



**BINUS UNIVERSITY**

**BINUS INTERNATIONAL**

**Final Project Cover Letter**

**(Individual Work)**

**Student Information:**

**Surname:** -

**Given Name:** Jeffrey

**Student ID:** 2602118484

**Course Code :** COMP6047001

**Course Name :** Algorithm and Programming

**Class :** L1AC

**Lecturer :** Jude Joseph Lamug Martinez, MCS

**Type of Assignment :** Final Project Report

**Submission Pattern**

**Due Date** : 16 January 2023

**Submission Date :** 15 January 2023

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer's instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

**Plagiarism/Cheating**

BiNus International seriously regards all forms of plagiarism, cheating, and collusion as academic offences which may result in severe penalties, including loss/drop of marks, course/class discontinuity, and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

**Declaration of Originality**

By signing this assignment, I understand, accept, and consent to BiNus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student: Jeffrey

## **A. Background**

As part of this algorithm and programming course, we are expected to create a comprehensive application that uses all of the Python knowledge and skills we have acquired during the semester to solve a certain problem. We are also expected to apply concepts that is beyond our lesson in class. As a result, I decided to pick up several concepts/topics that I have not learned before, such as Django, API, and data visualization.

A problem that I intend to solve is neither accurate nor up-to-date information of some websites about weather conditions and prediction. Another issue could be that the user interface is difficult to navigate or not user-friendly. Additionally, some websites are not providing enough detail or information about the forecast, such as hourly updates.

Hence, I provide a solution with a web app that improves the accuracy and up-to-date of the weather information by utilizing a reliable weather API. Moreover, I try to make my web app as interactive as possible so that users can easily navigate through all the pages and features. In addition, I include hourly forecasts and air quality to increase the detail of the application.

## **B. Project Specification**

### **1. Introduction**

After contemplating several ideas and topics for my final project, I decided to make a friendly weather web app that utilizes Django for the operation combined with data visualization. The objectives of this web app are users can check the air quality and weather condition of a certain city or place they are staying, and they can also utilize it to forecast the weather for the next few days such as the temperature and sky conditions of their place.

In the beginning of the running program of my web app, users will be directed to the main page with the navigation bar at the top section and the content in the rest of the page. Several pages/features available: 'main page', 'about page', 'aqi page', 'forecast page', 'login and register page'. In order to access all the features in this web app, users will either need to register their account if they are newbies or log into their own account. Their account is automatically saved in Django's database. After filling in their own credentials, they will be redirected to the main page where they can click on the about page containing the detail of the web app. They can find out the states and

cities available to be checked of the air quality. The main functions where this web app focus on are the aqi page and the forecast page. In aqi page, it shows up the default 'Jakarta' air quality and its weather condition. The background color and description of the aqi will change according to the range of air quality of a city inputted by the users. At the bottom of the page, the icon of weather condition will also modified based on the climate derived from an API. As for the forecast page, the users will need to input a place, the number of forecasted days starting from the scale 1 until 5 days and select the data (temperature/sky) they want to visualize. All the data in this web app acquired from many free yet reliable APIs.

## **2. Modules/Library/Framework**

- **Django - Framework**

Django is a free and open-source web framework written in Python. I chose Django over the other web framework (streamlit, flask, justpy) as I find it more challenging and it can maintain high-quality web applications with minimal effort. Moreover, it offers many third-party packages. There are a lot of Django modules that I use such as render, redirect, reverse, UserCreationForm, messages, login\_required

- **Virtualenv - Tool**

Virtualenv is a tool used to create isolated Python environments. I used it to keep my global Python installation clean and more manageable. It also helps to avoid conflicts between dependencies of different projects that I was working on.

- **JSON - Module**

I use the JSON module because I need to work with JSON (JavaScript Object Notation) data retrieved from the API, which is commonly used for data interchange on the web.

- **Requests - Library**

This library allows me to send HTTP requests and handle the response. In this case, I use it to make a request to a weather API to retrieve data.

- **pytz - Library**

I used this library to convert and provide timezone-related functionality. In this case, I utilize it to set it into Asia/Jakarta timezone.

- **Datetime - Module**

This is a built-in module of the Python standard library where I use it to work with date and time values.

- **Pandas - Library**

Pandas is a library for data manipulation and analysis. It provides powerful data structures, such as DataFrames, and tools for working with data, such as reading and writing to different file formats. I use it to create a dataframe with my data.

- **Plotly Express and Plotly Offline - Modules**

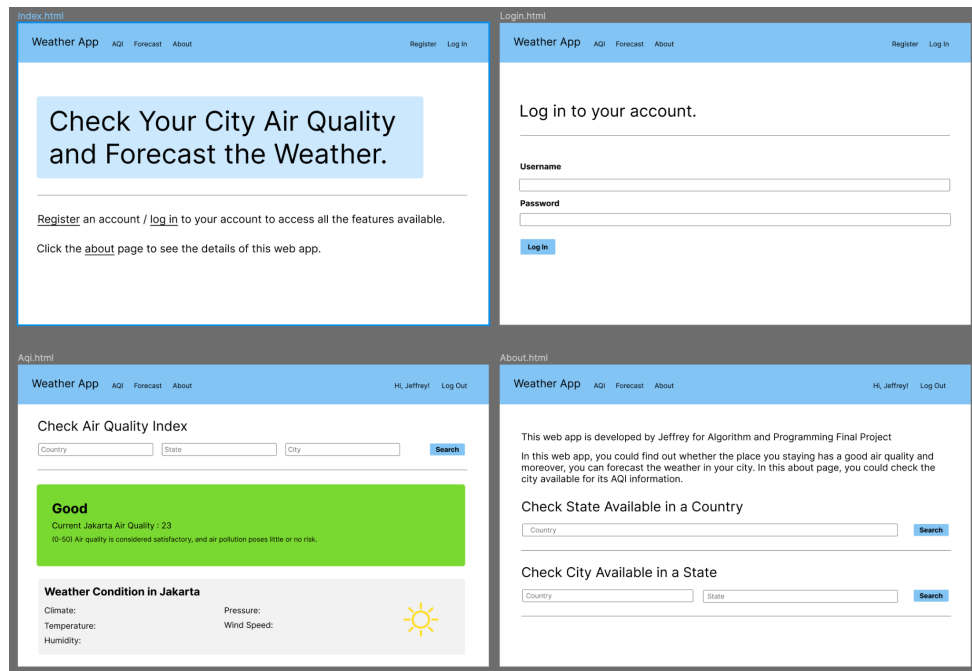
These modules are used to create interactive visualizations, in this case, I use them to create a visual thus interactive representation (plots) of the weather data. The plotly.offline module allows me to generate the plot and save it as an HTML file, which can be opened in a web browser.

