University of Science and Technology of Southern Philippines

College of Information Technology and Computing Department of Computer Science

Simple Weather Information Retrieval Application

By:

Jimenez, Cherry Lee H.

Lucagbo, Eros P.

Maulod, Zayq Rashid F.

December 12, 2024

1. Project Overview

1.1. Objective

The project aims to develop a simple tool designed to provide users with **basic weather information** for a **specific city**. The primary goal is to create a straightforward system where users can quickly and easily access key weather details without unnecessary complexity. By integrating the OpenWeatherMap API, the application ensures that users can retrieve essential data about the current weather conditions in an efficient and reliable manner.

1.2. Key Features

1. **Basic Weather Information:** Users can input a city name to view essential weather details such as temperature, humidity, and general conditions.
2. **User-Friendly Interface:** A minimal and straightforward interface allows users to interact with the application effortlessly.
3. **Error Messages:** Provides simple feedback if the city name is incorrect or data cannot be retrieved.
4. **API Integration:** Fetches weather data from OpenWeatherMap API to deliver accurate and up-to-date information.
5. **Lightweight Design:** The application focuses on core functionality, ensuring fast performance and ease of use.

2. System Architecture

2.1. Client (Front-end)

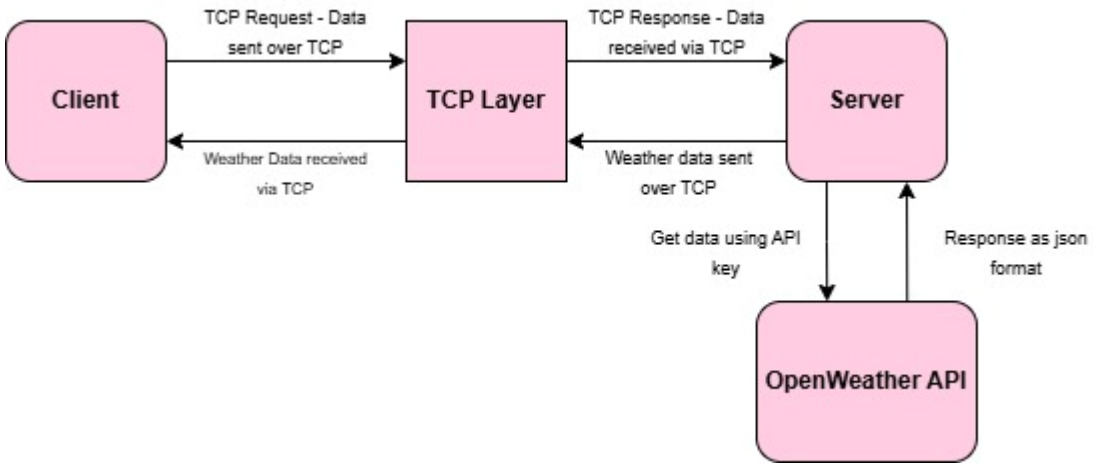
1. **Functionality:** The client provides a web-based interface where users can input the city name and view the retrieved weather information. It then displays the weather details fetched from the server.
2. **Interaction:** The client sends user input (city name) to the server via an HTTP POST request and receives weather data in response, which it renders using HTML templates.

2.2. Server (Backend)

1. **Functionality:** The server processes user requests by fetching weather information from the OpenWeatherMap API based on the city name provided by the client. It formats and returns the data in a structured response.
2. **Technology:** The server is built with Python, using the socket module for communication and the requests library for API integration.
3. **Interaction:** The server listens for client connections, processes the received city name, retrieves weather data from the API, and sends the formatted response back to the client for display.

2.3. Protocol (TCP/IP)

1. **Functionality:** Ensures reliable communication between the client and server by managing the transmission of requests and responses.
2. **Technology:** Utilizes the TCP/IP protocol suite for establishing and maintaining a connection between the client and server.
3. **Interaction:** Handles the transmission of the city name from the client to the server and the delivery of weather data back to the client.



