Project Documentation
Weather and Air Quality Application
AeroSky

Version 2.0 – Final submission 01.12.2024

Group Kirsi:

Anh Tran 150511795

Anna Luu 424956

Tuan Nguyen 152060244

Kirsi Huhtakallio K424394

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, giving users a clear overview of current outdoor conditions, helping them make 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 in the home view and displays:

- Current view: current data of weather and air quality of the input city. User can also add/remove a city to/from their favorite list of cities by clicking the heart icon on the top right of the view.
- Data view: near weekly weather forecast and visualization of weather and air quality data in a period. Displayed parameters can be adjusted dynamically with checkboxes and date-pickers.
- Favorite view: list of saved cities as cards. User can save their frequently checked cities from current view and quickly see current temperature and weather situation of cities. Use can also delete a city from favorite list.
- History view: search history as table. User can look up their search history based on period of search, or city name. Results are displayed with search ID, city name, date of search and temperature.
- Account view: account information stored in the database. User can see account information, username and password. They can also modify username and password to log in to the app.

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 the 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

- JavaFXML: 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. Installation Guide and Instruction

Instruction and installation guide can also be found in the project README.md

Prerequisites

Before proceeding, ensure the following software is installed on your system:

1. Java SDK 17: Download and install the Java SDK 17 from https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html
2. Maven: Download and install Maven from https://maven.apache.org/download.cgi
3. Docker: Install Docker Desktop or Docker Engine from https://www.docker.com/
4. IntelliJ IDEA: Download and install IntelliJ IDEA community edition IDE from https://www.jetbrains.com/idea/download/

Clone the Repository

You can clone the repository using either IntelliJ IDEA or the terminal:

1. Using IntelliJ IDEA:
    - Open IntelliJ IDEA

- Go to **File > New > Project from Version Control**
- Enter the repository URL: https://course-gitlab.tuni.fi/compse110-fall2024/kirsi-group
- Click Clone.

2. Using the Terminal:
   Open a terminal and execute the following commands:

git clone https://course-gitlab.tuni.fi/compse110-fall2024/kirsi-group
cd kirsi-group/WeatherApp/WeatherApp

Install the Database
To set up the database, Docker is required. Run the following command in the WeatherApp/WeatherApp directory:

docker-compose up -d

This command uses Docker Compose to start the database container in detached mode. Verify that the database is running with:

docker ps

Look for the database container in the list.

Build the Project
To build the project using Maven:

1. Navigate to the project directory (WeatherApp/WeatherApp) if not already there.
2. Run the following command:

mvn clean install

This will clean any previous builds and install all dependencies.

Run the Application

1. Open the project in IntelliJ IDEA
2. Locate the main class WeatherSystemApplication.java
3. Right-click on WeatherSystemApplication.java and select Run 'WeatherSystemApplication'
4. Alternatively, run the application from the terminal:

mvn spring-boot:run

5. Log in with username user1 and password 123456 to test the app

Common Issues and Solutions

- Java version issue: If you see errors related to Java version, ensure the environment JAVA_HOME points to the JDK 17 installation path. Uninstall existing version of java and reinstall JDK 17.
- Docker not running: If the database container fails to start, ensure Docker is running and retry the **docker-compose up -d command**.
- Maven dependencies not resolving: Run **mvn dependency:purge-local-repository** to clear local Maven cache and then retry the build.

5. High-Level Description

5.1 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.

Table 1. Core 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 • Manages UI Components • Loads and Switch Layouts • State Management
	Current Controller	Manages the current layout of the user searched city's current weather and air quality related functions	• Initialization • Gets Data Input • Fetches Weather and Air Quality Data • Updates UI • Sets City Name • Adds to Favorites

	LogIn Controller	Manages log-in related functions	• Intializes Login view • Handles user login • Sets Alert for successful/failed login • Loads app's main view
	Favorite Controller	JavaFX controller displays the list of the saved favorites	• Initialization • Populates and creates city cards • Deletes city cards
	Account Controller	Manages user account related functionalities	• Initializes Account view • Displays user information • Saves updated account information
	History Controller	Manages users previous search -related functions	• Initializes History view • Displays previous searches from a user picked timeline
Business logic (Services) (Model)	Weather Service	Implement core business logic related to weather data	• Fetches current weather data from external APIs or databases. • Processes weather data to meet the application's requirements. • Provides weather data to controllers as needed.
	AirQuality Service	Define methods for obtaining air quality information based on geographical coordinates (longitude and latitude).	• Fetches Air quality information • Pollution List
	Forecast Service	Provides an abstraction for fetching weather forecast information based on a given location	• Fetches weather forecast information for a specified location.
	Favorite Service	Provides an abstraction to handle favorites	• Fetches list of favorites for user id

		based on userId and weatherInfoDto	• Toggles delete and gets information if favorite is saved
	History Service	Defines the contract for managing the weather search history	• Retrieving and saving user search history related to previous user made searches
	User Service	Provides the contract for handling user-related operations in the weather application.	• User Login • Update user information
	AirApiCall	To encapsulate the logic required to make an external API call to fetch weather data based on location, allowing for easier and more reusable integration of weather data into the application.	• External API Interaction: Facilitate 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.
	ApiTemplate		
	ForecastApi Call		
	WeatherApi Call		
	WeathetAnd AirQuality	Defines a contract that any implementing class	• Decoupling

	Facade	must follow and abstraction.	• 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 Model	BaseEntity	Defines the structure of the data and map to the database tables	• Database Mapping • Encapsulation of Data • Foundation for Business Logic • Inheritance
	Favourite Entity		
	History Entity		
	UserEntity		
Data Access	Favorite Repository	Responsible for interacting with the database	• Retrieves all favorite entities associated with a specific user by executing a custom SQL query. • Finds a specific favorite record for a user based on the city name. • Inherits standard CRUD operations
	History Repository		
	User Repository		
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		
	ForecastInfo Dto		
	HumidityDto		
	LocationDto		
	MainDto		
	SysDto		
	Temperature Dto		
	WeatherDto		
	WeatherFull Dto		
	WeatherInfo Dto		
	WindDto		

