return jsonData  
  
@app.route('/fetchTimeline')  
def locationTimeline():  
 locationName = request.args.get('locationName')  
 locationInfo = wf.getLocationInfobyLocationName(conn, locationName)  
 dataList, timeList = [], []  
 availableSampleTypeIds, availableSampleTypeNames = wf.getSampleInfo(conn)  
 for i in range(len(availableSampleTypeIds)):  
 data, time = grabTimeline(locationInfo[0][0], i+1, cur)  
 dataList.append(data)  
 timeList.append(time)  
 sensorDict = wf.createDictionaryOfData(availableSampleTypeNames, dataList, timeList)  
 jsonData = {"data": sensorDict}  
 return jsonData  
  
@app.route('/timeline', methods=['POST', 'GET'])  
def timelinePage():  
 jsonPost = wf.createJsonForPageLoading(typeNames, request)  
 return render\_template('LocationTimeline.html', post=jsonPost)  
  
@app.route('/fetchMachineLearningPredictions')  
def machineLearningPredictions():  
 locationName = request.args.get('locationName')  
 period = int(request.args.get('period'))  
 periodType = request.args.get('periodType')  
 locationInfo = wf.getLocationInfobyLocationName(conn, locationName)  
 dataList, timeList = [], []  
 availableSampleTypeIds, availableSampleTypeNames = wf.getSampleInfo(conn)  
 for i in range(len(availableSampleTypeIds)):  
 data, time = machineLearning(locationInfo[0][0], i+1, cur, period, periodType)  
 dataList.append(data)  
 timeList.append(time)  
 sensorDict = wf.createDictionaryOfData(availableSampleTypeNames, dataList, timeList)  
 jsonData = {"data": sensorDict}  
 return jsonData  
  
@app.route('/hourlyPrediction', methods=['POST', 'GET'])  
def hourlyPage():  
 jsonPost = wf.createJsonForPageLoading(typeNames, request)  
 return render\_template('LocationHourly.html', post=jsonPost)  
  
@app.route('/dailyPrediction', methods=['POST', 'GET'])  
def dailyPage():  
 jsonPost = wf.createJsonForPageLoading(typeNames, request)  
 return render\_template('LocationDaily.html', post=jsonPost)  
  
@app.route('/UserPage?userDetails=<userDetails>')  
def userPage(userDetails):  
 username = userDetails.split(",")[0]  
 linkedID = wf.getLinkedAccountIDByName(conn, username)  
 if(len(linkedID) > 0):  
 stationName, stationAPI = wf.getNameAndPasswordByLinkedID(conn, linkedID)  
 userDetails = wf.makeUserDetailsString(userDetails, stationName, stationAPI)  
 else:  
 userDetails = wf.makeUserDetailsString(userDetails, 'None', 'None')  
 return render\_template('UserPage.html', info=list(userDetails.split(",")))  
  
@app.route('/LoginPage')  
def loginPage():  
 return render\_template('Login.html')  
  
@app.route('/LoginReceiver', methods=['POST', 'GET'])  
def loginRequest():  
 if request.method == 'POST':  
 username, password = wf.getNameAndPassword(request)  
 hashPassword = hashlib.sha256(password.encode()).hexdigest()  
 userID = wf.getDeviceIDByNameAndPassword(conn, username, hashPassword)  
 if userID == None or username == '' or password == '':  
 return redirect(url\_for('loginPage'))  
 else:  
 imageCount = wf.getSampleCountByDeviceID(conn, userID)  
 if imageCount > 0:  
 return redirect(url\_for('userPage', userDetails=username+','+str(imageCount)))  
 else:  
 return redirect(url\_for('userPage', userDetails=username+','+str(imageCount)))  
  
@app.route('/RegisterReceiver', methods=['POST', 'GET'])  
def registerRequest():  
 if request.method == 'POST':  
 username, password = wf.getNameAndPassword(request)  
 checkPassword = request.form['checkPassword']  
 deviceCount = wf.getDeviceCountByDeviceName(conn, username)  
 if password == checkPassword and deviceCount == 0 and username != '' and password != '':  
 hashPassword = hashlib.sha256(password.encode()).hexdigest()  
 wf.addNewDevice(conn, username, hashPassword, 'User', None)  
 return redirect(url\_for('userPage', userDetails=username+','+str(0)))  
 else:  
 return redirect(url\_for('loginPage'))  
  
@app.route('/LocationReceiver', methods=['POST', 'GET'])  
def getSelectedLocation():  
 if request.method == 'POST':  
 locationName = request.form.get('locationSelect')  
 if locationName == "none":  
 return redirect(url\_for('home'))  
 else:  
 return redirect(url\_for('locationPage', locationName=locationName))  
  
@app.route('/addLocation', methods=['POST', 'GET'])  
def addNewLocation():  
 if request.method == 'POST':  
 locationName = request.form.get('locationName')  
 try:  
 latitude, longitude = float(request.form.get('latitude')), float(request.form.get('longitude'))  
 wf.longitudeAndLatitudeValidation(conn, locationName, latitude, longitude)  
 except:  
 return "Latitude or longitude is not a valid data type (floats or integers only)"  
  
@app.route('/addStation', methods=['POST', 'GET'])  
def addNewStation():  
 userDetails = request.args.get('userDetails').split(',')  
 if request.method == 'POST':  
 locationName = request.json['location'].upper()  
 locationID = wf.getLocationInfobyLocationName(conn, locationName)[0][0]  
 if(locationID != None):  
 stationName = wf.generateStationAPIKey(conn, locationName)  
 hashPassword = hashlib.sha256(stationName.encode()).hexdigest()  
 wf.addNewDevice(conn, stationName, hashPassword, 'Station', locationID)  
 stationLinkID = wf.getIDOfStation(conn, hashPassword)  
 wf.linkIDWithStation(conn, stationLinkID, userDetails[0])  
 else:  
 return "Location not found"  
 return {"stationName": stationName, "stationPass": hashPassword}  
  
@app.route('/reportWarning', methods=['GET', 'POST'])  
def reportWarning():  
 reportTime = datetime.datetime.now()  
 reportTimeStr = reportTime.strftime('%Y-%m-%d, %H:%M:%S')  
 userDetails = request.args.get('userDetails').split(',')  
 if request.method == 'POST':  
 userInfo = wf.getLocationAndDeviceIDByName(conn, userDetails[0])  
 if userInfo[0][1] != None:  
 wf.addSamples(conn, userInfo[0][0], 5, userInfo[0][1], reportTimeStr, "True")  
 return {"message": "Report submitted"}  
 else:  
 return {"message": "No home location set"}  
  
@app.route('/imageReceiver', methods=['POST', 'GET'])  
def receiveImage():  
 if request.method == 'POST':  
 locationName = request.form['location'].upper()  
 username = request.form['hiddenUsername']  
 if locationName == "":  
 locationID = wf.getLocationAndDeviceIDByName(conn, username)[0][1]  
 if locationID == None:  
 return "Not a valid location name"  
 else:  
 locationID = wf.getLocationInfobyLocationName(conn, locationName)[0]  
 if locationID == None:  
 return "Not a valid location name"  
 locationID = locationID[0]  
 file = request.files['imageUpload']  
 setHome = False  
 if request.form.get('setHome'):  
 setHome = True  
 savePath = os.path.join("Images", datetime.datetime.now().strftime("%Y-%m-%d-%H%M%S")+".jpg")  
 file.save(savePath)  
 try:  
 #image analysis algorithm  
 percentageCover, condition, timestamp = wf.imageAnalysisSequence(savePath, True)  
 userID = wf.getDeviceIDByNameAndType(conn, username, 'User')  
 wf.addSamples(conn, userID, 4, locationID, timestamp, percentageCover)  
 if setHome:  
 wf.setHomeLocationOfDevice(conn, locationID, userID)  
 except:  
 return "Error, invalid file type"  
 return "File upload finished, info : "+str(percentageCover)+" "+condition  
  
