



# WEATHRU: UNDER THE WEATHER



*A Report submitted in partial fulfillment of the requirement for*

*The Award of the Degree of*

**Bachelor of Science (HONS)**

**IN**

**Computer Science**

Supervisor(s)

Dr. RAKHI GARG

Associate Professor

Dr. SARVESH PANDEY

Assistant Professor

Submitted by-

RIYANSHA SINGH

18229CMP005

RUBY SINGH

18229CMP006

**Department of Computer Science**

**MAHILA MAHA VIDYALAYA, BANARAS HINDU UNIVERSITY 221005**

**2021**

# CERTIFICATE

This is to certify that **RIYANSHA SINGH (18229CMP005)** and **RUBY SINGH(18229CMP006)** have certified Out the project under my supervision on theTopic “WEATHRU” during academic session 2020-2021.

As for as known to us ,the work reported in this project report has not been submitted to any institite/university for the award of any degree/deploma.

DR. RAKHI GARG

DR. SARVESH PANDEY

# CANDIDATE'S DECLARATION

We the undersigned solemnly declare that the project report is based on our own work carried out during the course our study under the supervision of **DR. RAKHI GARG** and **DR. SARVESH PANDEY**. We assert the statements made and conclusions drawn are an outcome of my research work. We further certify that

**I.** The work contained in the report is original and has been done by us under the general supervision of my supervisor.

**II.** The work has not been submitted to any other Institution for any other degree/diploma/certificate in this university or any other University of India or abroad

**III.** We have followed the guidelines provided by the university in writing the report.

**IV.** Whenever we have used materials (data, theoretical analysis, and text) from other sources, we have given due credit to them in the text of the report and giving their details in the references.

Riyansha

Ruby

Date: 05.07.21

Signature of the candidates

## TO WHOMSOEVER IT MAY CONCERN

This is to certify that the candidate has worked under my supervision and the Above statement made by the candidate is correct to the best of my knowledge.

Supervisor(s)

# ACKNOWLEDGEMENT

It is our privilege to express our sincerest regards to our project supervisors, **Dr. Rakhi Garg** and **Dr. Sarvesh Pandey** for their valuable inputs, able guidance, whole-hearted cooperation, constructive criticisms, for encouraging us to present the project on the topic “WEATHRU” and constantly motivating us to work harder during the course of the project.

Last but not the least, we express our thanks to our friends for their cooperation and support.

**Riyansha Singh**

**18229CMP005**

**Ruby Singh**

**18229CMP006**

# ABSTRACT

Nowadays we face a huge problem that knowing real weather status instantly in such places we need to know. It is complex and often challenging skill that involves observing and processing vast amounts of data.

The main concept of this project is to let users know the real weather conditions of their area including hourly forecast, daily forecast and air pollution level by using Public APIs. We also let users know current weather alerts and warnings on global level through our proposed system.

The first part of this project involves comparison of four different Public APIs and selecting the most suitable one.

The second part deals with the development of database which will contain the results of API call i.e. location, wind speed, humidity, temperature, pressure, date and time of call.

The last part is an implementation of one system architecture to integrate all our initial work. This section uses the concept of **Data Engineering**.

We are maintaining two more databases one to store information of Users registered on our system and other to store all the feedback of the Users related to our functioning.

# TABLE OF CONTENT

## 1. Introduction

- 1.1 Background and Motivation
- 1.2 Aim and Objective
- 1.3 Salient Features of the Project
- 1.4 Conclusion

## 2. Compare with Similar Works

- 2.1 Introduction
- 2.2 Present Weather Application

## 3. Technology Adapted

- 3.1 Introduction
- 3.2 Apache Airflow
- 3.3 Python Flask
- 3.4 PostgreSQL
- 3.5 Airflow Python Client
- 3.6 SQLAlchemy
- 3.7 HTML
- 3.8 CSS
- 3.9 Jinja2

## 4. Design

- 4.1 Overview
- 4.2 Backend Architecture

## 5. Working with API

- 5.1 Reasons for having API

- 5.2 Benefits of API over Web scraping
  - 5.3 Weather API comparison
  - 5.4 Selection of public API
  - 5.5 Other API used in the project
- 6. Backend development
  - 6.1 Introduction
  - 6.2 System Requirements
  - 6.3 Setting Up the Backend
  - 6.4 Methodology
  - 6.5 Conclusion
- 7. Database Design
  - 7.1 Purpose
  - 7.2 ER Schema
  - 7.3 ER Diagram
  - 7.4 Setting up a PostgreSQL Database
  - 7.5 Creating Tables
- 8. User Interface Design
  - 8.1 Frontend Web development
  - 8.2 Routes used in the Project
  - 8.3 System and Software Specifications
  - 8.4 Designing
  - 8.5 Views for Webpage
- 9. Summary and Conclusion
- 10. Advantages and Limitations of Proposed Architecture
- 11. Future Works

## 12. References

# 1.Introduction

In this chapter we hope to provide a brief description on the background and motivation about our project, point out the aim and objectives and to present proposed solution through our project.

## 1.1 Background

It is very important to get educated about current weather situations of a particular location as preferred since it affects day to day life to everyone. It is more effective if we can quickly update current weather status of a required location, as it makes easy to handle our day to day activities and livelihood too.

A huge problem that we are facing is that of getting accurate weather conditions from place to place in India being surrounded by seas from three sides, faces variations in weather status place to place and poor Air quality as well. The people need to be aware of the environment where they are breathing. Therefore weather forecasting is two steps process involving observation and forecasting.

## 1.2 Aim

To make a software that can be used to get know the real time as well as future weather updates for the required location using efficient Weather APIs. Also to make the system capable of informing of any global weather warnings and alerts.

## 1.3 Objective:

- Study about weather forecasting system and application.
- Study about technologies like web application, android app which can overcome problems.



- Design and develop a system which can properly get updates of the weather status.
- Evaluate proposed solutions.

## **1.4 Salient Features of the Project**

### **☐ Locations Covered**

At present, we have covered locations in three major countries of the world **Australia, India, USA** . Users can get weather updates of prime locations of these countries.

In future we can easily extend our existing database to include more countries and location.

### **☐ Functionalities**

- Users can get real time weather updates of the location including temperature, pressure, humidity, wind speed, sky cover.
- Similar information regarding weather is available for users interested in Hourly and 4 days Daily forecast.
- To let users know about air pollution levels in their area, we provide them with AQI levels through user friendly illustration.
- Latest weather news and alerts is also a crucial feature of our system.
- Users visiting our website has facility to register themselves to get weather updates of their current location directly through their email.
- Users can even submit their valuable feedback related to our functioning.

## **1.5 Conclusion**

Users can easily benefit from our system as we involve the use of fast, efficient and accurate public APIs.

## **2. Compare with similar works**

### **2.1 Introduction**

There is a large number of weather application used in the world like Accuweather, Yahoo! Weather, Weather live and so on. They provide their user with weather information. In this chapter we apply a comparison between our approach and them.

### **2.2 Present Weather Application**

#### **☐ Accuweather**

This free app is designed for iPhone users. It includes something called “iPhone weather station,” which allows you to receive alerts for warnings about severe weather, and it can even forecast out to 15 days, as opposed to the traditional 10 days limit.

Though it is very useful for iPhone users, but in India almost all people are android users so it seems to be meaningless and here we come with the solution “WEATHRU”. In Accuweather, the screen has more data which can confuse user from getting actual information. In “WEATHRU” we are providing user aesthetically pleasant design and it focuses on being minimal and easy to understand

#### **☐ Weather 2020**

It is the first and only source capable of producing accurate, long-range forecasts for businesses and consumers. It has Weather forecast for the whole world. They have a free tier plan for use limited by zip code boundaries.

But their period of prediction is inaccurate, particularly in the autumn season where “WEATHRU” gives Sharp information via open weather map API in all seasons.

## **❑ DARK SKY**

Dark Sky is the most accurate source of hyper local weather information. Hourly and daily forecast data are updated every hour. Severe weather alerts are updated in real-time as well.

But its services are restricted to individuals over 13 years of age. Whereas, WEATHRU does not possess this kind of restrictions, its services are available for everyone.

## **3. Technology Adapted**

### **3.1 Introduction**

This chapter provides specifics and details about the technology that we have adapted to solve the problem through implementing our proposed solution. Furthermore, it will point the reasons and the ways that these techniques and technology are appropriate for the proposed solution.

We used following frameworks and techniques for the implementation of our proposed structure.

- Apache Airflow
- Python Flask
- PostgreSQL
- Airflow Python Client
- SQLAlchemy
- HTML
- CSS
- Jinja2

### **3.2 Apache Airflow**

The rising industries motivates the need for workflow management systems (WMS) in modern practices, and provides a wish list of features and functions which leads them to choose Apache Airflow as our WMS of their choice.

Airflow is a WMS that defines tasks and their dependencies as code, executes those tasks on a regular schedule, and distributes task execution across worker processes. Airflow offers an excellent UI that displays the states of currently active and past tasks, shows diagnostic information about task execution, and allows the user to manually manage the execution and state of tasks.

## ❑ Workflows are “DAGs”

Workflows in Airflow are collections of tasks that have directional dependencies. Specifically, Airflow uses directed acyclic graphs — or DAG for short — to represent a workflow. Each node in the graph is a task, and edges define dependencies amongst tasks (The graph is enforced to be acyclic so that there are no circular dependencies that can cause infinite execution loops) where each task purpose is defined by the use of Operators.

## ❑ Operators

The Operators tell *what* is there to be done. The Operator should be atomic, describing a single task in a workflow that doesn't need to share anything with other operators.

The basic operators provided by this platform include:

- **BashOperator:** for executing a bash command
- **PythonOperator:** to call Python functions
- **EmailOperator:** for sending emails
- **SimpleHttpOperator:** for calling HTTP requests and receiving the response-text
- **DBoperators(e.g. MySQLOperator, SqliteOperator, PostgresOperator, MsSqlOperator, OracleOperator, etc.)** for executing SQL commands
- **Sensor:** to wait for a certain event (like a file or a row in the database) or time.

A task is the instance of the operator, like:


```
task_1= PythonOperator(task_id='report_blackouts', python_callable=eneas_check, dag=dag)
```

Here, the task\_1 is an instance of PythonOperator that has been assigned a task\_id, a python\_callable function, and some DAG.

## ❑ DAG Creation

Airflow provides a very easy mechanism to define DAGs: a developer defines his DAG in a Python script. The DAG is then automatically loaded into the DAG engine and scheduled for its first run. Modifying a DAG is as easy as modifying the Python script! The ease with which a developer can get started on Airflow contributes greatly to its draw.

Once your DAG has been loaded into the engine, you will see it on the Airflow home screen. On this page, you can easily hide your DAG from the scheduler by toggling an on/off switch — this is useful if one of your downstream systems is undergoing lengthy maintenance. Although Airflow handles failures, sometimes it’s best to just disable the DAG to avoid unnecessary failure alerts. In the screen shot below, the “cousin domains” DAG is disabled.