3. Code Structure

PROJECT/
├── **app/**
│ ├── **static/**
│ │ ├── **style.css**
template/
├── **base.html, city.html**
init.py # initialize flask
routes.py # client side
├──**tcp-server/**
│ ├──**server.py** # our main server
├── **run.py** # run the application. It used threading to run both the frontend and the server

4. Client-Side Implementation

4.1. HTML Structure

Form Submission Handler:

- Prevent default form submission to allow custom handling.
- Retrieve the input values from the form fields.
- Clear previous messages like success or error messages.

- Validate the input, e.g., ensuring the city name is entered correctly.
- Send a POST request to the backend API (/weather) with the city name.

Form Submission Workflow:

- The user enters a city name, like "New York", into the form.
- The form triggers the submit event, and the handler is invoked.
- The city name is validated to ensure it’s not empty and contains only letters.
- If valid, the city name is sent to the backend API via a POST request.
- The backend processes the request and returns weather data, which is displayed to the user.
- If the city is invalid or something goes wrong, an error message is shown.

4.2. CSS Implementation

- **Font Import**: The Google font 'Familjen Grotesk' is imported for usage throughout the webpage, with support for different font weights and italics.
- **Body Styling**:
  - Background color set to a dark shade (#0B131E).
  - Text color set to white for contrast.
  - Padding applied at the top to create space.
- **Title Styling**:
  - Text is centered and styled with a large font size (67px) and bold weight.
  - Bottom margin added for spacing.
- **Container Styling**:
  - A flexible container with a column layout, centered using margin: auto.
  - Width set to 816px and height to 330px, with a rounded background (#202B3B).
  - Padding applied inside and a gap between child elements.
- **Paragraph Styling**:
  - Font size set to 24px with a lighter font weight for readability.
- **Input Fields**:
  - Padding applied for comfortable interaction.
  - Font size set to 26px to ensure input text is large enough.
- **Button Styling**:
  - Button positioned with top margin and given a font size of 25px.
  - Padding, width, and border radius applied to create a rounded, clickable button.
- **Info Containers**:
  - .info1 uses flex display to space elements with justify-content: space-between.
  - .info2 also uses flexbox to evenly space elements and center content with a background and rounded corners.
- **Text in Info Containers**:
  - The .cityname and .temp classes have large font sizes (64px) and bold weights

for prominence.
    - .info2 p has a font size of 40px.
- **Icon Styling**:
    - The .icon class uses flexbox to center images and text inside.
- **Circle Styling**:
    - A circular container created using border-radius: 50%, with height and width both set to 150px.
- **Image Styling**:
    - Images inside the .icon class are constrained to a size of 100px by 100px, using object-fit: contain for proper scaling.
- **Popup Notification**:
    - Hidden by default, the notification is styled with a red background, white text, and rounded corners.
    - A shadow is added to give a floating effect.
    - The notification shows when the show class is added, with an animation that fades in and out.
- **Fade-In/Fade-Out Animation**:
    - Defined using @keyframes, the notification fades in, remains visible, and fades out smoothly

4.3. Key Client-Side Functions

- **Form Validation:** Ensures that the data entered by the user is valid before submission.
- **Form Submission Handler:** Manages the form submission process, including data validation and sending requests to the server.
- **Display Message:** Provides user feedback on form validation, success, or error messages.
- **Sending a POST Request:** Handles communication with the backend by sending data and receiving responses.
- **Display Weather Data:** Formats and displays the weather data from the backend response.
- **Reset Form Fields:** Clears the form after submission.
- **Loading Indicator:** Manages the display of a loading indicator while waiting for the response.
- **Clear Messages:** Clears any displayed messages (error or success).
- Input Formatting: Ensures the input data is in the correct format before validation.