@app.route('/DataReceiver', methods=['POST', 'GET'])  
def appendData():  
 if request.method == 'POST':  
 query = request.json  
 login = query['login']  
 name, key = login['stationName'], login['stationKey']  
 stationID, locationID = wf.getDeviceIDAndLocationIDByNameTypeAndKey(conn, name, 'Station', key)  
 if stationID != None:  
 if query['command'] == 'send':  
 data = query['data']  
 timestamp = query['timestamp']  
 sampleTypes = query['types']  
 for i in range(len(sampleTypes)):  
 if(sampleTypes[i] == 'media'):  
 imgRawB64 = data['media']  
 savePath = wf.saveAndLoadImg(imgRawB64, timestamp)  
 percentageCover, condition = wf.imageAnalysisSequence(savePath, False)  
 wf.addSamples(conn, stationID, 4, locationID, timestamp, percentageCover)  
 typeID = wf.getTypeIDFromListOfTypeNames(conn, sampleTypes, i)  
 if (typeID != None):  
 wf.addSamples(conn, stationID, typeID, locationID, timestamp, data[sampleTypes[i]])  
 return data  
 elif query['command'] == 'addType':  
 typeName, typeUnits = query['name'], query['units']  
 wf.addNewSampleType(conn, typeName, typeUnits)  
 return "Successfully added data type "+typeName+" of units "+typeUnits  
 else:  
 return "Station not registered"  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 cur.execute("SELECT TypeName FROM SampleType")  
 typeNames = cur.fetchall()  
 typeNames = wf.g.sqliteTupleToList(typeNames, 0)  
 app.run(host="0.0.0.0", debug=False)

**Web server functions**

import Utility.General as g  
import Utility.Prediction as p  
import datetime, random, os, base64, io  
from PIL import Image  
  
# Fetches all location IDs and their respective names from the Locations table  
def getLocationIDandLocationName(conn):  
 locationCursor = conn.cursor()  
 locationCursor.execute("SELECT LocationID, LocationName ID FROM Locations")  
 locationList = locationCursor.fetchall()  
 cleanLocationIDList = g.sqliteTupleToList(locationList, 0)  
 cleanLocationNameList = g.sqliteTupleToList(locationList, 1)  
 return cleanLocationIDList, cleanLocationNameList  
  
# Fetches all location information by their name  
def getLocationInfobyLocationName(conn, locationName):  
 locationCursor = conn.cursor()  
 locationCursor.execute("SELECT LocationID, Latitude, Longitude FROM Locations WHERE LocationName = ?", (locationName,))  
 locationInfo = locationCursor.fetchall()  
 locationID, locationLatitude, locationLongitude = g.sqliteTupleToList(locationInfo, 0)\  
 ,g.sqliteTupleToList(locationInfo, 1),g.sqliteTupleToList(locationInfo, 2)  
 if len(locationID) > 0:  
 return locationID, locationLatitude, locationLongitude  
 else:  
 return None, None, None  
  
# Returns the total number of samples for each location in a list of location IDs  
def getSampleCountByLocation(conn, locationIDList):  
 sampleCursor = conn.cursor()  
 sampleCountList = []  
 for i in range(len(locationIDList)):  
 sampleCursor.execute("SELECT COUNT(\*) FROM Samples WHERE LocationID=?",(locationIDList[i],))  
 sampleCount = sampleCursor.fetchall()  
 sampleCountList.append(sampleCount[0][0])  
 return sampleCountList  
  
# Returns the sample info for all samples in the table  
def getSampleInfo(conn):  
 sampleCursor = conn.cursor()  
 sampleCursor.execute("SELECT TypeID, TypeName FROM SampleType")  
 availableSampleTypes = sampleCursor.fetchall()  
 availableSampleTypeIds, availableSampleTypeNames = g.sqliteTupleToList(availableSampleTypes, 0)\  
 , g.sqliteTupleToList(availableSampleTypes, 1)  
 return availableSampleTypeIds, availableSampleTypeNames  
  
  
# Creates a dictionary of the data to be used in json  
def createDictionaryOfData(availableSampleTypeNames, data, time):  
 sensorDict = {}  
 for i in range(len(availableSampleTypeNames)):  
 sensorDict.update({availableSampleTypeNames[i]: {"data": data[i], "time": time[i]}})  
 return sensorDict  
  
# Creates a json dictionary of the location name and sample types to be used on the webpage  
def createJsonForPageLoading(typeNames, request):  
 locationName = request.args.get('locationName')  
 jsonPost = {"locationName": locationName, "sampleTypes": tuple(typeNames)}  
 return jsonPost  
  
# Returns the linked account ID by the username  
def getLinkedAccountIDByName(conn, username):  
 deviceCursor = conn.cursor()  
 deviceCursor.execute("SELECT LinkedAccountID FROM RegisteredDevices WHERE Name=?", (username,))  
 linkedID = g.sqliteTupleToList(deviceCursor.fetchall(), 0)  
 return linkedID  
  
# Returns the name and password of the linked account to the station  
def getNameAndPasswordByLinkedID(conn, linkedID):  
 deviceCursor = conn.cursor()  
 linkedID = linkedID[0]  
 deviceCursor.execute("SELECT Name,Password FROM RegisteredDevices WHERE DeviceID=?", (linkedID,))  
 stationDetails = deviceCursor.fetchall()  
 if len(stationDetails) > 0:  
 stationName, stationAPI = g.sqliteTupleToList(stationDetails, 0)[0], g.sqliteTupleToList(stationDetails, 1)[0]  
 return stationName, stationAPI  
 else:  
 return 'None', 'None'  
  
# Creates a string of the user's details for use in the user page  
def makeUserDetailsString(userDetails, stationName, stationAPI):  
 userDetails = userDetails + ',' + stationName + ',' + stationAPI  
 return userDetails  
  
# Returns the deviceID from the sqlite database by finding the matching username and password  
def getDeviceIDByNameAndPassword(conn, name, password):  
 deviceCursor = conn.cursor()  
 deviceCursor.execute("SELECT DeviceID FROM RegisteredDevices WHERE Name=? AND Password=?", (name, password))  
 deviceID = deviceCursor.fetchall()  
 deviceID = g.sqliteTupleToList(deviceID, 0)  
 if len(deviceID) > 0:  
 return deviceID[0]  
 else:  
 return None  
  
# Returns the number of samples/images contributed by a user  
def getSampleCountByDeviceID(conn, deviceID):  
 sampleCursor = conn.cursor()  
 sampleCursor.execute("SELECT COUNT(\*) FROM Samples WHERE DeviceID=?", (deviceID,))  
 imageCount = sampleCursor.fetchall()  
 imageCount = g.sqliteTupleToList(imageCount, 0)  
 return imageCount[0]  
  
# Returns the username and password from the html form  
def getNameAndPassword(request):  
 username = request.form['username']  
 password = request.form['password']  
 return username, password  
  
# Adds a new device/user to the RegisteredDevices table  
def addNewDevice(conn, name, password, type, locationID):  
 deviceCursor = conn.cursor()  
 deviceCursor.execute("INSERT INTO RegisteredDevices (Name, Password, Type, LocationID) VALUES (?, ?, ?, ?)",  
 (name, password, type, locationID))  
 conn.commit()  
  
# Checks that no duplicate locations are uploaded  
def checkIfLocationExists(conn, name):  
 locationCursor = conn.cursor()  
 locationCursor.execute("SELECT COUNT(\*) FROM Locations WHERE LocationName = ?"(name,))  
 locationCount = locationCursor.fetchall()  
 if locationCount[0][0] == 0:  
 return True  
 else:  
 return False  
  
# Adds a new location by name, latitude and longitude to the locations table  
def addNewLocation(conn, name, latitude, longitude):  
 locationCursor = conn.cursor()  
 locationCursor.execute("INSERT INTO Locations (LocationName, Latitude, Longitude) VALUES (?,?,?)",  
 (name.upper(), latitude, longitude,))  
 conn.commit()  
  
# Checks that the longitude and latitude values are valid by checking that they are within the correct values  
def longitudeAndLatitudeValidation(conn, name, latitude, longitude):  
 if (latitude < 90 and latitude > -90) and (longitude < 180 and longitude > -180) and checkIfLocationExists(conn, name):  
 addNewLocation(conn, name, latitude, longitude)  
 return "New location " + name + " added to database"  
 else:  
 return "Latitude or longitude is not in range of values"  
  
# Checks that no devices exist with the same username  
def getDeviceCountByDeviceName(conn, name):  
 deviceCursor = conn.cursor()  
 deviceCursor.execute("SELECT COUNT(\*) FROM RegisteredDevices WHERE Name=?", (name,))  
 deviceCount = deviceCursor.fetchall()  
 return deviceCount[0][0]  
  
# Produces a station API key (hash code of the station name)  
def generateStationAPIKey(conn, locationName):  
 isUnique = False  
 while isUnique == False:  
 currentDate = datetime.datetime.now().strftime("%d%m%Y")  
 stationName = currentDate + locationName + str(random.randint(0, 9999))  
 deviceCount = getDeviceCountByDeviceName(conn, stationName)  
 if (deviceCount == 0):  
 isUnique = True  
 return stationName  
  