 **Airflow**

DAGs

Data Profiling

Browse

Admin

Docs

22:16 UTC

DAGs

Show 100 entries

Search:

		DAG	Owner	Statuses	Links
1	<input type="checkbox"/>	cousin_domains	mbodola	<div><div>7</div><div></div><div></div><div></div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div>
2	<input checked="" type="checkbox"/>	ep_demo	sanand	<div><div>203</div><div></div><div></div><div></div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div>
3	<input checked="" type="checkbox"/>	forwarders_model	kmandich	<div><div>20</div><div></div><div>7</div><div>5</div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div>
4	<input checked="" type="checkbox"/>	refresh_asn_data	kmandich	<div><div>30</div><div></div><div></div><div></div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div>

Showing 1 to 4 of 4 entries

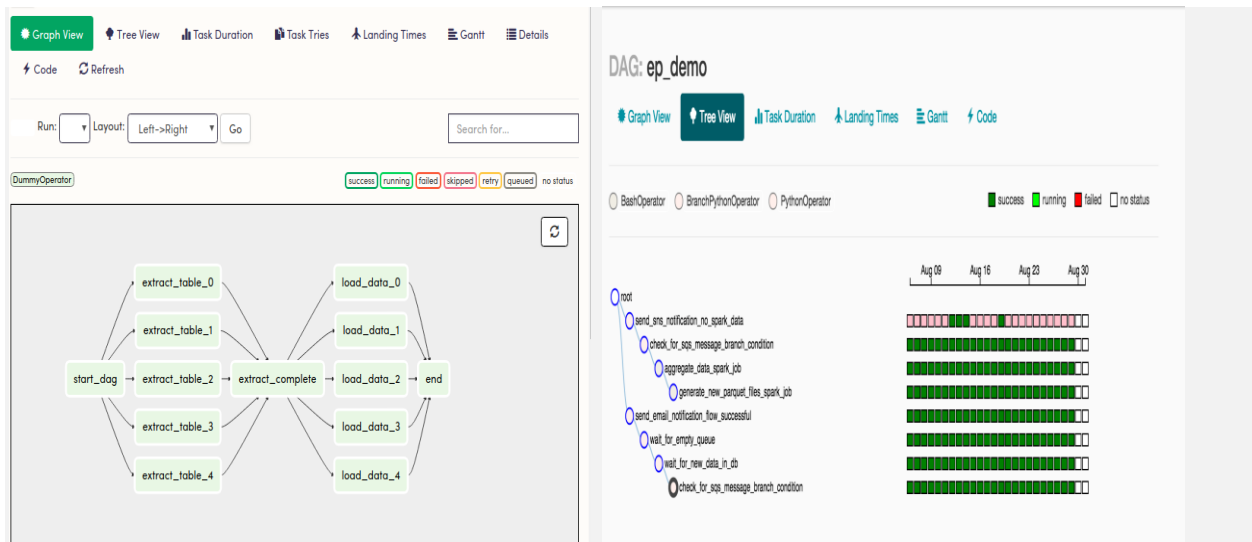
Previous

1

Next

Airflow provides a few handy views of your DAG. The first is the Graph View as shown which define the dependencies among various tasks described by the DAG script.



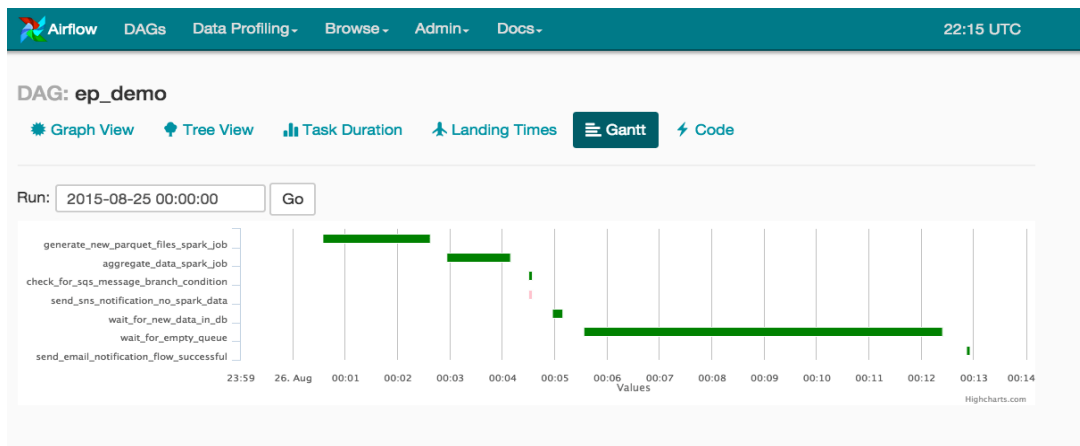


Over time, we can quickly assess the run status for multiple days by referring to Airflow’s Tree View. In the image above, a vertical column of squares relates to all the tasks in a day’s run of a DAG. For July 26, the run completed successfully because all squares are forest green!

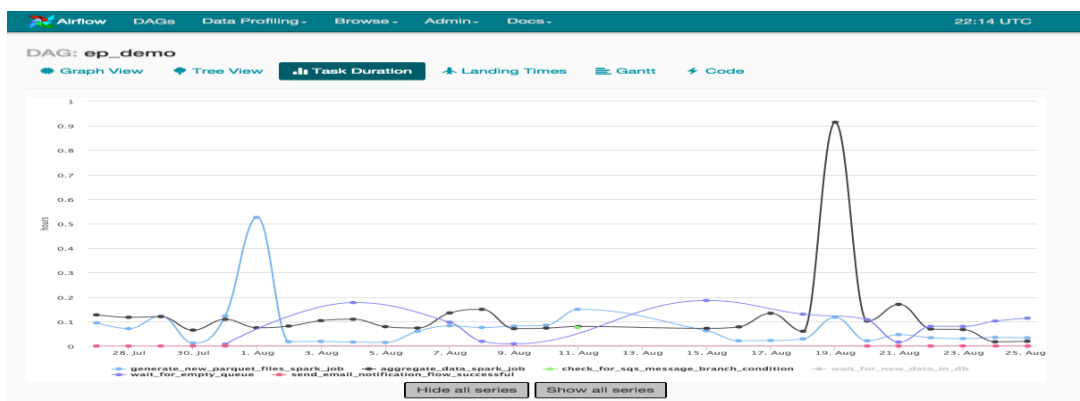
## ❑ DAG Metrics and Insights

For every DAG execution, Airflow captures the run state, including any parameters and configuration used for that run and provides this run state at your fingertips. We can leverage this run-state to capture information such as model versions for different versions of machine-learned models that we use in our pipeline. This helps us in both issue diagnosis and attribution.

In terms of pipeline execution, we care about speeding up our pipelines. In the Gantt chart below, we can view how long our scheduled jobs take in hopes of speeding them up.



We also care about the time variability of runs. For example, will a run always take 30 minutes to complete or does the time vary wildly?



Since Airflow is easy to set up (example via **pip install** if you only want releases) as an admin. It has a terrific UI. It is developer friendly as it allows a developer to set up a simple DAG and test it in minutes, we consider Airflow as the star of our project.

### 3.3 Python Flask

The user interaction with our application is considerable; therefore, we should guarantee that it ensures all aspects that it should be enriched with. Moreover, Airflow is easily integrable with Flask hence we thought to adapt it for developing our client-side web applications. Flask is a micro web framework written in Python. It is classified as a microframework because it does not require particular tools or libraries. It has no database abstraction layer, form validation, or any other



components where pre-existing third-party libraries provide common functions. However, Flask supports extensions that can add application features as if they were implemented in Flask itself. Extensions exist for object-relational mappers, form validation, upload handling, various open authentication technologies and several common framework related tools.

### 3.4 PostgreSQL

The weather forecasting system frequently involves with information updates and retrievals. This occurs due to users requesting API's for weather updates in their area. This process involves a huge amount of data that flow from and to database. PostgreSQL databases are not limited in size and can handle multiple concurrent writes. PostgreSQL manages concurrency through a system known as ***multiversion concurrency control (MVCC)***, which gives each transaction a "snapshot" of the database, allowing changes to be made without being visible to other transactions until the changes are committed. This largely eliminates the need for read locks, and ensures the database maintains the ACID (atomicity, consistency, isolation, durability) principles in an efficient manner.

### 3.5 Airflow Python Client

The thing to note is that Airflow is built for static flows. So if our requirements demand dynamic workflow creation Airflow will not fit the bill. We were not able to render an attractive GUI for Weather on the spot application using just Airflow, so we have to integrate it with a Python framework Flask to present our workflow controlled by Airflow. This integration was only possible with Airflow Python Client. Airflow Python Client uses REST API endpoints supported by Airflow to trigger a workflow (DAG) manually whenever user initiates an action.

### 3.6 SQLAlchemy

SQLAlchemy is a high performing and accurate python SQL toolkit that facilitates the communication between Python programs and database. In our case it helped our web application to communicate with PostgreSQL database in the backend. A benefit many developers enjoy with SQLAlchemy is that it allows them to write

python code in their project to map from the database schema to the applications python objects.

### **3.7 HTML**

HTML is the code that we used to structure the web page and its content.

