Design Document for .NET MAUI Desktop App - Weather and Public Transport Integration

Table of Contents

1. Introduction

This design document outlines the structure and components of a .NET MAUI desktop application that integrates data from three different REST APIs: positionstack, weather forecast and public transport.he primary goal of the application is to provide users with weather information when arriving at their destinations, allowing them to find public transport routes from place A to B while considering configurable weather data.

2. Application overview

The application will consist of the following major features:

- Will convert street addresses to latitude and longitude coordinates.
- Weather information display for the arrival location. This gives information how you should prepare for the trip.
- Public transport route planning between two user-defined locations.
- --Configurable weather data to influence route selection.--
- User-friendly graphical user interface (GUI).

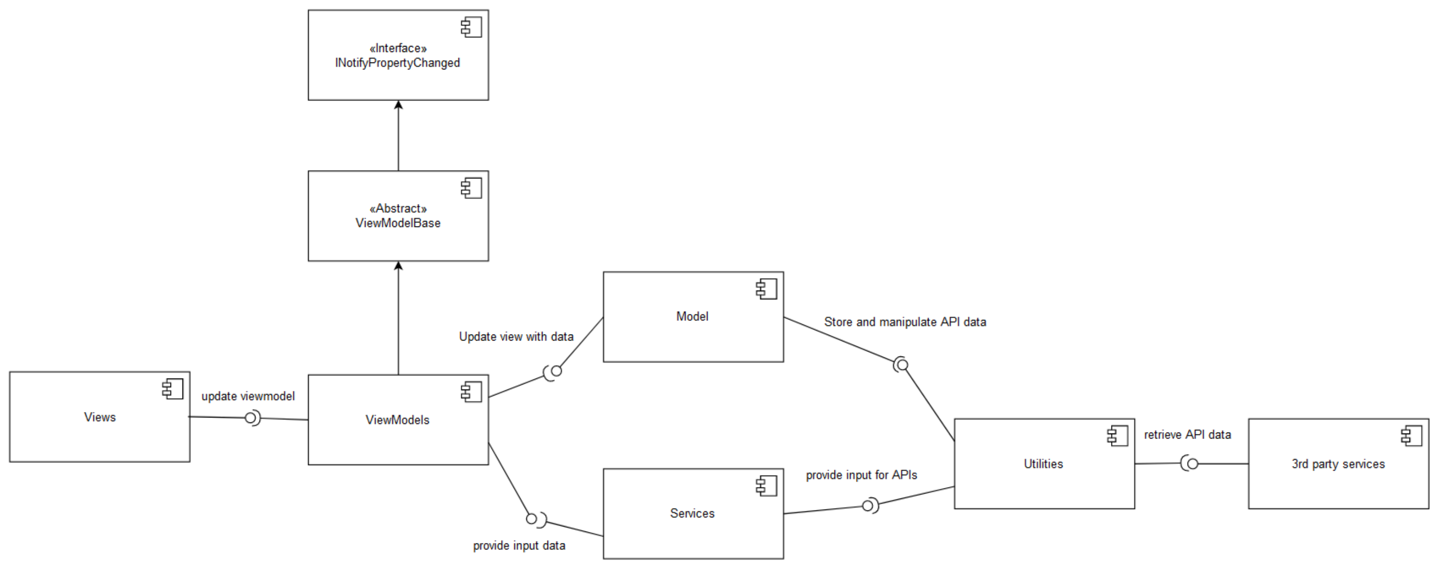
3. System Architecture

The proposed system architecture follows a client-server model, where the client application communicates third party weather and public transport APIs to retrieve data. The key architectural components include:

- Client Application: The .NET MAUI desktop application responsible for presenting the user interface and handling user interactions.
- API Integration Layer: This layer manages communication with the external REST APIs. It includes modules to retrieve weather forecasts and public transport route information.
- Data Processing Layer: Responsible for processing and formatting data received from APIs for presentation in the user interface.
- User Interface Layer: The graphical user interface, which displays weather information and allows users to plan public transport routes. It interacts with the API integration layer and the data processing layer.
- There is no other backend on this project. All data is fetched from third party APIs by the client application.

.

4. Key Components and Interfaces



Views

The visible part of the application. User interacts with view to use the functionalities of the application. Its responsibility is to change whenever the state in viewmodel changes.

ViewModel

Connects model to view. Its responsibility is to bind data and provide said data from model to view.

Model

Used to store data during applications use. Its responsibility is to handle data storage and manipulation of said data.

