**Weather and Pollen data for IP Address**

Kristina Lashchuk

University of South Carolina Upstate

Department of Informatics, Spartanburg, SC USA

lashchk@email.uscupstate.edu

**Abstract**

Everyone has days when fresh air would be enjoyable, but many are unable to enjoy the fresh air due to weather conditions, allergies, or both. Although many applications exist that provide weather and pollen data, they do not provide much detailed data. In this work, we designed and developed an API that will provide more detailed pollen and weather data using the location of an IP address. Open-Meteo, ip-api, and Pollen APIs were used to create this API called WPI.

1. **Introduction**

There are days when fresh air would be enjoyable for everyone. Although, for many individuals, there are days they are unable to enjoy fresh air due to weather conditions, allergies, or both. Of course, some applications can be used to check the weather and pollen to determine if they can enjoy some fresh air. For instance, individuals can use the Meteomatics API to fetch historical, current, and forecast data for any location (n.d.). The Meteomatics API provides weather data, from temperature to precipitation, and pollen concentration data for types such as birch. However, many of these applications, such as the Meteomatics API, that provide weather and pollen data do not provide much detailed data.

In this paper, WPI, a weather and pollen data for IP addresses API, is designed and developed. While there are existing APIs that provide weather and pollen data for different locations, WPI will provide more detailed data. For example, when requesting pollen data, users will receive all the available pollen types, including their color, category, seasonality, index, index description, health recommendations, and plant description for the requested location. In creating WPI, the APIs Open-Meteo, ip-api, and Pollen API are used. Open-Meteo was used for weather data, ip-api for location data, and Pollen API for pollen data.

1. **Motivations and Background**

In this section, there will be a description of the terminology used in this paper and a motivating example for IPW.

*II.I Terminology*

* **API (application programming interface)** is a set of rules that allows communication between software applications to exchange data, features, and functionality. APIs also allow users to incorporate other applications, share only necessary information, and share data and functions with third parties (Goodwin, 2024).
* **API key** is a code of letters and numbers that identifies and authenticates a user. It allows users to connect with an API and allows developers to control the use of their APIs.
* **Internet Protocol address (IP address)** is a numerical identifier assigned to a device connected to the internet. It allows a device to send and receive data, communicate with other devices, identify the host or network being used, and identify the device's location.
* **Internet Service Provider (ISP)** is a company that provides access to the internet, such as Verizon.
* **Latitude (Lat)** is a coordinate that identifies the distance north or south of the Earth’s equator. It ranges from -90 degrees at the south pole to 90 degrees at the north pole.
* **Longitude (Long)** is a coordinate that identifies the distance east or west of the meridian at Greenwich, England. It ranges from 0 degrees at the meridian to 180 degrees eastward and -180 degrees westward.
* **Temperature (temperature\_2m)** is the degree of heat in an object or substance that can be measured in Fahrenheit or Celsius. In this paper, temperature will be measured in Fahrenheit and at 2 meters above ground.
* **Humidity (relative\_humidity\_2m)** is the amount of water vapor in the air at 2 meters above ground**.**
* **Precipitation** is the sum of rain, showers, and snow for the last hour.
* **Snowfall** is the amount of snow that has fallen for the last hour in centimeters.
* **Universal Pollen Index (UPI)** is a global unified index that uses a scale of 0 (None) to 5 (Very High). UPI helps simplify comparing pollen in different locations.
* **DayInfo** provides the date of the represented forecast.
* **PollenTypeInfo** provides the seasonality of available pollen types, UPI, and health recommendations for the requested location.
* **PlantInfo** provides the names and descriptions of plants at the requested location, including their seasonality, appearance, and cross-reactions.

*II.II Motivating Example*

Given an IP address, it is processed through the ip-api API, which returns the latitude and longitude coordinates. The ip-api API allows users to look at other information that is related to IPV4 and IPv6 addresses, such as time zone, country, region, etc. The latitude and longitude coordinates are used to make a request to the Open-Meteo API and fetch the temperature data. The Open-Meteo API gives users access to current weather data, such as temperature and rain, for any location on Earth. The coordinates are also used to make a request to the Pollen API and fetch pollen information, including types, indexes, and health recommendations. This API allows users to identify and learn more about the plants that affect their allergies.