### **3.8 CSS**

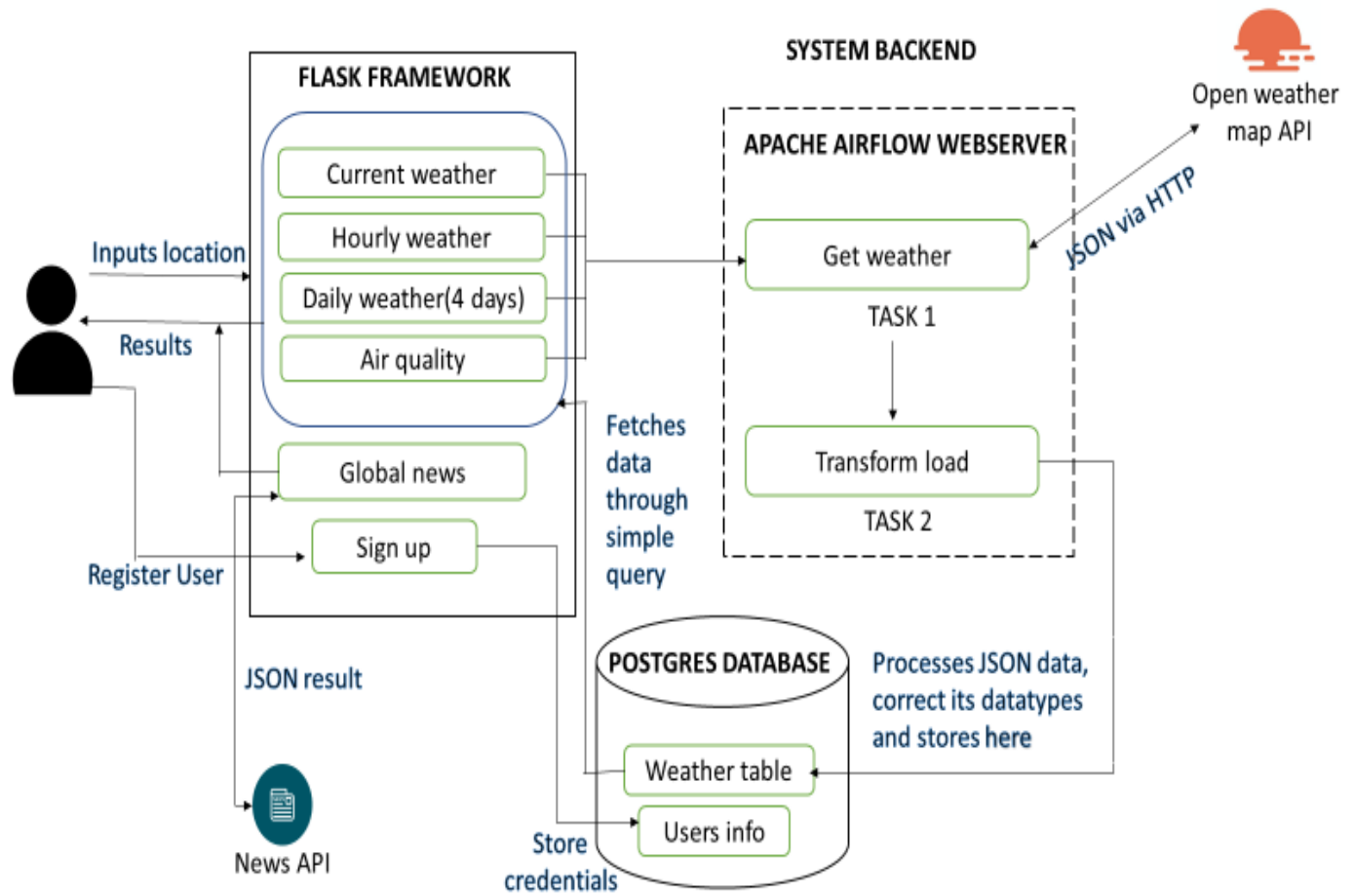
CSS is the language we used for describing the presentation of web pages, including colors, layout, and fonts.

### **3.9 Jinja 2**

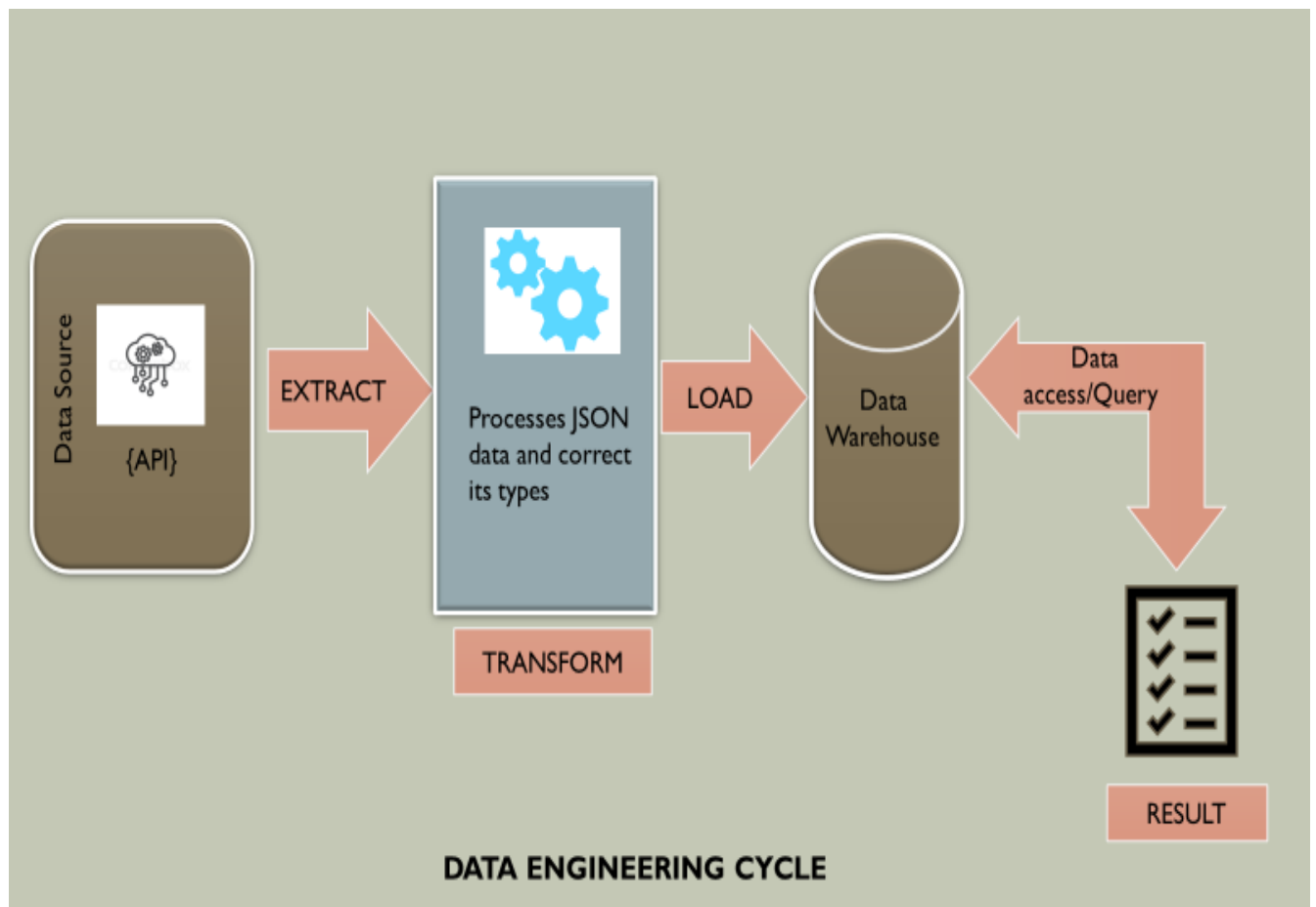
In order to feed users data fetched from database or the API returned JSON data through simple HTML, we employed Jinja2 as our templating engine. Jinja is a fast, expressive, extensible templating engine. Special placeholders in the template allowed us to write code similar to python syntax. Then the template is passed data to render the final document.

# 4. Design

## 4.1 Overview



## 4.2 Backend Architecture



## 5. Working with API

API stands for Application Programming Interface. An API is a software intermediary that allows two applications to talk to each other. In other words, an API is the messenger that delivers your request to the provider that you're requesting it from and then delivers the response back to you.

### 5.1 Reason for having API

APIs make it possible for sites to let other apps and developers use their data for their own applications and purposes.

They work by “exposing” some limited internal functions and features so that applications can share data without developers having direct access to behind-the-scenes code and their large databases.

### 5.2 Benefits of API over Web Scraping

- Since websites change their layout every day, you would have to change your scraping code from time to time to make sure that everything keeps working.
- When regular and similar bulk data extraction is the requirement, API can be the way to go. It can help in automating the data extraction process, including all kinds of documents from pdfs to images and invoices

### 5.3 Weather API comparison

This is a comparison study of different public APIs that can be used to obtain weather data for our application.

The comparison is based on four weather APIs and below is a short description of them.

- **Accuweather API**

The Accuweather API delivers detailed current, historical, and forecasted weather information for locations all over the world. Developers could build a wide range of innovative and engaging weather data-powered applications using AccuWeather. The free trial and **paid plans** include current conditions, 24-hour historical current conditions, forecasts, and indices.

However, they are not capable of making large number of free API calls per day.

- **Weatherbit API**

Weatherbit offers 5 different APIs for forecasts, historical data, and other weather data such as air quality, soil temperature, and soil moisture. Collecting data from weather stations, Weatherbit uses machine learning and AI to help predict the weather.

Boasting a 95% uptime and highly responsive API servers, Weatherbit provides a free limited-functionality account for a single API key. If one is looking to create a commercial app, note that the free API subscription will not be enough and will have to upgrade to one of the paid plans.

- **Aeris API**

Aeris Weather API provides access to weather data and forecasts as well as storm reports, earthquake warnings, and other unique data for premium subscribers.

- **Openweathermap API**

OpenweatherMap offers weather data APIs for different types of timeline data. In a solution inspired by crowdsourcing projects like Wikipedia, weather data is collected from meteorological broadcast services worldwide and over 40,000 weather stations. This freemium solution also has a feature-limited free option that allows access to the 5 days/3 hour forecast API, as well as weather alerts and a weather map.

## 5.4 Selection of public weather API

We chose **Openweathermap API** as our public API because of its user-friendly features, easy to access API endpoints and its long-term free plan.

### 5.4.1 Generating API key

We need to get an API key to identify ourselves, this authenticates us to the services of Openweathermap. To get it-

- Visit <http://openweathermap.org> and sign up to create a free account.
- Visit your profile and click my API key to start.
- Your secret API Key will be shown. Copy it and save it securely for future purpose.

This API key will be activated and ready to use within an hour.

### 5.4.2 API endpoints available with openweathermap

- Current weather

<https://api.openweathermap.org/data/2.5/weather?q={city name}&appid={API key}>

- Hourly and Daily Forecast

<https://api.openweathermap.org/data/2.5/onecall?lat={lat}&lon={lon}&exclude={part}&appid={API key}>

## 5.5 Other API used in the project

Apart from openweathermap API, we have used two more API in our project to make it more presentable.

- **Weatherbit API**

We have used this API to enable our Users get real time air pollution levels in their area. This Air Quality API returns a 3 day / hourly forecast of air quality condition as for any location in the world. Additionally, this API returns an air quality index score (AQI) which we are interested in.

Endpoint used

<https://api.weatherbit.io/v2.0/forecast/airquality?lat=38.0&lon=78.0&key=APIKEY>

We did not use openweathermap Air Pollution API since it returns air pollution levels at any location on a scale of 1 to 5 where 1 stands for Good and 5 stands for Very Poor which Users might be less likely to be interested in.

- **News API**

In order to keep our users updated with weather alerts and news around the globe and make our application more appealing we have provided them with facility to get up-to-the-minute latest weather news by simply clicking News tab on our web GUI.

Endpoint used

[https://newsapi.org/v2/everything?q=tesla&from=2021-06-01&sortBy=publishedAt&apiKey=API\\_KEY](https://newsapi.org/v2/everything?q=tesla&from=2021-06-01&sortBy=publishedAt&apiKey=API_KEY)



## 6. Backend development

### 6.1 Introduction

We have used Apache Airflow for developing our backend because it provides easy workflow management system.

Through this we go over the operations of data engineering called Extract, Transform, Load (ETL) and show how they can be automated and scheduled using [Apache Airflow](#). Extracting data can be done in a multitude of ways, but one of the most common ways is to query a [WEB API](#). If the query is successful, then we will receive data back from the API's server. Often times the data we get back is in the form of [JSON](#). JSON can pretty much be thought of a semi-structured data or as a dictionary where the dictionary keys and values are strings. Since the data is a dictionary of strings this means we must *transform* it before storing or *loading* into a database. Airflow is a platform to schedule and monitor workflows and in this post I will show you how to use it to extract the daily weather in New York from the [OpenWeatherMap](#) API, convert the temperature to Celsius and load the data in a simple [PostgreSQL](#) database

Before using it, we must install its dependencies and check system compatibilities for it.

