Personal Development Journey

Day 0: Research & Planning

When I got the task to build a weather application, my first instinct was to do some reconnaissance. I downloaded several popular weather apps to my phone—the default iOS weather app, AccuWeather, and a few others—just to understand how they work, what the UI flow looks like, and how users navigate through them. It's always helpful to see real-world examples before diving into your own implementation.

Next came the technical setup: configuring the Flutter project, organizing the folder structure, and getting the Android emulator running on my machine. Since I already had some experience with Provider architecture in Flutter, I quickly refreshed my knowledge by reviewing the official Flutter documentation.

The biggest challenge was finding the right API. I spent a good amount of time researching free weather APIs. After consulting with an AI assistant (thanks, Copilot!), OpenWeatherMap was recommended as the best option. However, their free tier had some endpoints behind a paywall. The turning point came when I discovered they offer a Student program with additional endpoints unlocked. Using my student privilege, I set up a student account and got an API key.

I created the first middleware—essentially a repository layer to manage communication between my frontend and their API. I wrote a few test methods, started with simple debugging by printing the URL, and once I confirmed the API was working, I copied the JSON response and used an online JSON-to-Dart converter to generate my data models: `GeoData` and `WeatherData`.

My first test was searching for "Sarajevo." The API returned Sarajevo's coordinates perfectly. As a fun note—Mostar always shows up as warmer than Sarajevo, which is historically accurate!

Day 1: Core Functionality & UI Development

With the API proven to work, I shifted focus to creating proper data parsers. For each API response, I implemented `toString()` overrides in my models so I could easily inspect the parsed data in debug logs.

Once the parsing was solid, I tested it end-to-end: typed "Sarajevo" into the app and watched the temperature appear on the home screen. That moment of seeing real API data displayed in the app was incredibly satisfying.

Now came the creative part—designing the UI. I spent time researching existing weather apps and design inspiration online. I'd say 50% of the design came from real-world apps, and the other 50% from my own vision of what a minimalist, clean weather app should look like. I wanted the temperature to be the star of the show, with a small, elegant animation showing current conditions outside.

The animation challenge was significant. Should I use static icons or smooth animations? I decided to invest in quality—I found Lottie animations from LottieFiles and selected beautiful 3D weather animations. Working with Copilot, I mapped out which weather conditions needed which animations:

- Thunderstorms → thunder.json
- Snow → snow.json
- Rain → rain.json
- Mist/Fog → mist.json
- Clouds → cloudy_day.json

- Clear sky → clear_day.json
- Loading state → loading_weather.json

For the user flow, I envisioned a permission screen asking whether the user allows location access. If they decline, they can search for their city manually. Once location is granted or a city is selected, the app shows the home screen with current weather and detailed information.

I built the permission screen and home screen UI using a modern design system: gradient backgrounds flowing from sky-blue (#38B6FF) through mid-blue (#1565C0) down to deep navy (#060C22), paired with glassmorphism effects and Google Fonts (Poppins for headers, Roboto for body text).

Then came the logic implementation. This required updating Android and iOS configuration files to request location permissions. Interestingly, I initially had a permission issue—even after the user granted permission, I couldn't access the API because I hadn't properly handled the permission state. Copilot caught this mistake, and once fixed, location data flowed perfectly.

For city search, I decided to build a search bar at the top that opens a dialog when tapped. The dialog shows available cities. I found a JSON file containing thousands of world cities and implemented smart search with debouncing to avoid memory issues. The autocomplete works smoothly, suggesting cities as the user types.

I also conceptualized a favorites system where users could bookmark cities and access them instantly from a dedicated section. While I didn't implement this fully yet, it's on my roadmap.

Spent much of the remaining time debugging issues and refining the experience.

Day 2: Polish, Testing & Documentation

Day 2 was all about quality assurance. I tested the app extensively—on Chrome (web), Android emulator, and my personal Android device. Everything worked beautifully.

For the final UI touches, I implemented a draggable bottom panel that slides up to show detailed forecast information. I also added a pull-to-refresh gesture so users can manually update the weather data.

The rest of the day (until 5 PM) was dedicated to refactoring the home screen code for readability, writing comprehensive project documentation, and testing edge cases to ensure the app handled errors gracefully.

Final touches implemented:

- Draggable panel for detailed forecast access
- Pull-to-refresh for manual data refresh
- Error boundary with user-friendly messages
- Responsive design across all screen sizes
- Proper animation lifecycle management
- State management with clear data flow