1. **Proposed Approach**

This API begins with a request made in the driver application to the custom API. The request includes the base, app version, and the resource weather with an IP address, as shown in the code below. Once the request is made, the IP address is passed through the Weather resource into the get function.

BASE = "http://127.0.0.1:5000/"

APP\_VERSION = "v1/"

response = requests.get(BASE + APP\_VERSION + 'weather/{enter an IP Address here})

print(response)

In the get function, the ip-api API URL is separated into three pieces. URL1\_1 contains the first portion of the ip-api API URL, and URL1\_2 contains the IP address entered into the driver application request. URL1\_3 contains the last portion of the URL listing the fields to return. The fields listed to be returned are status, message, regionName, city, lat, lon, and isp. These three URLs are then combined into URL1 to create a complete URL. Then a request is made using URL1 in the variable called response. If the status code of the response is 200, then the request is saved, in JSON format, to the jresponse variable. The latitude coordinate from the request is saved to the variable lat, and the longitude is saved to the lon variable. Then the IP address, latitude, longitude, city, and regionName are printed. If the status code of the response is not 200, then “Failed to retrieve data” and the status code are printed. This first portion of the get function, which gets the latitude and longitude coordinates of the requested IP address, is shown below.

class Weather(Resource):

    def get(self, IP\_address):

        URL1\_1 = "http://ip-api.com/json/"

        URL1\_2 = IP\_address

        URL1\_3 = "?fields=status,message,regionName,city,lat,lon,isp"

        URL1 = (URL1\_1 + URL1\_2 + URL1\_3)

        response = requests.get(URL1)

        if (response.status\_code == 200):

            jresponse = response.json()

            lat = jresponse["lat"]

            lon = jresponse["lon"]

            print(IP\_address)

            print(lat)

            print(lon)

            print(jresponse["city"])

            print(jresponse["regionName"])

        else:

            print("Failed to retrieve data:", response.status\_code)

Once the latitude and longitude coordinates are saved, the Open-Meteo API URL is separated into five pieces. URL2\_1 contains the first portion of the Open-Meteo API URL that states the forecasted weather is to be returned, and latitude equals. URL2\_2 contains the latitude coordinate, retrieved from the ip-api API URL, and is converted to string format. The middle portion of the Open-Meteo API URL, stating longitude, is contained in URL2\_3. URL2\_4 is the longitude coordinate, converted to string format, from the ip-api API URL. The last portion of the Open-Meteo URL is contained in URL2\_5. It lists the current temperature, relative humidity, precipitation, and snowfall to be returned. It states the temperature is to be returned in Fahrenheit units as well. URL1\_1, 2, 3, 4, and 5 are combined into the variable URL2, creating the complete Open\_Meteo API URL. Then a request is made using URL2 in the variable called response\_2. If the status code of response\_2 is 200, the data in response\_2 is printed in JSON format. If the status code is not 200, then “Failed to retrieve data” and the status code are printed. This portion of the get function, which gathers weather data, is shown below.

URL2\_1 = "https://api.open-meteo.com/v1/forecast?latitude="

URL2\_2 = str(lat)

URL2\_3 = "&longitude="

URL2\_4 = str(lon)

URL2\_5 = "&current=temperature\_2m,relative\_humidity\_2m,precipitation,snowfall&temperature\_unit=fahrenheit"

URL2 = (URL2\_1 + URL2\_2 + URL2\_3 + URL2\_4 + URL2\_5)

response\_2 = requests.get(URL2)

if (response\_2.status\_code == 200):

    print(response\_2.json())

else:

    print("Failed to retrieve data:", response\_2.status\_code)

When the weather data is retrieved, the Pollen API URL is separated into five pieces as well. The first portion of the Pollen API URL is contained in URL3\_1, which returns the pollen forecasted and requires you to add your API key. URL3\_2 contains the same thing as URL2\_2. In URL3\_3, it states location.latitude, and in URL3\_4, it is the same as in URL2\_4. The last portion of the Pollen API URL stating the number of days to be forecasted is contained in URL3\_5. The five URLs are combined in URL3 creating the properly working Pollen API URL. A request is then made using URL3 and saved in the response\_3 variable. If the status code for response\_3 is 200, then the data is printed in JSON format. However, if the status code is not 200, then “Failed to retrieve data” and the status code are printed. The last portion of the code that gathers pollen data is shown below.