### **3. Important Folder/Files**

- 'users' folder - contains the template and function for user login and register, including login page, register page, and views.py function.
- 'weather' folder - contains the startapp for django.
- 'weatherapp' folder - contains the template and function for my web app features such as home page, base html file, aqi page, forecast page, about page and views.py function.
- 'documentation' folder - contains all the solution design and images needed, including this final project report.
- 'static' folder - contains all the images/icons I used for my web app.

## C. Solution Design

### 1. Sketchboard



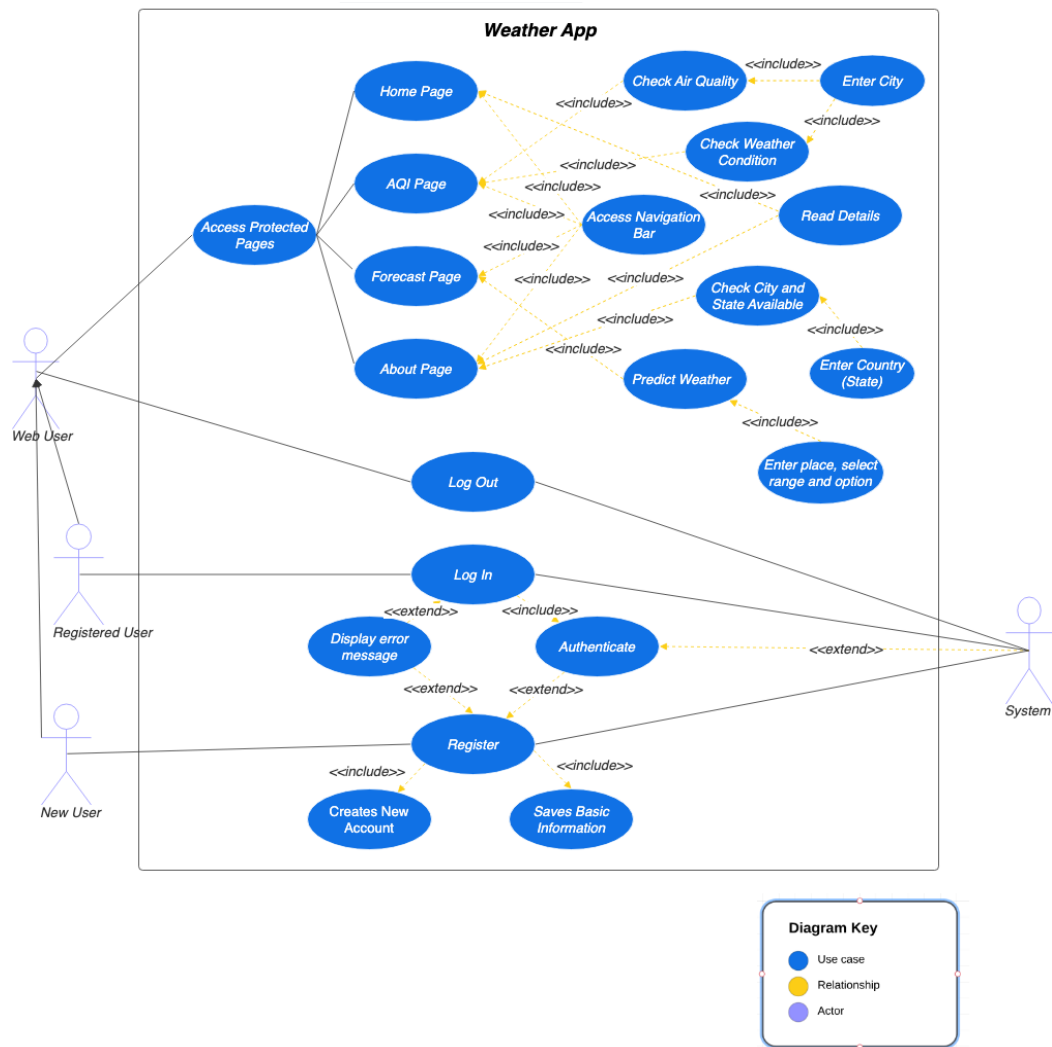
This is the mockup that I made in figma before I started to design my website.

For the styling and functionality of my website, I used bootstrap css and js framework in my HTML file.

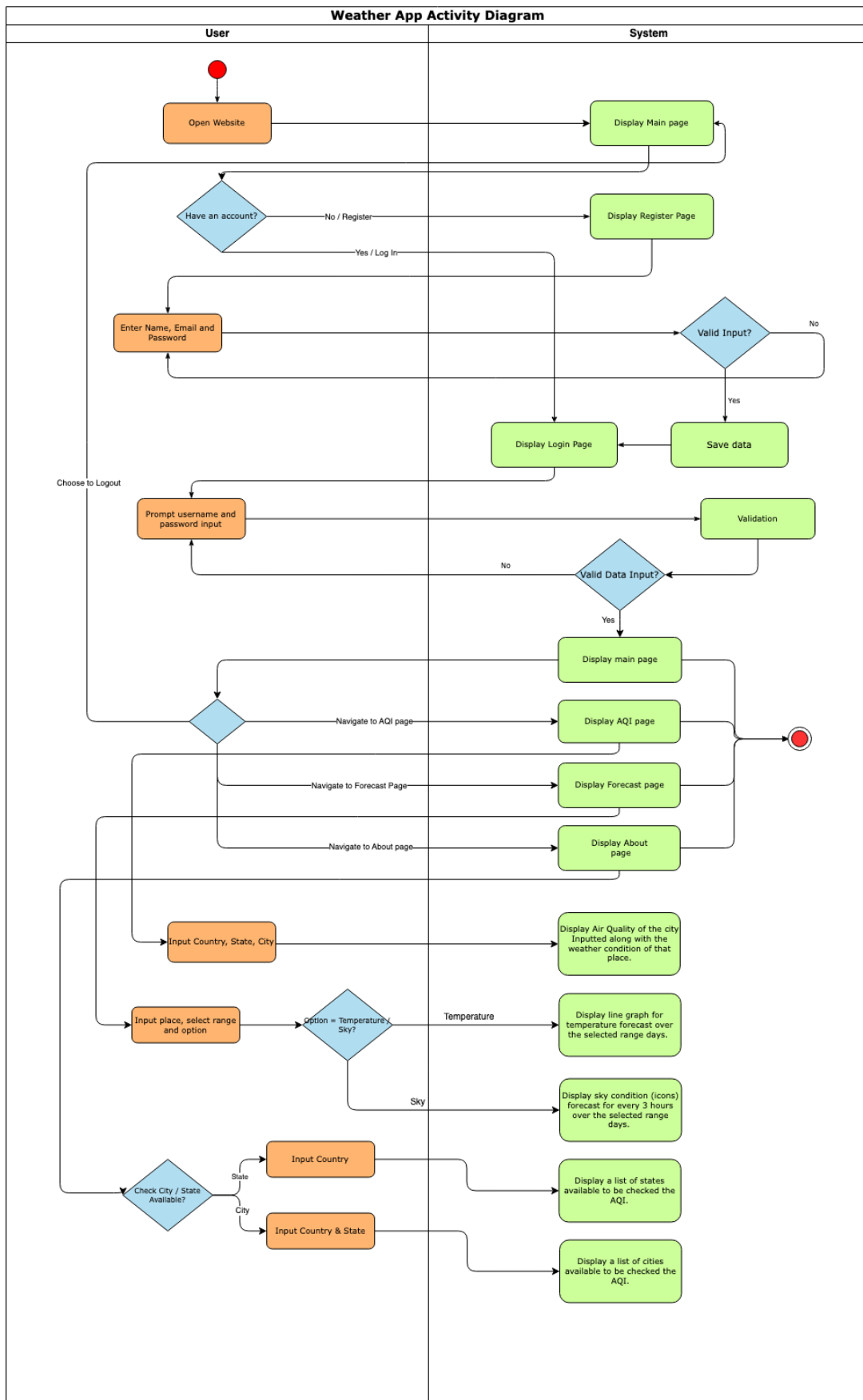
Layouts of this web app:

