Project Documentation
Weather and Air Quality Application
AeroSky

Version 1.0 – Midterm submission 27.10.2024

Group Kirsi:

<Name> <Student number>

Table of Contents

1. Introduction

The purpose of this document is to outline the high-level design for the AeroSky application – Weather and Air Quality application. This project is part of the course COMP.SE.110 Software Design.

Desktop application provides users with real-time weather conditions along with air quality data for a selected location. The data is visualized through interactive charts, maps, and alerts, giving users a clear overview of current outdoor conditions, helping them make informed decisions for their activities, health, and safety.

AeroSky is designed to provide real-time weather updates and forecasts using the Spring framework and Lombok. The application is built with modern Java technologies to ensure scalability, maintainability, and a rich user experience via a JavaFX UI.

The application architecture follows the **Model-View-Controller (MVC)** and **modular and layered** approach, which separates concerns across data handling (model), user interface (view), and user interaction (controller). It uses Java libraries for HTTP requests and JSON handling, with JavaFX providing the interactive, graphical UI.

2. Key Features

The app takes city name as input and displays:

- Current view: current data of weather and air quality of the input city.
- Data view: near weekly weather forecast and visualization of weather and air quality data daily, weekly or monthly. Displayed parameters can be adjusted dynamically with buttons
- Favorites: lists of saved cities as cards. Users can save their frequently checked cities and quickly see current temperature and weather situation of cities.

Besides, the app shows notifications for high AQI levels or severe weather conditions. It also suggests activities based on the current weather and air quality.

3. Technology

Java is used for backend, and JavaFXML is used for frontend user interface. Two APIs are OpenWeathermapAPI and AirVisualAPI. Many design patterns are used for improving the code structure.

APIs

- OpenWeather API: Open API for fetching weather current and forecast data. Input is the name of city. The API version used in this project is 2.5. Documentation: https://openweathermap.org/api/one-call-api

- AirVisual API: Open API for fetching air quality data. Inputs are latitude and longitude of the city, which are fetched from OpenWeather API. The version used in this project is 2.0. Documentation: https://api-docs.iqair.com/#275daeb0-d34d-408a-bc03-f74d097a4eae

Dependencies and Relationships

- JavaFX: Provides the graphical user interface.
- FxWeaver: Integrates JavaFX with Spring Framework
- Spring Frameworks: Backend
- Lombok: Reduces boilerplate code in Java classes.
- Java SDK 17: Latest Java features and performance improvements.

4. High-Level Description

Components and Modules

The application is structured following a modular and layered architecture approach.

Dividing code files into smaller sub-parts and layers by separating and isolating them from each other makes the project more manageable as a whole.

Table 1. Components and modules of project.

Layer	**Component**	**Purppose**	**Responsibilities**
Main Application (Entry point)	Weather System Application	Acts as the entry point for the Spring Boot application and JavaFX application.	• Boots the Spring application context. • Launches the JavaFX application.
Presentation (View and Controllers)	WeatherApp	Initializes and manages the JavaFX UI	• Set up the primary stage (window) for the JavaFX application. • Load the initial UI layout. • Handle user interactions and GUI updates.
	WebClient Config	Spring configuration class responsible for creating and configuring a WebClient bean	• Configuration Class • Creating and configuring WebClient

	Weather Display Support	Assist with updating JavaFX UI components with weather information. It acts as a helper or utility class.	• Updating UI Components with Weather Data • Data Formatting • Encapsulation
	MainView Controller	Manages the primary user interface	• Initialize Method: Prepares the controller with initial UI setup. • Event Handlers: Methods to change the displayed layout when buttons are clicked.
	Settings Controller	Manages the settings UI	• Handle user input and preferences. • Save and retrieve user settings. • Interact with services to update settings in the application.
	Home Controller	Manages the home view of the application, focusing on user interaction for initiating searches.	• @FXML Fields: Binds JavaFX UI components from the FXML file to this controller. • initialize Method: Sets up initial state of the search input field. • search Method: Handles the search button click event, storing search input and changing the layout to show results.
	Data Controller	Manages the data-centric view, controlling elements like the line chart and several forecast labels and icons.	• @FXML Fields: Binds JavaFX UI components from the FXML file to this controller. • initialize Method: Sets up the line chart