URL3\_1 = "https://pollen.googleapis.com/v1/forecast:lookup?key={Enter your API key here}&location.longitude="

URL3\_2 = str(lon)

URL3\_3 = "&location.latitude="

URL3\_4 = str(lat)

URL3\_5 = "&days=1"

URL3 = (URL3\_1 + URL3\_2 + URL3\_3 + URL3\_4 + URL3\_5)

response\_3 = requests.get(URL3)

if (response\_3.status\_code == 200):

    print(response\_3.json())

else:

    print("Failed to retrieve data:", response\_3.status\_code)

1. **Discussion**

The development of the IWP API started with choosing three APIs from a list provided through GitHub. This list included a variety of APIs, from animal-based to weather-based. The process of choosing three APIs was difficult; however, the final ones chosen were Open-Meteo, ip-api, and Pollen API. The process was difficult because the list contained many APIs that were image- or random fact-based, or the link included did not work. At first, three weather APIs were chosen, but it was unclear how these APIs could be used to create a new API. With this, the list of APIs was referred to again, and IP address APIs were discovered. When these APIs were discovered, an idea was developed that the new API could use one of the IP address APIs to get coordinates that could be used to get data from other APIs.

Another issue that was raised was with API keys. During the development of the new API, it was discovered that many of the APIs in the list required an API key. One of the APIs that was considered in the development of IWP was OpenWeather, which did require an API key. However, the goal of accessing the data in this API was not accomplished even after creating an account with this API and creating a key. Due to limited time, another API was chosen that did not require an API key, called Open-Meteo. Apiip was another API that required an API key but did not work. Since Apiip did not work, ip-api was chosen to gain access to IP address information. Although the API key did not work for OpenWeather and Apiip, it did work for the Pollen API. By using a Google account, an API key was created through the APIs & Services section under the Credentials menu.

The development of the new API continued with the creation of a custom API and a driver application. The custom API and the driver application were developed to get the latitude and longitude coordinates from an IP address and then used to get weather and pollen data. Once the custom API and the driver application were created and run for the first time, the API did not work. There were multiple issues with the custom API and the driver application that caused the API to not work.

With the driver application, the first issue was the absence of models for each of the APIs and the new API. To fix this, four models were created called: IP\_Locator, Weather, Pollen, and IWP. The IP\_Locator model, based on the ip\_api Api, has the following columns: isp, city, regionName, lat, and lon. The Weather model with the columns temperature\_2m, relative\_humidity\_2m, precipitation, and snowfall is based on the Open-Meteo API. The third model, Pollen, based on the Pollen API, has the columns DayInfo, PollenTypeInfo, and PlantInfo. The last model, IWP, for this new API includes: isp, city, regionName, temperature\_2m, relative\_humidity\_2m, precipitation, and snowfall.

An error was received after creating the models and running the driver application. The error stated that db, used when making the models, was not defined. To resolve this, dependencies were imported, and application and database variables were defined as shown below. Once models, dependencies, and application and database variables were added, the driver application ran with no issues.

import requests

from flask import Flask

from flask\_restful import Api, Resource, reqparse, abort, fields, marshal\_with

from flask\_sqlalchemy import SQLAlchemy

app = Flask(\_\_name\_\_)

api = Api(app)

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///database.db'

db = SQLAlchemy(app)

app\_version = "v1/"

The custom API was created with the dependencies and application and database variables shown above, resource fields, a weather resource, and a debugger function. The first time the custom API was run, it did not work due to missing and incorrect coding. The missing code was registering the resource called Weather with the API, as shown below.

api.add\_resource(Weather, "/" + app\_version + "weather/<path:IP\_address>")

The incorrect coding that was not allowing the custom API to work was in the Weather resource. When a request was made in the driver application with an IP address included, the IP address was not being properly read and used in the Weather resource. To allow the Weather resource to properly read and use an IP address, a URL was separated into pieces. The separate pieces were then combined to make a working URL. This was applied to all three API URLs. For example, the ip-api URL was separated into three pieces and then combined to create a URL.