- Navigation bar on the top section of each page.
- Home page
- Login page : containing form for user authentication
- Register page : containing form for creating account
- Aqi page : containing air quality for a certain city and the weather condition.
- Forecast page : containing temperature data that is visualized in line chart and sky condition for each three hours
- About page : containing information and details about the page.

## 2. Use Case Diagram



## 3. Activity Diagram



#### 4. Class Diagram



CreateUserForm	Meta	UsersConfig
	fields : list model	default_auto_field : str name : str

AQIProcessor	AirVisualAPI	ForecastAPI	ForecastProcessor	LookupConfig
city_aqi_api	api_key : str	api_key : str	forecast_api	default_auto_field : str name : str
process_aqi() weather_condition()	get_aqi(country, state, city) get_cities(country, state) get_states(country)	get_forecast_api(place)	get_data(forecast_days)	

## D. Essential Algorithms

- **Setting Up Django**

- Install Virtualenv: I created a new virtual environment by installing virtualenv using command 'pip install virtualenv' in my terminal. I also activated the virtual environment everytime I want to work on my Django project using command 'source bin/activate'.
- Install Django: I installed Django using pip by running python3 -m pip install Django in my terminal.
- Create a new project: I created a new project by running the command django-admin startproject weatherapp2 in my terminal. This would create a new directory called weatherapp2 in my current directory with the basic file structure for a Django project.
- Create a new app: Within my project directory, I created a new app by running the command python manage.py startapp weatherapp. This will create a new directory called weatherapp with the basic file structure for a Django app.
- Configure the database: In my project's settings.py file, I configured the database settings using a SQLite database.
- Run migrations: After configuring the database, I made migrations to create the necessary database tables by running the command python manage.py migrate in my terminal.
- Create views: In my app's views.py file, I created views to handle the different pages of my web application. views are Python functions that take a web request and return a web response.

- Create URLs: In my project's urls.py file, I created URLs for my views. URLs are the addresses that users type into their web browsers to access different pages of your web application.
- Create templates: I created HTML templates in my app's templates directory. These templates will be rendered by views to create the final HTML that is sent to the user's web browser.
- Start the development server: I started the development server by running python manage.py runserver. This will start the server and make my web application available at <http://127.0.0.1:8000/> in my web browser.

- **AirVisualAPI Class**

```
# class created for making requests to air quality API and retrieve data from it
class AirVisualAPI:
    # initialize an object's state that store my api key
    def __init__(self):
        self.api_key = "1bc7b8c0-9a65-4935-a911-42dffefbe512"
```

The AirVisualAPI class is a wrapper for the AirVisual API. This class is used to make requests to the API and retrieve data. The `__init__` method is the constructor for the class. It sets the `api_key` attribute to my api key used for retrieving data.

```
# defining a method for getting states available that takes in country argument
def get_states(self, country):
    states = []
    state_available_api_request =
requests.get("http://api.airvisual.com/v2/states?country=" + country +
"&key=" + self.api_key) # make a GET request to the API
    try:
        state_api = json.loads(state_available_api_request.content)
# parse the API Call JSON response to a string and returns a Python dictionary
        if state_api['status'] == "success":
            for state in state_api['data']:
                states.append(state['state']) # list out all the states available for the given country
    except Exception as e:
        state_api = "Error..."
    return state_api, states # return two values 'state_api' and 'states'
```

This code defines a method `get_states` for the `AirVisualAPI` class. The method takes in a single argument, `country`, and returns a tuple of two values: `state_api` and `states`. The method first initializes an empty list called `states`. Then, it makes an API request to the <http://api.airvisual.com/v2/states> endpoint, passing in the `country` and `self.api_key` as query parameters. If the request is successful, the method will parse the JSON response using the `json.loads()` method, which parses a JSON string and

returns a Python dictionary. If the request is successful and the key status in the response is equal to "success", it then iterates through the data key of the response, appending the value of the state key of each item to the states list. 'state\_api' is the json dictionary returned from the API call, and 'states' is the list of states that are available for the given country. If an exception occurs, the method will return "Error..." instead of the state\_api.

```
# defining a method for getting cities available that takes in country and
state argument
def get_cities(self, country, state):
    cities = []
    city_available_api_request =
requests.get("http://api.airvisual.com/v2/cities?state=" + state +
"&country=" + country + "&key=" + self.api_key)    # make a GET request to
the API
    try:
        city_api = json.loads(city_available_api_request.content)
# parse the API Call JSON response to a string and returns a Python
dictionary
        if city_api['status'] == "success":
            for city in city_api['data']:
                cities.append(city['city'])    # list out all the cities
available for the given country and state
        except Exception as e:
            city_api = "Error..."
    return city_api, cities    # return two values 'city_api' and 'cities'
```

This code defines a method get\_cities is used to list the cities available for the given country and state. The steps are the same as before, make an API call by get request and returns it into a JSON dictionary. From that JSON data, if the key status in the response equals to success, it then iterates through the data key and append the city key to cities list.

```
# defining a method for retrieving aqi json dictionary for a specific city,
state, country
def get_aqi(self, country, state, city):
    city_aqi_api_request =
requests.get("http://api.airvisual.com/v2/city?city=" + city + "&state=" +
state + "&country=" + country + "&key=" + self.api_key)    # make a GET
request to the API
    try:
        city_aqi_api = json.loads(city_aqi_api_request.content)
# parse the API Call JSON response to a string and returns a Python
dictionary
        except Exception as e:
            city_aqi_api = "Error..."
    return city_aqi_api
# return python dictionary containing aqi data
```

This code defines a method get\_aqi for the AirVisualAPI class. The method takes in three arguments: country, state, and city, and returns a single value: city\_aqi\_api. The method makes an API request to the http://api.airvisual.com/v2/city endpoint, passing

in the city, state, country, and self.api\_key as query parameters. If the request is successful, the method will parse the JSON response using the json.loads() method, which parses a JSON string and returns a Python object. Finally, the method returns the city\_aqi\_api variable, which is the json object returned from the API call.