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

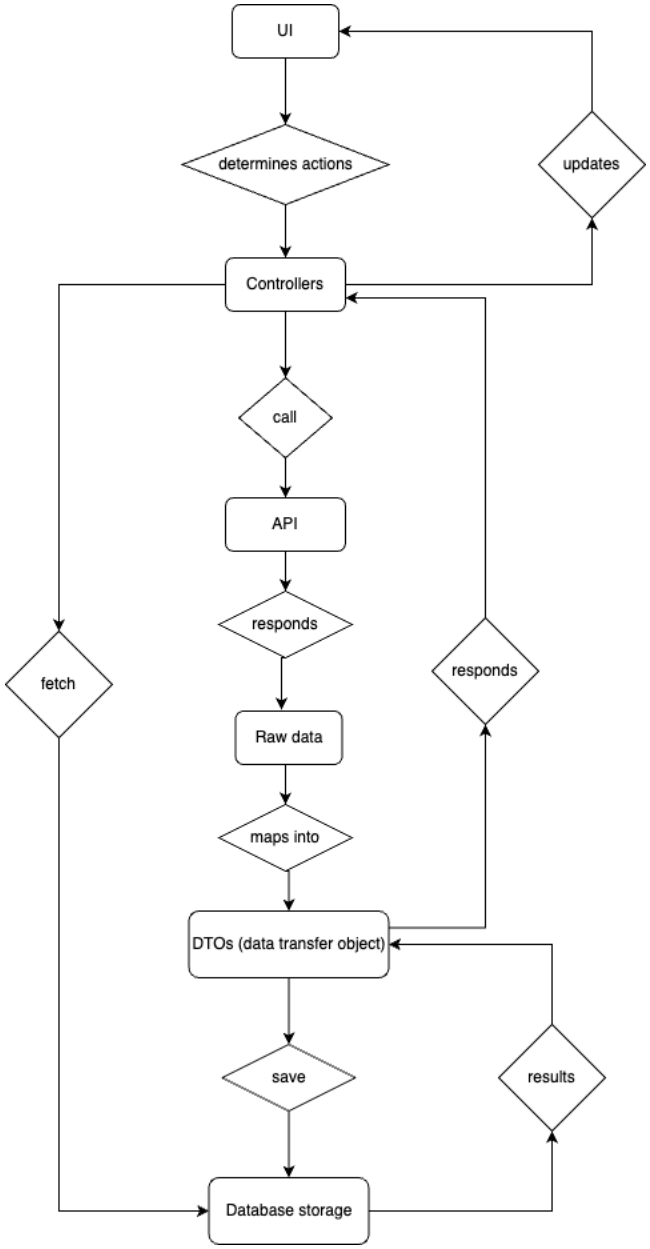


Figure 1. Flow diagram of the app.

5.2 Interfaces (internal)

The figure below shows the flows between components through interfaces in our app. There are interfaces for services, façade implementation to retrieve data from APIs, and interfaces for Repository – the database. We used façade design pattern to improve the accessibility of users to services. Controllers can access to services through the interfaces, retrieve data and update UI. Similarly, services access database through the interfaces.

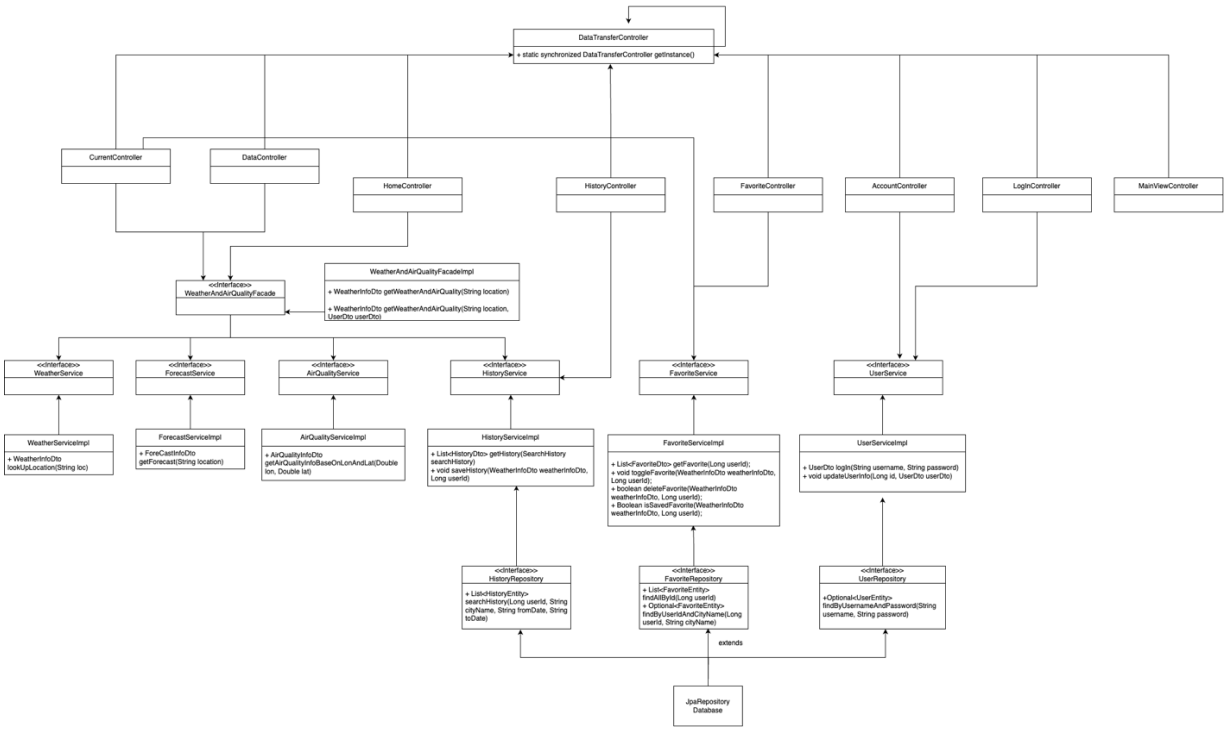


Figure 2. Flow of internal interfaces diagram of the app.

6. 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 and 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]

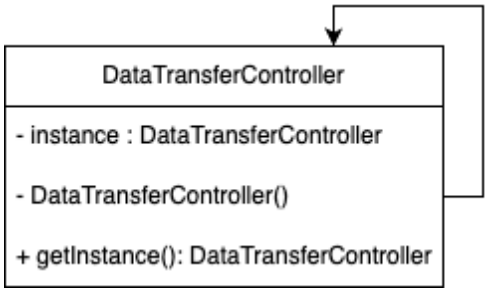


Figure 3. The Singleton pattern applied in this project for DataTransferController class ensures sharing resources smoothly between controllers.