			with initial temperature data.
	DataTransfer Controller	Ensures that there is only one instance of itself (Singleton pattern) across the entire application. This is useful for managing shared UI components as well as maintaining the application's state	• Ensures Single Instance • Lazy Initialization • Manage UI Components • Load and Switch Layouts • State Management
	Forecast Controller	JavaFX controller responsible for handling the forecast layout of the weather application and updating the UI with weather data.	• Initialization • Get Data Input • Fetch Weather and Air Quality Data • Update UI • Set City Name • Add to Favorites
Business logic (Services) (Model)	Weather Service	Implement core business logic related to weather data	• Fetch current weather data from external APIs or databases. • Process weather data to meet the application's requirements. • Provide weather data to controllers as needed.
	AirQuality Service	Define methods for obtaining air quality information based on geographical coordinates (longitude and latitude).	•
	Forecast Service	provides an abstraction for fetching weather forecast information based on a given location	• Fetches weather forecast information for a specified location.
	AirApiCall	To encapsulate the logic required to make an external API call to fetch weather data based on	• External API Interaction: Facilitate

	ApiTemplate	location, allowing for easier and more reusable integration of weather data into the application.	interactions with an external weather API. • Configuration Management: Handle configuration values such as API URL and access token. • Data Retrieval: Retrieve weather information based on specified parameters (like location). • Template Specialization: Provide specific endpoint details and response type for the weather API by extending a generic API template class.
	ForecastApiCall		
	WeatherApiCall		
	WeathetAndAirQualityFacade	Defines a contract that any implementing class must follow and abstraction.	• Decoupling • Simplified Interface
	WeathetAndAirQualityFacadeImpl	Provides a simpler, unified interface to a set of interfaces in a subsystem. It makes the subsystem easier to use for the clients by hiding its complexity.	• Simplifies client interaction • Decoupling clients from subsystems • Reduces dependencies
Data transfer	AirDataDto	Data transfer Objects are used within the application to transfer data between different layers such as controllers, services, and views	• Encapsulation • Data Transfer • Abstracts and decouples • Simplification • Validation and Formatting • Network Efficiency
	AirForecastDto		
	AirQualityInfoDto		
	CurrentDataDto		
	HistoryDto		
	LocationDto		

	PollutantDto		
	PollutionDto		
	UnitsDto		
	WeatherDto		
	CloudDto		
	CurrentWeatherDto		
	ForecastDto		
	ForecastInfoDto		
	HumidityDto		
	LocationDto		
	MainDto		
	SysDto		
	TemperatureDto		
	WeatherDto		
	WeatherFullDto		
	WeatherInfoDto		
	WindDto		

Separate attachment AeroskyUML.pdf shows full class diagram how classes communicate.

5. Design Patterns and Principles

To ensure the application is modular, reusable and maintainable, several design patterns and principles were implemented.

Singleton pattern: ensures that there's only one instance of the DataTransferController class across the application. It is useful to manage shared resources such as UI components, states. Lazy initialization is used – the instance is created only when it is needed. In other words, the instance is not created when the application starts but only when getInstance() is called. [1]

Façade pattern: provides simplified interface for complex subsystems. This pattern is used so that clients do not need to understand the details and relationships of services and to use them. It decouples the client from the client from the subsystems. Clients can access services via façade interface. [2]