- **AQIProcessor Class**

```
# class created for processing all the aqi data retrieved from airvisual API
class AQIProcessor:
    # initializes an attribute that is assigned the response from the
    AirVisual API
    def __init__(self, city_aqi_api):
        self.city_aqi_api = city_aqi_api

    # defining a method for determining the category, color, and description
    of the AQI based on the AQI value.
    def process_aqi(self):
        aqi_categories = {          # create a dictionary that maps AQI ranges to
AQI categories, category colors, and descriptions.
            (0, 50): ("Good", "good", "(0-50) Air quality is considered
satisfactory, and air pollution poses little or no risk."),
            (51, 100): ("Moderate", "moderate", "(51-100) Air quality is
acceptable. However, there may be a risk for some people, particularly those
who are unusually sensitive to air pollution."),
            (101, 150): ("Unhealthy for Sensitive Group", "usg", "(101-150)
Members of sensitive groups may experience health effects. The general
public is less likely to be affected."),
            (151, 200): ("Unhealthy", "unhealthy", "(151-200) Some members of the
general public may experience health effects; members of sensitive groups
may experience more serious health effects."),
            (201, 300): ("Very Unhealthy", "veryunhealthy", "(201-300) Health
alert: The risk of health effects is increased for everyone."),
            (301, 500): ("Hazardous", "hazardous", "(301-500) Health warning of
emergency conditions: everyone is more likely to be affected."),
        }

        if self.city_aqi_api['status'] == 'success':
            aqi = self.city_aqi_api['data']['current']['pollution']['aqius']
            for aqi_range, (category, category_color, category_description)
in aqi_categories.items():
                if aqi_range[0] <= aqi <= aqi_range[1]:
                    return category, category_color, category_description,
aqi

            return "", "", "", ""          # returns an empty tuple if the API
request is not successful or aqi value is not found in any of the range
```

This code defines a class AQIProcessor that takes a single argument city\_aqi\_api in its constructor (\_\_init\_\_) and set it as an attribute of the class.

The class contains method process\_aqi which takes no arguments. The method uses a dictionary aqi\_categories to map ranges of AQI values to corresponding categories, category colors, and category descriptions. It retrieves the AQI value from the city\_aqi\_api object and assigns it to the variable aqi. Then, it iterates through the aqi\_categories dictionary and for each AQI range, it compares the aqi with the range's

lower and upper bounds. If the aqi falls within the range, the method returns the corresponding category, category color, category description, and the aqi value.

```
# defining a method for retrieving the weather conditions for the specific
city
def weather_condition(self):
    if self.city_aqi_api['status'] == 'success':
        temperature =
self.city_aqi_api['data']['current']['weather']['tp']
        pressure = self.city_aqi_api['data']['current']['weather']['pr']
        humidity = self.city_aqi_api['data']['current']['weather']['hu']
        wind_speed =
self.city_aqi_api['data']['current']['weather']['ws']
        climate_icons =
self.city_aqi_api['data']['current']['weather']['ic']
        climate_dict = {          # create a dictionary that stores the
description of climate_icons and the icon path
            "01d" : ("Clear Sky", "images/01d.png"),
            "01n" : ("Clear Sky", "images/01n.png"),
            "02d" : ("Few Clouds", "images/02d.png"),
            "02n" : ("Few Clouds", "images/02n.png"),
            "03d" : ("Scattered Clouds", "images/03d.png"),
            "03n" : ("Scattered Clouds", "images/03d.png"),
            "04d" : ("Broken Clouds", "images/04d.png"),
            "04n" : ("Broken Clouds", "images/04d.png"),
            "09d" : ("Shower Rain", "images/09d.png"),
            "09n" : ("Shower Rain", "images/09d.png"),
            "10d" : ("Rain", "images/10d.png"),
            "10n" : ("Rain", "images/10n.png"),
            "11d" : ("Thunderstorm", "images/11d.png"),
            "11n" : ("Thunderstorm", "images/11d.png"),
            "13d" : ("Snow", "images/13d.png"),
            "13n" : ("Snow", "images/13d.png"),
            "50d" : ("Mist", "images/50d.png"),
            "50n" : ("Mist", "images/50d.png"),
        }
        for icons, (climate, icon_link) in climate_dict.items():
            if climate_icons == icons:
                return temperature, pressure, humidity, wind_speed,
climate, icon_link
        return "", "", "", "", "", ""          # returns an empty tuple if the
API request is not successful or icon is not found in any of the dictionary
```

This code defines a method `weather_condition` for the `AQIProcessor` class that takes no arguments and returns a tuple of six values: temperature, pressure, humidity, wind\_speed, climate, and icon\_link. The method first checks whether the status key of the `city_aqi_api` attribute is equal to "success". If it is, it retrieves temperature, pressure, humidity, wind\_speed, and the climate icon code from the `city_aqi_api` object and assigns it to the corresponding variables.

- **Rendering Template Function (views.index)**

```
# function to handle requests made and returns index.html template to the
client to be rendered.
def index(request):
    return render(request, 'weatherapp/index.html', {})
```

This function uses the render function from the django.shortcuts library to render a specific HTML template, 'weatherapp/index.html', and return it as the response to the client. The second argument is the template that we want to render. The third argument is a dictionary of key-value pairs that will be passed as variables to the template. In this case, it is an empty dictionary {}.