### 6.2 System Requirements

- Windows 10 Pro
- Ubuntu18.04 LTS (minimum 4 GB RAM required)
- Python 3.6.9

- Apache Airflow 2.0.1

## 6.3 Setting up the backend

By default, Airflow webserver runs on <http://localhost:8080>. Following are the steps to be followed to get backend Airflow webserver running-

- **Step 1:** Open Ubuntu 18.04 lts terminal and initiate the airflow database.

```
root@Dell: ~  
root@Dell:~# airflow db init  
DB: postgresql+psycopg2://postgres:***@localhost:5432/airflow  
[2021-07-02 00:03:56,710] {db.py:674} INFO - Creating tables  
INFO [alembic.runtime.migration] Context impl PostgresqlImpl.  
INFO [alembic.runtime.migration] Will assume transactional DDL.
```

- **Step 2:** After initiation start Airflow webserver.

```
root@Dell:~# airflow webserver -p 8080
```

```
      _ _ _ _ _  
      | |   | |   | |   | |   | |   | |   | |   | |   | |  
      |_|___|_|___|_|___|_|___|_|___|_|___|_|___|_|___|_|  
      / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \  
     /   /   /   /   /   /   /   /   /   /   /   /   /   /  
    /___/___/___/___/___/___/___/___/___/___/___/___/___/  
[2021-07-02 00:08:37,049] {dagbag.py:448} INFO - Filling up the DagBag from /dev/null  
Traceback (most recent call last):
```

- **Step 3:** Open a separate Ubuntu terminal and start the Airflow scheduler

```
root@Dell: ~  
root@Dell:~# airflow scheduler  
  
[2021-07-02 00:12:01,987] {scheduler_job.py:1247} INFO - Starting the scheduler  
[2021-07-02 00:12:01,989] {scheduler_job.py:1252} INFO - Processing each file at most -1 times  
[2021-07-02 00:12:02,102] {dag_processing.py:250} INFO - Launched DagFileProcessorManager with pid: 83
```

- **Step 4:** Airflow will be running on localhost:8080

## 6.4 Methodology

By using Airflow, we have tried to build a workflow management and scheduling system which will get triggered once user inputs his location parameter to get results out of the web.

### ❑ Weatherdag.py

This is the main caller that will control the system and perform all tasks defined in the assignment. This is stored as a python file we will call by convention DAG definition file.

Step by Step buildup of DAG definition file:

- In this file, the first thing we do is to import all the necessary libraries:

```
1  from datetime import datetime, timedelta
2  from airflow import DAG
3  from airflow.operators.bash_operator import BashOperator
4  from airflow.operators import PythonOperator
5  import os
6  from airflow.hooks import PostgresHook
7  import json
8  import numpy as np
9
```

- Next, we define the Python function that will transform and load our JSON data object into our database. This can be seen below in the function, `load_data`:

```

getweather.py > ...
2 import json
3 from datetime import datetime, timedelta
4 import os
5
6 def get_weather():
7     """
8     Query openweathermap.com's API and to get the weather for
9     Varanasi and then dump the json to the /src/data/ directory
10    with the file name "<today's date>.json"
11    """
12    with open('/c/Users/Admin/airflow/DAGS/src/get_city.json') as JsonFile:
13        data = json.load(JsonFile)
14        JsonData = data["weather_input"]
15        count = len(JsonData)
16        loc = (JsonData[count-1].get("city"))
17        print(loc)
18
19    url = 'http://api.openweathermap.org/data/2.5/weather?q={}&units=imperial&appid=a728a370130711e2e8192f0aaa7ecb4b'
20
21
22    result = requests.get(url.format(loc))
23
24    if result.status_code == 200 :
25
26        json_data= result.json()
27        file_name= str(datetime.now().date()) + '.json'
28        tot_name= os.path.join(os.path.dirname(__file__), 'data', file_name)
29
30        with open(tot_name, 'w') as outputfile:
31            json.dump(json_data, outputfile)
32
33    if __name__ == "__main__":
34        get_weather()

```

we instantiate a PostgresHook object and pass our postgres connection id, `weather_id` (will be discussed), to the constructor. We then get the current day's date so what we can load the appropriate JSON data from the API request of this day.

Once we load the data, we can observe that it is a dictionary with string key-value pairs. We then transform the values in this dictionary and check to make sure that the numerical values are not NaNs using NumPy's `isnan` function. If there are any NaNs in the numerical data, we flag this data as invalid.

Next, we cast all the individual data field values into a tuple which we then pass as a parameter along with the SQL insertion command, `insert_cmd`, into the PostgresHook object's run method. The run method then inserts the data into the database.

- We then define our default parameters, `default_parameter`. This is a dictionary that includes such information as the owner of the DAG and how many times, and how frequently to retry running the DAG if it fails

- Next, we instantiate our DAG. The `dag_id` needs to be unique. The `dag_id` will be passed off to all the tasks which needs to be completed during their instantiation. Every DAG will have a datetime object called the `start_date` which should be a future date as well as `timedelta` object, `schedule_interval`, that dictates how often to run the DAG.
- Next, we instantiate a BashOperator `task1` which becomes a task that executes the API call through `getweather.py` script.
- We next instantiate a PythonOperator, `task2`, that transforms and loads the JSON data that was pulled from API into the database.

Notice that we pass the function, `load_data` through the constructor as a keyword parameter, `python_callable`.

- Finally, we set the up the DAG pipeline by saying one task depends on the other being completed. We can declare that `task1` needs to be completed before `task2` can be started by using the following notation at the bottom of your `.py` script

```
task1 >> task2
```

```

60 # Define the default dag arguments.
61 default_args = {
62     'owner' : 'Mike',
63     'depends_on_past' : False,
64     'email' : ['mdh266@gmail.com'],
65     'email_on_failure' : False,
66     'email_on_retry' : False,
67     'retries' : 5,
68     'retry_delay' : timedelta(minutes=1)
69 }
70
71
72 # Define the dag, the start date and how frequently it runs.
73 # I chose the dag to run everyday by using 1440 minutes.
74 dag = DAG(
75     dag_id='weatherDag',
76     default_args=default_args,
77     start_date=datetime(2017,8,24),
78     schedule_interval=timedelta(minutes=1440))
79
80
81 # First task is to query get the weather from openweathermap.org.
82 task1 = BashOperator(
83     task_id='get_weather',
84     bash_command='python ~/airflow/dags/src/getWeather.py' ,
85     dag=dag)
86
87
88 # Second task is to process the data and load into the database.
89 task2 = PythonOperator(
90     task_id='transform_load',
91     provide_context=True,
92     python_callable=load_data,
93     dag=dag)
94
95 # Set task1 "upstream" of task2, i.e. task1 must be completed
96 # before task2 can be started.
97 task1 >> task2
98

```

## ▪ task 1: Calling an API in python

To use a Web API to get data, you make a request to a remote web server, and retrieve the data you need. In Python, this is done using the requests module. Below I wrote a module, [getWeather.py](#), that uses a GET request to obtain the weather.

After the request has been made, I check to see if it was successful by checking the [status\\_code](#),

```
result.status_code == 200
```

otherwise, I print an error. If the request is successful, then weather data is returned and is then dumped into a JSON file with a name that is the current date using the [JSON](#) package.

The above code is stored in a file title [getWeather.py](#)

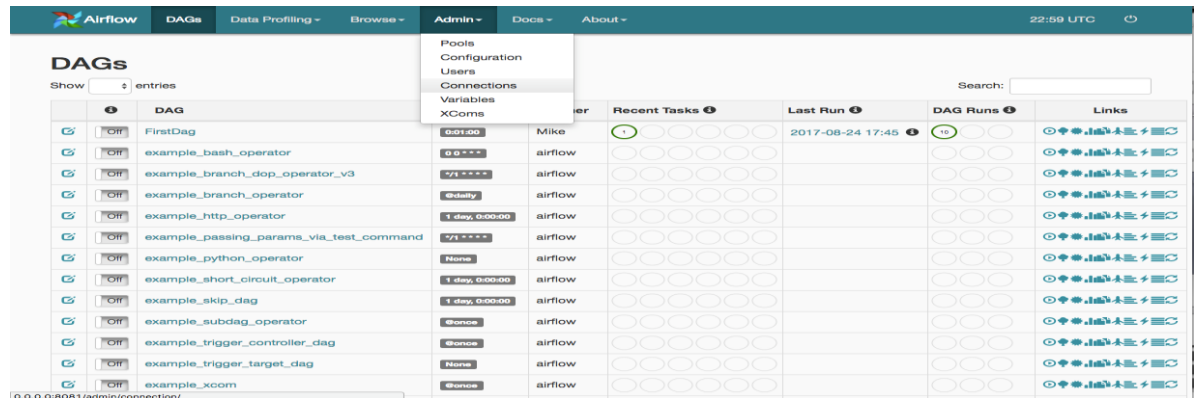
Note that this is the exact Bash command that we used to have Airflow collect daily weather data.

## ▪ task 2: Loading data in database

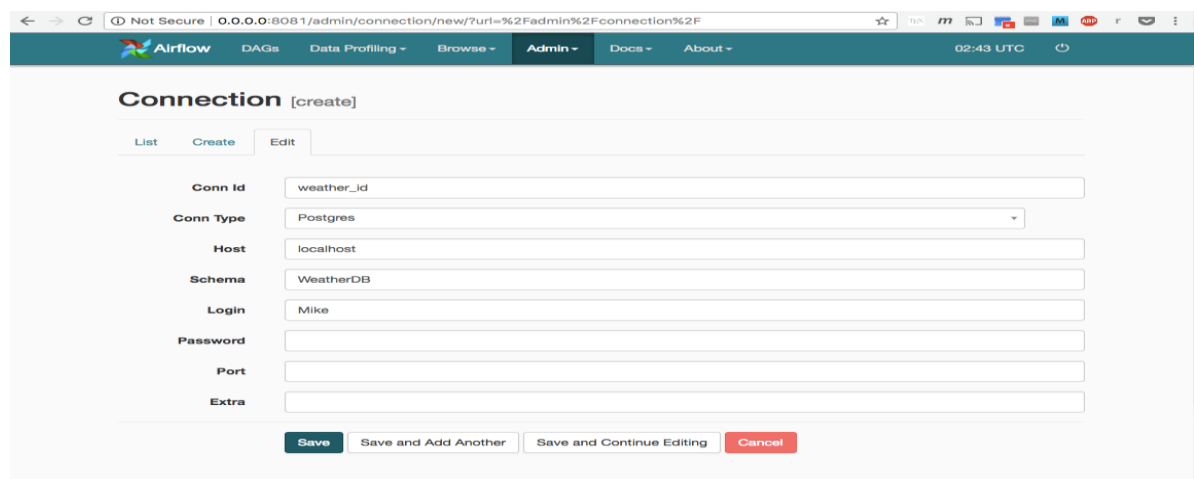
We call the `load_data` python function for implementation of task 2 with id 'transform\_load'.