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]

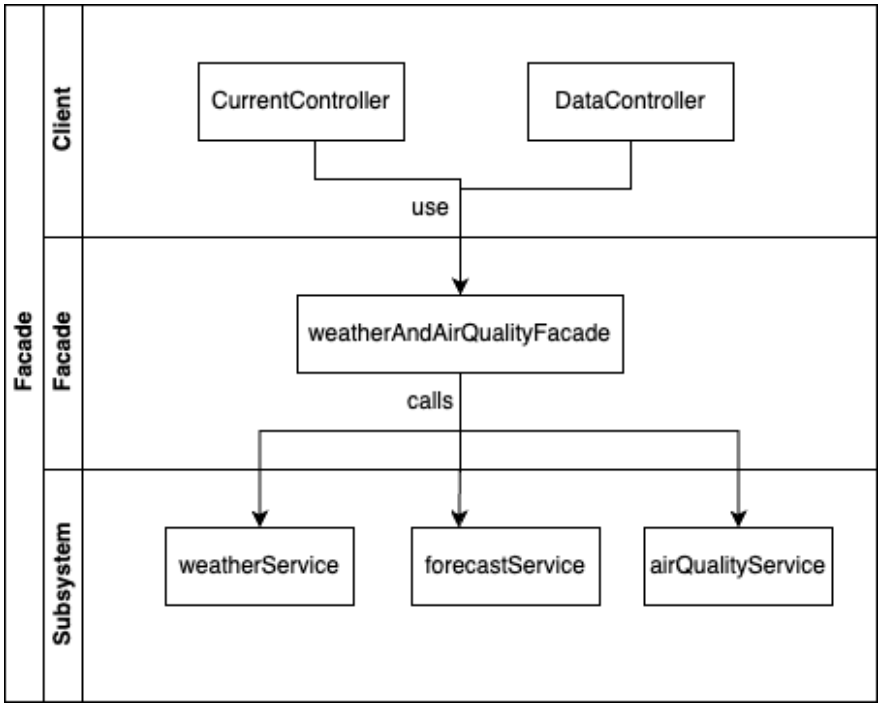


Figure 4. Façade design pattern applied in this project.

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]

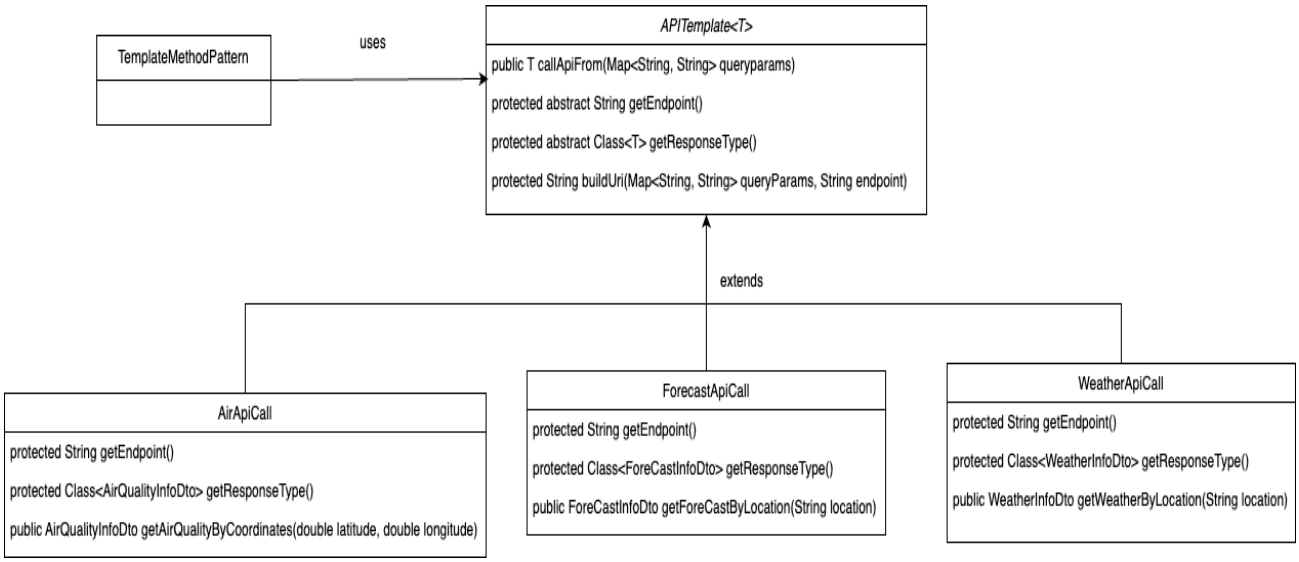


Figure 5. Template method used in this project.

Builder Pattern: a creational design pattern that helps construct complex objects step by step in a systematic and flexible way. It separates the construction of an object from its representation, allowing the same construction process to create different representations of the object. This pattern is particularly useful when objects require many configurations or optional parameters, making it challenging to manage with traditional constructors. A builder pattern can be used to create immutable objects. A typical builder structure consists of Director (or Client), Builder Interface, Concrete Builder, and Product. It is also possible to combine Builder and Concrete Builder into a static nested class inside Product. [7]

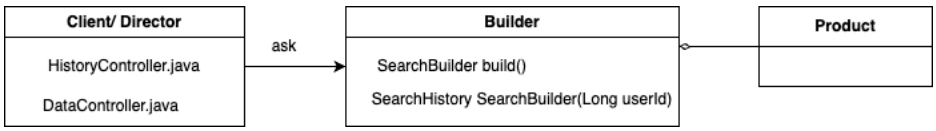


Figure 6. Builder pattern is used for History Search Feature in this project.

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. [8] The flow diagram in Figure 1 shows the MVC pattern of this app.

7. 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, and a favorite button to add/remove city from the favorite list.
    - A data view that displays weather and air quality data with interactive graphs.
    - A favorite view that shows list of saved cities.
    - A history view that user can search their search history in the app within a period.
    - An account view that user can manage their information and modify username and password if needed.

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.

8. Design Decisions and Justifications

8.1 Technical Decisions

The following technical 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. **JavaFXML for UI**: JavaFXML 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 JavaFXML 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.

8.2 Design Decisions

In this project, we used many design patterns and principles to ensure that our code structure is clean and maintainable. The summary of patterns, principles and their justifications can be found in the table below.

Table 2. Summary table of used patterns/ principles, classifications and why they are needed and included in this project.

Name of pattern/ principle	Classification	Justification
Singleton	Creational design pattern	There are many shared resources that different controllers need to access and use. Therefore, a data transfer class can help share resources smoothly. Singleton pattern is applied here to ensure that there's only one instance with the purpose of managing shared resources in the program. It helps manage the resources efficiently. Lazy initialization is used so that instance is initialized only when getInstance() function is called.
Façade	Structural design pattern	In this project, controllers need to access different services since they control different functionalities of the app. One way is to access separate services every time they are needed. A façade design pattern could be applied here to make life easier. The interface acts as a single stop for all services, so controllers can call services in WeatherAndAirQuality interface without handling each separate services.
Single Responsible Principle	SOLID Principle	We want our code architecture to be modular, efficient and maintainable. Therefore, each class should be designed so that it handles only one responsibility of the program. Mixing