- **Rendering Template Function (views.about)**

```
# decorator to check whether the user is logged in, and if not, redirects
the user to login_url
@login_required(login_url='users/login')
# function to handle requests made and returns about.html template to the
client to be rendered
def about(request):
    airvisual = AirVisualAPI()      # create an instance of AirVisualAPI class
    if request.method == "POST":
        if 'find-state' in request.POST:      # if find-state button is
pressed, return the states and state_api
            country = request.POST['country-for-state']
            state_api, states = airvisual.get_states(country)
            return render(request, 'weatherapp/about.html', {'state_api':
state_api, 'states': states})
        elif 'find-city' in request.POST:      # if find-city button is
pressed, return the cities and city_api
            country = request.POST['country-for-city']
            state = request.POST['state']
            city_api, cities = airvisual.get_cities(country, state)
            return render(request, 'weatherapp/about.html', {'city_api':
city_api, 'cities': cities})
    return render(request, 'weatherapp/about.html', {})      # returns an
empty context data if request method is not POST
```

This code defines a function about that is used to handle requests made to the '/about' URL of a Django application. It requires the user to be logged in to access this view, if the user is not logged in, it redirects them to 'users/login' page.

When the function is accessed, it first creates an instance of the AirVisualAPI class. If the request method is 'POST', it checks which button is pressed by checking the key 'find-state' or 'find-city' in the request.POST object. Depending on which button is pressed, it retrieves the selected country or state from the request.POST object and passes it to the get\_states or get\_cities method of the airvisual object. It then renders the 'weatherapp/about.html' template with the returned values of state\_api, states, city\_api, cities as context data. If the request method is not 'POST', it renders the 'weatherapp/about.html' template with an empty context data.

- **Rendering Template Function (views.aqi)**

```
# decorator to check whether the user is logged in, and if not, redirects
the user to login_url
@login_required(login_url='users/login')
```

```

# function to handle requests made and returns about.html template to the
client to be rendered
def aqi(request):
    airvisual = AirVisualAPI()          # create an instance of AirVisualAPI
class
    if request.method == "POST":
        country = request.POST['country-aqi']
        state = request.POST['state-aqi']
        city = request.POST['city-aqi']
        city_aqi_api = airvisual.get_aqi(country, state, city)      # get AQI
data by passing all the parameters to get_aqi method
    else:          # default value if the request method is not post
        city_aqi_api = airvisual.get_aqi("Indonesia", "Jakarta", "Jakarta")

    # create an instance of AQIProcessor class
    aqi_processor = AQIProcessor(city_aqi_api)

    # assign returned values of process_aqi method to the corresponding
variables
    category, category_color, category_description, aqi =
aqi_processor.process_aqi()

    # assign returned values of weather_condition method to the corresponding
variables
    temperature, pressure, humidity, wind_speed, climate, icon_link =
aqi_processor.weather_condition()
    jakartaTz = pytz.timezone("Asia/Jakarta")
    current_datetime = datetime.now(jakartaTz).strftime("%b %d, %Y at %H:%M
%p %Z")      # set the timezone to Jakarta timezone
    return render(request, 'weatherapp/aqi.html', {      # render all the
returned values as context data
        'city_aqi_api': city_aqi_api,
        'category': category,
        'category_color': category_color,
        'category_description': category_description,
        'aqi': aqi,
        'temperature': temperature,
        'pressure': pressure,
        'humidity': humidity,
        'wind_speed': wind_speed,
        'climate': climate,
        'icon_link': icon_link,
        'current_datetime': current_datetime
    })

```

When this function is accessed, it first creates an instance of the AirVisualAPI class. If the request method is 'POST', it retrieves the selected country, state, and city from the request.POST object and passes it to the get\_aqi method of the airvisual object to get the AQI data. If the request method is not 'POST', it uses the default value of "Indonesia", "Jakarta", "Jakarta" as an argument for get\_aqi method. Then, it creates an instance of the AQIProcessor class and passes the city\_aqi\_api as an argument. Using the created instance, it calls process\_aqi and weather\_condition method and assigns the returned values to the corresponding variables. The function also retrieves the time in Jakarta timezone and assigns it to current\_datetime variable.

Finally, it renders the 'weatherapp/aqi.html' template with the returned values as context data.

- **ForecastAPI Class**

```
# class created for making requests to forecasted temperature and sky
condition API and retrieve data from it
class ForecastAPI:
    # initialize an object's state that store my api key
    def __init__(self):
        self.api_key = "141710af2113bab9f55ef73e1bcd33d5"

    # defining a method for retrieving forecast data dictionary for a
    specific place
    def get_forecast_api(self, place):
        forecast_api_request =
requests.get("http://api.openweathermap.org/data/2.5/forecast?q=" + place +
"&appid=" + self.api_key) # make a GET request to the API
        try:
            forecast_api = json.loads(forecast_api_request.content) #
parse the API Call JSON response to a string and returns a Python dictionary
        except Exception as e:
            forecast_api = "Error..."
        return forecast_api # return python dictionary containing
forecasted data
```

This code defines a class ForecastAPI that is used to retrieve weather forecast data from OpenWeatherMap API. The class has one attribute api\_key which is a string containing the API key that will be used to make requests to the OpenWeatherMap API.

The class has one method get\_forecast\_api which takes in a single argument place, which is the name of the place for which the forecast data is to be retrieved. The method uses the requests library to send a GET request to the OpenWeatherMap API with the place and api\_key as parameters and assigns the response to the variable forecast\_api\_request. Finally, it returns the variable forecast\_api which contains the forecast data in JSON format.

- **ForecastProcessor Class**

```
# class created for processing the forecasted data retrieved from api
openweathermap
class ForecastProcessor:
    # initializes an attribute that is assigned the response from the
Forecast API
    def __init__(self, forecast_api):
        self.forecast_api = forecast_api

    # defining a method for filtering the data retrieved from the API
    def get_data(self, forecast_days):
        filtered_data = self.forecast_api['list']
```



```
nr_values = 8 * int(forecast_days)
return filtered_data[:nr_values]
```

This code defines a class `ForecastProcessor` which is used to process and filter the forecast data retrieved from the OpenWeatherMap API. The class has one attribute `forecast_api` which is assigned the response data from the OpenWeatherMap API.

The class has one method `get_data` which takes in a single argument `forecast_days` which is the number of days for which the forecast data is required. It returns the slice of `filtered_data` from the start to the calculated number of values.