## ➤ Creating connection with database in Airflow

We need to create a connection to the database (postgres\_conn\_id). We do this by going the Airflow Webserver in our web-browser and clicking on Admin tab and then choosing connections as shown below:



Next, click on the create link and enter the information relevant to create our Postgres connection.



We then click save and can now use [weather\\_id](#) as our [postgres\\_conn\\_id](#) connection id.

## **6.5 Conclusion**

Airflow is an extremely useful tool for building data pipelines and scheduling jobs in Python. Through our backend work of 'WEATHRU' application, we performed an ETL job using Airflow.



## 7. Database Design

### 7.1 Purpose

Though we are directly using OpenWeatherMap API to fetch our results, we still needed the support for Database.

- API will give results in JSON format which is a little difficult to work with as it exists as an array of strings and it can only be accessed with some dot/bracket notation. Hence using database ensures efficient retrieval of information through simple query.
- Additionally, using a database over an API can add security, authorization, authentication, validation, business constraints, abstractions, limiting access to only required capabilities, simplification, software models and many more.

### 7.2 ER Schema

- **Weather information table**

The screenshot shows a database management tool interface for configuring a table named 'weather\_table'. The 'Columns' tab is selected, displaying a table with the following columns:

	Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?
<input checked="" type="checkbox"/>	city	text			No	No
<input checked="" type="checkbox"/>	country	text			No	No
<input checked="" type="checkbox"/>	latitude	real			No	No
<input checked="" type="checkbox"/>	longitude	real			No	No
<input checked="" type="checkbox"/>	todays_date	date			No	No
<input checked="" type="checkbox"/>	humidity	real			No	No
<input checked="" type="checkbox"/>	pressure	real			No	No
<input checked="" type="checkbox"/>	min_temp	real			No	No
<input checked="" type="checkbox"/>	max_temp	real			No	No
<input checked="" type="checkbox"/>	temp	real			No	No
<input checked="" type="checkbox"/>	weather	text			No	No
<input checked="" type="checkbox"/>	date_time	timestamp without time zone			No	No

At the bottom of the interface, there are buttons for 'Cancel', 'Reset', and 'Save'.







- **Users information table**

login\_users

General
Columns
Advanced
Constraints
Parameters
Security
SQL

Inherited from table(s)
Select to inherit from...

Columns
+

	Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?
 	username	text			<input type="checkbox"/> No	<input type="checkbox"/> No
 	email	text			<input type="checkbox"/> No	<input type="checkbox"/> No
 	password	text			<input type="checkbox"/> No	<input type="checkbox"/> No

i
?

Cancel
Reset
Save









- Users Feedback table

feedback

General
Columns
Advanced
Constraints
Parameters
Security
SQL

Inherited from table(s)
Select to inherit from...

Columns
+

	Name	Data type	Length/Precision	Scale	Not NULL?	Primary key?
 	name	text			<input type="checkbox"/> No	<input type="checkbox"/> No
 	email	text			<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes
 	message	text			<input type="checkbox"/> No	<input type="checkbox"/> No
 	reply	text			<input type="checkbox"/> No	<input type="checkbox"/> No

i
?

Cancel
Reset
Save

This weather feedback table is Admin controlled. Any feedback or query asked by the user is stored in column ‘message’ and replies to it is stored in column ‘reply’

### 7.3 ER Diagram



## 7.4 Setting Up a PostgreSQL database

- **Step 1:** The first thing we will need to do is import the libraries to interact with SQL in Python. These are `sqlalchemy` and `sqlalchemy_utils`.
- **Step 2:** Then we create the database engine by filling values in given general command

`dialect+driver://username:password@host:port/database`

The dialect in our case will be `postgresql` and driver will be `psycopg2`.

- **Step 3:** Next, we create the database if it doesn't already exist.
- **Step 4:** We connect to our database, dbname with the psycopg2 library using the command

```
conn = psycopg2.connect(database = dbname, user = username,  
password = pass)
```

- **Step 5:** We now get our cursor object

```
cur = conn.cursor()
```

- **Step 6:** In order to select rows from database, we use `cur.execute ()` and to view the rows from the table we use `cur.fetchall()`.

## 7.5 Creating tables

❑ Weather\_table

```

1  from flask_sqlalchemy import sqlalchemy
2  from sqlalchemy import create_engine
3  from sqlalchemy_utils import database_exists, create_database
4  import psycopg2
5
6  def make_database():
7      """
8          Make the Postgres database and create the table
9      """
10
11     dbname = 'weatherDB'
12     username = 'postgres'
13     password = 'riyansha'
14     tablename = 'weather_table'
15
16
17     engine = create_engine('postgresql+psycopg2://%s:%s@localhost/%s'%(username,password,dbname))
18
19     if not database_exists(engine.url):
20         create_database(engine.url)
21
22     conn = psycopg2.connect(database = dbname, user = username, password = password)
23
24     cur = conn.cursor()
25
26
27     create_table = """CREATE TABLE IF NOT EXISTS %s
28         (
29             city            TEXT,
30             country         TEXT,
31             latitude        REAL,
32             longitude       REAL,
33             todays_date     DATE,
34             humidity        REAL,
35             pressure        REAL,
36             min_temp        REAL,
37             max_temp        REAL,
38             temp            REAL,
39             weather         TEXT
40         )
41     """ % tablename
42
43     cur.execute(create_table)
44     conn.commit()
45     conn.close()
46
47     if __name__ == "__main__":
48         make_database()

```

## ❑ Users\_info

```

22     conn = psycopg2.connect(database = dbname, user = username, password = password)
23
24     cur = conn.cursor()
25
26     create_table = """CREATE TABLE IF NOT EXISTS %s
27         (
28             username      TEXT,
29             email          TEXT,
30             password       TEXT
31         )
32         """ % tablename
33
34     cur.execute(create_table)
35     conn.commit()
36     conn.close()
37
38 if __name__ == "__main__":
39     make_database()

```

## ❑ Feedback

```

3 from sqlalchemy_utils import database_exists, create_database
4 import psycopg2
5
6 def make_database():
7
8     dbname = 'Users_info'
9     username = 'postgres'
10    password = 'riyansha'
11    tablename = 'feedback'
12
13
14    engine = create_engine('postgresql+psycopg2://%s:%s@localhost/%s'%(username,password,dbname))
15
16    if not database_exists(engine.url):
17        create_database(engine.url)
18
19    conn = psycopg2.connect(database = dbname, user = username, password = password)
20
21    cur = conn.cursor()
22
23    create_table = """CREATE TABLE IF NOT EXISTS %s
24        (
25            Name           TEXT,
26            email          TEXT,
27            Message        TEXT,
28            Reply          Text
29        )
30        """ % tablename
31
32    cur.execute(create_table)
33    conn.commit()
34    conn.close()

```

# 8. User Interface Design

## 8.1 Frontend web development

While there are numerous technology that build web applications that serve dynamic content, one that has really caught the attention of the development community is Flask web framework. Local access link for Flask is localhost:5000.

We have `render_template()` to pass static HTML web pages to our website and Flask routing technique to help user remember the URLs.

### 8.1.1 Routes used in the Project

- **“/” and “/results”** – URL will display the home page where users can search location and get current weather results.
- **“/signin” and “/success”**– URL will render signup.html page where users can enter his details which on submit will first get verified from database. If found correct user will get logged in that session.
- **“/signout”** – URL will release the session variable and user gets signed out of that session.
- **“/daily”** – URL will display coming 4 days forecast.
- **“/hourly”** – URL will display hourly forecast for current day at an interval of 4 hours.
- **“/air\_quality”** – URL will display AQI levels in the requested location.
- **“/news”** – URL will give latest global weather alerts and news.
- **“/contacts”** – URL will render contact.html page where users can post their queries and feedback related to our functioning.
- **“/about”** – URL will give information related to our website by rendering about.html page

## 8.2 System and Software Specifications

- Processor: Intel Core i3 third gen and above
- RAM: 4 GB
- Operating system: Windows 10 and above

- Python Flask 3.6
- Airflow Python Client

## 8.3 Designing

### ○ Importing necessary libraries and building connection with database

We have imported Psycopg2 library to build connection with database using registered username and password in PgAdmin4 module. By default the webserver for database runs on port 5432.

```

1  from flask import Flask, render_template, request, redirect, url_for, session
2  import airflow_client.client
3  from airflow_client.client.api import config_api, dag_api, dag_run_api
4  from airflow_client.client.model.dag_run import DAGRun
5  from airflow_client.client.model.error import Error
6  from flask_sqlalchemy import SQLAlchemy
7  from datetime import datetime
8  import psycopg2
9  from jinja2 import Template

try:
    conn = psycopg2.connect(database="WeatherDB", user="postgres", password="riyansha", host="localhost", port="5432")
    print("connected")

except:
    print("unable to connect")

try:
    con = psycopg2.connect(database="Users_info", user="postgres", password="riyansha", host="localhost", port="5432")
    print("connected")

except:
    print("unable to connect")

```

### ○ Defining Airflow trigger function

Since our backend exist on a different server than our main web GUI we had to make use of a third-party client i.e. **Airflow python Client** to externally trigger our DAG behind whenever user submits his location requirements.



```

39 def trigger():
40     configuration = airflow_client.client.Configuration(
41         host = "http://localhost:8080/api/v1",
42         username = 'riyansha',
43         password = 'singh@123'
44     )
45
46     dag_id = 'weatherdag'
47
48     with airflow_client.client.ApiClient(configuration) as api_client:
49
50         dag_run_api_instance = dag_run_api.DAGRunApi(api_client)
51         try:
52             dag_run= DAGRun(
53                 #dag_run_id='manual__2021-05-29T17:19:19.211364+00:00',
54                 dag_id=dag_id,
55                 external_trigger=True,
56             )
57             api_response = dag_run_api_instance.post_dag_run(dag_id, dag_run)
58             pprint(api_response)
59         except airflow_client.client.exceptions.OpenApiException as e:
60             print("Exception when calling DAGRunAPI->post_dag_run: %s\n" %e)
61

```

## ○ Home page

We will get information about the weather conditions of the last stored location in the database whenever this page is accessed by user.

```

52 app = Flask(__name__)
53 app.secret_key=os.urandom(24)
54
55 @app.route("/", methods=['POST', 'GET'])
56 def home():
57     cursor = conn.cursor()
58     cursor.execute("SELECT DISTINCT ON (todays_date) * FROM public.weather_table ORDER BY todays_date desc, date_time desc NULLS LAST, city limit 1;")
59     result = cursor.fetchall()
60     print(result)
61     return render_template('home.html', result = result)

```

## ○ Today weather conditions

This page accepts user's location requirements and then calls the trigger() to display real time weather information.

```

73 @app.route("/results", methods=['POST', 'GET'])
74 def render_results():
75     country = request.form['country']
76     place = request.form['location']
77
78
79     with open('get_city.json') as json_file:
80         data = json.load(json_file)
81         temp = data["weather_input"]
82         y = {"city": place }
83         temp.append(y)
84
85     write_json(data)
86
87     trigger()
88
89     time.sleep(40)
90
91     cursor = conn.cursor()
92     cursor.execute("SELECT DISTINCT ON (todays_date) * FROM public.weather_table ORDER BY todays_date desc, date_time desc NULLS LAST, city limit 1;")
93     result = cursor.fetchall()
94
95     print(result)
96     time.sleep(10)
97     return redirect(url_for('home', result = result, place = place, country = country))