		responsibilities in classes can make code less modular and maintenance harder.
Dependency Injection	Design pattern	The app should be testable and maintainable, so dependencies should be managed efficiently. Instead of harcoding dependencies, we can inject mock or implementations of dependencies to write unit tests. Spring framework makes it easy to configure and replace dependencies at runtime. DI also helps to improve consistency and reduce redundancy in the code.
Open/Closed Principle	SOLID principle	We want our classes to be extended without the risk of introducing bugs into existing source code. However, new features or behaviors should be added by extending existing classes rather than rewriting them. Following Open/Closed Principle helps here. A class should be open for extension but closed for modification. This principle promotes maintainability, flexibility and scalability of the code. In the code, new functionality could be added by creating a new interface (use of abstraction) so it is open for extension.
Interface Segregation Principle	SOLID principle	The code should also be clean and clear, avoiding unused methods and unnecessary code clutter. Therefore, interfaces should be specific to the needs of the implementing classes rather than being too broad or general. Applying this principle makes the code easier to understand and implement with smaller, specific interfaces, also enhances maintainability. In our code, each interface has a single, well-defined responsibility, and classes depend only on the functionality they actually use.
Template Design Pattern	Behavioral design pattern	We use two APIs (weather and air quality) and have three main things that need to be fetched from them (air quality, weather and forecast). Even though the fetched data is different, the structures of API calls share some common logic such as error handling and URI construction. Therefore, template design pattern can be applied to define the structure of calling APIs in a superclass while allowing subclasses to override specific steps of the algorithm without changing its overall structure.

Builder Pattern	Creational design pattern	In the History view, users can look up their search history in the database within a time range, after a date or before a date, or based on the city name. Some parameters could be optional when creating an object to search, so it is harder to manage with traditional constructors. Builder pattern can be applied here to construct complex objects step by step in a systematic and flexible way. It can be used to create immutable objects.
Model-Control-View Pattern	Architectural design pattern	A widely used design pattern for structuring software app. We found MVC suitable for a layered, modular and maintainable code. Views and controllers are independent of the model, allowing them to be reused or tested in isolation. The components interact through well-defined interfaces, reducing interdependencies.

For the UI of the app, it should be user-friendly with enough functions for users to see the data and have some ideas about weather and air quality in their cities. The name AeroSky implies that this app's main purpose is about air and weather.
We decided some basic functions based on requirements and some common applications:

- home view for searching,
- current view for displaying result,
- data view for exploring more about data in the input city,
- history view for looking up search history,
- favorite view for storing/ displaying favorite cities of user
- account view for managing user information saved in the database.

A left-hand side navigation tab with buttons helps navigating through the app. Since the theme is about weather and air, color pallet should be relevant to Earth, sky, creating a peaceful, fresh and natural vibe. Our chosen color pallet is shown below, with codes as CDE8E5, EEF7FF, 7AB2B2, 4D869C.

Figure 7. Color pallet for the app's UI.

To ensure a smooth UX, we also implemented some warnings and notifications for some activities such as logging in, deleting favorite city. The app is also designed to be interactive in some views such as interactive graphs in data view, search in history view, modifiable information fields in account view.

9. Graphical User Interface (GUI)

In this project, we use JavaFXML to build GUI in desktop. This decision seemed feasible as the given project scope was to build a desktop application instead of web application with java backend.