- **Rendering Template Function (`views.forecast`)**

```
# decorator to check whether the user is logged in, and if not, redirects
the user to login_url
@login_required(login_url='users/login')
# function to handle requests made and returns forecast.html template to the
client to be rendered
def forecast(request):
    forecast = ForecastAPI()          # create an instance of ForecastAPI class
    if request.method == "POST":
        place = request.POST['place']
        days = request.POST['days']
        option = request.POST['option']
    else:
        # default value if the request method is not post
        place = "Jakarta"
        days = "1"
        option = "Temperature"

    if days == "1":
        tostring = f"{option} for the next {days} day in {place}"
    else:
        tostring = f"{option} for the next {days} days in {place}"

    # get forecasted data by passing all the parameters to get_forecast_api
    method
    forecast_api = forecast.get_forecast_api(place)

    # create an instance of ForecastProcessor class
    forecast_data = ForecastProcessor(forecast_api)

    # assign returned values of get_data method to the filtered_data variable
    filtered_data = forecast_data.get_data(days)
```

The function creates an instance of the `ForecastAPI` class and checks if the request method is a POST request. If the request method is a POST request, it assigns the values of the request data to the variables `place`, `days`, and `option`. If the request method is not a POST request, it assigns default values for the `place`, `days`, and `option` variables. Then it calls the `get_forecast_api` method of the `ForecastAPI` class passing in the `place` variable and assigns the result to the `forecast_api` variable. Then, it creates an instance of the `ForecastProcessor` class passing in the `forecast_api` variable. It then

calls the `get_data` method of the `ForecastProcessor` class, passing in the `days` variable and assigns the result to the `filtered_data` variable.

```
if option == "Temperature":
    # list and extracts the temperature and date values, converts the
    temperature values from Kelvin to Celsius
    temperatures = [data['main']['temp'] for data in filtered_data]
    dates = [data['dt_txt'] for data in filtered_data]
    new_temperatures = [temp-273 for temp in temperatures]

    # Create a dataframe with the dates and temperatures lists as columns
    df = pd.DataFrame()
    df['Dates'] = dates
    df['Temperatures (C)'] = new_temperatures

    # Pass the dataframe to the px.line() function to create a line plot
    figure = px.line(df, x='Dates', y='Temperatures (C)', labels={"x":
    "Date", "y": "Temperature (C)"})
    # assigns the generated plot to the temperature_plot variable
    temperature_plot = plot.figure, output_type="div")

    return render(request, 'weatherapp/forecast.html', {'place': place,
    'option': option, 'tostring': tostring, 'plot_div': temperature_plot})
```

It checks the value of the `option` variable. If it is "Temperature", it iterates over the `filtered_datalist` and extracts the temperature and date values, converts the temperature values from Kelvin to Celsius and creates a Pandas dataframe with the dates and temperatures. It then uses the Plotly library to create a line plot of the temperature data and assigns the generated plot to the `temperature_plot` variable. Finally, it renders the `forecast.html` template and passes in the `place`, `option`, `tostring`, and `plot_div` variables to be used in the template.

```
elif option == "Sky":
    images = {
        "Clear": "images/clear.png",
        "Clouds": "images/cloud.png",
        "Rain": "images/rain.png",
        "Snow": "images/snow.png"
    }

    # extracts the sky condition and date values, and creates a
    dictionary with date as key and sky condition as value.
    sky_conditions = [data['weather'][0]['main'] for data in
    filtered_data]
    image_paths = []
    for condition in sky_conditions:
        image_paths.append(images[condition])

    dates = [data['dt_txt'] for data in filtered_data]
    sky = {}
    for i in range(len(sky_conditions)):
        sky[dates[i]] = image_paths[i]

    return render(request, 'weatherapp/forecast.html', {'forecast_api':
    forecast_api, 'place': place, 'option': option, 'tostring': tostring,
    'image_paths': image_paths, 'sky': sky})
```

```
return render(request, 'weatherapp/forecast.html', {})
```

If the option variable is "Sky", it will iterate over the filtered\_data list and extracts the sky condition and date values, and creates a dictionary with date as key and sky condition as value. It then returns the forecast.html template with the place, option, tostring, image\_paths and sky variables to be used in the template.

- **URLs.py**

```
from django.urls import path
from .views import index, about, aqi, forecast

urlpatterns = [
    path('', index, name='index'), # Home page
    path('about', about, name='about'), # Page showing details of the web app
    path('aqi', aqi, name='aqi'), # Page showing AQI and Weather of a city
    path('forecast', forecast, name='forecast') # Page showing weather
forecast of a city
]
```

This code is defining the URLs for the web application. It imports the views (index, about, aqi, forecast) from the views.py file.

It uses the path() function from the django.urls module to define the URLs for the application. Each path() function call is for a different webpage of the application and each one takes in 3 arguments:

1. The first argument is the URL pattern, this is the part of the URL that will come after the domain name in the browser. For example, in the first call, the pattern is empty, so when the user visits the website, the URL will be the domain name only.
2. The second argument is the view function that will handle the request made to that URL. For example, in the first call, the view function is 'index' which is the index view.
3. The third argument is the name of the URL pattern, this is used to refer to the URL pattern in my html file.

So, when a user navigates to the home page, the browser sends a request to the server and the request matches the first path which is '/' the index view will handle the request and return the index.html template to the browser to be rendered. And so for other page/urls.

- **Rendering Template function (views.login)**

```
# function to handle user login requests
def loginPage(request):
```

```

    if request.user.is_authenticated:                                # if user is already
authenticated, will be redirected to index page
        return redirect(reverse('weatherapp:index'))
    else:
        if request.method == "POST":
            username = request.POST.get('username')
            password = request.POST.get('password')

            user = authenticate(request, username=username,
password=password)          # call the authenticate method to check the
credentials

            if user is not None:
                login(request, username)
                return redirect(reverse('weatherapp:index'))      # an user
will be redirected to index page if authenticated
            else:
                messages.info(request, 'Username OR password is incorrect')
# shows message if credentials do not match

        context = {}
        return render(request, 'users/login.html', context)

```

The loginPage view handles user login requests. It first checks if the user is already authenticated and if so, redirects them to the index page. If the user is not authenticated, the view checks if the request method is "POST" which means that the user has submitted the login form. It then retrieves the entered username and password, calls the authenticate method to check if the entered credentials match any of the existing user and if they do, it logs in the user and redirects them to the index page. If the credentials do not match, it shows an error message to the user.

- **Rendering Template function (views.register)**

```

# function to handle user registration requests
def register(request):
    if request.user.is_authenticated:
        return redirect(reverse('weatherapp:index'))
    else:
        form = CreateUserForm()          # create a form object of
CreateUserForm
        if request.method == 'POST':
            form = CreateUserForm(request.POST)
            if form.is_valid():          # check if the data is valid
                form.save()              # saves the new user to the database
                user = form.cleaned_data.get('username')
                # show a success message and redirect the user to login page
                messages.success(request, 'Account was created for ' + user)
                return redirect(reverse('users:login'))

        context = {'form': form}
        return render(request, 'users/register.html', context)

```

The register view handles user registration requests. It first checks if the user is already authenticated and if so, redirects them to the index page. If the user is not authenticated, it creates a form object of the CreateUserForm, checks if the request

method is "POST" which means that the user has submitted the registration form, and if so, it retrieves the entered data, validates it and saves the new user to the database. If the form is valid, it shows a success message and redirects the user to the login page. If the form is not valid, it shows the form with error messages.