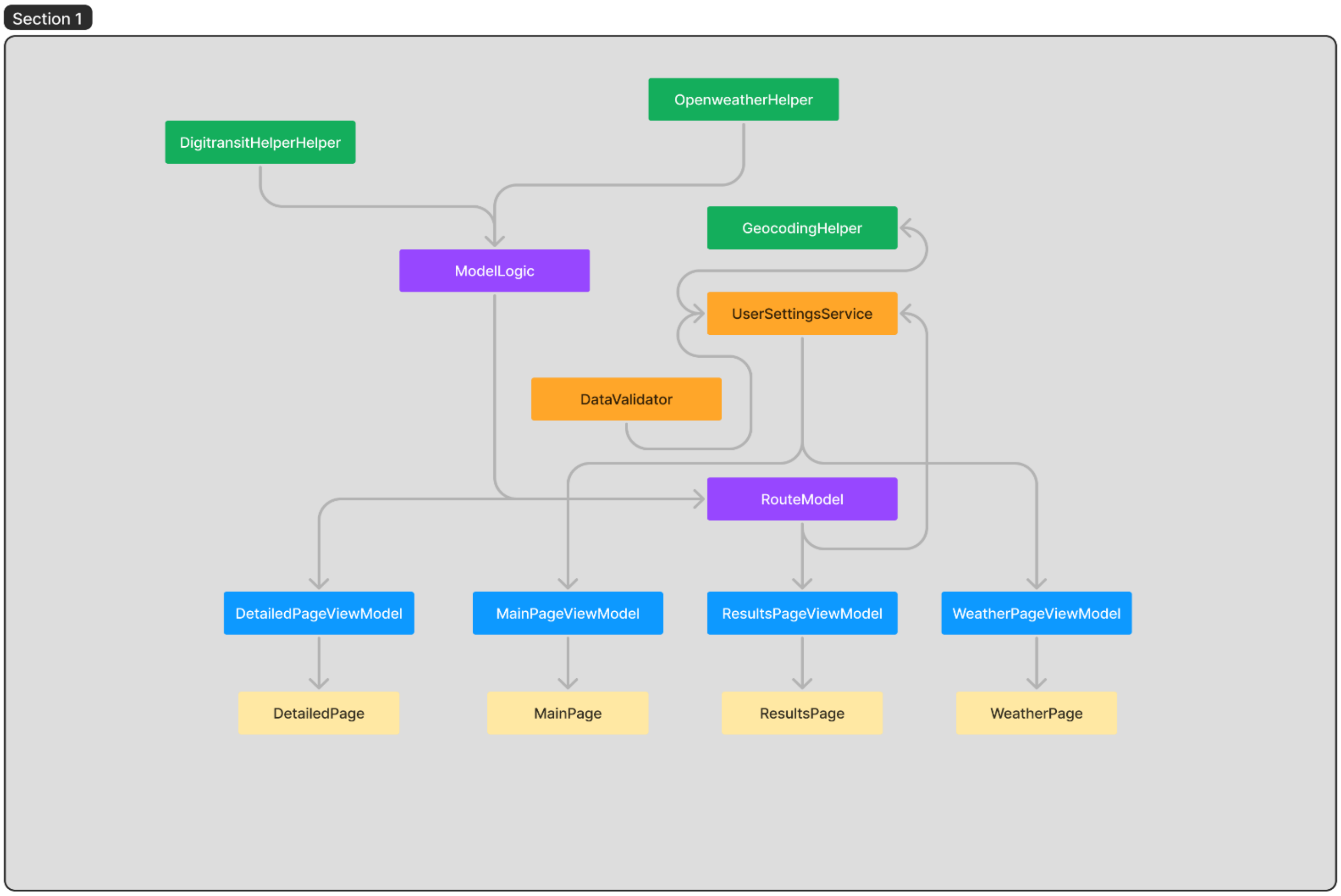
Services

Handles temporary storage of user settings given by user. Its responsibility is to only store user input and provide this input to utilities.

Utilities

Handles communication with third party APIs. Its responsibility is to take required data from services to make successful API calls and then forward that data to model.

Boundaries and interfaces (internal)



5. API Integration

We plan to use the following APIs:

OpenWeatherMap: We will be using OpenWeatherMap for weather forecasting. It's an easy to use REST-API and suits well for our project's needs. It gives accurate enough data for our use case.

