NOVOGRADAC

NovoWeatherApp Documentation

Francis Tran

January 30, 2025

1 Introduction

This implementation of a weather app is built using Django and allows users to enter a U.S. ZIP code to retrieve real-time weather data. Weather data is fetched from OpenWeatherMap's Current Weather API and stored in a SQLite database. The system uses caching to avoid redundant API calls and provides a user-friendly web interface.

2 Key Features

2.1 User Input Handling

When designing my application, error prevention was a high priority. A good way to mitigate errors is by making sure users do not do anything they're not supposed to. I ensured this through form validation to ensure that users can only input valid ZIP codes. This means that any ZIP code that is too short, too long, isn't numeric, or doesn't exist in the U.S. will not be saved in the database. Error messages appear dynamically below the input field, guiding users to enter a valid ZIP code. If the OpenWeatherMap API is unreachable or returns an error, the application notifies the user instead of crashing.

2.2 API Integration

Accurate and current weather data is carefully extracted from OpenWeatherMap API responses. Users can be assured that the weather information they are receiving is reliable, while proper implementation makes certain that no unnecessary and potentially costly API calls are made.

2.3 Simple Web Page

I have integrated Bulma CSS for clean, elegant styling on the web page. Users can easily input their desired ZIP code and see their searched locations on the same page. The template has been designed with simplicity in mind, guaranteeing that users can do what they need to quickly, efficiently, and without being distracted by clutter.

2.4 Caching

I have implemented a simple caching approach to reduce extraneous API calls and improve efficiency in the application. Data is stored for one day, after which a new API call is required. If a user enters a ZIP code that has already been cached for the day, the app retrieves it from the database instead of making an API call.

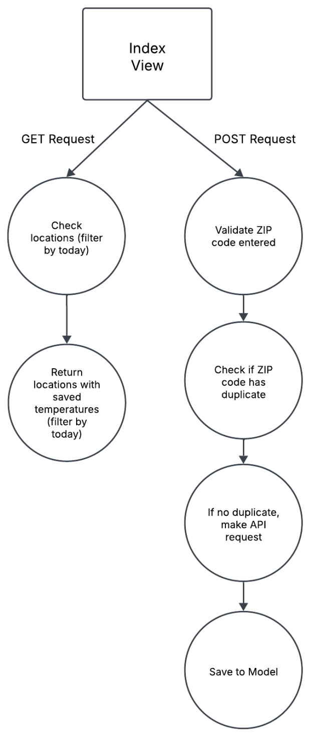
Index View
GET Request
POST Request
Check locations (filter by today)
Return locations with saved temperatures (filter by today)
Validate ZIP code entered
Check if ZIP code has duplicate
If no duplicate, make API request
Save to Model

Figure 1: Caching Diagram

3 Project Structure

3.1 Model-View-Template Architecture

Django's Model-View-Template architecture for my project seamlessly fits into this use case. The Model is used to store weather data, such as location name, ZIP code, high and low temperature, and an icon taken from API Responses handled by the View. The View then requests this weather data from the model and passes it to the Template where it is then displayed on the web page. The Template passes users' ZIP Code inputs to the View where it makes API requests to get weather data.

3.2 Design Rationale

I chose Django and its Model-View-Template (MVT) structure for its seamless integration with the OpenWeatherMap API, built-in ORM, admin panel, and form handling, making it efficient for managing weather data. While I considered React for the frontend, Django's templating was sufficient for this simple application,

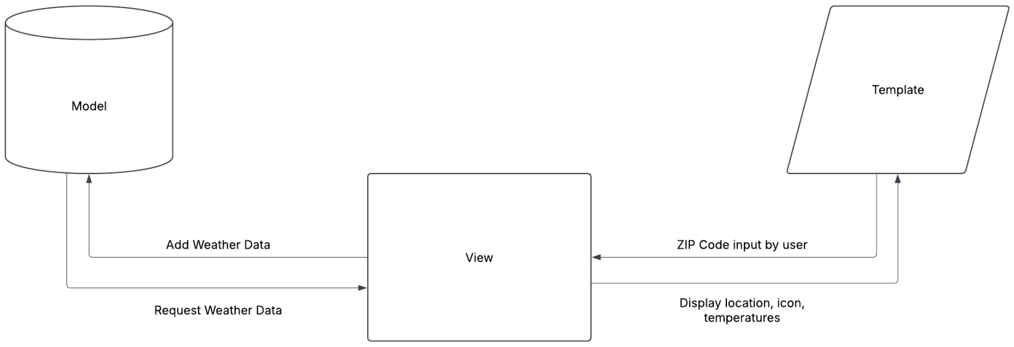

Figure 2: MVT Diagram

reducing complexity. SQLite was chosen for its lightweight yet scalable nature, allowing future expansion. Though Django lacks the dynamic interactivity of a React-based application, its robust backend capabilities and quick implementation made it the best choice for this project.

4 Potential Enhancements

Without the constraints of the limited scope and time-frame of this project, there are some useful and interesting features I would like to add in the future.

4.1 International Locations

In its current implementation, the web-app is only to take and return weather information for locations in the United States. An expansion of this feature to allow for global locations across all countries would be a useful addition that would vastly expand the potential audience of this application. With an increased audience also come the challenges associated with scalability, such as increased API calls and more database storage which are issues I expect to overcome.

4.2 User Profiles

With the addition of different user profiles, users can store and potentially customize their dashboard as they see fit. For example, a user could add their own location, and with the implementation of international locations as well, add the locations of their friends and family members across the globe. A different user who travels often could add the locations of their most frequent destinations. This allows for more usability and customization for any client. User authentication could be handled with Django's built-in authentication system.

4.3 Additional APIs

Integration with APIs, such as OpenAI's GPT API could enable natural language processing and speech recognition, increasing accessibility for users who may not be able to type. This could also allow for users to be able to search using not only ZIP codes, but with city names in different languages.