```

## ○ Hourly weather conditions

```

193 @app.route("/hourly", methods=['POST', 'GET'])
194 def hourly_result():
195     cursor = conn.cursor()
196     cursor.execute("SELECT DISTINCT ON (todays_date) * FROM public.weather_table ORDER BY todays_date desc, date_time desc NULLS LAST, city limit 1;")
197     result = cursor.fetchall()
198
199     place = result[0][0]
200     latin = result[0][2]
201     lontin = result[0][3]
202
203     weather_url = requests.get(f'https://api.openweathermap.org/data/2.5/onecall?lat={latin}&lon={lontin}&exclude=current,minutely,daily&appid=a728a370130711e2e8192f0aaa7e')
204     weather_data = weather_url.json()
205     day_1_temp = int(weather_data['hourly'][1]['temp']-273)
206     day_1_con = weather_data['hourly'][1]['weather'][0]['description']
207     day_1_time = weather_data['hourly'][1]['dt']
208     day_1_icon = weather_data['hourly'][1]['weather'][0]['icon']
209
210     time1 = datetime.fromtimestamp(day_1_time).strftime('%I %p')
211     time2 = datetime.fromtimestamp(day_2_time).strftime('%I %p')
212     time3 = datetime.fromtimestamp(day_3_time).strftime('%I %p')
213     time4 = datetime.fromtimestamp(day_4_time).strftime('%I %p')
214
215     return render_template('hourly.html', place = place, day_1_temp = day_1_temp, day_1_con = day_1_con, time1 = time1, day_2_temp = day_2_temp, day_2_con = day_2_con,
216
217
218
219

```

## ○ Daily weather conditions

```

154 @app.route("/daily", methods=['POST', 'GET'])
155 def daily_result():
156     cursor = conn.cursor()
157     cursor.execute("SELECT DISTINCT ON (todays_date) * FROM public.weather_table ORDER BY todays_date desc, date_time desc NULLS LAST, city limit 1;")
158     result = cursor.fetchall()
159     print(result)
160     place = result[0][0]
161     latin = result[0][2]
162     lontin = result[0][3]
163
164
165     weather_url = requests.get(f'https://api.openweathermap.org/data/2.5/onecall?lat={latin}&lon={lontin}&exclude=current,minutely,hourly&appid=a728a370130711e2e8192f0aaa74')
166     weather_data = weather_url.json()
167     day_1_temp = int(weather_data['daily'][0]['temp']['day']-273)
168     day_1_con = weather_data['daily'][0]['weather'][0]['description']
169     day_1_wind = weather_data['daily'][0]['wind_speed']
170     day_1_icon = weather_data['daily'][0]['weather'][0]['icon']
171
172     date2 = datetime.fromtimestamp(day_2_date).strftime('%a %d %b')
173     date3 = datetime.fromtimestamp(day_3_date).strftime('%a %d %b')
174     date4 = datetime.fromtimestamp(day_4_date).strftime('%a %d %b')
175
176
177     return render_template('daily.html', place = place, day_1_temp = day_1_temp, day_1_con = day_1_con, day_1_wind = day_1_wind, day_2_temp = day_2_temp,

```

## ○ Air quality

```

139 @app.route("/air_quality", methods=['POST', 'GET'])
140 def air_quality():
141     cursor = conn.cursor()
142     cursor.execute("SELECT DISTINCT ON (todays_date) * FROM public.weather_table ORDER BY todays_date desc, date_time desc NULLS LAST, city limit 1;")
143     result = cursor.fetchall()
144     place = result[0][0]
145     country = result[0][1]
146     latin = result[0][2]
147     lotin = result[0][3]
148     weather_url = requests.get(f'https://api.weatherbit.io/v2.0/current/airquality?lat={latin}&lon={lotin}&key=6d9d3d4fc0a34f569aa0c56e27eb7121')
149     weather_data = weather_url.json()
150     first = weather_data['data'][0]['aqi']
151     return render_template('air_quality.html', first = first, place = place, country = country)

```

## ○ Sign in

```

100 @app.route("/signup", methods=['POST', 'GET'])
101 def signup():
102     return render_template('signup.html')
103
104 @app.route("/success", methods=['POST', 'GET'])
105 def contact():
106     username = request.form['user']
107     email = request.form['email']
108     password = request.form['pass']
109
110     sql = "INSERT INTO login_users (username, email, password) VALUES (%s, %s, %s);"
111     insert = (username, email, password)
112     cursor = con.cursor()
113     cursor.execute(sql, insert)
114     con.commit()
115     cursor.execute("SELECT * FROM login_users where email LIKE {};".format(email))
116     myuser = cursor.fetchall()
117     session['username'] = myuser[0][0]
118     return redirect(url_for('home'))

```

## ○ Sign out

```
---
120     @app.route('/signout')
121     def signout():
122         session.pop('username')
123         return redirect('/')
124
```

## ○ News

```
125     news_url = 'https://newsapi.org/v2/everything?q=weather alerts&apiKey=7f0493be147448a3b081fd0d368c034e'
126
127     def get_data(news_url):
128         response = requests.get(news_url)
129         resp = response.json()
130         articles = resp['articles']
131         return articles
132
133     articles = get_data(news_url)
134
135     @app.route("/news", methods=['POST', 'GET'])
136     def news():
137         return render_template('news.html', articles = articles)
```

## ○ Feedback page

```
230     @app.route("/contact", methods=['POST', 'GET'])
231     def contact_info():
232         return render_template('contact.html')
233
```

## ○ About page

```
234     @app.route("/about", methods=['POST', 'GET'])
235     def about():
236         return render_template('about.html')
237
238     if __name__ == '__main__':
239         app.run(debug = True)
```

