Smart Weather Forecasting System

1: Project Planning and Conceptualization

The first step in creating the Smart Weather Forecasting System was planning and defining the overall concept of the project. Here are the key components I aimed to develop:

1. **Objective:**
    - Create an IoT-based system to monitor real-time weather data such as temperature, humidity, wind speed, and wind direction.
    - Provide real-time weather alerts via SMS and activate a buzzer when severe conditions are detected (e.g., stormy weather).
    - Develop a web dashboard that displays the collected data and a 7-day weather forecast.
2. **Technologies:**
    - **Raspberry Pi**: To collect weather data from sensors.
    - **Flask**: For building a backend API to serve weather data.
    - **Twilio**: To send SMS alerts for severe weather conditions.
    - **GPIO**: To interface with a buzzer for alerting the user.
    - **Frontend (HTML, CSS, JavaScript)**: For building the web interface that displays the weather data.
3. **Hardware Components:**
    - Raspberry Pi 4
    - DHT11 temperature and humidity sensor
    - Wind speed sensor
    - Buzzer (connected to Raspberry Pi GPIO pins)
4. **Functionality:**
    - Collect and process weather data from the sensors.
    - Display the current weather and a 7-day forecast.
    - Trigger an alert (SMS and buzzer) when weather conditions are severe.

2: Hardware Setup and Sensor Configuration

The next step involved setting up the physical components of the system.

1. **Raspberry Pi Setup:**
    - Installed **Raspberry Pi OS** on the Raspberry Pi.

- Set up the Raspberry Pi on a local network to access it remotely.
- Configured the GPIO pins for sensor and buzzer control.

2. **Connecting Sensors:**
   - **DHT11 Sensor (Temperature and Humidity)**:
     Connected the DHT11 sensor to the Raspberry Pi's GPIO pins to monitor temperature and humidity levels.
   - **Wind Speed Sensor**:
     Used a wind speed sensor to capture wind speed data. This sensor was also connected to the GPIO pins of the Raspberry Pi.
   - **Buzzer**:
     A buzzer was connected to one of the GPIO pins to sound an alert when critical weather conditions (e.g., storm or heavy wind) were detected.
3. **Testing Sensors:**
   - Tested the sensors individually to ensure they were providing accurate readings (e.g., temperature, humidity, and wind speed).

3: Backend Development with Flask

The next stage involved building the backend API to handle the weather data and provide the necessary functionality.

1. **Flask API Setup:**
   - Set up a basic Flask application on the Raspberry Pi to handle HTTP requests and serve weather data.
   - Installed necessary Python libraries like Flask, twilio, RPi.GPIO, and random to facilitate data generation and communication.
   - Developed routes to handle:
     - **/weather-data**: Endpoint to return current weather data (temperature, humidity, wind speed, wind direction, and weather description).
     - **/7-day-forecast**: Endpoint to return a 7-day weather forecast.
2. **Simulating Weather Data:**
   - Used Python's random module to simulate weather readings.
   - Configured different ranges for normal and high weather conditions (e.g., temperature, humidity, wind speed).
   - Developed a function to generate random weather data based on these ranges.
3. **SMS Alerts (Twilio Integration):**
   - Used the Twilio API to send SMS alerts when the system detects adverse weather conditions (like "Rainy," "Windy," or "Stormy").

   - Configured the Twilio client with the necessary credentials (Account SID, Auth Token, and Phone Numbers).

4. **Buzzer Activation:**
   - Programmed the buzzer to activate when extreme weather conditions were detected.
   - Set the buzzer to sound for 2 seconds when triggered.

4: Frontend Development (HTML, CSS, JavaScript)

The next task was to build the web-based dashboard where users could view the real-time weather data and the 7-day forecast.