4.4. Code Snippet Overview

Routes.py:

```
from flask import Blueprint, render_template, request

import socket

import json


bp = Blueprint('main', __name__)


@bp.route('/')

def index():

    return render_template('base.html')


@bp.route('/weather', methods=['POST'])

def weather():

    city = request.form['city']


    weather_info = get_weather_info(city)


    # If error message returned, display it

    if isinstance(weather_info, str) and
weather_info.startswith("error"):

        error_message = weather_info

        return render_template('base.html',
error_message=error_message)

    else:

        weather_info, icon, description, temp, feels_like, humidity,
wind_speed, time = weather_info

        return render_template('city.html', cityname=city,
weather_info=weather_info, icon=icon, description=description,

                               temp=temp, feels_like=feels_like,
humidity=humidity, wind_speed=wind_speed, time=time)
```

```
def get_weather_info(city):
    host = '192.168.13.88'  # Server IP address
    port = 8080              # Port to connect to

    try:
        # Create socket connection
        client_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
        client_socket.settimeout(5)  # Set a timeout for the connection

        # Try connecting to the server
        client_socket.connect((host, port))

        # Send the city name
        client_socket.send(city.encode())

        # Receive the weather data
        weather_info = client_socket.recv(1024).decode()

        # Close the socket
        client_socket.close()
        if weather_info == '\"error\"':
            return "error: City not found or invalid data."

        # Split and validate the received data
        weather_info_parts = weather_info.split('|')
        if len(weather_info_parts) < 8:
            return "error: Incomplete data received."

        temp = weather_info_parts[1]
        feels_like = weather_info_parts[2]
```

```
        humidity = weather_info_parts[3]

        description = weather_info_parts[4]

        time = weather_info_parts[5]

        icon =
f"https://openweathermap.org/img/wn/{weather_info_parts[6]}.png"

        wind_speed = weather_info_parts[7]



        return weather_info, icon, description, temp, feels_like,
humidity, wind_speed, time



    except socket.timeout:

        return "error: Timeout occurred while trying to connect to the
server."

    except socket.error as e:

        return f"error: Unable to connect to the server. Error:
{str(e)}"

    except Exception as e:

        return f"error: An unexpected error occurred. Error: {str(e)}"
```

5. Server-Side Implementation

5.1. Key Server Functions

The server-side uses **HTTP (Hypertext Transfer Protocol)** to enable communication between the server and the external **OpenWeatherMap API**. HTTP is a standardized application-layer protocol used for sending and receiving data over the web.

Process

1. **Data Request to the Server:**
    a. After the user inputs the correct city
    b. The web app establishes a connection with the server using a TCP socket.
    c. It sends the city name to the server as part of this connection.
2. **Server Processes the Request:**
    a. Upon receiving the city name, the server makes an HTTP request to an external weather API (OpenWeatherMap).
    b. The server retrieves weather data for the specified city from the API.
3. **Server Formats the Data:**
    a. The server parses the API's response, extracting details like temperature, humidity, weather description, and wind speed.

b. It formats this data into a structured format that can be easily transmitted back to the web app.

4. **Response Sent to the Web App:**
   a. The server sends the formatted weather data back to the web app over the same TCP connection
5. **Web App Parses the Data:**
   a. The web app receives the weather data from the server and processes it into individual pieces of information (e.g., temperature, description)

5.2. Code snippets

Server.py

```
import socket
import requests
import json
import datetime

def convert_to_local_time(timestamp, timezone_offset):
    # Convert Unix timestamp to UTC time
    utc_time = datetime.datetime.utcfromtimestamp(timestamp)

    # Convert timezone offset from seconds to timedelta
    time_diff = datetime.timedelta(seconds=timezone_offset)

    # Adjust time to local timezone
    local_time = utc_time + time_diff

    # Format local time into a human-readable string
    formatted_time = local_time.strftime("%Y-%m-%d %H:%M:%S")

    return formatted_time

def fetch_weather(city):
    api_key = "2daef3a757430743c872e0ad0a0352e5"  # Get your OpenWeatherMap
API key
    base_url = "http://api.openweathermap.org/data/2.5/weather"
    complete_url = f"{base_url}?q={city}&appid={api_key}&units=metric"

    response = requests.get(complete_url)
    data = response.json()
    print(data)
    if data["cod"] == 200:

        main_data = data["main"]
        timezone = data["timezone"]
        dt = data["dt"]
        time = convert_to_local_time(timezone, dt)
        weather_data = data["weather"][0]
        wind = data["wind"]
        temp = main_data['temp']
        feels_like = main_data['feels_like']
        humidity =  main_data['humidity']
```

```
        description = weather_data['description']
        icon = weather_data['icon']
        wind_speed = round(wind['speed'] * 3.6)

       weather_info                                                    =
f"|{temp}|{feels_like}|{humidity}|{description}|{time}|{icon}|{wind_speed}|
"
    else:
        weather_info = "City not found!"

    return weather_info

def server_program():
    host = '192.168.13.88'  # Server IP address
    port = 8080       # Port to bind

    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind((host, port))
    server_socket.listen(2)  # Listen for one client
    print(f"Server listening on {host}:{port}")

    conn, addr = server_socket.accept()
    print(f"Connection from {addr} established.")

    city = conn.recv(1024).decode()  # Receive city name from client

    weather_info = fetch_weather(city)  # Fetch weather information
    conn.send(json.dumps(weather_info).encode())
    conn.close()

if __name__ == "__main__":
    server_program()
```