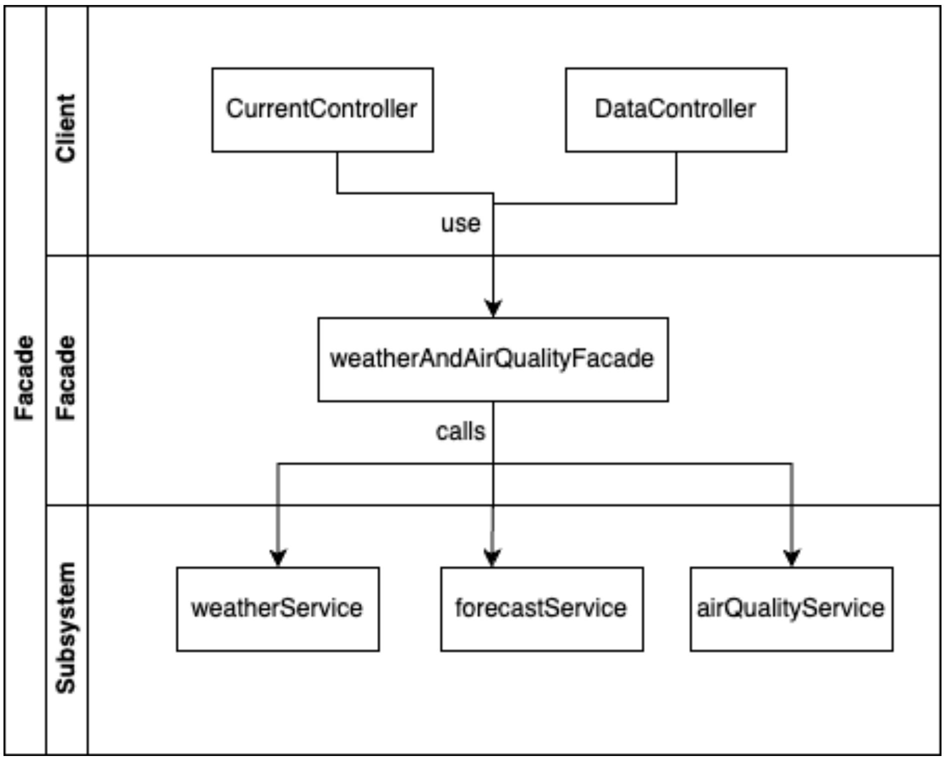


Figure 1. Façade design pattern applied in this project.

Single Responsibility Principle (SRP): Each class is designed with a single responsibility, so it is easier to test and maintain. By adhering to SRP, the project is highly modular. For example, the WeatherServiceImpl class is responsible only for retrieving weather data and handling fallbacks when the requested location is not available. It does not handle UI, database logic or other concerns. [3]

Dependency Injection (DI): DI is applied to manage dependencies and enhance flexibility. The dependency, API clients for weather and air quality services, are injected into the WeatherServiceImpl class using constructor-based dependency injection, allowing them to be easily mocked for testing purposes. The code is also easier to scale because components can be modified without changing dependent code. [4]

Open/Closed Principle: A class should be open for extension but closed for modification. In other words, class should be designed so that it allows new functionality to be added without modifying the existing code. It helps prevent bugs from ruining stable code. In this project, interfaces are created for services, outlining methods for fetching data. Any new API provider has its own interface. Each API also has its own implementation of the interface. These approaches allow service classes to remain unchanged while being easily extended with new functionality. [5]

Template Design Pattern: a behavioral design pattern used to define the structure of an algorithm in a superclass while allowing subclasses to override specific steps of the algorithm without changing its overall structure. In this project, Template Design Pattern is used for common API call structure. ApiTemplate is an abstract class, providing template for making API calls. It handles common logic such as error handling and URI construction, allowing subclasses to implement specific API call details. Subclasses must implement details for their specific API calls. [6]