# Returns the ID of the station by checking for the APIKey  
def getIDOfStation(conn, password):  
 deviceCursor = conn.cursor()  
 deviceCursor.execute("SELECT DeviceID FROM RegisteredDevices WHERE Password=? AND Type='Station'", (password,))  
 stationID = deviceCursor.fetchall()  
 stationID = g.sqliteTupleToList(stationID, 0)[0]  
 return stationID  
  
# Updates the linked account ID of the user so that they have an associated weather station to them  
def linkIDWithStation(conn, stationID, name):  
 deviceCursor = conn.cursor()  
 deviceCursor.execute("UPDATE RegisteredDevices SET LinkedAccountID=? WHERE Name=?", (stationID, name,))  
 conn.commit()  
  
# Returns the location and device ID where the device name = name  
def getLocationAndDeviceIDByName(conn, name):  
 deviceCursor = conn.cursor()  
 deviceCursor.execute("SELECT DeviceID, LocationID FROM RegisteredDevices WHERE Name=?", (name,))  
 userInfo = deviceCursor.fetchall()  
 return userInfo  
  
# Adds a sample to the sample table  
def addSamples(conn, deviceID, typeID, locationID, timestamp, value):  
 sampleCursor = conn.cursor()  
 sampleCursor.execute("INSERT INTO Samples(DeviceID, TypeID, LocationID, Timestamp, Value) VALUES(?,?,?,?,?)",  
 (deviceID, typeID, locationID, timestamp, value))  
 conn.commit()  
  
# A sequence of commands performed to return the percentage cloud cover in the image  
def imageAnalysisSequence(savePath, fetchTime):  
 skyShot = p.CloudCover(savePath)  
 skyShot.linearScan()  
 percentageCover = skyShot.calcCoverPercentage()  
 condition = skyShot.determineCondition()  
 if fetchTime:  
 timestamp = skyShot.timestamp  
 return percentageCover, condition, timestamp  
 else:  
 return percentageCover, condition  
  
# Returns the deviceID by checking against the name and type of the device  
def getDeviceIDByNameAndType(conn, name, type):  
 deviceCursor = conn.cursor()  
 deviceCursor.execute("SELECT DeviceID FROM RegisteredDevices WHERE Name=? AND Type=?", (name, type,))  
 deviceID = deviceCursor.fetchall()[0][0]  
 return deviceID  
  
# Sets a user's home location for ease of image submission and extreme weather reports  
def setHomeLocationOfDevice(conn, locationID, deviceID):  
 deviceCursor = conn.cursor()  
 deviceCursor.execute("UPDATE RegisteredDevices SET LocationID=? WHERE DeviceID=? ", (locationID, deviceID,))  
 conn.commit()  
  
# Decodes the image from base64 so that it is readable by the pillow library for measuring cloud cover  
def saveAndLoadImg(imgRaw, timestamp):  
 imgReadable = base64.b64decode(imgRaw.encode('utf-8'))  
 img = Image.open(io.BytesIO(imgReadable))  
 datetimeTimestamp = datetime.datetime.strptime(timestamp, '%Y-%m-%d, %H:%M:%S')  
 savePath = os.path.join("Images", datetimeTimestamp.strftime("%Y-%m-%d-%H%M%S") + ".jpg")  
 img.save(savePath)  
 return savePath  
  
# Returns the type ID from the type name  
def getTypeIDFromListOfTypeNames(conn, sampleTypes, i):  
 sampleCursor = conn.cursor()  
 sampleCursor.execute("SELECT TypeID FROM SampleType WHERE TypeName=?", (sampleTypes[i].upper(),))  
 typeID = sampleCursor.fetchall()  
 return typeID[0][0]  
  
# Adds a new type of sample to the SampleType database  
def addNewSampleType(conn, name, units):  
 typeCursor = conn.cursor()  
 typeCursor.execute("INSERT INTO SampleType(TypeName, Units) VALUES(?,?)", (name.upper(), units))  
 conn.commit()  
  
# returns the device and location ID of a device from the name, password and type of the device  
def getDeviceIDAndLocationIDByNameTypeAndKey(conn, name, type, key):  
 deviceCursor = conn.cursor()  
 deviceCursor.execute("SELECT DeviceID, LocationID FROM RegisteredDevices WHERE Name=? AND Password=? AND Type=?",  
 (name, key,type,))  
 IDs = deviceCursor.fetchall()  
 deviceID, locationID = IDs[0][0], IDs[0][1]  
 return deviceID, locationID

**General functions**

def sqliteTupleToList(list, j):  
 rfList = [] # Queries from the sql database are received as tuples so it must be refined to an ordinary list  
 for i in range(len(list)):  
 rfList.append(list[i][j])  
 return rfList

**Prediction**

import numpy as np  
from PIL import Image  
import datetime, os  
import sqlite3 as sql  
import pandas as pd  
import statsmodels.tsa.statespace.sarimax as sm  
  
class CloudCover:  
 def \_\_init\_\_(self, fileName):  
 # Converts the image into an array for easier analysis  
 self.refImage = Image.open(fileName)  
 self.refLoad = self.refImage.load()  
 self.totalClear, self.totalCloud, self.coverPercentage = 0,0,0  
 self.xMaximum, self.yMaximum = self.refImage.size  
 self.timestamp = datetime.datetime.now().strftime('%Y-%m-%d, %H:%M:%S')  
 def calcCoverPercentage(self):  
 self.coverPercentage = (self.totalCloud/(self.totalClear+self.totalCloud))\*100  
 return self.coverPercentage  
 def determineCondition(self):  
 if self.coverPercentage > 90:  
 return "Overcast"  
 elif self.coverPercentage < 90 and self.coverPercentage > 60:  
 return "Mostly cloudy"  
 elif self.coverPercentage < 60 and self.coverPercentage > 30:  
 return "Slightly cloudy"  
 else:  
 return "Clear"  
 def classifyPixel(self, xPixel, yPixel):  
 # Determines whether the pixel is cloudy (grey or white) or sky (blue)  
 r,g,b = self.refLoad[xPixel,yPixel]  
 totalPixelValue = r+g+b  
 if (r,g,b) != (0,0,0):  
 # Percentage blue constant is defined as 0.45  
 if (b/totalPixelValue) > 0.45:  
 self.totalClear += 1  
 else:  
 self.totalCloud += 1  
 def linearScan(self):  
 # Performs a linear scan across the image, row upon row  
 yTemp = 0  
 for i in range(self.yMaximum):  
 xTemp = 0  
 for j in range(self.xMaximum):  
 # Calls the classifyPixel function  
 self.classifyPixel(xTemp, yTemp)  
 xTemp += 1  
 yTemp += 1  
  