6. Protocol Design

The custom protocol used for communication between the client (Flask web app) and the server (weather-fetcher server) follows a request-response pattern where the client sends a city name, and the server processes the request, fetches weather data, and sends the results back. The protocol is designed for simplicity, ensuring reliability and clarity in how data is exchanged.

6.1. Protocol Outline

1. **Handshake:** Upon connecting, the client sends the city name request to the server, and the server responds with the weather data or an error message.
2. **Weather Request:** The client sends a city name as a message (string) over the TCP connection. The city name is the primary request to fetch weather information.
3. **Weather Data Transfer:** The server responds with a formatted string containing weather data, including temperature, humidity, description, and other relevant weather details, separated by delimiters (|).

4. **Error Handling:** If the city is not found or an error occurs, the server sends a specific error message (e.g., "City not found!"). The client can then handle this response by displaying an appropriate error message to the user.

6.2. Protocol Flow

1. **Client Sends City Request:**
    a. The client (Flask app) sends the city name to the server. The message is sent as a plain string.
    b. Example: **Cagayan De Oro City**
2. **Server Processes Request**:
    a. The server receives the city name and uses it to query the OpenWeatherMap API.
    b. The server processes the returned data, converts the timestamp to local time, and formats the weather information.
3. **Server Sends Weather Data**:
    a. The server sends back the weather data in a specific format (delimited by |), which includes:
        i. Temperature
        ii. Feels like
        iii. Humidity
        iv. Description
        v. Time (local time)
        vi. Weather icon URL
        vii. Wind speed

6.3. Example Protocol Flow

1. **Client Sends Request**:
    a. The client (Flask app) sends the city name to the server London
2. **Server Processes Request**:
    a. The server queries the OpenWeatherMap API for the weather data for "Cagayan De Oro City"
    b. The server receives the data and processes it (e.g., converts timestamps, formats weather data).
3. **Server Sends Weather Data**:
    a. The server sends the processed weather data back to the client.
4. **Client Displays Data**:
    a. The client (Flask app) receives the weather data and displays it on the web page for the user to see.

7. Error Handling & Logging

7.1. Error Handling Strategies

Scenario	Type	Message Displayed
City found	Success	"Weather data displayed on the results page"
City not found	Error	"City not found. Please try again."
Empty or invalid input	Error	"Please enter a valid city name."
Server connection issue	Error	"Unable to connect to the server. Please try again later."
API key invalid/missing	Error	"Server configuration issue. Please try again later."
Rate limit exceeded	Error	"Service is temporarily unavailable. Please try again."
API timeout or network issue	Error	"Unable to fetch weather data. Please try later."