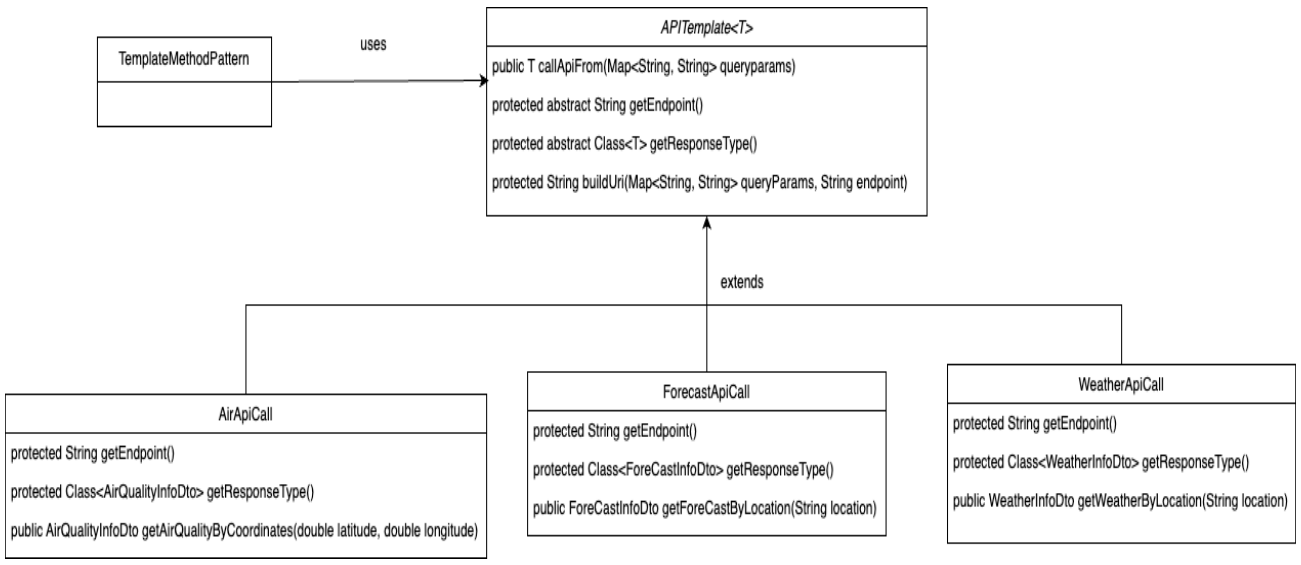

TemplateMethodPattern
uses
APITemplate<T>
public T callApiFrom(Map<String, String> queryparams)
protected abstract String getEndpoint()
protected abstract Class<T> getResponseType()
protected String buildUri(Map<String, String> queryParams, String endpoint)
extends
AirApiCall
protected String getEndpoint()
protected Class<AirQualityInfoDto> getResponseType()
public AirQualityInfoDto getAirQualityByCoordinates(double latitude, double longitude)
ForecastApiCall
protected String getEndpoint()
protected Class<ForeCastInfoDto> getResponseType()
public ForeCastInfoDto getForeCastByLocation(String location)
WeatherApiCall
protected String getEndpoint()
protected Class<WeatherInfoDto> getResponseType()
public WeatherInfoDto getWeatherByLocation(String location)

Figure 2. Template method used in this project.

Model-View-Control (MVC): Separates the application's data handling (Model), user interface (View), and control flow (Controller) to make it easier to manage and extend each part independently. [7]

6. System Architecture (MVC Pattern)

Model-View-Controller pattern was chosen for its simplicity and effectiveness both for design/architectural aspects and dividing implementation work with the team.

Model (Data Layer)

- This layer will handle the API calls and data parsing from the APIs.
- The model will have classes for API calls and data handling.

View (User Interface Layer)

- The **JavaFXML** UI will visualize the data fetched by the model.
- Views will include:
  - A home view that shows a search text box
  - A current view that displays current weather and air quality data with city name, current time.
  - A data view that displays weather and air quality data with interactive graphs.
  - A favorite view that shows list of saved cities for quick access
  - Notifications and alerts based on health and activity recommendations.

Controller (Interaction Layer)

- The controller will handle user interactions and manage communication between the view and model.
- For example, when the user enters a location or clicks the "Search" button, the controller will fetch data from the model and update the view with new visualizations.

7. Design Decisions and Justifications

The following decisions were made to enhance usability, performance, and future extensibility:

1. Project design started as an idea of a simple **Model-View-Controller** architecture has grown also to be **modular and layered**. This ensures separation of concerns, scalability and maintainability. Both patterns help to maintain modularity by isolating each functional area (data, view, control), simplifying future modifications and testing. These patterns also help in dividing work among the project team.
2. **JavaFX for UI**: JavaFX provides native support for Java desktop applications and is ideal for creating interactive, visually appealing user interfaces. Project scope was to make a desktop application that has Java backend, so JavaFX seemed natural approach for Desktop UI.
3. **Spring Framework**: Provides dependency injection, streamlined development, and powerful features.
4. **APIs Selection (OpenWeather and AirVisual)**: Both APIs provide reliable, free, and accessible data without requiring API keys, ensuring ease of integration and development.