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

Final UI prototype of the app

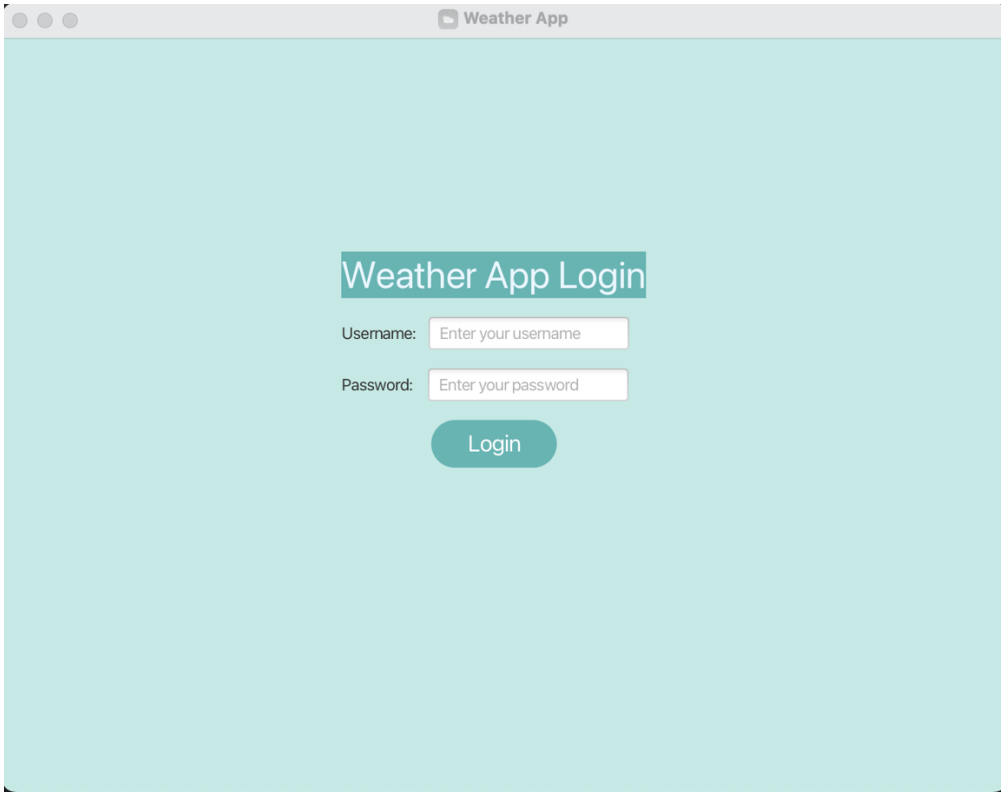


Figure 8. Log In view

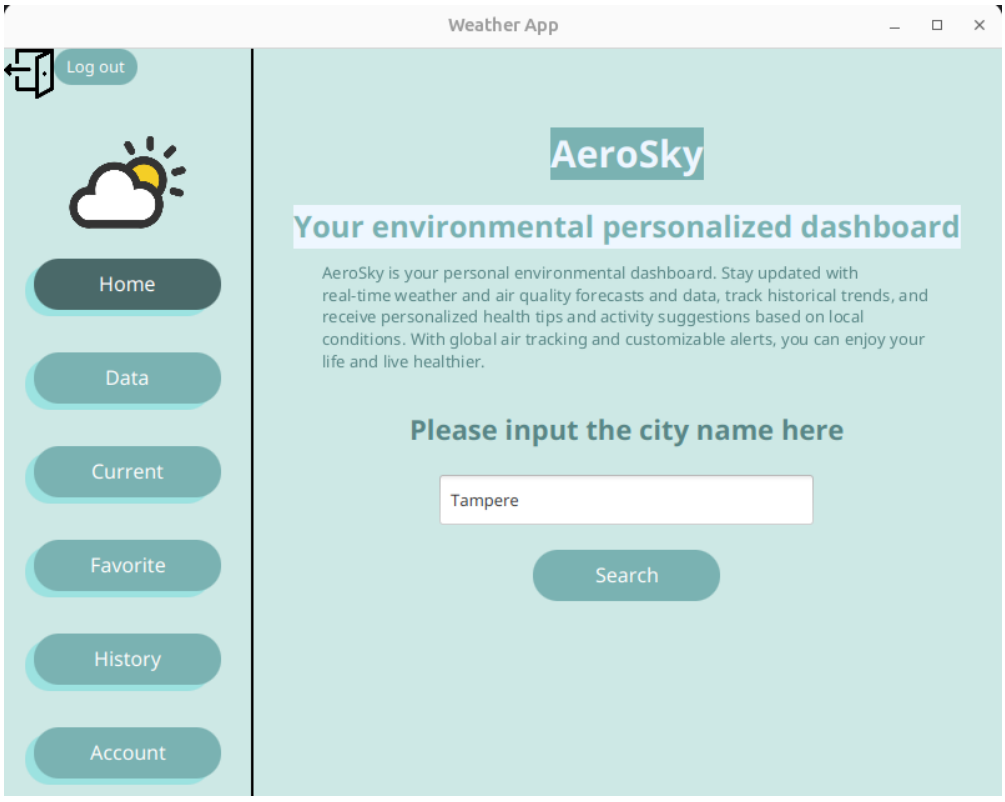


Figure 9. Home view

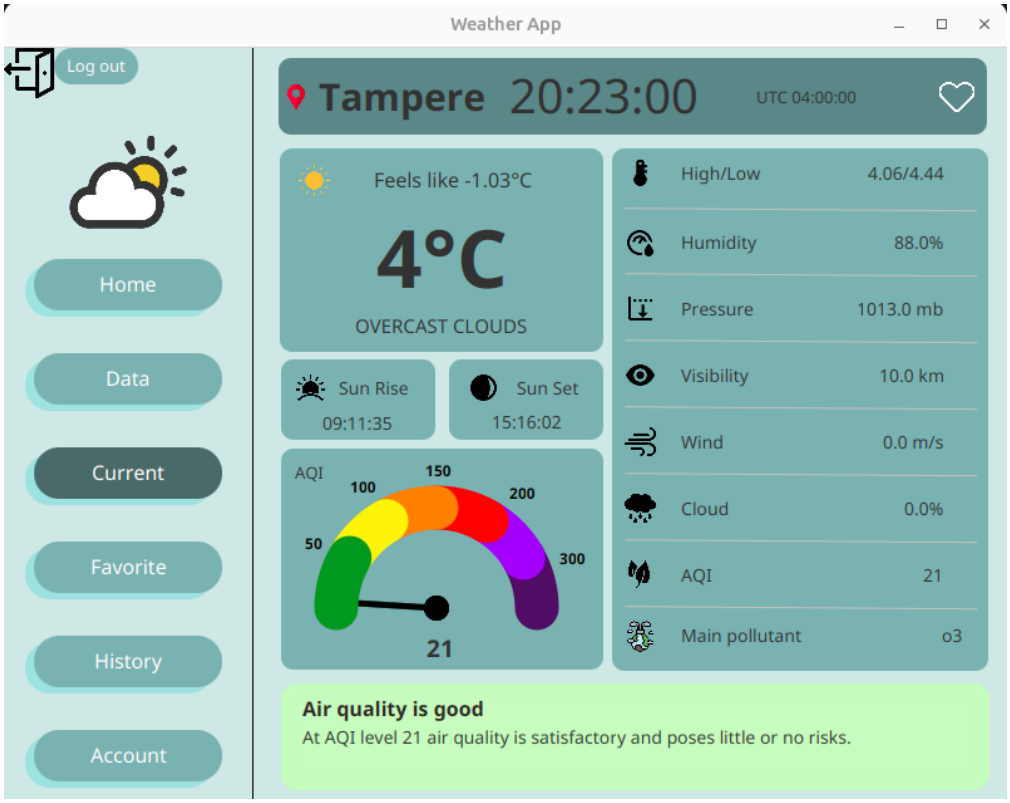


Figure 10. Current view

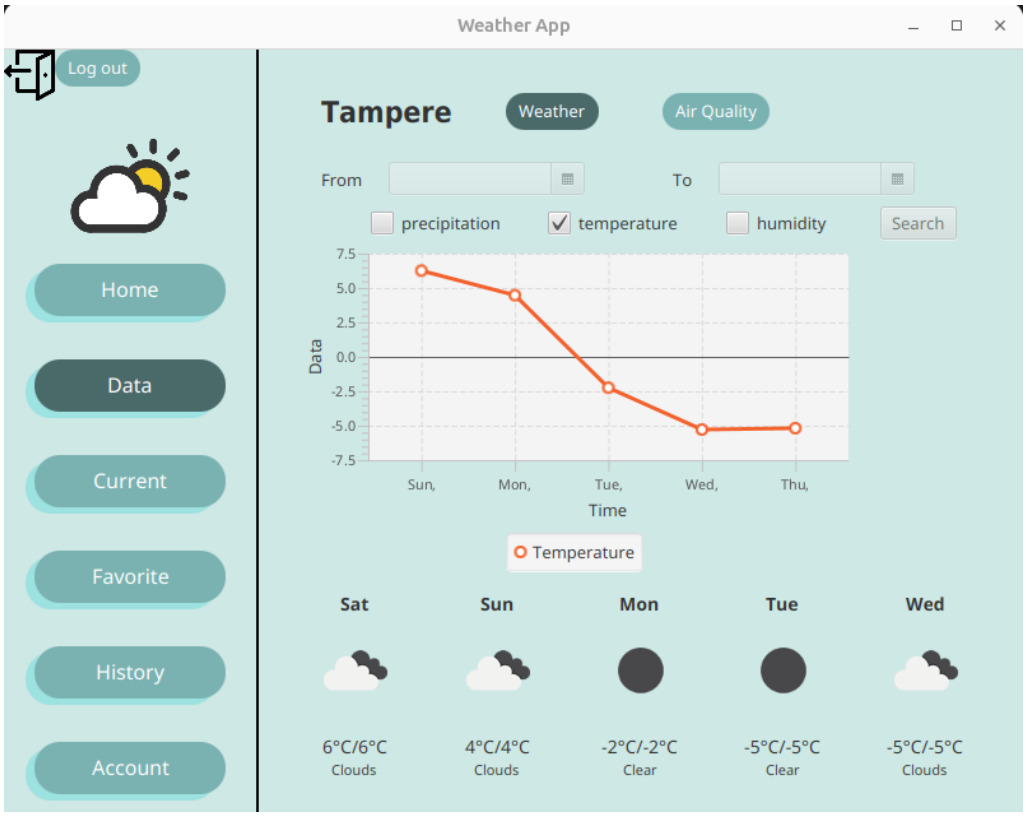


Figure 11. Data view

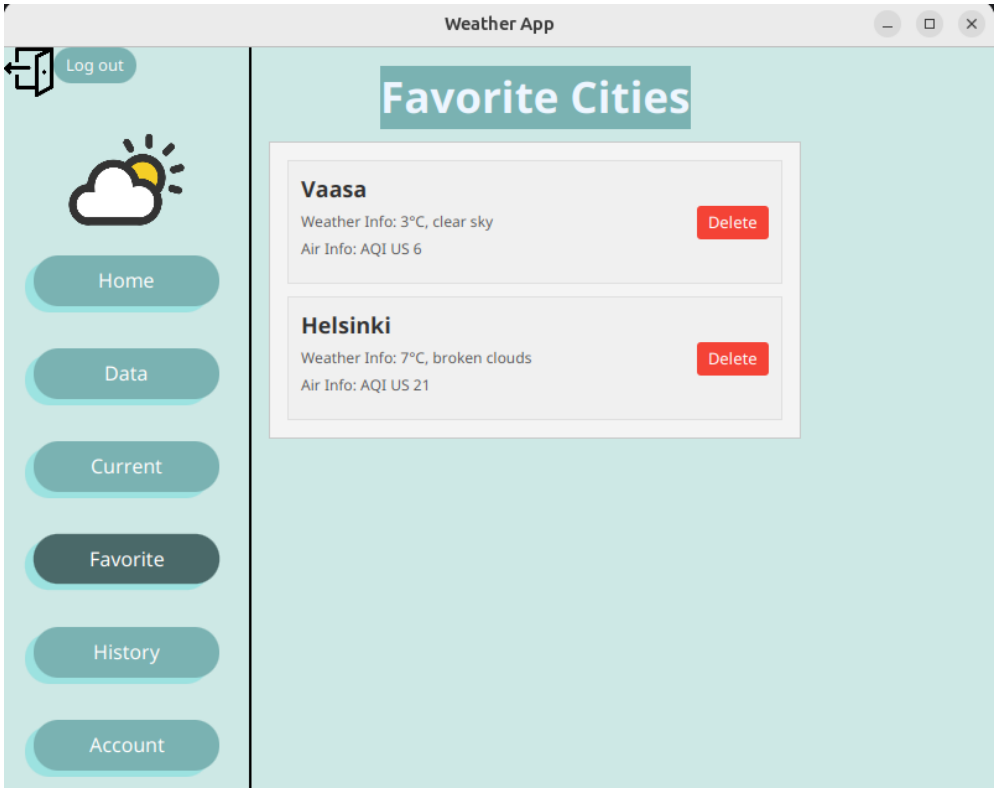


Figure 12. Favorite view

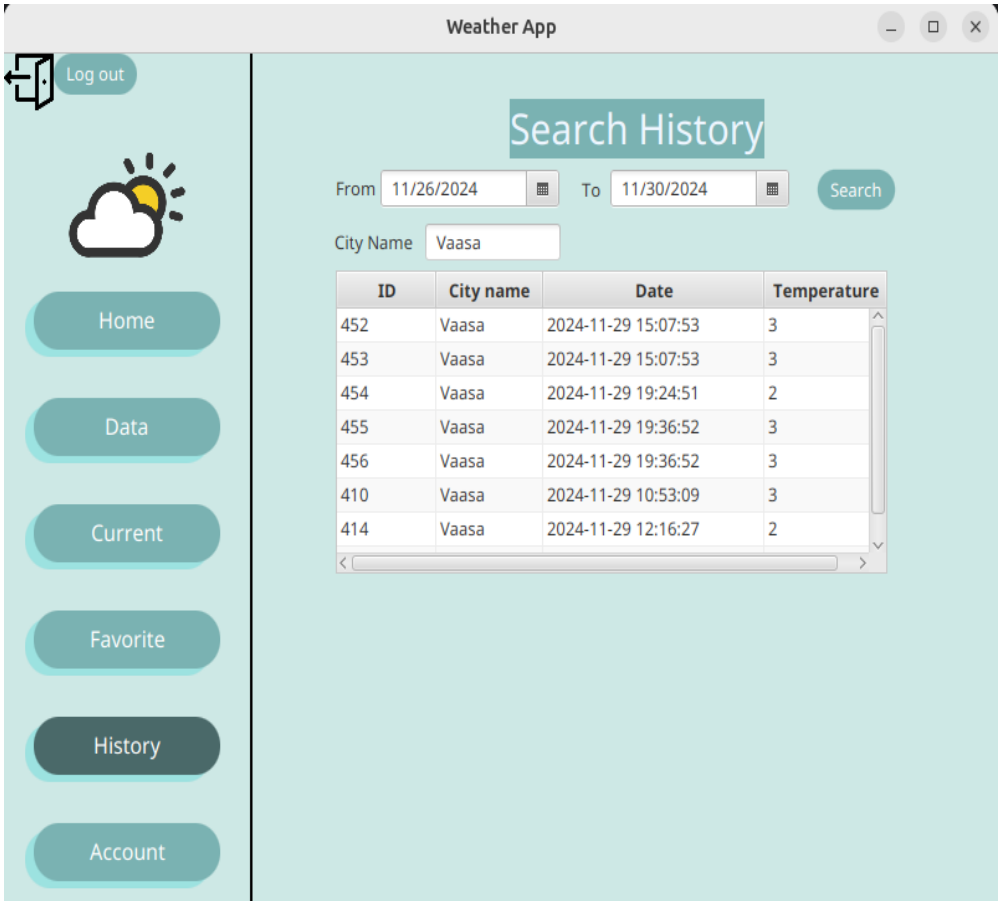


Figure 13. History view

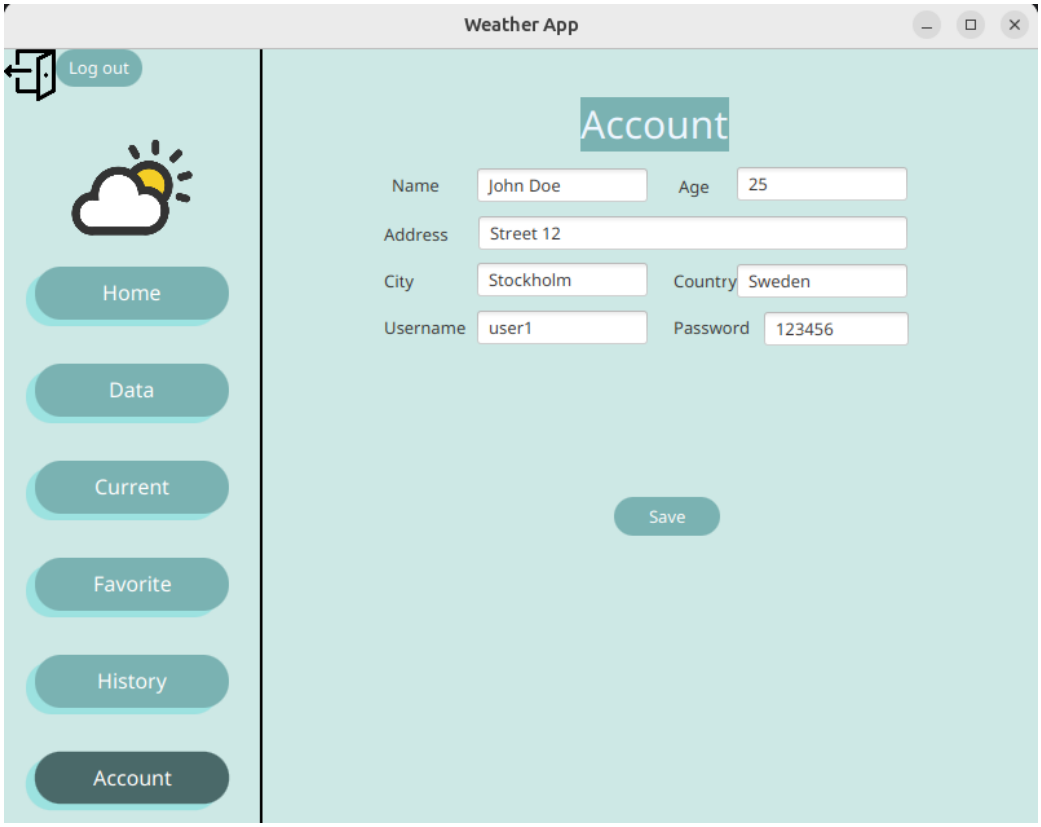


Figure 14. Account view

9. Future Implementation and Enhancements

Some future implementations and enhancements in the future.

- Finalizing data visualization part to display the data more meaningfully.
- Extending new features such as Alerts and Notifications to give users alerts about air quality dangerous classification, severe weather conditions and give advice for activities outdoor or indoor.
- Improving graphical user interface (GUI) for more aesthetic UI and intuitive user-friendly user experience (UX).
- Basic features can be broadened If the paid versions of both APIs can be accessed, the data of air quality forecast as well as history can be fetched to display graph of air quality forecast in Data view as well as commercial level air pollution information.

10. AI uses

In this project, we use ChatGPT for business logic and idea recommendations, debugging codes, and documentation idea recommendations. AI was also used at times for code review in which IntelliJ AI Assistant pointed out whether the application structure followed layered and modular architecture.