URL1\_1 = "http://ip-api.com/json/"

URL1\_2 = IP\_address

URL1\_3 = "?fields=status,message,regionName,city,lat,lon,isp"

URL1 = (URL1\_1 + URL1\_2 + URL1\_3)

Following the revision of the URLs, more issues occurred. The first issue was that the latitude and longitude coordinates were not being printed from the ip-api API request. The package Pandas was used to try fixing the issue but was unsuccessful. The attempt went further by trying to normalize the data in a data frame using the code *df = pd.json\_normalize(data).* Normalizing the data in a data frame still did not fix the issue. Changing where it says response to jresponse and using the format *jresponse[“”]* in the code, as shown below, helped allow the latitude and longitude coordinates to be printed.

jresponse = response.json()

lat = jresponse["lat"]

lon = jresponse["lon"]

The other issue that occurred after the URLs were revised was in the Open-Meteo and Pollen APIs’ URLs. The error received was that the latitude and longitude sections of the URLs were unable to be concatenated from float to string. The only way to link the URL sections together is from string to string. This means the latitude and longitude sections of the URL needed to be converted to string format, as shown below. This was then applied to the Pollen API URL as well. Once these issues were resolved, the custom API printed the latitude and longitude coordinates, weather data, and pollen data.

URL2\_2 = str(lat)

URL2\_4 = str(lon)

1. **Related Work**

As mentioned previously, there are existing APIs that provide weather, pollen, or weather and pollen data for different locations. Another API that provides both weather and pollen data for any location by using latitude and longitude coordinates is Tomorrow.io. Tomorrow.io has more than 80 different data fields, including weather, pollen, and air quality, that a user can use to build anything (Tomorrowdocs, 2023). Hygraph, a dashboard that allows users to join different services and data sources into a single API, provides another example (Woloszyn, 2021). The example provided by Hygraph is similar to the API developed in this paper; however, it does not get the latitude and longitude coordinates from an IP address. Instead, in this example, the latitude and longitude coordinates are retrieved from the location details of the user of Hygraph (Hygraph Team, 2023).

1. **Conclusion and Future Work**

In this paper, IWP was introduced as an API that can get the latitude and longitude coordinates of an IP address and then use the coordinates to get weather and pollen data. Through testing the code by trial and error using a personal IP address, the API showed the weather and pollen data for the location of the personal IP address requested. This API allows users to determine if weather, allergies, or both will prevent them from enjoying fresh air by providing more detailed weather and pollen data.

In the future, IWP can be developed further to gather more data. More data that can be gathered includes data on air quality, wildfire tracking, environmental alerts, weather alerts, and natural disasters such as hurricanes. This data can be gathered from IWP, with the same method used to gather the weather and pollen data, by using other APIs. For instance, air quality data for a particular location based on latitude and longitude coordinates can be gathered using an API called OpenAQ or BreezoMeter. The API BreezoMeter can also be used to gather data on wildfire tracking and environmental alerts, such as if air quality is expected to get worse. For weather alerts, like flood warnings and instructions on how to proceed, an API called WeatherAPI can be used to gather the data.

**References**

Goodwin, M. (2024, April 9). *What is an API?* IBM. Retrieved April 23, 2024, from https://www.ibm.com/topics/api

[Hygraph Team] (2023, January 25). *How to integrate multiple REST APIs*. Hygrpah. Retrieved April 21, 2024, from https://hygraph.com/blog/integrate-rest-apis

[Meteomatics] (n.d.). *Particles*. Retrieved April 1, 2024, from https://www.meteomatics.com/en/api/available-parameters/particles/#pollen\_concentrations

[Tomorrowdocs] (2021). *Weather data layers*. Retrieved April 21, 2024, from https://docs.tomorrow.io/reference/weather-data-layers

Woloszyn, D. (2021, June 7). *Hygraph.* Benjamas.io. Retrieved April 21, 2024, from https://bejamas.io/discovery/headless-cms/hygraph/