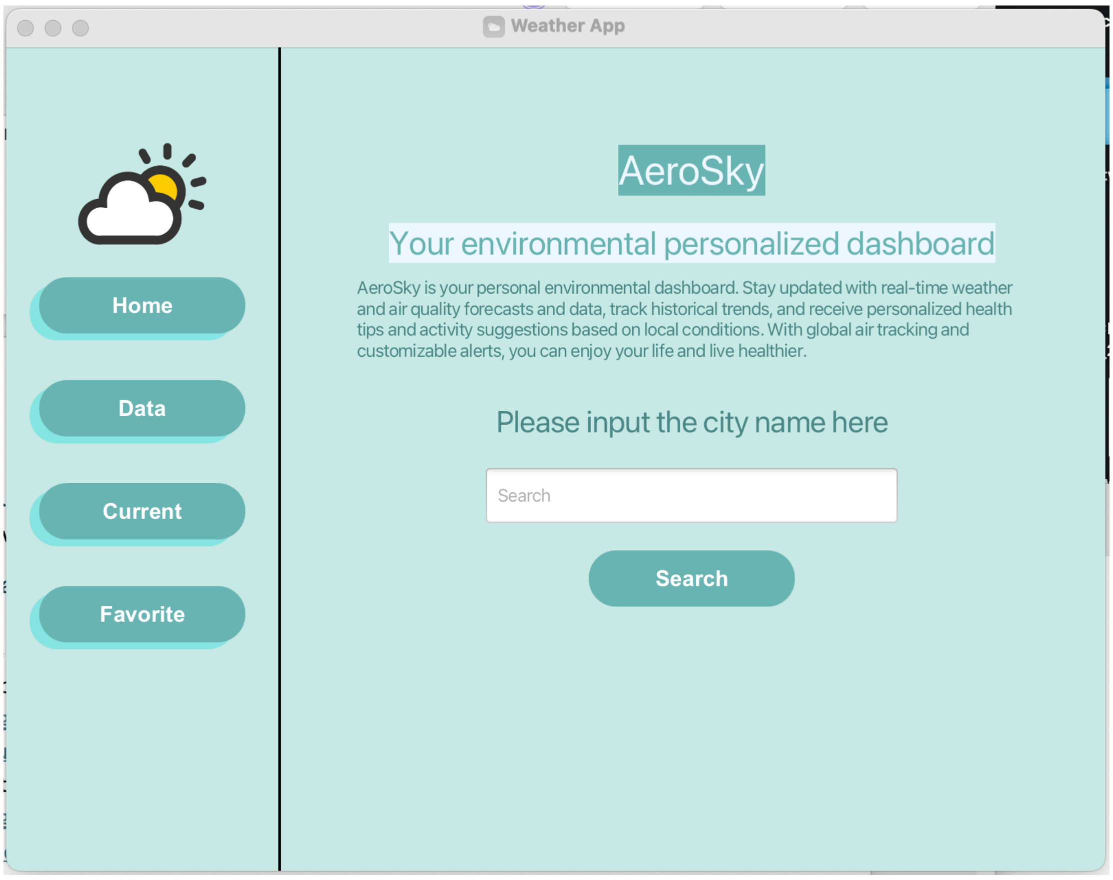
8. Graphical User Interface (GUI)

In this project, we use JavaFXML to build GUI in desktop.

Prototyping phase

The interactive prototype in Figma shows how our application should work. However, there are some modifications during implementation from the initial Figma prototype.
Prototype can be accessed via this Figma link: https://www.figma.com/design/vNLrs1o29MTktmaIWPGHQF/Weather?node-id=0-1&t=fV5GU5NuJTSApl81-1
Interactive prototype: https://www.figma.com/proto/vNLrs1o29MTktmaIWPGHQF/Weather?node-id=14-228&starting-point-node-id=1%3A2&t=FutlsaCYfsm3SZ6f-1