1. **HTML Structure:**
   - Created the main layout using HTML, which included:
     - **Navigation Bar**: Links to different pages (e.g., Home, About).
     - **Weather Dashboard**: Displayed the current weather conditions such as temperature, humidity, wind speed, wind direction, and weather description.
     - **7-Day Forecast Table**: Displayed the temperature and weather conditions for the next seven days.
2. **CSS Styling:**
   - Designed a clean, user-friendly interface with a modern look.
   - Styled the dashboard and forecast table with colors, fonts, and effects to make the website visually appealing.
   - Made the interface responsive to ensure it worked well on different screen sizes.
3. **JavaScript (Data Fetching):**
   - Used JavaScript's fetch() function to make asynchronous requests to the Flask API and fetch real-time weather data.
   - Updated the dashboard automatically every 2 seconds to ensure the weather data was always up-to-date.
   - Used JavaScript to populate the 7-day forecast table with the appropriate data from the /7-day-forecast endpoint.
4. **Frontend-Backend Integration:**
   - Integrated the frontend and backend by ensuring that the weather data from the Flask API was correctly displayed on the webpage.

5:Testing and Debugging

Once the system was built, thorough testing was essential to ensure everything worked smoothly.

1. **Testing Weather Data Fetching:**

- Checked that the frontend was correctly receiving and displaying real-time data (e.g., temperature, humidity, wind speed).
- Verified that the 7-day forecast was being populated dynamically and updated correctly.

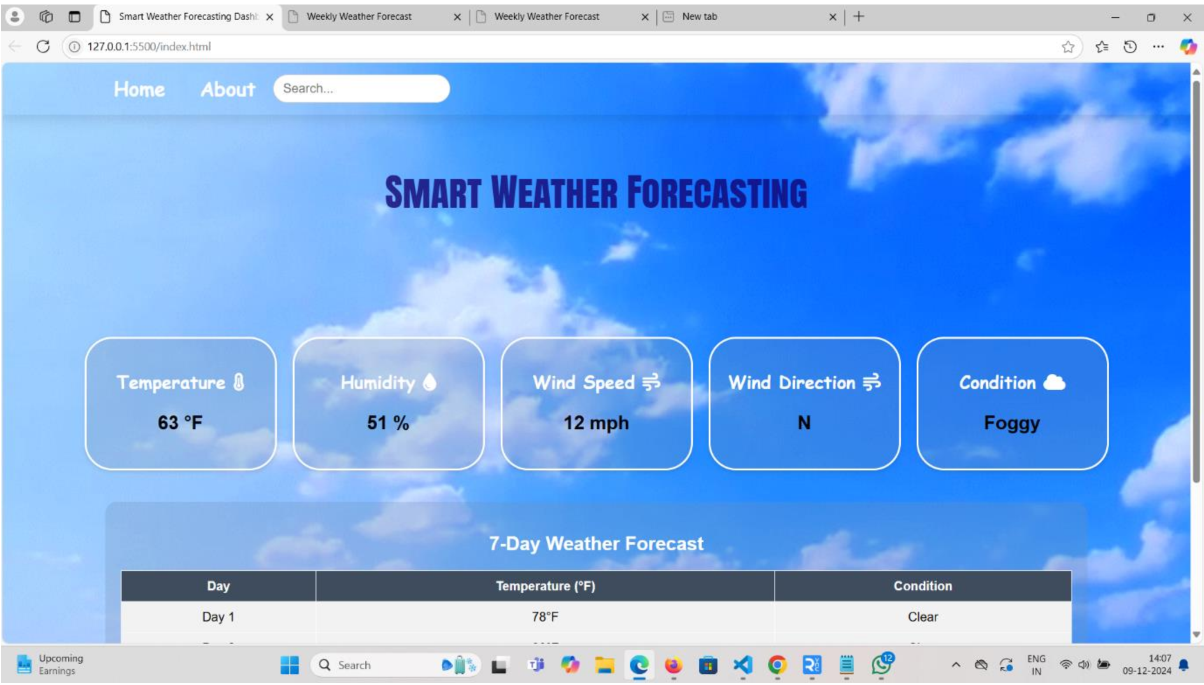
2. **Testing Alerts and Buzzer:**
   - Simulated extreme weather conditions (e.g., high wind speed or rain) and confirmed that the SMS alerts were sent and the buzzer activated.
   - Ensured that the alerts worked both locally (on the Raspberry Pi) and remotely (via Twilio).
3. **Bug Fixing:**
   - Fixed any bugs related to the API responses, frontend rendering issues, or incorrect data being displayed.
   - Ensured that the weather data was accurate and updated consistently.