- **Rendering Template function (views.logout)**

```
# function to handle user logout requests
def logoutUser(request):
    logout(request)
    return redirect(reverse('users:login'))
```

The logoutUser view handles user logout requests. It calls the logout method to log out the current user and redirects them to the login page.

- ❖ There are still a lot of methods and functions that I implemented in my project, such as python embedded tag in my html template file, but I would just mention the important functions I used to handle my web requests and return a response.

## E. Evaluation and Reflection

As of now, I have not deployed my web app, so I have to run my app locally on my terminal. In terms of features and designs, the app functions as per my plan. One of the bugs that is still unsolved is that when I try to submit the forecast form without inputting any place or a place that is unavailable, my Django web app will run into an error.

```
KeyError at /forecast
'list'

Request Method: POST
Request URL: http://127.0.0.1:8000/forecast
Django Version: 4.1.4
Exception Type: KeyError
Exception Value: 'list'
Exception Location: /Users/jeff2709/Documents/Binus/Semester1/weatherapp2/venv/weather/weatherapp/views.py, line 191, in get_data
Raised during: weatherapp.views.forecast
Python Executable: /Users/jeff2709/Documents/Binus/Semester1/weatherapp2/venv/bin/python3
Python Version: 3.9.13
Python Path:
  ['Users/jeff2709/Documents/Binus/Semester1/weatherapp2/venv/weather',
   /Users/jeff2709/opt/anaconda3/lib/python3.9.zip,
   /Users/jeff2709/opt/anaconda3/lib/python3.9,
   /Users/jeff2709/opt/anaconda3/lib/python3.9/lib-dynload,
   /Users/jeff2709/Documents/Binus/Semester1/weatherapp2/venv/lib/python3.9/site-packages']
Server time: Sun, 15 Jan 2023 14:38:39 +0000

Traceback Switch to copy-and-paste view
/Users/jeff2709/Documents/Binus/Semester1/weatherapp2/venv/lib/python3.9/site-packages/django/core/handlers/exception.py, line 55, in inner
    55.         response = get_response(request)
    ▶ Local vars

/Users/jeff2709/Documents/Binus/Semester1/weatherapp2/venv/lib/python3.9/site-packages/django/core/handlers/base.py, line 197, in _get_response
    197.         response = wrapped_callback(request, *callback_args, **callback_kwargs)
    ▶ Local vars

/Users/jeff2709/Documents/Binus/Semester1/weatherapp2/venv/lib/python3.9/site-packages/django/contrib/auth/decorators.py, line 23, in _wrapped_view
    23.         return view_func(request, *args, **kwargs)
    ▶ Local vars

/Users/jeff2709/Documents/Binus/Semester1/weatherapp2/venv/weather/weatherapp/views.py, line 221, in forecast
    221.         filtered_data = forecast_data.get_data(days)
    ▶ Local vars

/Users/jeff2709/Documents/Binus/Semester1/weatherapp2/venv/weather/weatherapp/views.py, line 191, in get_data
    191.         filtered_data = self.forecast_api['list']
    ▶ Local vars

Request information
USER      jeff2709
GET       No GET data
```

There are still many features and conditions that can be improved in this web application. I would like to fix the bug that happened on the forecast page as well as change some duplicative code.

### *Reflection*

It took me about two to three weeks to contemplate several ideas that I wanted to work on, but I finally decided to make a weather web app, which I think is qualified enough for a beginner's coding final project and for an introduction course to programming. I learned how to connect to third-party APIs, get data from those APIs, pull it back into my web app, filter the data, do different things based on what the APIs return, and do some logic while working on this project. Moreover, I also learned how to visualize the filtered data and plot it in my web app.

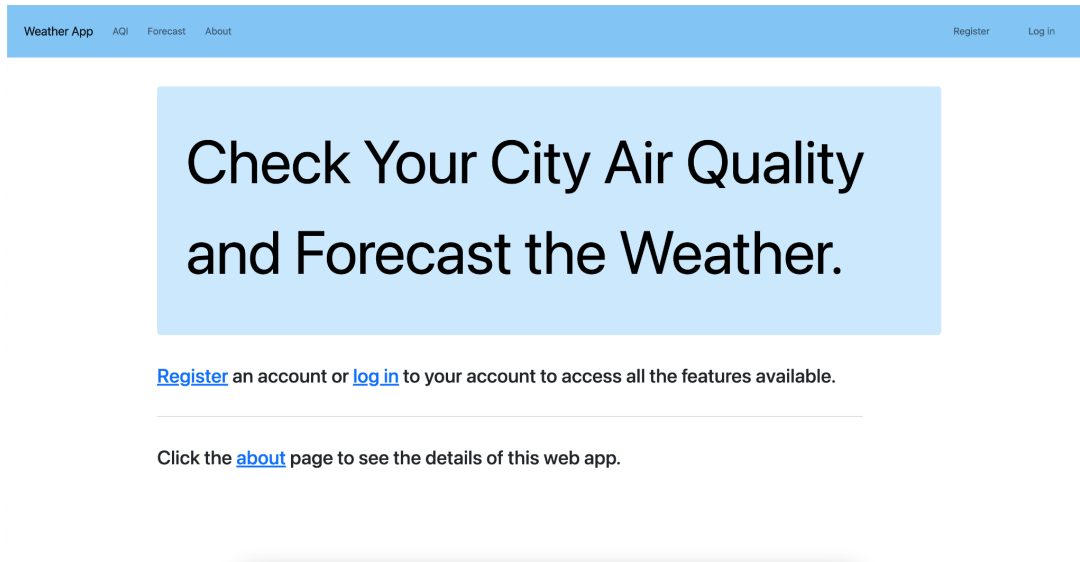
During the creation of this project, I have faced a lot of challenges and ran into many issues working with Django because there are many functions in Django that I have to spend quite some time understanding well in order to have my web app function. To overcome those problems, I tried to watch many YouTube videos about Django. Besides

the aid of YouTube, Stackoverflow has also been really helpful to me because it offered many possible ways to fix my errors and showed me how to debug them.

In conclusion, I am satisfied with the outcome of this project, considering I had no experience in coding before entering university. The functionality and design of the features work as I expected them to. In addition, I applied the concepts beyond what has been taught in class: Django (including working with the backend), retrieving data from APIs, visualizing data, and many more. Working on this project is a challenging and rewarding experience. Another thing that I have learned from this project is to have better time management so I do not feel pressured when the deadline looms. In the future, I hope to improve my coding skills.