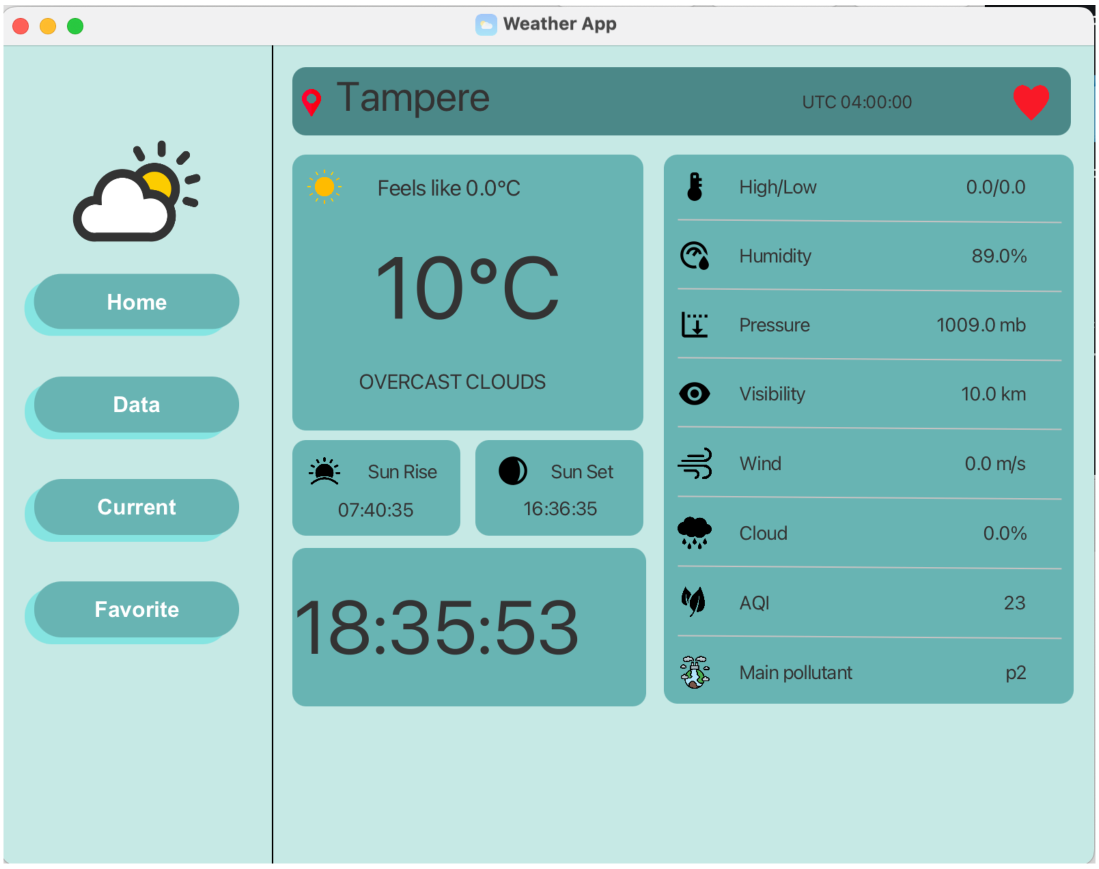
Current views of the app


Weather App
Home
Data
Current
Favorite
AeroSky
Your environmental personalized dashboard

AeroSky is your personal environmental dashboard. Stay updated with real-time weather and air quality forecasts and data, track historical trends, and receive personalized health tips and activity suggestions based on local conditions. With global air tracking and customizable alerts, you can enjoy your life and live healthier.