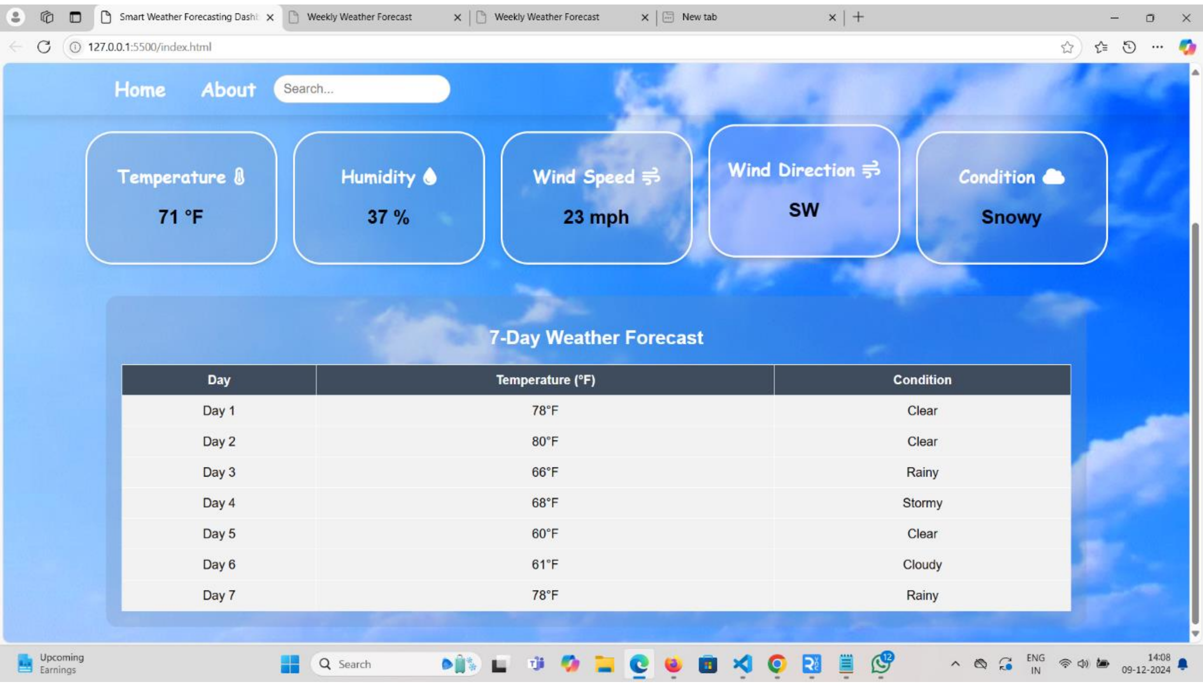
6: Deployment and Finalization

After the system was fully tested, I moved on to deploying it for real-time use.

1. **Deploying the Flask API:**
   - Deployed the Flask app to the Raspberry Pi, making sure it was running as a background service.
   - Configured the Raspberry Pi to automatically start the Flask app on boot.
2. **Making the Web Interface Accessible:**
   - Set up port forwarding on my router to make the Flask app accessible via the local network.
   - Tested the frontend on multiple devices (laptops, tablets, and smartphones) to ensure it was responsive.
3. **Final Adjustments:**
   - Made any final tweaks to improve the user interface and user experience, such as enhancing styling and adding additional weather parameters.
   - Prepared a user guide and documentation to explain how the system works and how to set it up.