8. Documentation

8.1. How to Run the Application

1. Requirements:
   - **Install Python:**
     - Download and install the latest version of Python from python.org. Ensure the installation includes the pip package manager.
   - **Install an IDE:**
     - Use an IDE capable of running Python code, such as Visual Studio Code (VS Code), PyCharm, or Jupyter Notebook.
   - **Install Dependencies:** Open a terminal or command prompt and install the required libraries using pip:
     - pip install requests
     - pip install flask

2. Steps to Run:
   - **Setting up the Server:**
     - Open the server.py file in your IDE.
     - Replace the placeholder **api_key** in the **fetch_weather** function with your OpenWeatherMap API key.
     - Modify the host and port variables in **server.py** if necessary (default: 192.168.13.88 and 8080).
     - Start the server:
       - python server.py
     - Confirm the server is running by checking for a message like: Server listening on 192.168.13.88:8080

- **Run the Flask Application:**
  - Open the **route.py** file in your IDE.
  - Ensure the **host** and **port** variables in the **get_weather_info** function match the server's configuration.
  - Start the Flask application:
    - flask run
  - Open a web browser and navigate to http://127.0.0.1:5000/.

3. Using the Application
   - On the homepage, input a city name and click "Get Weather."
   - The application will display the current weather information for the entered city, including:
     - Temperature
     - Feels Like
     - Humidity
     - Weather Description
     - Wind Speed
     - Local Time

8.2. API References

1. Server Functions:
   - convert_to_local_time(timestamp: int, timezone_offset: int) -> str

   Converts a Unix timestamp to a local time string.

   Parameters:
     - timestamp: Unix timestamp.
     - timezone_offset: Timezone offset in seconds.

       Returns: A string representing the local time.

   - fetch_weather(city: str) -> str

   Fetches weather information for a given city from the OpenWeatherMap API.

   Parameters:
     - city: Name of the city.

       Returns: Weather information as a formatted string. If the city is not found, return "City not found!".

   - server_program()

   Initializes and runs the TCP server.

   Exceptions: Handles socket errors or client disconnections gracefully.

2. Flask App Functions:
   - index()

     Route: /

     Renders the homepage where users can input a city name.

- weather()
  Route: /weather
  Handles POST requests, sends the city name to the TCP server, and receives weather data.
  Parameters:
  - city: City name entered by the user in the form.
    Returns: Renders a template (city.html) with weather details.

- get_weather_info(city: str) -> Tuple
  Sends a city name to the TCP server and parses the returned weather data.
  Parameters:
  - city: Name of the city.
    Returns: Parsed weather data as individual components (e.g., temperature, description, icon).

8.3. Common Issues and Troubleshooting

1. Common Errors
   a. Error: "City not found!"
      - Verify the city name is correct.
      - Check if the OpenWeatherMap API is reachable.
   b. Error: "Connection refused"
      - Ensure the TCP server (server.py) is running and accessible at the specified IP and port.
   c. Error: "KeyError: 'main'" or similar
      - The API response may differ if the city is invalid. Verify the API key and the query.

2. Debugging Tips
   a. Use print() statements in the server and Flask app to log the flow of data.
   b. Check the server's log to ensure the API requests are successful.

8.4 Test Instructions

To test the weather-fetching application, first ensure that Python 3.x and the requests library are installed. For testing the weather fetching functionality on the server, create a test_server.py file containing unit tests for the fetch_weather function. The tests should check if the function returns weather data for valid cities and the "City not found!" message for invalid cities. Run this test by executing python test_server.py. Additionally, for testing the client-server communication, create a test_client_server.py file to mock socket communication between the server and client. This test will ensure that data is sent and received correctly. Run this test by executing python test_client_server.py. These tests verify that the server fetches weather data correctly and the client-server interaction works as expected.

9. Testing

To ensure the robustness of this weather-fetching application, we developed and executed unit tests targeting critical functionalities:

- **Connection Tests:** we verified that the client successfully connects to the server and appropriately handles scenarios where the server is unavailable or unreachable. This ensures reliable communication between the components.
- **Data Transfer Tests:** we confirmed that the server correctly processes weather requests from the client and sends accurate weather information back. This ensures the integrity of the data being transferred and displayed to the user.
- **Edge Case Handling:** we tested for various edge cases, including invalid or empty city names, network timeouts, and unexpected disconnections during client-server communication. These tests ensure the application remains stable and provides meaningful error messages when issues occur.

The tests were conducted using Python's unittest framework, ensuring that the application's core functionalities perform as expected. This process helped identify and resolve issues, making the application more robust and user-friendly.