AI can be beneficial in software design and implementation. It can help in tasks such as code formatting, refactoring code, writing simple test cases, or generating suggestions, so developers can focus on complex logic behind software. AI is also useful in reducing errors like syntax errors or suggesting solutions to code structure problems. Especially, junior developers or students benefit from those suggestions since they usually don't have much experience to handle design and errors for complex software. In large software projects, AI can be used to automate repetitive tasks, improving the efficiency of developers. However, some people take advantage of AI without knowing its disadvantages. Over-relying on AI tools may lead to skills degradation, and blind trust in AI solutions leads to incorrect or suboptimal implementations. AI may not fully understand the nuances or specific requirements of a project, leading to irrelevant or inappropriate suggestions. Not mention concerns about security. Therefore, AI is beneficial but it should be used carefully.

11. Self-evaluation

At the beginning of this project, we had ideas about a smart planner that allows user to search data about weather and forecast, receive activity recommendations based on their interests and locations. However, we had difficulties in finding a suitable API for event data and location data. Visualizing data as a map also takes more time than normal plots. Moreover, we had to think about complete business ideas behind the app. How can users receive information about the events in their places? What is the logic of activity recommendations for users?

After the first 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 location 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.

For mid-term submission, data can be fetched from 2 APIs. Basic interactions between APIs and business model were formed. Some features were coded to demonstrate our ideas. Even though we hadn't worked on database and detailed app logics, the UI and ideas about functions of the app were quite complete in this phase.

After the second meeting with the teaching assistant, we continued developing the rest of the functionalities and database. At that stage, the purposes of our app were searching for current weather and air quality of cities, searching data in a period and weather forecast in a week, adding cities to a favorite list, managing account information. We added history, favorite and account management; improved business logic, database structure and UI/UX. Throughout the process of developing this app, we had changed the

business ideas, logics, code designs three times to improve functionality and usability of the app. The result is quite complete, and it satisfies all necessary functional requirements. We also learned so much about design patterns and principles in developing an app and how they help improve the code structure to become more professional, modular, and maintainable in the future. Even though the final business ideas are not all the same as the initial brainstorming ideas, we kept all important functionalities and ideas from the first stage and developed them well enough. In the later stage of this project, we also implemented and built more functions successfully.

However, free Air Quality API ended up having constraints to the initial data visualization designed purpose, which was a shame, so for a lesson learned: research API constraints more thoroughly and pay attention to which APIs are truly open source to the core. As populating line chart trends with API data was not possible, it was decided that populating the chart should be done using saved previous user search data.