Please input the city name here
Search
Search

Figure 3. Home view


Weather App

Home

Data

Current

Favorite

Tampere UTC 04:00:00

Feels like 0.0°C

10°C

OVERCAST CLOUDS

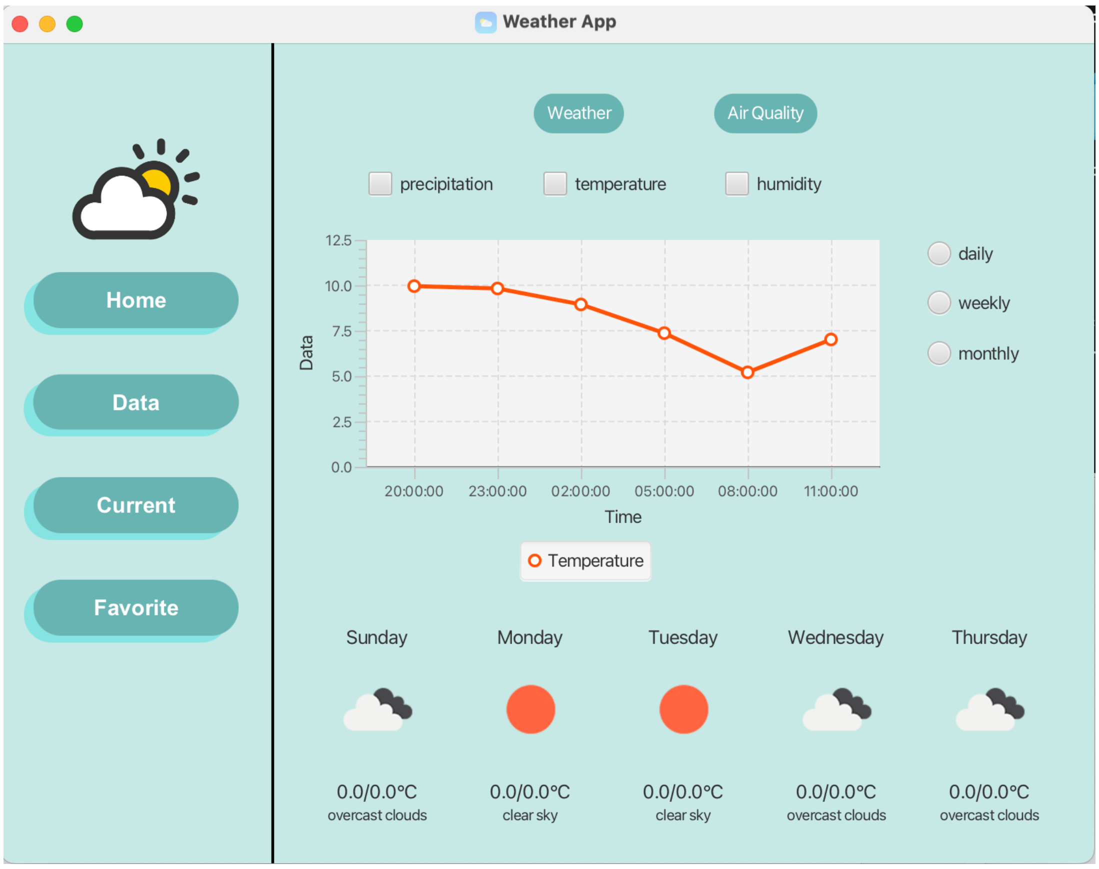
Sun Rise 07:40:35

Sun Set 16:36:35

18:35:53

High/Low	0.0/0.0
Humidity	89.0%
Pressure	1009.0 mb
Visibility	10.0 km
Wind	0.0 m/s
Cloud	0.0%
AQI	23
Main pollutant	p2

Figure 4. Current view


Weather App

Home

Data

Current

Favorite

Weather

Air Quality

precipitation

temperature

humidity

daily

weekly

monthly

Sunday	Monday	Tuesday	Wednesday	Thursday
0.0/0.0°C	0.0/0.0°C	0.0/0.0°C	0.0/0.0°C	0.0/0.0°C
overcast clouds	clear sky	clear sky	overcast clouds	overcast clouds