class prediction:  
 def \_\_init\_\_(self, locationID, cur):  
 self.locationID = locationID  
 self.cur = cur  
 def grab(self, sampleType, period):  
 self.cur.execute('SELECT Timestamp, Value FROM Samples WHERE TypeID=? AND LocationID=?',(sampleType, self.locationID,))  
 dataset = self.cur.fetchall()  
 currentTime = datetime.datetime.now()  
 data = []  
 time = []  
 for i in range(len(dataset)):  
 sampleTime = datetime.datetime.strptime(dataset[i][0], '%Y-%m-%d, %H:%M:%S')  
 deltaTime = (sampleTime- currentTime).total\_seconds()  
 if deltaTime >= -period and deltaTime <= 0:  
 count = 0  
 for j in range(len(data)):  
 if time[j] == dataset[i][0]:  
 data.append((dataset[i][1]+data[j])/2)  
 time.append(dataset[i][0])  
 del data[j], time[j]  
 else:  
 count += 1  
 if count == len(data):  
 data.append(dataset[i][1])  
 time.append(dataset[i][0])  
 return data, time  
 def prepareDataset(self, dataset, periodType):  
 prepDataset = pd.DataFrame(dataset)  
 prepDataset = prepDataset.set\_index('Datetime')  
 prepDataset = prepDataset.resample(periodType).ffill().reset\_index()  
 prepDataset = prepDataset.set\_index('Datetime')  
 return prepDataset  
 def timeSeriesForecast(self, sampleType, period, periodType, locationID):  
 data, time = self.grab(sampleType, 100000000000000)  
 if(len(data) > 1 and type(data[0]) == float):  
 trainDataSet = {'Datetime': pd.to\_datetime(time), 'Data': data}  
 df\_trainDataSet = self.prepareDataset(trainDataSet, periodType)  
 if not os.path.exists(periodType+'Models\model-'+str(sampleType)+'-'+str(locationID)+'.pkl'):  
 mod = sm.SARIMAX(df\_trainDataSet,order=(1,1,1), seasonal\_order=(0,1,0, 12), trend='t',  
 enforce\_stationarity=False, enforce\_invertibility=False)  
 results = mod.fit()  
 results.save(periodType+'Models\model-'+str(sampleType)+'-'+str(locationID)+'.pkl')  
 forecast = results.forecast(steps=period, dynamic=False)  
 else:  
 results = sm.SARIMAXResults.load(periodType+'Models\model-'+str(sampleType)+'-'+str(locationID)+'.pkl')  
 forecast = results.forecast(steps=period, dynamic=False)  
 return forecast  
 else:  
 return "None"  
 def linearRegression(self, sampleType, period):  
 self.cur.execute('SELECT Timestamp, Value FROM Samples WHERE TypeID=? AND LocationID=?',(sampleType, self.locationID,))  
 dataset = self.cur.fetchall()  
 currentTime = datetime.datetime.now()  
 x\_train, y\_train = np.array([]), np.array([])  
 for i in range(len(dataset)):  
 sampleTime = datetime.datetime.strptime(dataset[i][0], '%Y-%m-%d, %H:%M:%S')  
 deltaTime = (currentTime-sampleTime).total\_seconds()  
 if deltaTime <= period:  
 x\_train = np.append(x\_train, [(period-deltaTime)])  
 y\_train = np.append(y\_train, [(dataset[i][1])])  
 if(len(dataset) > 0):  
 if(type(dataset[0][1]) == float):  
 self.n = len(x\_train)  
 meanX = np.mean(x\_train)  
 meanY = np.mean(y\_train)  
 XY = np.sum(np.multiply(y\_train, x\_train)) - self.n \* meanY \* meanX  
 XX = np.sum(np.multiply(x\_train, x\_train)) - self.n \* meanX \* meanX  
 self.m = XY / XX  
 self.c = meanY - self.m \* meanX  
 if len(y\_train) > 1:  
 return y\_train[len(y\_train)-1]  
 if len(y\_train) == 1:  
 self.m = 0  
 self.c = y\_train[len(y\_train)-1]  
 return y\_train[len(y\_train)-1]  
 else:  
 return "null"  
 else:  
 self.m = 0  
 return "null"  
 else:  
 self.m = 0  
 return "null"  
 def hourPrediction(self, timeAfterHour):  
 predictedTemp = self.m \* (3600+(timeAfterHour\*60)) + self.c  
 return predictedTemp  
 def checkTrend(self):  
 if self.m < 0:  
 return "decrease"  
 if self.m == 0:  
 return "stay constant"  
 if self.m > 0:  
 return "increase"  
  
def bubbleSort(data, time):  
 currentTime = datetime.datetime.now()  
 for i in range(len(data)):  
 for j in range(len(data)-(i+1)):  
 deltaTime1 = (datetime.datetime.strptime(time[j+i], '%Y-%m-%d, %H:%M:%S') - currentTime).total\_seconds()  
 deltaTime2 = (datetime.datetime.strptime(time[j+i+1], '%Y-%m-%d, %H:%M:%S') - currentTime).total\_seconds()  
 if deltaTime1 > deltaTime2:  
 tempStoreData, tempStoretime = data[i+j], time[i+j]  
 time[i+j], data[i+j] = time[i+j+1], data[i+j+1]  
 time[i + j + 1], data[i + j + 1] = tempStoretime, tempStoreData  
 return data, time  
  
def makeTimestamp(timeAccessed, i, time):  
 newTimestamp = timeAccessed + datetime.timedelta(minutes=i)  
 newTimestamp = newTimestamp.strftime('%Y-%m-%d, %H:%M:%S')  
 time.append(newTimestamp)  
  
def appendValues(data, ID, period, dataset):  
 latestValue = dataset.linearRegression(ID, period)  
 time = []  
 timeAccessed = datetime.datetime.now()  
 if not (type(latestValue) is str):  
 oldTemperatures, oldTime = dataset.grab(ID, period)  
 oldTemperatures, oldTime = bubbleSort(oldTemperatures, oldTime)  
 for i in range(len(oldTemperatures)):  
 data.append(oldTemperatures[i])  
 time.append(oldTime[i])  
 for j in range(60):  
 data.append(dataset.hourPrediction(j))  
 makeTimestamp(timeAccessed, j, time)  
 else:  
 data.append("Not enough data")  
 return data, time, latestValue  
  
def minuteCast(locationID, cur, sampleTypes):  
 availableData = []  
 dataset = prediction(locationID, cur)  
 availableTimes, latestValues, trendInfoList = [], [], []  
 for i in range(len(sampleTypes)):  
 temporaryList = []  
 temporaryList, time, latestValue = appendValues(temporaryList, i+1, 3600, dataset)  
 availableData.append(tuple(temporaryList))  
 availableTimes.append(time)  
 latestValues.append(latestValue)  
 trendInfoList.append(dataset.checkTrend())  
 return availableData, availableTimes, latestValues, trendInfoList  
  
def grabTimeline(locationID, sampleType, cur):  
 dataset = prediction(locationID, cur)  
 data, time = dataset.grab(sampleType, 100000000000000000000)  
 return data, time  
  
def machineLearning(locationID, sampleType, cur, period, periodType):  
 time = []  
 dataset = prediction(locationID, cur)  
 forecast = dataset.timeSeriesForecast(sampleType, period, periodType, locationID)  
 if isinstance(forecast, str):  
 data, time = [0,0]  
 else:  
 data = forecast.values  
 for i in range(len(forecast.index)):  
 time.append(datetime.datetime.strftime(forecast.index[i], '%Y-%m-%d, %H:%M:%S'))  
 data = data.tolist()  
 return data, time  
  
def checkStormWarning(cur, locationID, periodOfConcern):  
 dataset = prediction(locationID, cur)  
 data, time = dataset.grab(5, periodOfConcern)  
 if(len(time) == 0):  
 return "None"  
 else:  
 return time[0]

**Raspberry pi “send data” program**

import requests, json, bmp180, time  
import datetime as dt  
import os, base64  
  
def getMedia():  
 os.system('libcamera-jpeg -o cloudImage.jpg --shutter 2000 --width 640 --height 480')  
 with open("cloudImage.jpg", "rb") as img:  
 imgBytes = img.read()  
 imgDecoded = base64.b64encode(imgBytes).decode("utf8")  
 return imgDecoded  
  
def writeToBuffer(query):  
 with open("../../buffer.txt", "a") as buffer:  
 buffer.write(str(query)+"\n")  
 buffer.close()  
  
def sendData(query, url):  
 try:  
 jsonData = requests.post(url, json=query)  
 print(jsonData.text)  
 return True  
 except:  
 print("server not online")  
 writeToBuffer(query)  
 return False  
  
def sendBuffer(url):  
 with open("../../buffer.txt", "r+") as buffer:  
 queries = buffer.readlines()  
 if(len(queries) > 0):  
 count = 0  
 for i in range(len(queries)):  
 query = (queries[i][0:len(queries[i])-1].replace("'","\"")).replace("(","[").replace(")","]")  
 if(sendData(json.loads(query), url) or queries[i] == ""):  
 count += 1  
 time.sleep(0.0001)  
 if(count == len(queries)):  
 buffer.truncate(0)  
 print("sent "+str(count)+" queries and missed "+str(len(queries)-count)+"queries")  
 buffer.close()  
   
url = "http://192.168.1.72:5000/DataReceiver"  
location = "PURLEY"  
stationName = "09082022PURLEY0001"   
stationKey = "MY\_KEY"   
  
command = input("What command would you like to use? (send, addType)")  
  
if(command == "send"):  
 sampleType = input("What type of data are you sending? (floats, both(media and floats))")  
 if(sampleType == "both"):  
 while True:   
 timestamp = dt.datetime.now().strftime("%Y-%m-%d, %H:%M:%S")  
 temperature, pressure = bmp180.readBmp180()  
 imgDecoded = getMedia()  
 query = {"command": "send", "types": ("temperature", "pressure", "media"), "data": ({"temperature": temperature, "pressure": pressure, "media": (imgDecoded)}), "login": ({"stationName": stationName, "stationKey": stationKey}), "timestamp": timestamp}  
 if(sendData(query, url)):  
 sendBuffer(url)  
 time.sleep(60)  
 elif(sampleType == "floats"):  
 while True:  
 timestamp = dt.datetime.now().strftime("%Y-%m-%d, %H:%M:%S")  
 temperature, pressure = bmp180.readBmp180()  
 query = {"command": "send", "types": ("temperature", "pressure"), "data": ({"temperature": temperature, "pressure": pressure}), "login": ({"stationName": stationName, "stationKey": stationKey}), "timestamp": timestamp}  
 if(sendData(query, url)):  
 sendBuffer(url)  
 time.sleep(60)  
elif(command == "addType"):  
 name = input("What is the name of the sample type?")  
 units = input("What is the sample type's units?")  
 query = {"command": "addType", "name": name, "units": units}  
 try:  
 jsonData = requests.post(url, json=query)  
 print(jsonData.text)  
 except:  
 print("Server currently unavailable")  
else:  
 print("Invalid command")