Screenshots:





```
 * Serving Flask app "untitledwethertwilio" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
INFO:werkzeug: * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
INFO:werkzeug: * Restarting with stat
WARNING:werkzeug: * Debugger is active!
INFO:werkzeug: * Debugger PIN: 248-540-139
INFO:twilio.http_client:-- BEGIN Twilio API Request --
INFO:twilio.http_client:POST Request: https://api.twilio.com/2010-04-01/Accounts/ACee152dbbee18b8790d57680553729e47/Messages.j
son
INFO:twilio.http_client:Headers:
INFO:twilio.http_client:Content-Type : application/x-www-form-urlencoded
INFO:twilio.http_client:User-Agent : twilio-python/9.3.7 (Linux aarch64) Python/3.9.2
INFO:twilio.http_client:X-Twilio-Client : python-9.3.7
INFO:twilio.http_client:Accept-Charset : utf-8
INFO:twilio.http_client:Accept : application/json
INFO:twilio.http_client:-- END Twilio API Request --
DEBUG:urllib3.connectionpool:Starting new HTTPS connection (1): api.twilio.com:443
DEBUG:urllib3.connectionpool:https://api.twilio.com:443 "POST /2010-04-01/Accounts/ACee152dbbee18b8790d57680553729e47/Messages
.json HTTP/1.1" 201 850
INFO:twilio.http_client:Response Status Code: 201
INFO:twilio.http_client:Response Headers: {'Content-Type': 'application/json;charset=utf-8', 'Content-Length': '850', 'Connect
ion': 'keep-alive', 'Date': 'Mon, 09 Dec 2024 08:36:38 GMT', 'X-Powered-By': 'AT-5000', 'Twilio-Concurrent-Requests': '1', 'Tw
ilio-Request-Id': 'RQ649e6cdc7895c685f07274f55893ce9d', 'Twilio-Request-Duration': '0.141', 'X-Home-Region': 'us1', 'X-Api-Dom
ain': 'api.twilio.com', 'Strict-Transport-Security': 'max-age=31536000', 'Access-Control-Allow-Origin': '*', 'Access-Control-A
llow-Headers': 'Accept, Authorization, Content-Type, If-Match, If-Modified-Since, If-None-Match, If-Unmodified-Since, Idempote
ncy-Key, X-Pre-Auth-Context, X-Target-Region', 'Access-Control-Allow-Methods': 'GET, POST, PATCH, PUT, DELETE, OPTIONS', 'Acce
ss-Control-Expose-Headers': 'ETag, Twilio-Request-Id', 'Access-Control-Allow-Credentials': 'true', 'X-Shenanigans': 'none', 'X
-Cache': 'Miss from cloudfront', 'Via': '1.1 eeb60fee72923d35b96c344ca988f3aa.cloudfront.net (CloudFront)', 'X-Amz-Cf-Pop': 'M
AA51-P3', 'X-Amz-Cf-Id': 'Jb6Vs0a2S_zvzryzGq0WEswObgcGH_kZlCGnbo8ku5TCxKEF2OlVvQ=='}
INFO:root:Sent message: SM649e6cdc7895c685f07274f55893ce9d
INFO:root:Buzzer ON
INFO:root:Buzzer OFF
DEBUG:root:Weather Data: {'temperature': 71, 'humidity': 35, 'wind_speed': 7, 'wind_direction': 'W', 'wind_degree': 128, 'weat
her_description': 'Windy'}
INFO:werkzeug:192.168.137.67 - - [09/Dec/2024 14:06:42] "GET /weather-data HTTP/1.1" 200 -
INFO:twilio.http_client:-- BEGIN Twilio API Request --
INFO:twilio.http_client:POST Request: https://api.twilio.com/2010-04-01/Accounts/ACee152dbbee18b8790d57680553729e47/Messages.j
son
```