Figure 5. Data view

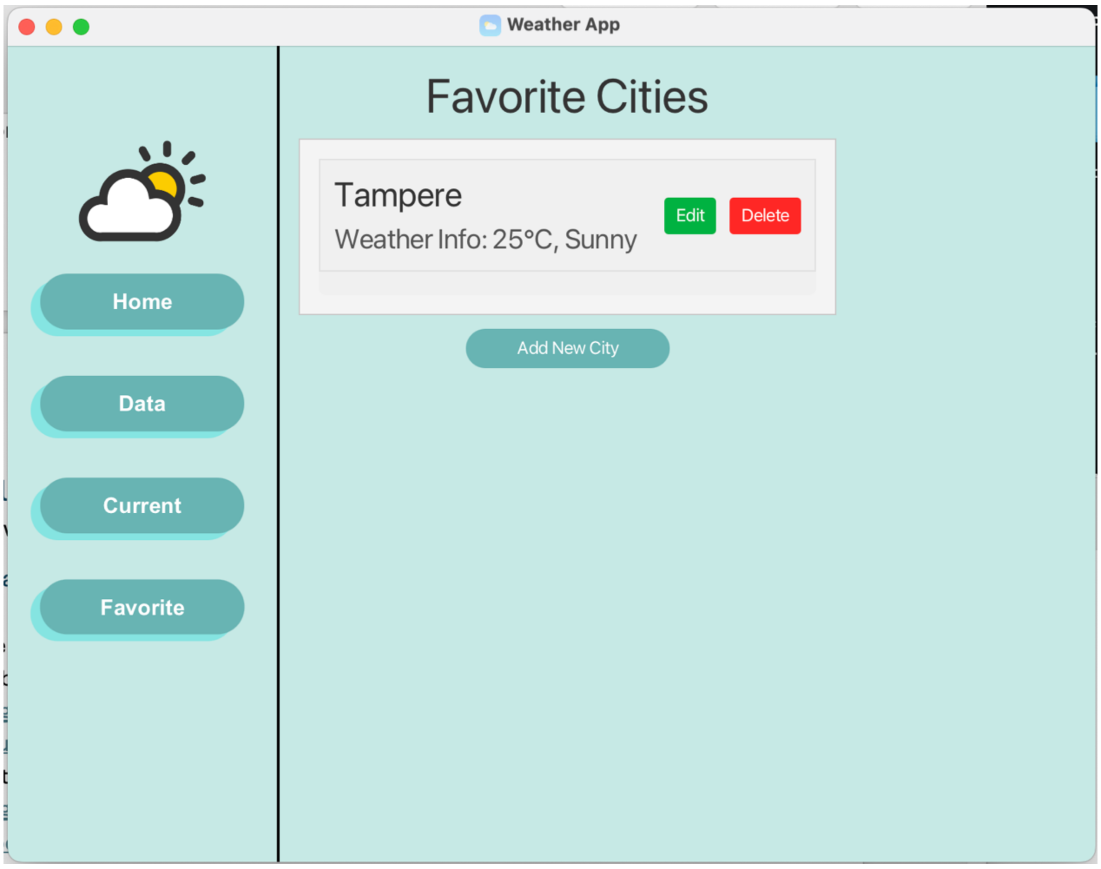


Figure 6. Favorite view

9. Future Implementation and Enhancements

Some future implementations and enhancements for the final submission.

- Completing data visualization part.
- Implementing Settings view, Alerts and Notifications features using builder pattern. This pattern is suitable for complex objects with many pieces of information. Some attributes might be optional, depending on the available data of the API. Instead of using lengthy constructor with many parameters, builder pattern could be used to simplify the object creation process. [8]
- Hardcoding business logic of activities and alerts suggestions.
- Implementing Favorite view and data logic of saving cities.
- Improving GUI for better aesthetic UI and user-friendly UX.

10. AI uses

In this project, we use ChatGPT for business logic and idea recommendations, debugging codes, and documentation idea recommendations.

11. Self-reflection

After meeting with the teaching assistant, we realized that one of our initially chosen APIs did not align well with the project requirements. The guidelines specify that both APIs should be used meaningfully and equally. To address this, we updated our design by replacing the second API with an Air Quality API. This change allows us to integrate and visualize weather and air quality data more effectively, ensuring a balanced and purposeful use of both APIs.

References

[1] Refactoring Guru, Singleton, accessed on 27.10.2024, available at https://refactoring.guru/design-patterns/singleton

[2] Refactoring Guru, Façade, accessed on 27.10.2024, available at https://refactoring.guru/design-patterns/facade

[3] Stackify, SOLID Design Principles Explained: The Single Resplonsibility Principle, accessed on 27.10.2024, available at https://stackify.com/solid-design-principles/

[4] Stackify, Dependency Injection, accessed on 27.10.2024, available at https://stackify.com/dependency-injection/

[5] Stackify, SOLID Design Principles Explained: The Open/Closed Principle, accessed on 27.10.2024, available at https://stackify.com/solid-design-open-closed-principle/

[6] Refactoring Guru, Template Method, accessed on 27.10.2024, available at https://refactoring.guru/design-patterns/template-method

[7] GeeksForGeeks, MVC Design Pattern, accessed on 27.10.2024, available at https://www.geeksforgeeks.org/mvc-design-pattern/

[8] Refactoring Guru, Template Method, accessed on 27.10.2024, available at https://refactoring.guru/design-patterns/builder