**HTML Templates and JSS**

* Home page
* Login page
* User page
* Location main page
* Location timeline page
* Location hourly prediction page
* Location daily prediction page

<!DOCTYPE HTML>  
<html>  
<head>  
 <link rel="stylesheet" type="text/css" href="{{ url\_for('static',filename='styles/graphics.css') }}">  
</head>  
<body>  
 <div class="navigationBar">  
 <a class="active" href="{{ url\_for('home') }}">Home</a>  
 <a href="{{ url\_for('loginPage') }}">Login</a>  
 <form action="http://127.0.0.1:5000/LocationReceiver" method="post">  
 <select name="locationSelect" id="locationSelect">  
 <option value="none">Locations : </option>  
 </select>  
 <input type="submit">  
 </form>  
 </div>  
 <div class="navbackground"></div>  
 <div class="geoChart" id="geoChart"></div>  
 <script type="text/javascript" src="https://www.google.com/jsapi"></script>  
 <script>  
 google.charts.load('visualization', {'packages':['geochart'],'mapsApiKey': 'MY\_KEY'});  
 google.charts.setOnLoadCallback(drawLocationsMap);  
 var post = {{ post|tojson }};  
 var select = document.getElementById("locationSelect");  
 function drawLocationsMap() {  
 var post = {{ post|tojson }};  
 data = [['City','Sample count']];  
 for(var i=0; i < post.sampleCounts.length; i++){  
 data.push([post.locationNames[i],post.sampleCounts[i]]);  
 }  
 console.log(data);  
 var dataTable = google.visualization.arrayToDataTable(data);  
 var options = {  
 title: 'Geo map',  
 sizeAxis: {minValue: 0, maxSize: 15},  
 region: 'world',  
 displayMode: 'markers',  
 markerOpacity: 0.8,  
 legend: {texStyle: {color: 'black', fontSize: 16}},  
 colorAxis: {colors: ['yellow', 'purple']}  
 };  
 var chart = new google.visualization.GeoChart(document.getElementById('geoChart'));  
 chart.draw(dataTable, options);  
 }  
 for (var i=0; i < post.locationNames.length; i++){  
 var option = document.createElement("option");  
 option.value = post.locationNames[i];  
 option.innerHTML = post.locationNames[i];  
 select.appendChild(option);  
 }  
 </script>  
</body>  
</html>

-------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------

<!DOCTYPE HTML>  
<html>  
<head>  
 <link rel="stylesheet" type="text/css" href="{{ url\_for('static',filename='styles/graphics.css') }}">  
 <link rel="stylesheet" type="text/css" href="{{ url\_for('static',filename='styles/loginGraphics.css') }}">  
</head>  
<body>  
 <div class="navigationBar">  
 <a href="{{ url\_for('home') }}">Home</a>  
 <a class="active" href="{{ url\_for('loginPage') }}">Login</a>  
 </div>  
 <div class="navbackground"></div>  
 <form action="http://127.0.0.1:5000/LoginReceiver" method="post">  
 <div class="login">  
 <h2>Login</h2>  
 <p>Username : </p><p><input type="text" name="username"/></p>  
 <p>Password : </p><p><input type="password" name="password"/></p>  
 <input type="submit" value="login"/>  
 </div>  
 </form>  
 <form action="http://127.0.0.1:5000/RegisterReceiver" method="post">  
 <div class="login">  
 <h2>Register</h2>  
 <p>Username : </p><p><input type="text" name="username"></p>  
 <p>Password : </p><p><input type="password" name="password"></p>  
 <p>Repeat password : </p><p><input type="password" name="checkPassword"></p>  
 <input type="submit" value="sign up"/>  
 </div>  
 </form>  
</body>  
</html>

-------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------

<!DOCTYPE HTML>  
<html>  
<head>  
 <link rel="stylesheet" type="text/css" href="{{ url\_for('static',filename='styles/graphics.css') }}">  
 <link rel="stylesheet" type="text/css" href="{{ url\_for('static',filename='styles/userPageGraphics.css') }}">  
</head>  
<body>  
 <div class="navigationBar">  
 <a href="{{ url\_for('home') }}">Home</a>  
 <a class="active" href="{{ url\_for('loginPage') }}">Login</a>  
 </div>  
 <div class="navbackground"></div>  
 <div class="box">  
 <form action="http://127.0.0.1:5000/imageReceiver" method="post" enctype="multipart/form-data">  
 <p id = "username" name="username"></p>  
 <p id = "imageCount" name="imageCount"></p>  
 <input type="file" name="imageUpload">  
 <p>Location (town name) : </p>  
 <input type="text" name="location"><br>  
 <input type="checkbox" id="setHome" name="setHome">  
 <label for="setHome">Set as home?</label><br>  
 <input type="submit" value="submit image">  
 <input type="hidden" name="hiddenUsername" id="hiddenUsername" value="">  
 </form>  
 </div>  
 <div class="box">  
 <h2>Location not found?</h2>  
 <form action="http://127.0.0.1:5000/addLocation" method="post">  
 <p>Location name : </p><input type="text" name="locationName">  
 <p>Latitude (decimal) : </p><input type="text" name="latitude">  
 <p>Longitude (decimal) : </p><input type="text" name="longitude"><br>  
 <input type="submit" value="add new location">  
 </form>  
 </div>  
 <div class="box">  
 <h2>Add your own station!</h2>  
 <p>Location name : </p><input type="text" id="stationLocationName" name="locationName">  
 <button id="submitApplication" onclick="generateAPI()">Generate api key and station name</button>  
 <input id="name" type="hidden">  
 <input id="APIKEY" type="hidden">  
 <input id="revealAPI" type="hidden" onclick="unhide()"><label>Show api key</label>  
 </div>  
 <div class="stormWarning">  
 <img src="{{ url\_for('static',filename='styles/ExtremeWeather.png') }}", onclick="reportExtremeWeather()">  
 </div>  
 <p id="errorMessage"></p>  
 <script>  
 var welcome = document.getElementById("username");  
 var imageCount = document.getElementById("imageCount");  
 var stationName = document.getElementById("name");  
 var APIKey = document.getElementById("APIKEY");  
 var showAPIKey = document.getElementById("revealAPI");  
 var errorMessage = document.getElementById("errorMessage");  
 var stationLocation = document.getElementById("stationLocationName");  
 var post = {{ info|tojson }};  
 console.log(post);  
 document.getElementById("hiddenUsername").value = post[0];  
 welcome.innerHTML = "User : "+post[0];  
 imageCount.innerHTML = "Images contributed : "+post[1];  
 if(post[2] != "None"){  
 stationName.type = "text";  
 APIKey.type = "password";  
 showAPIKey.type = "checkbox";  
 stationName.value = post[2];  
 APIKey.value = post[3];  
 }  
 function generateAPI(){  
 fetch("http://127.0.0.1:5000/addStation?userDetails="+[post[0],post[1]], {  
 method: "POST",  
 body: JSON.stringify({  
 location: stationLocation.value  
 }),  
 headers: {  
 "Content-type": "application/json; charset=UTF-8"  
 }  
 })  
 .then(response => response.json())  
 .then(function (response) {  
 stationName.type = "text";  
 APIKey.type = "password";  
 showAPIKey.type = "checkbox";  
 stationName.value = response["stationName"];  
 APIKey.value = response["stationPass"];  
 });  
 }  
 function reportExtremeWeather(){  
 fetch("http://127.0.0.1:5000/reportWarning?userDetails="+[post[0],post[1]], {  
 method: "POST",  
 headers: {  
 "Content-type": "application/json; charset=UTF-8"  
 }  
 })  
 .then(response => response.json())  
 .then(function (response) {  
 errorMessage.value = response["message"];  
 console.log(response["message"]);  
 });  
 }  
 function unhide(){  
 if(APIKey.type == "password"){  
 APIKey.type = "text";  
 }else{  
 APIKey.type = "password";  
 }  
 }  
 </script>  
</body>  
</html>