Digitransit API: We will be using digitransit API for fetching routes from place A to B inside Finland. This API offers detailed view of different routes.

Positionstack API: We will be using Positionstack API to convert street addresses to latitude and longitude coordinates. This is used to fetch data from Openweathermap and Digitransit APIs.

6. User Interface

The user interface will be designed to be intuitive and user-friendly. We will employ modern design principles for a responsive and visually appealing desktop application. Visual components will include:

- From-to screen which contains autocompleted input fields and submit button along with modern visualization.
- Travel settings which contain sliders for user input. User can change different variables of the weather configuration.
- Result view which contains search results of the trip. Result view is scrollable list which contains results of the search. Most matching results are shown at the top.
- Detailed result view containing visual presentation of the data, including graphs.

7. Configuration and Customization

Users will be able to customize their weather-related preferences, such as temperature ranges, precipitation thresholds, and preferred weather conditions for route planning. These configurations will influence the suggested routes. User will also have possibility to save and load previous searches.

8. Design Patterns and Approaches

Technology stack:

Choice of C# and .NET MAUI: C# is a versatile and widely-used language, and .NET MAUI provides a cross-platform framework for building native applications. This choice allows for code reusability across multiple platforms.

SOLID principles:

Single Responsibility Principle (SRP): Each class or module **is** designed to have a single responsibility, promoting a more modular and maintainable codebase.

Open/Closed Principle (OCP): **Designed** components to be easily extensible without modifying existing code.

Dependency Inversion Principle (DIP): Dependency injection **is** used to invert the dependencies, making high-level modules independent of low-level modules.

Responsibility Division:

Separation of Concerns: The application **is** structured to separate concerns such as UI, business logic, and data access. This helps in modularizing the codebase and makes it easier to understand, maintain, and extend.

Service Layer**: Our application has separate service layer** to handle business logic separately from the UI. This enhances maintainability.

Other design decisions:

MVVM (Model-View-ViewModel): We utilize MVVM architecture to separate the presentation logic from XAML UI. This is the main purpose of using MVVM, since the application doesn't have a backend, no model approach is required.

Dependency Injection: Implement dependency injection to promote loose coupling between components and facilitate unit testing.

RESTful API Consumption: Adopt RESTful principles for communicating with external APIs, ensuring a standardized and scalable approach.

9. Self-Evaluation

Regarding functionalities of our program, we were able to stick mostly to our original design. Somewhere down the road we realized we need a third API to provide us with the necessary data for the rest of the functionalities to work (other APIs relied on geocoding information). We were able to add some functionalities that weren't in our original design, such as saving the last search to the address input fields when you close and reopen the program. We also added a bit more details to the route details page.

10. Usage of Artificial intelligence

10.1 ChatGPT for Ideation and Planning

ChatGPT, powered by OpenAI's GPT-3.5 architecture, plays a pivotal role in shaping the conceptualization and planning of our project. Here's how we leverage ChatGPT:

- Idea Generation: We initially used ChatGPT to brainstorm project ideas. It assisted us in exploring various concepts, provided insights, and helped refine our initial vision.
- Design Assistance: ChatGPT has been instrumental in drafting documents, including this design document. It assists in structuring content, suggesting relevant sections, and refining the language for better clarity.
- Problem Solving: Throughout the project, ChatGPT serves as a valuable resource for troubleshooting and problem-solving. It aids in understanding and resolving technical challenges.
- Prototyping Guidance: ChatGPT assists in creating rough prototypes and mockups, allowing us to visualize and iterate on our ideas more effectively.

10.2 GitHub Copilot for Efficient Coding

GitHub Copilot is a code completion tool that utilizes AI to assist developers in writing code more efficiently. We employ GitHub Copilot in our coding phase for various tasks:

- Code Generation: Copilot generates code snippets, functions, and even entire classes based on the context of our coding project. This accelerates development by automating repetitive coding tasks.

- Code Suggestions: Copilot provides real-time code suggestions as we type, enhancing code quality and adherence to best practices. It offers insights into coding patterns and standards.
- Documentation Assistance: GitHub Copilot aids in creating code documentation, making it easier for team members to understand and maintain the codebase.
- Error Handling: It assists in identifying and rectifying errors, reducing debugging time and ensuring a more stable codebase.

11. Conclusion

This design document outlines the key components, interfaces, and architectural considerations for the development of a .NET MAUI desktop application that integrates weather forecast and public transport information. As development progresses, we will refine these designs and adapt to emerging requirements and best practices in software design.