---

## 8.4 Views for Webpage

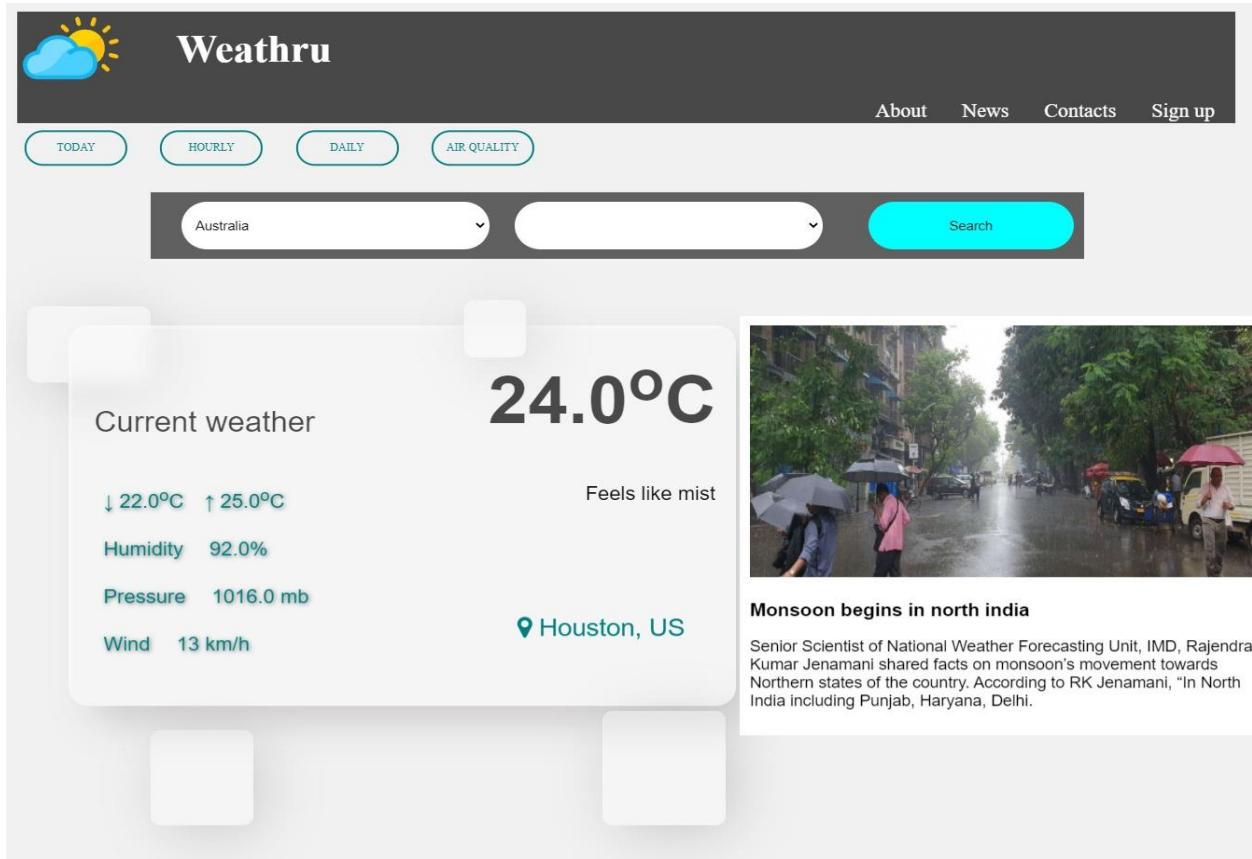


Fig 1: Home page

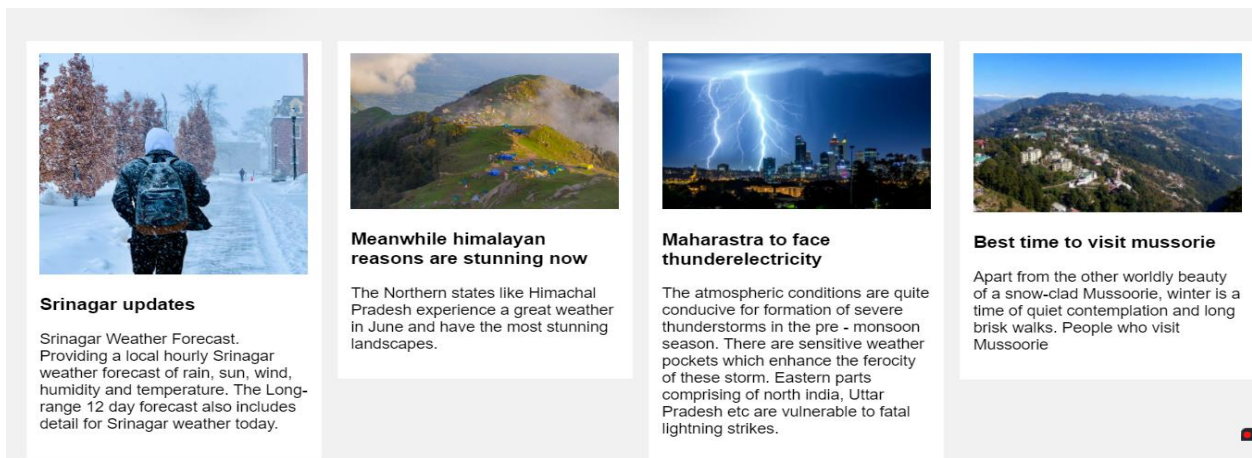
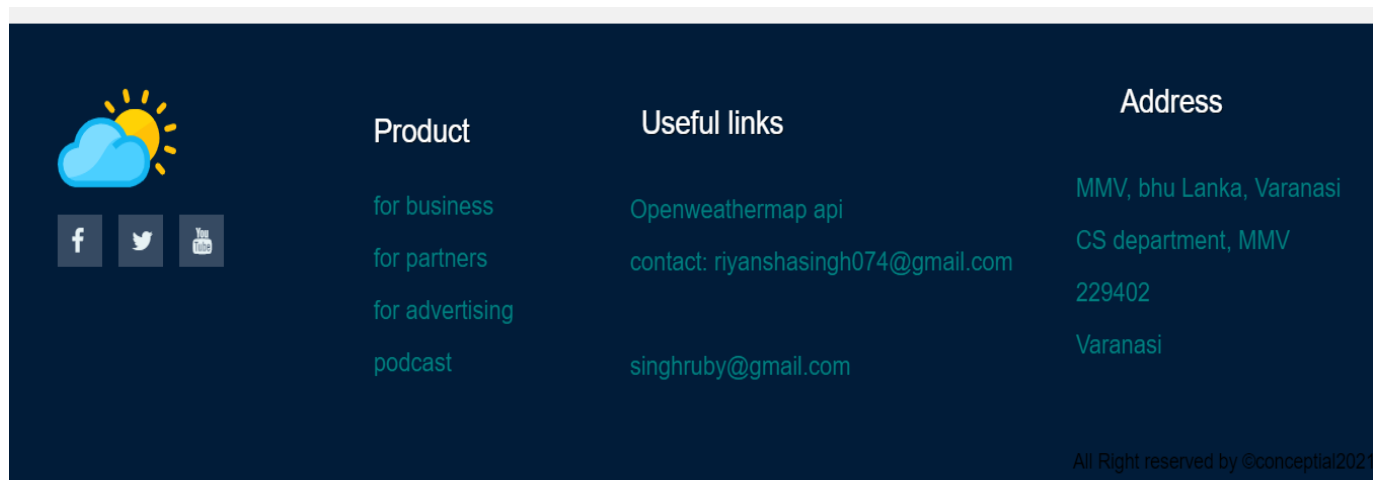
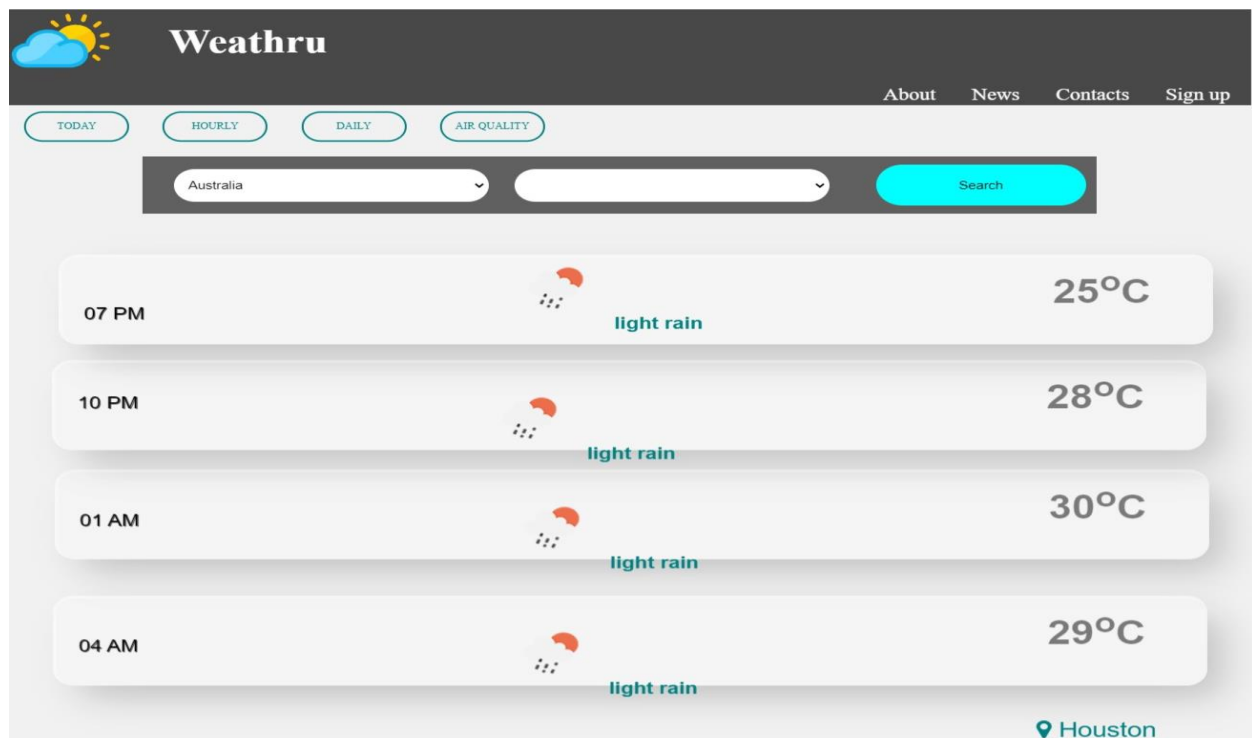