-------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------

<!DOCTYPE HTML>  
<html>  
<head>  
 <link rel="stylesheet" type="text/css" href="{{ url\_for('static',filename='styles/graphics.css') }}">  
 <link rel="stylesheet" type="text/css" href="{{ url\_for('static',filename='styles/minuteCast.css') }}">  
</head>  
<body>  
 <div class="navigationBar">  
 <a href="{{ url\_for('home') }}">Home</a>  
 <a href="{{ url\_for('hourlyPage', locationName=post.locationName) }}">Hourly Forecast</a>  
 <a href="{{ url\_for('dailyPage', locationName=post.locationName) }}">Daily Forecast</a>  
 <a href="{{ url\_for('timelinePage', locationName=post.locationName) }}">Timeline</a>  
 <a href="{{ url\_for('loginPage') }}">Login</a>  
 </div>  
 <div class="navbackground"></div>  
 <div class="summaryBox">  
 <a id="location"></a>  
 <p id="latLong" class="info"></p>  
 <p id="temperature" class="info"></p>  
 <p id="pressure" class="info"></p>  
 <p id="summary"></p>  
 </div>  
 <div class="warning">  
 <a id="warning"></a>  
 </div>  
 <form id="sampleTypePicker">  
 <select id="sampleTypeSelect"></select>  
 <input type="submit" value="submit">  
 </form>  
 <div class="chart" id="chartDisplay"></div>  
 <table class="charts-css bar" id="MinuteCastTemp"></table>  
 <script type="text/javascript" src="https://www.gstatic.com/charts/loader.js"></script>  
 <script type="text/javascript">  
 google.charts.load('current', {'packages':['corechart']});  
 google.charts.setOnLoadCallback(loadDataForChart);  
 var temperatureValue = document.getElementById('temperature');  
 var pressureValue = document.getElementById('pressure');  
 var locationNameDisplay = document.getElementById('location');  
 var latLongDisplay = document.getElementById('latLong');  
 var summary = document.getElementById('summary');  
 var sampleTypeSelectBox = document.getElementById('sampleTypeSelect');  
 var sampleTypePicker = document.getElementById('sampleTypePicker');  
 var warning = document.getElementById('warning');  
 var jsonRequest;  
 var dataSelection = 'TEMPERATURE';  
 var post = {{ post|tojson }};  
 if(post.stormWarning != 'None'){  
 warning.innerHTML = "Extreme weather warning notice from "+post.stormWarning;  
 document.body.style.backgroundImage = "url({{url\_for('static',filename='styles/Storm.jpg')}})";  
 }  
 for(var i = 0; i < post.sampleTypes.length; i++){  
 var option = document.createElement("option");  
 option.value = post.sampleTypes[i];  
 option.innerHTML = post.sampleTypes[i];  
 sampleTypeSelectBox.appendChild(option);  
 }  
 sampleTypePicker.addEventListener('submit', function(e) {  
 e.preventDefault();  
 dataSelection = sampleTypeSelectBox.options[sampleTypeSelectBox.selectedIndex].value;  
 sampleTypeChart(dataSelection, jsonRequest);  
 })  
 function drawChart(sampleType, time, typeName, chart) {  
 var data = new google.visualization.DataTable();  
 data.addColumn("date", "Time and date");  
 data.addColumn("number", typeName);  
 for(var i=0; i < sampleType.length; i++){  
 dateTime = new Date(Date.UTC(parseInt(time[i].substring(0,4)), parseInt(time[i].substring(5,7))-1, parseInt(time[i].substring(8,10)), parseInt(time[i].substring(12,14))-1, parseInt(time[i].substring(15,17)), parseInt(time[i].substring(18,20))));  
 data.addRow([dateTime, sampleType[i]]);  
 }  
 var options = {  
 title: typeName,  
 legend: { position: 'bottom' }  
 };  
 var chart = new google.visualization.LineChart(document.getElementById(chart));  
 chart.draw(data, options);  
 }  
 function sampleTypeChart(dataSelection, jsonRequest){  
 console.log(dataSelection);  
 if(jsonRequest.data[dataSelection].data.length > 1){  
 drawChart(jsonRequest.data[dataSelection].data,jsonRequest.data[dataSelection].time, dataSelection, "chartDisplay");  
 }  
 }  
 function loadDataForChart(){  
 fetch('/fetchData?locationName='+post.locationName)  
 .then(function (response) {  
 return response.json();  
 }).then(function (jsonData) {  
 jsonRequest = jsonData;  
 temperatureValue.innerHTML = "Temperature: "+jsonRequest.data.TEMPERATURE.latestValue+" celsius";  
 pressureValue.innerHTML = "Pressure: "+jsonRequest.data.PRESSURE.latestValue+" hPa";  
 locationNameDisplay.innerHTML = jsonRequest.location.locationName;  
 latLongDisplay.innerHTML = "Latitude: "+jsonRequest.location.latitude+"<br>Longitude: "+jsonRequest.location.longitude;  
 if(jsonRequest.data.PRESSURE.trend == "increase"){  
 pressureInfluence = "lower";  
 }else{  
 pressureInfluence = "higher";  
 }  
 summary.innerHTML = "Temperature is likely to "+jsonRequest.data.TEMPERATURE.trend+" over the next hour. Pressure is likely to "+jsonRequest.data.PRESSURE.trend+", resulting in a "+pressureInfluence+" chance of cloud cover across the next hour.";  
 if(jsonRequest.data.TEMPERATURE.latestValue >= 30){  
 document.body.style.backgroundImage = "url({{url\_for('static',filename='styles/Dry.jpg')}})";  
 } else if(jsonRequest.data.TEMPERATURE.latestValues >= 20 & jsonRequest.data.TEMPERATURE.latestValues <= 30){  
 document.body.style.backgroundImage = "url({{url\_for('static',filename='styles/Sunset2.jpg')}})";  
 } else if(jsonRequest.data.TEMPERATURE.latestValues >= 10 & jsonRequest.data.TEMPERATURE.latestValues <= 20){  
 document.body.style.backgroundImage = "url({{url\_for('static',filename='styles/spring.jpg')}})";  
 } else{  
 document.body.style.backgroundImage = "url({{url\_for('static',filename='styles/winter.jpg')}})";  
 }  
 if(post.stormWarning != 'None'){  
 warning.innerHTML = "Extreme weather warning notice from "+post.stormWarning;  
 document.body.style.backgroundImage = "url({{url\_for('static',filename='styles/Storm.jpg')}})";  
 }  
 sampleTypeChart(dataSelection, jsonRequest);  
 });  
 }  
 setInterval(loadDataForChart, 60000);  
 </script>  
</body>  
</html>

-------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------

<!DOCTYPE HTML>  
<html>  
<head>  
 <link rel="stylesheet" type="text/css" href="{{ url\_for('static',filename='styles/graphics.css') }}">  
 <link rel="stylesheet" type="text/css" href="{{ url\_for('static',filename='styles/timeline.css') }}">  
</head>  
<body>  
 <div class="navigationBar">  
 <a href="{{ url\_for('home') }}">Home</a>  
 <a href="{{ url\_for('hourlyPage', locationName=post.locationName) }}">Hourly Forecast</a>  
 <a href="{{ url\_for('dailyPage', locationName=post.locationName) }}">Daily Forecast</a>  
 <a class='active' href="{{ url\_for('timelinePage', locationName=locationName) }}">Timeline</a>  
 <a href="{{ url\_for('loginPage') }}">Login</a>  
 </div>  
 <div class="navbackground"></div>  
 <form id="sampleTypePicker">  
 <select id="sampleTypeSelect"></select>  
 <input type="submit" value="submit">  
 </form>  
 <div class="chart" id="chartDisplay"></div>  
 <table class="charts-css bar" id="MinuteCastTemp"></table>  
 <script type="text/javascript" src="https://www.gstatic.com/charts/loader.js"></script>  
 <script type="text/javascript">  
 google.charts.load('current', {'packages':['corechart']});  
 google.charts.setOnLoadCallback(loadDataForChart);  
 var post = {{ post|tojson }}  
 var sampleTypeSelectBox = document.getElementById("sampleTypeSelect");  
 var sampleTypePicker = document.getElementById('sampleTypePicker');  
 var dataSelection = "TEMPERATURE";  
 var jsonRequest;  
 for(var i = 0; i < post.sampleTypes.length; i++){  
 var option = document.createElement("option");  
 option.value = post.sampleTypes[i];  
 option.innerHTML = post.sampleTypes[i];  
 sampleTypeSelectBox.appendChild(option);  
 }  
 sampleTypePicker.addEventListener('submit', function(e) {  
 e.preventDefault();  
 dataSelection = sampleTypeSelectBox.options[sampleTypeSelectBox.selectedIndex].value;  
 sampleTypeChart(dataSelection, jsonRequest);  
 })  
 function drawChart(sampleType, time, typeName, chart) {  
 var data = new google.visualization.DataTable();  
 data.addColumn("date", "Time");  
 data.addColumn("number", typeName);  
 for(var i=0; i < sampleType.length; i++){  
 dateTime = new Date(Date.UTC(parseInt(time[i].substring(0,4)), parseInt(time[i].substring(5,7))-1, parseInt(time[i].substring(8,10)), parseInt(time[i].substring(12,14))-1, parseInt(time[i].substring(15,17)), parseInt(time[i].substring(18,20))));  
 data.addRow([dateTime, sampleType[i]])  
 }  
 var options = {  
 title: typeName,  
 legend: { position: 'bottom' }  
 };  
  