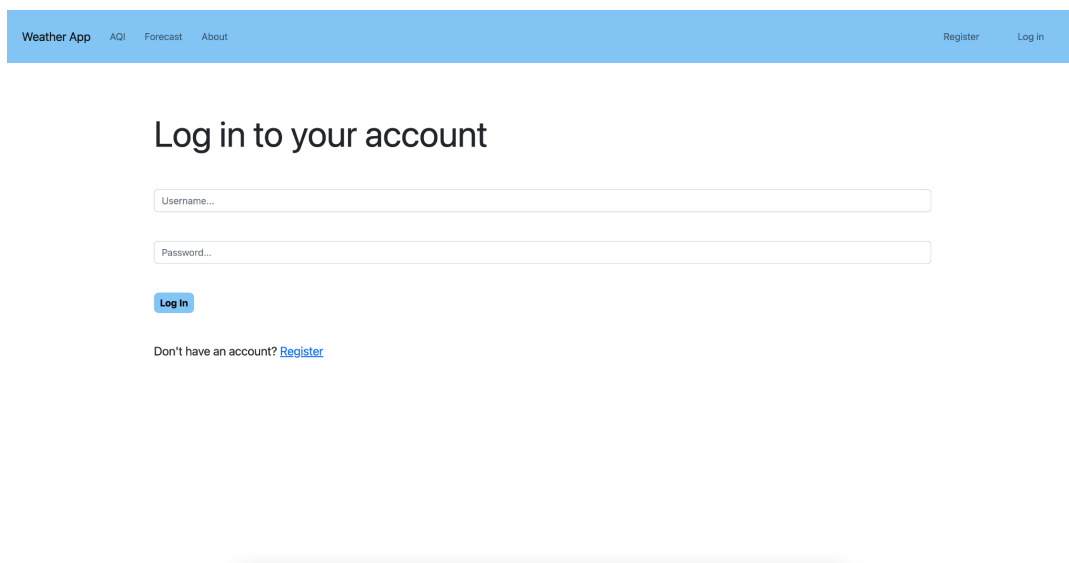
## F. Working Program Evidence

### *Home Page*



The screenshot shows the home page of a web application. At the top is a blue navigation bar with links: "Weather App", "AQI", "Forecast", "About", "Register", and "Log in". The main content area has a light blue background with the text "Check Your City Air Quality and Forecast the Weather." in large, bold, black font. Below this, there is a line of text: "Register an account or log in to your account to access all the features available." followed by a horizontal line. Under the line, it says "Click the about page to see the details of this web app." with "about" as a blue link. The page has a subtle shadow effect at the bottom.

### *Login Page*



The screenshot shows the login page of the web application. It has the same blue navigation bar as the home page. The main content area is white with the heading "Log in to your account" in bold black font. Below the heading are two input fields: "Username..." and "Password...". Under the password field is a blue "Log in" button. At the bottom, there is a link: "Don't have an account? Register". The page has a subtle shadow effect at the bottom.

### *Register Page*



## Register an Account

Register Account

Already have an account? [Login](#)

## AQI Page

## Check the Air Quality and the Weather Condition of a city

Search

### Moderate

Current Jakarta Air Quality: 57, (Main Pollutant: PM2.5)

(51-100) Air quality is acceptable. However, there may be a risk for some people, particularly those who are unusually sensitive to air pollution.

The current datetime is: Jan 15, 2023 at 03:03 AM WIB

### Weather Condition in Jakarta

Climate: Few Clouds

Temperature: 26°C

Humidity: 82%

Pressure: 1009 mbar

Wind Speed: 3.09 km/h



## Forecast Page - Temperature

## Weather Forecast for the Next Days

Enter a place

Place

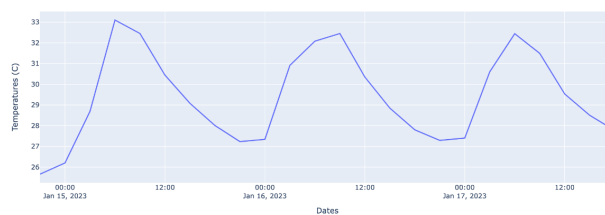
Forecast Days (Select the number of forecasted days [1-5])

Select data to view

Temperature

Forecast

### Temperature for the next 3 days in Jakarta



## Forecast Page - Sky

### Weather Forecast for the Next Days

Enter a place

Place


Forecast Days (Select the number of forecasted days [1-5])

Select data to view


Temperature

Forecast


Sky for the next 3 days in Jakarta




2023-01-14 21:00:00




2023-01-15 00:00:00




2023-01-15 03:00:00




2023-01-15 06:00:00




2023-01-15 09:00:00




2023-01-15 12:00:00




2023-01-15 15:00:00




2023-01-15 18:00:00




2023-01-15 21:00:00




2023-01-16 00:00:00




2023-01-16 03:00:00




2023-01-16 06:00:00




2023-01-16 09:00:00




2023-01-16 12:00:00




2023-01-16 15:00:00




2023-01-16 18:00:00





2023-01-16 21:00:00





2023-01-17 00:00:00














## About Page

[Weather App](#) [AQI](#) [Forecast](#) [About](#)

Welcome, jeff2709. [Log out](#)

This web app is developed by Jeffrey for Algorithm and Programming Final Project.

In this web app, you could find out whether the place you staying has a good air quality and moreover, you can forecast the weather in your city. In this about page, you could check the city available for its AQI information.

### Check State Available in a Country

Enter country

Search

1. Aceh

2. Bali

3. Bangka Belitung

4. Banten

5. Bengkulu

6. Central Java

7. Central Kalimantan

8. Central Sulawesi

9. East Java

10. East Kalimantan

## G. References

- Stackoverflow -> troubleshoot
- <https://www.postman.com/> -> testing my API data
- [https://www.youtube.com/playlist?list=PL-51WBLyFTg2vW-\\_6XBoUpE7vpmoR3ztQ](https://www.youtube.com/playlist?list=PL-51WBLyFTg2vW-_6XBoUpE7vpmoR3ztQ) ,  
[https://www.youtube.com/watch?v=teG-MXNFeUI&list=PL\\_99hMDIL4d3KvzWadeMCc7y9bn3E5ix0](https://www.youtube.com/watch?v=teG-MXNFeUI&list=PL_99hMDIL4d3KvzWadeMCc7y9bn3E5ix0) -> learning django
- <https://www.youtube.com/watch?v=mgX0Iz4q0bE> -> data visualization
- <https://www.youtube.com/watch?v=Fwbn1HMsyng&t=1070s> -> data visualization (plotly) in django
- <https://www.iqair.com/id/commercial-air-quality-monitors/api> -> air visual (aqi) api
- <https://openweathermap.org/forecast5> -> forecast api
- Figma, LucidChart, Diagrams.net -> tool used for design
- Pixel -> images (icons)