Fig 2: Gallery Section in home and other Pages



**Fig 3: Footer of Home and other Pages**



**Fig 4: Hourly weather**

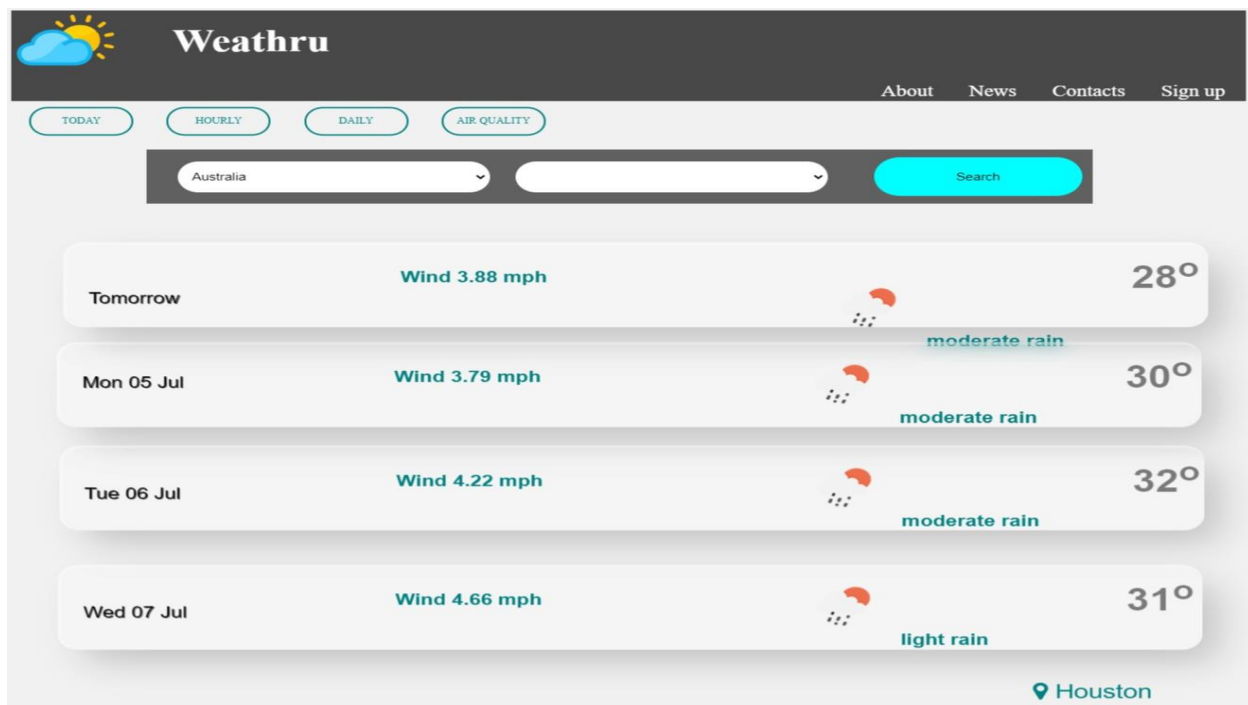


Fig 5: Daily Forecast

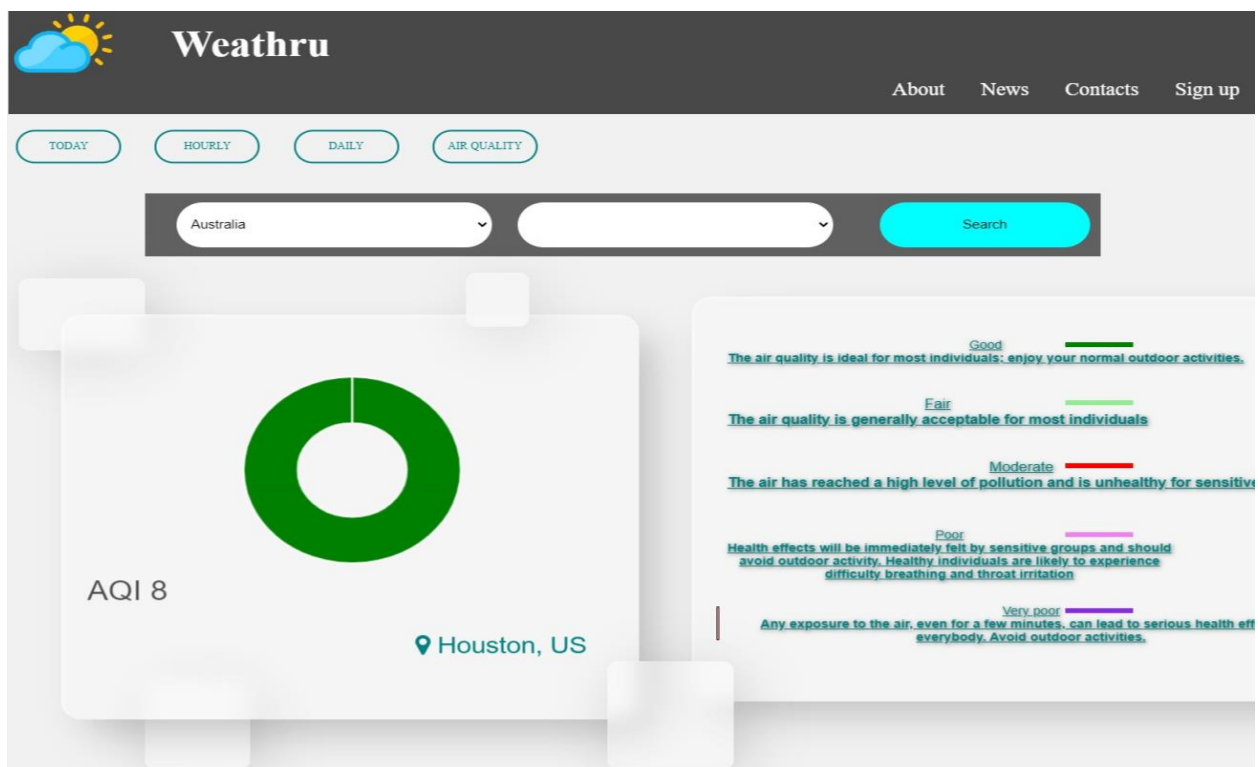
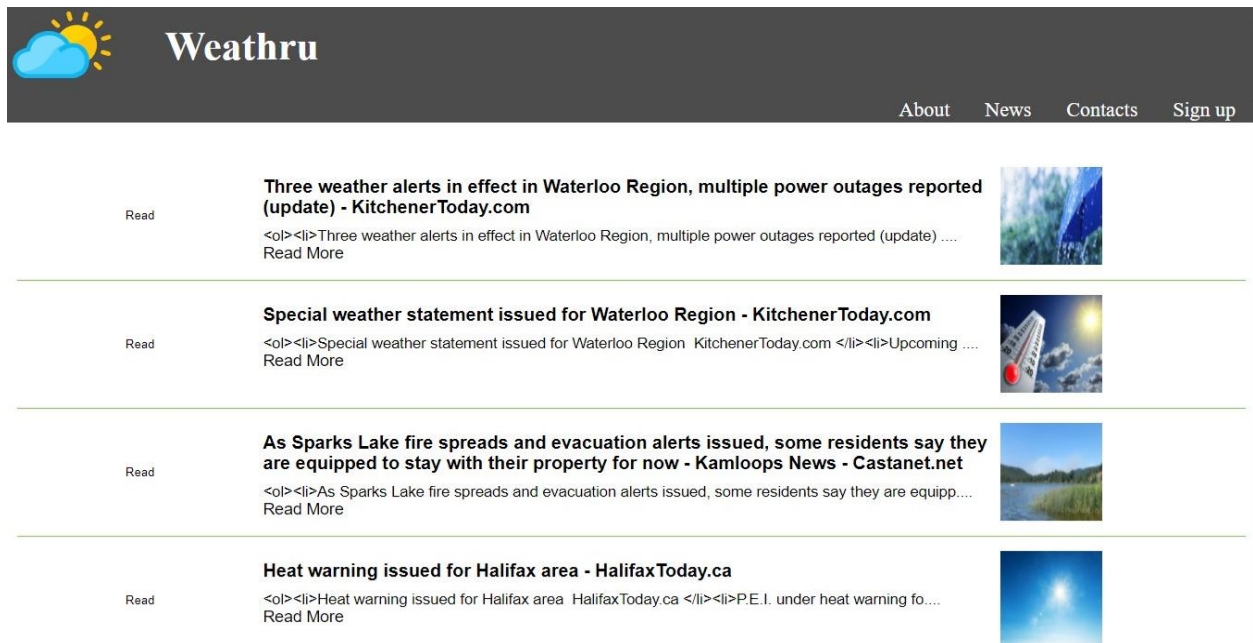
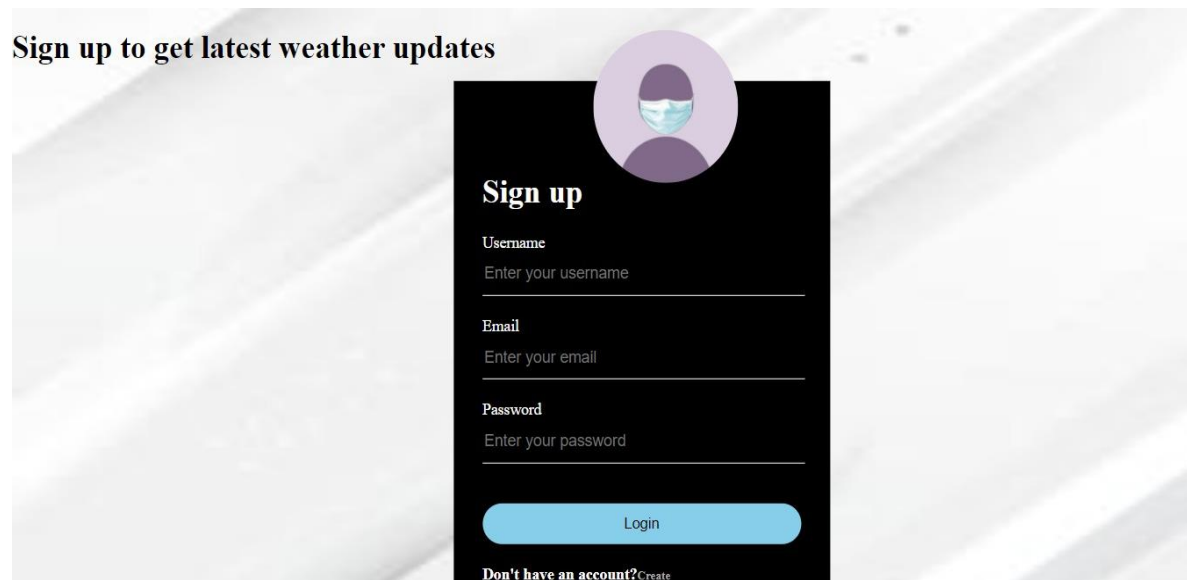


Fig 6: Air quality

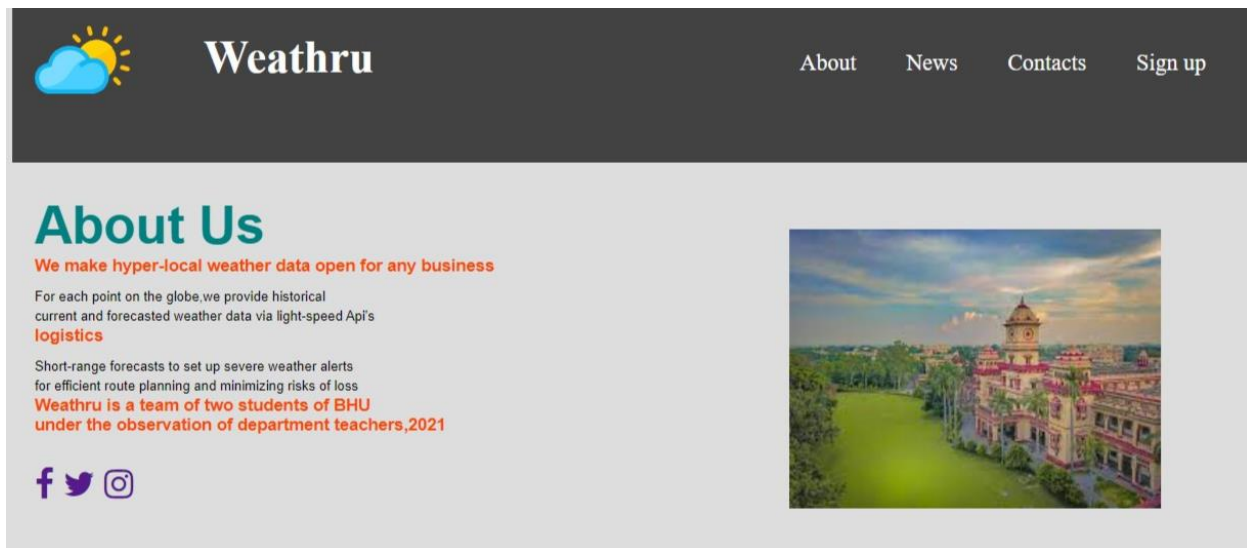


**Fig 7: Global Weather alerts**



**Fig 8: Page for Sign up**





**Fig 9: About Us**

**Fig 10: Feedback page**

## **9. Summary and Conclusion**

WEATHRU is an online weather forecasting API based system. In this system any user across the globe can get weather information ranging from present day to upcoming days. The database maintained to store results of API call allows efficient retrieval of information whenever required. Credentials of registered users is also stored in separate database.

It meets all the requirements and justifies the purpose for which it has been build. The system has been moved to the steady state. The system is operated at a high level of efficiency and the users will be very much benefitted from it. The software provides accurate and true information to let users build a trustworthy relationship with us.

# 10. Advantages and limitations of Proposed Architecture

## 10.1 Advantages

- Attractive user interface to make interaction easy.
- Current, hourly, daily, air quality data is available at one place.
- It provides verified and accurate data through API.
- Weather alerts are also available via news API.
- Components can easily be replaced with other implementations.
- Developers can specialize and focus on different components without affecting others as all views are clearly separated.

## 10.2 Disadvantages

- API based system are a little slow at functioning as compared to normal ones.
- GPS technology is not used, hence location input is very much required.
- Historical weather information is not yet available.
- It can be costly to create a design and implement two different types of technology (Airflow and Flask) together for simple user interfaces.
- Apache Airflow can be too complex for simple applications.

## 11. Future works

- We can help users predict the amount of rainfall which can help them in agriculture and farming. This can be done by building a machine learning model.
- We can also add about historical weather conditions and extended forecast.
- We can send notifications to the registered users regarding weather updates of their area through email id provided.
- We can easily make it an android friendly application through minor changes in script.

By changing some of the features, this project can be used for various purposes.

## 12. References

- ❑ <https://openweathermap.org>
- ❑ <https://api.openweathermap.org>
- ❑ <https://weatherbit.io/v2.0/current/airquality>
- ❑ Foruzon, Computer Networks
- ❑ Pankaj Jalote, (2003) 'An Integrated Approach towards Software Engineering'
- ❑ <https://newsapi.org/v2>
- ❑ [www.slideshare.net](http://www.slideshare.net)
- ❑ [www.airflow.apache.org](http://www.airflow.apache.org)
- ❑ [www.github.com](http://www.github.com)
- ❑ [www.stackoverflow.com](http://www.stackoverflow.com)
- ❑ [www.realpython.com](http://www.realpython.com)
- ❑ Navathe, Elmasri, 'Fundamentals of Database Systems', Pearson Education
- ❑ [www.towardsdatascience.com](http://www.towardsdatascience.com)
- ❑ <http://michael-harmon.com/AirflowETL.html>
- ❑ [www.youtube.com](http://www.youtube.com)

# APPENDIX

## Individual Contribution to the Project

❑ Name of student: **RIYANSHA SINGH**

My work includes designing the backend system that fetches the data from API, performs Data Engineering on the obtained data and finally display the result to the end user. I had also worked on designing the Database to store weather data, user's information and feedback. Routing webpages through Flask, integrating frontend (Flask) with backend system (Airflow) were also important part of my work.

❑ Name of student: **RUBY SINGH**

My contribution is to design and develop web interface for the website. So I studied JAVASCRIPT, HTML, CSS and python for designing web interface. Mainly I worked on front end of the project. I have to figure out the user interaction occurred between users and the system. The design of these web interfaces should be compatible with users. Thus they have to be user-friendly.