 var chart = new google.visualization.LineChart(document.getElementById(chart));  
 chart.draw(data, options)  
 }  
 function sampleTypeChart(dataSelection, jsonRequest){  
 if(jsonRequest.data[dataSelection].data.length > 1){  
 drawChart(jsonRequest.data[dataSelection].data,jsonRequest.data[dataSelection].time, dataSelection, "chartDisplay");  
 }  
 }  
 function loadDataForChart(){  
 fetch('/fetchTimeline?locationName='+post.locationName)  
 .then(function (response) {  
 return response.json();  
 }).then(function (jsonData) {  
 jsonRequest = jsonData;  
 sampleTypeChart(dataSelection, jsonRequest);  
 });  
 }  
 setInterval(loadDataForChart, 60000);  
 </script>  
</body>  
</html>

-------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------

<!DOCTYPE HTML>  
<html>  
<head>  
 <link rel="stylesheet" type="text/css" href="{{ url\_for('static',filename='styles/graphics.css') }}">  
 <link rel="stylesheet" type="text/css" href="{{ url\_for('static',filename='styles/timeline.css') }}">  
</head>  
<body>  
 <div class="navigationBar">  
 <a href="{{ url\_for('home') }}">Home</a>  
 <a class='active' href="{{ url\_for('hourlyPage', locationName=post.locationName) }}">Hourly Forecast</a>  
 <a href="{{ url\_for('dailyPage', locationName=post.locationName) }}">Daily Forecast</a>  
 <a href="{{ url\_for('timelinePage', locationName=post.locationName) }}">Timeline</a>  
 <a href="{{ url\_for('loginPage') }}">Login</a>  
 </div>  
 <div class="navbackground"></div>  
 <form id="sampleTypePicker">  
 <select id="sampleTypeSelect"></select>  
 <input type="submit" value="submit">  
 </form>  
 <div class="chart" id="chartDisplay"></div>  
 <table class="charts-css bar" id="MinuteCastTemp"></table>  
 <script type="text/javascript" src="https://www.gstatic.com/charts/loader.js"></script>  
 <script type="text/javascript">  
 google.charts.load('current', {'packages':['corechart']});  
 google.charts.setOnLoadCallback(loadDataForChart);  
 var post = {{ post|tojson }}  
 var sampleTypeSelectBox = document.getElementById("sampleTypeSelect");  
 var sampleTypePicker = document.getElementById('sampleTypePicker');  
 var dataSelection = "TEMPERATURE";  
 var jsonRequest;  
 for(var i = 0; i < post.sampleTypes.length; i++){  
 var option = document.createElement("option");  
 option.value = post.sampleTypes[i];  
 option.innerHTML = post.sampleTypes[i];  
 sampleTypeSelectBox.appendChild(option);  
 }  
 sampleTypePicker.addEventListener('submit', function(e) {  
 e.preventDefault();  
 dataSelection = sampleTypeSelectBox.options[sampleTypeSelectBox.selectedIndex].value;  
 sampleTypeChart(dataSelection, jsonRequest);  
 })  
 function drawChart(sampleType, time, typeName, chart) {  
 var data = new google.visualization.DataTable();  
 data.addColumn("date", "Time");  
 data.addColumn("number", typeName);  
 for(var i=0; i < sampleType.length; i++){  
 dateTime = new Date(Date.UTC(parseInt(time[i].substring(0,4)), parseInt(time[i].substring(5,7))-1, parseInt(time[i].substring(8,10)), parseInt(time[i].substring(12,14))-1, parseInt(time[i].substring(15,17)), parseInt(time[i].substring(18,20))));  
 data.addRow([dateTime, sampleType[i]])  
 }  
 var options = {  
 title: typeName,  
 legend: { position: 'bottom' }  
 };  
  
 var chart = new google.visualization.LineChart(document.getElementById(chart));  
 chart.draw(data, options)  
 }  
 function sampleTypeChart(dataSelection, jsonRequest){  
 if(jsonRequest.data[dataSelection].data.length > 1){  
 drawChart(jsonRequest.data[dataSelection].data,jsonRequest.data[dataSelection].time, dataSelection, "chartDisplay");  
 }  
 }  
 function loadDataForChart(){  
 fetch('/fetchMachineLearningPredictions?locationName='+post.locationName+'&period=24&periodType=H')  
 .then(function (response) {  
 return response.json();  
 }).then(function (jsonData) {  
 jsonRequest = jsonData;  
 sampleTypeChart(dataSelection, jsonRequest);  
 });  
 }  
 </script>  
</body>  
</html>

-------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------

<!DOCTYPE HTML>  
<html>  
<head>  
 <link rel="stylesheet" type="text/css" href="{{ url\_for('static',filename='styles/graphics.css') }}">  
 <link rel="stylesheet" type="text/css" href="{{ url\_for('static',filename='styles/timeline.css') }}">  
</head>  
<body>  
 <div class="navigationBar">  
 <a href="{{ url\_for('home') }}">Home</a>  
 <a href="{{ url\_for('hourlyPage', locationName=post.locationName) }}">Hourly Forecast</a>  
 <a class='active' href="{{ url\_for('dailyPage', locationName=post.locationName) }}">Daily Forecast</a>  
 <a href="{{ url\_for('timelinePage', locationName=post.locationName) }}">Timeline</a>  
 <a href="{{ url\_for('loginPage') }}">Login</a>  
 </div>  
 <div class="navbackground"></div>  
 <form id="sampleTypePicker">  
 <select id="sampleTypeSelect"></select>  
 <input type="submit" value="submit">  
 </form>  
 <div class="chart" id="chartDisplay"></div>  
 <table class="charts-css bar" id="MinuteCastTemp"></table>  
 <script type="text/javascript" src="https://www.gstatic.com/charts/loader.js"></script>  
 <script type="text/javascript">  
 google.charts.load('current', {'packages':['corechart']});  
 google.charts.setOnLoadCallback(loadDataForChart);  
 var post = {{ post|tojson }}  
 var sampleTypeSelectBox = document.getElementById("sampleTypeSelect");  
 var sampleTypePicker = document.getElementById('sampleTypePicker');  
 var dataSelection = "TEMPERATURE";  
 var jsonRequest;  
 for(var i = 0; i < post.sampleTypes.length; i++){  
 var option = document.createElement("option");  
 option.value = post.sampleTypes[i];  
 option.innerHTML = post.sampleTypes[i];  
 sampleTypeSelectBox.appendChild(option);  
 }  
 sampleTypePicker.addEventListener('submit', function(e) {  
 e.preventDefault();  
 dataSelection = sampleTypeSelectBox.options[sampleTypeSelectBox.selectedIndex].value;  
 sampleTypeChart(dataSelection, jsonRequest);  
 })  
 function drawChart(sampleType, time, typeName, chart) {  
 var data = new google.visualization.DataTable();  
 data.addColumn("date", "Time");  
 data.addColumn("number", typeName);  
 for(var i=0; i < sampleType.length; i++){  
 dateTime = new Date(Date.UTC(parseInt(time[i].substring(0,4)), parseInt(time[i].substring(5,7))-1, parseInt(time[i].substring(8,10)), parseInt(time[i].substring(12,14))-1, parseInt(time[i].substring(15,17)), parseInt(time[i].substring(18,20))));  
 data.addRow([dateTime, sampleType[i]])  
 }  
 var options = {  
 title: typeName,  
 legend: { position: 'bottom' }  
 };  
  
 var chart = new google.visualization.LineChart(document.getElementById(chart));  
 chart.draw(data, options)  
 }  
 function sampleTypeChart(dataSelection, jsonRequest){  
 if(jsonRequest.data[dataSelection].data.length > 1){  
 drawChart(jsonRequest.data[dataSelection].data,jsonRequest.data[dataSelection].time, dataSelection, "chartDisplay");  
 }  
 }  
 function loadDataForChart(){  
 fetch('/fetchMachineLearningPredictions?locationName='+post.locationName+'&period=7&periodType=D')  
 .then(function (response) {  
 return response.json();  
 }).then(function (jsonData) {  
 jsonRequest = jsonData;  
 sampleTypeChart(dataSelection, jsonRequest);  
 });  
 }  
 </script>  
</body>  
</html>

**CSS styling**

* Graphics.css (general)
* Login Graphics
* User page graphics
* Minute cast graphics
* Timeline graphics

body {  
 background-color: #d6d4d4;  
 background-image: url(Sunset2.jpg);  
 background-repeat: no-repeat;  
 margin: 0;  
 padding: 0;  
}  
  
.navigationBar {  
 background-color: #7E38B7;  
}  
  
.navigationBar a {  
 float: left;  
 color: white;  
 background-color: #262626;  
 font-style: bold;  
 text-align: center;  
 font-family: Sans-Serif;  
 text-decoration: none;  
 padding: 10px 12px;  
}  
  
.navigationBar a:hover {  
 background-color: white;  
 color: black;  
}  
  
.navigationBar a.active {  
 background-color: #541675;  
 color: white;  
}  
  
.navigationBar select {  
 float: right;  
 font-family: Sans-Serif;  
 padding: 10px 12px;  
}  
  
.navigationBar input[type=submit] {  
 float: right;  
 font-family: Sans-Serif;  
 padding: 10px 12px;  
}  
  
.navbackground{  
 width: 1920px;  
 height: 39px;  
 background-color: #262626;  
}  
  
.geoChart{  
 position: relative;  
 top:30px;  
 margin-left: auto;  
 margin-right: auto;  
 width: 1080px;  
 height: 608px;  
}  
-------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------

.login {  
 width: 400px;  
 height: 300px;  
 font-family: Sans-Serif;  
 position: relative;  
 background-color : #737373;  
 top: 10px;  
 left: 20px;  
 float: left;  
 margin: 10px 10px 10px 10px;  
 border-radius: 10px;  
}  
  
.login h2{  
 left: 20px;  
 position: relative;  
}  
  
.login p{  
 left: 20px;  
 position: relative;  
}  
  
.login input[type=text] {  
 left: 20px;  
 position: relative;  
 border: none;  
}  
  
.login input[type=submit] {  
 left: 20px;  
 position: relative;  
 background-color: #541675;  
 color: white;  
 border-radius: 5px;  
 border-style: none;  
 border-width: 5px;  
}

-------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------

.box {  
 width: 320px;  
 height: 310px;  
 font-family: Sans-Serif;  
 position: relative;  
 background-color : #4dffcc;  
 opacity: 80%;  
 top: 10px;  
 left: 20px;  
 float: left;  
 margin: 10px 10px 10px 10px;  
 border-radius: 10px;  
}  
  
.box p{  
 left: 20px;  
 position: relative;  
}  
  
.box h2{  
 left:20px;  
 position: relative;  
}  
  
.box input{  
 left: 20px;  
 position: relative;  
}  
  
.box label{  
 left: 20px;  
 position: relative;  
}  
  
.box button{  
 left:20px;  
 position:relative;  
}  
  
.stormWarning {  
 float:right;  
 width: 50px;  
 height:50px;  
 opacity: 80%;  
 background-color: #ffc908;  
 border-radius: 10px;  
 margin: 10px 10px 10px 10px;  
 position:relative;  
 top: 150px;  
 right: 50px;  
}  
  
.stormWarning img {  
 position:relative;  
 height: 50px;  
 width: 50px;  
}  
.stormWarning a{  
 height: 50px;  
 width: 50px;  
}  
  
.stormWarning:hover{  
 animation-name:enlarge;  
 animation-duration: 2s;  
}  
  
.stormWarning img:hover{  
 animation-name:enlarge;  
 animation-duration: 2s;  
}  
  
@keyframes enlarge{  
 0%{  
 background-color:#ffc908;  
 width:50px;  
 height:50px;  
 border-radius: 10px;  
 }  
 50%{  
 background-color:#de0d37;  
 width:60px;  
 height:60px;  
 border-radius: 10px;  
 }  
 100%{  
 background-color:#ffc908;  
 width:50px;  
 height:50px;  
 border-radius: 10px;  
 }  
}

-------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------

.chart {  
 position:relative;  
 top:170px;  
 left:20px;  
 float:left;  
 height:450px;  
 width:800px;  
}  
  
.warning a {  
 position: relative;  
 left: 500px;  
 bottom: 100px;  
 color: white;  
 font-style: bold;  
 text-align: center;  
 font-family: Sans-Serif;  
 text-decoration: none;  
 font-size: 20px;  
}  
  
p {  
 float:left;  
 display:inline;  
}  
  
.summaryBox {  
 width: 400px;  
 height: 300px;  
 border-radius: 10px;  
 background-color: white;  
 opacity: 50%;  
 position: relative;  
 top: 50px;  
 left: 80px;  
}  
  
.summaryBox a {  
 text-decoration: none;  
 font-size: 26px;  
 position: relative;  
 top: 20px;  
 left: 20px;  
 font-family: Sans-Serif;  
 color:black;  
}  
  
.summaryBox p {  
 width: 350px;  
 height: 10px;  
 font-family: Sans-Serif;  
 position: relative;  
 top: 10px;  
 left: 20px;  
}  
  
select {  
 float: left;  
 position: relative;  
 top: 130px;  
 left: 80px;  
}  
  
input[type=submit] {  
 position: relative;  
 float: left;  
 top: 130px;  
 left: 80px;  
}

-------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------

.chart {  
 position:relative;  
 height:600px;  
 width:1500px;  
 margin:2px;  
}

-------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------- -------------

—------------------------------------------------------***Under Maintenance***—----------------------------------------------------

**Evaluation**

**Met objectives**

|  |  |  |
| --- | --- | --- |
| **Objective** | **Met?** | **Comment** |
| 1.1 |  |  |
| 1.2 |  |  |
| 1.3 |  |  |
| 1.4 |  |  |
| 1.5 |  |  |
| 1.6 |  |  |
| 1.7 |  |  |
| 1.8 |  |  |
| 2.1 |  |  |
| 2.1.1 |  |  |
| 2.1.2 |  |  |
| 2.2 |  |  |
| 2.2.1 |  |  |
| 2.2.2 |  |  |
| 2.2.3 |  |  |
| 2.3 |  |  |
| 3.1 | Yes |  |
| 3.2 | Yes |  |
| 3.2.1 | Yes |  |
| 3.2.2 | Yes |  |
| 3.2.3 | Yes |  |
| 3.3 | Yes |  |
| 3.4 | Yes |  |
| 4.1 | Yes |  |
| 4.2 | Yes |  |
| 4.3 | Yes |  |
| 4.4 |  |  |
| 5.1 |  |  |
| 5.2 |  |  |
| 5.3 |  |  |
| 5.4 |  |  |
| 5.5 |  |  |
| 5.6 |  |  |
| 5.7 |  |  |
| 5.8 |  |  |
| 5.9 |  |  |
| 6.1 |  |  |
| 6.2 |  |  |
| 6.3 |  |  |
| 6.4 |  |  |
| 6.5 |  |  |
| 6.6 |  |  |
| 6.6 |  |  |
| 6.7 |  |  |
| 6.8 |  |  |
| 6.9 |  |  |
| 7.1 | Yes |  |
| 7.1.1 | Yes |  |
| 7.1.2 | Yes |  |
| 7.1.3 | Yes |  |
| 7.1.4 | Yes |  |
| 7.1.5 | Yes |  |
| 7.2 | Yes |  |
| 7.2.1 | Yes |  |
| 7.2.1.1 | Yes |  |
| 7.2.1.2 | Yes |  |
| 7.2.2 | Yes |  |
| 7.2.3 | Yes |  |
| 7.2.4 | Yes |  |
| 7.2.5 | Yes |  |
| 7.2.6 | Yes |  |
| 7.2.7 | Yes |  |
| 7.2.8 | Yes |  |
| 7.2.9 | Yes |  |
| 7.3 | Yes |  |
| 7.3.1 | Yes |  |
| 7.3.2 | Yes |  |
| 7.3.3 | Yes |  |
| 7.3.4 | Yes |  |
| 7.3.5 | Yes |  |
| 7.3.6 | Yes |  |
| 7.3.7 | Yes |  |
| 7.4 |  |  |
| 7.4.1 |  |  |
| 7.4.2 |  |  |
| 7.4.3 |  |  |

**Feedback**

**Feedback analysis**

**Extensions**