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/

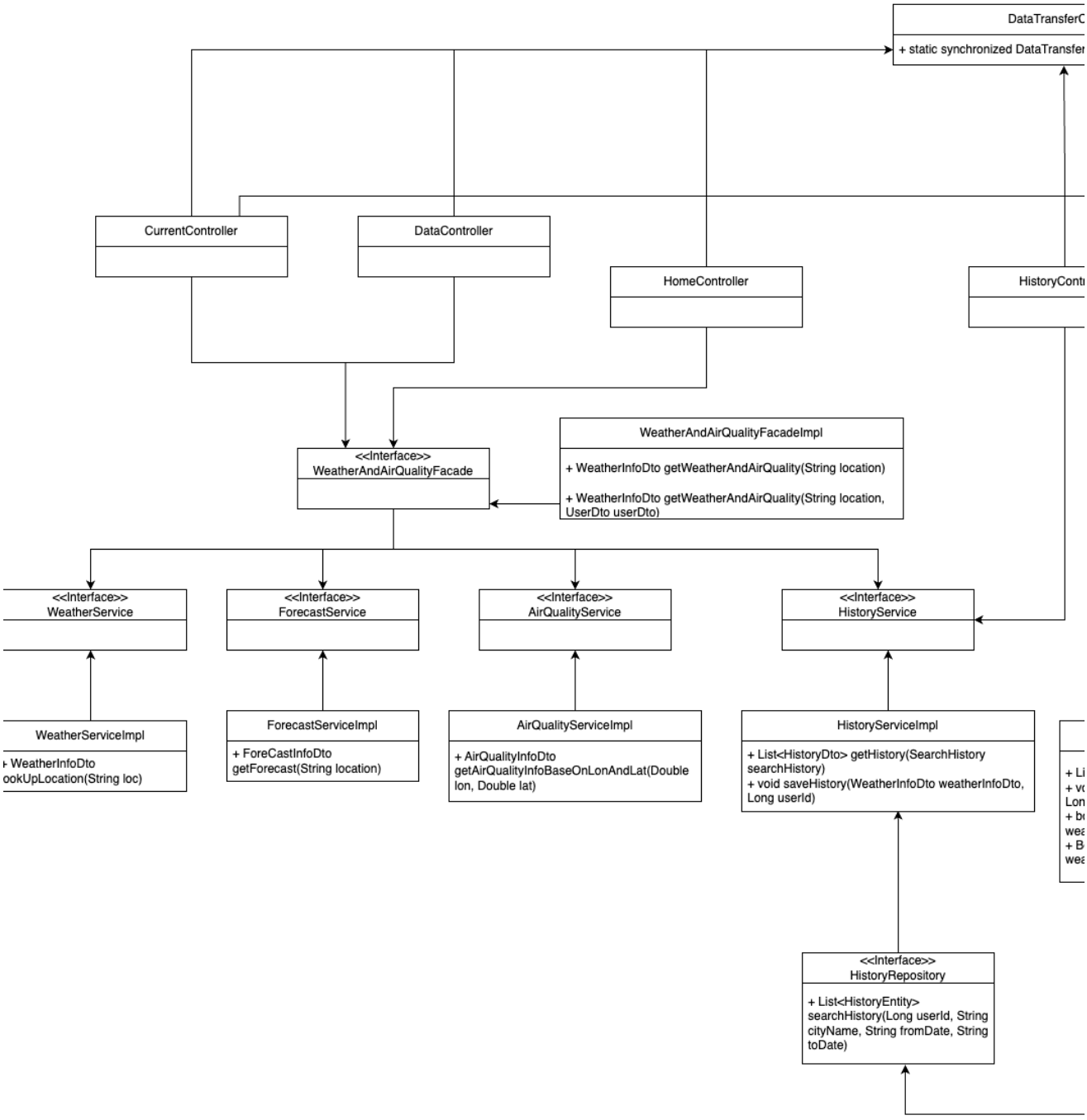
[6] Refactoring Guru, Template Method, accessed on 27.10.2024, available at https://refactoring.guru/design-patterns/template-method

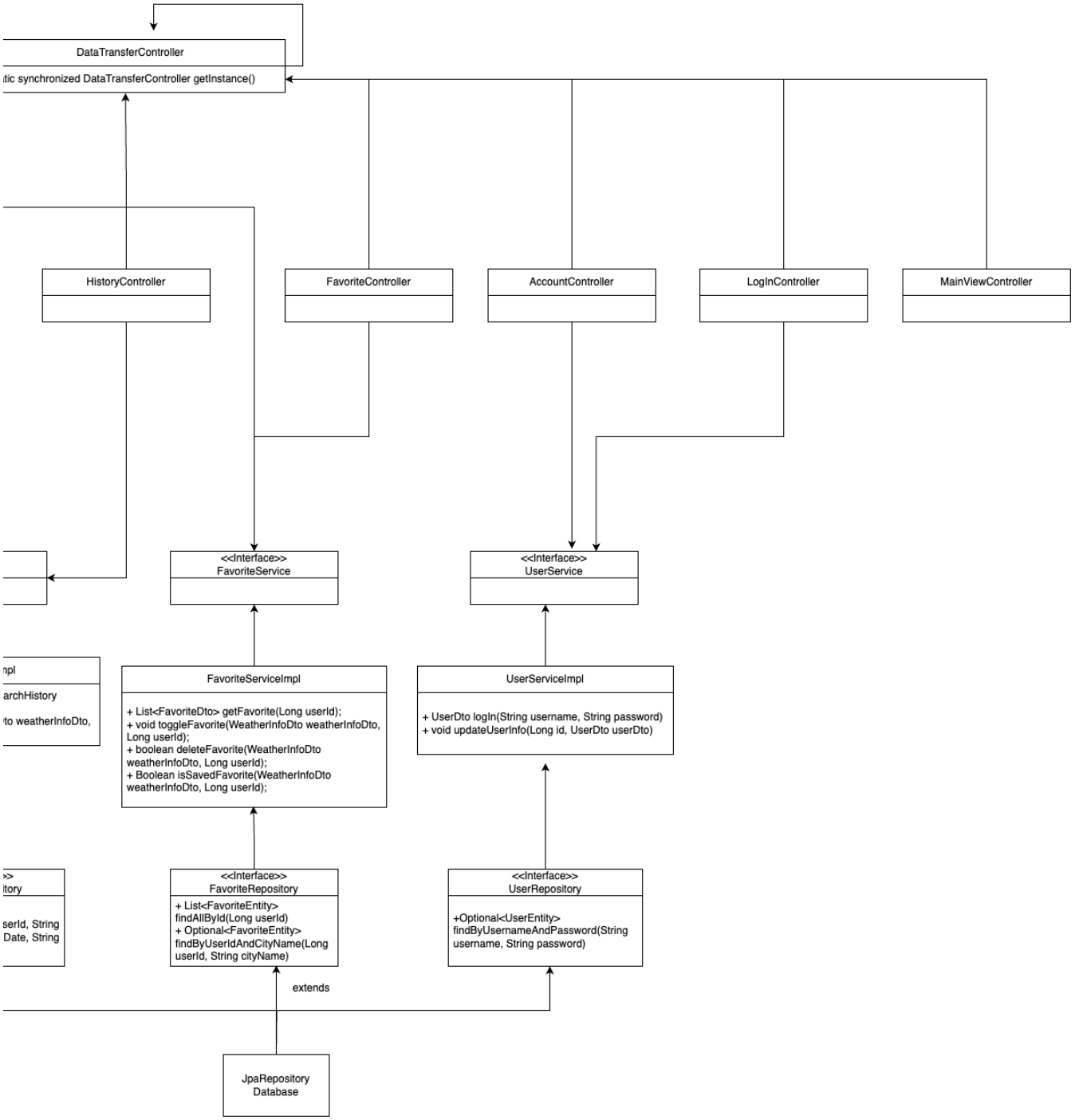
[7] Refactoring Guru, Builder, accessed on 27.10.2024, available at https://refactoring.guru/design-patterns/builder

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

Appendix

Internal interface flow chart




DataTransferController
tic synchronized DataTransferController getInstance()
HistoryController
FavoriteController
AccountController
LogInController
MainViewController
<<Interface>>
FavoriteService
<<Interface>>
UserService
FavoriteServiceImpl
+ List<FavoriteDto> getFavorite(Long userId);
+ void toggleFavorite(WeatherInfoDto weatherInfoDto, Long userId);
+ boolean deleteFavorite(WeatherInfoDto weatherInfoDto, Long userId);
+ Boolean isSavedFavorite(WeatherInfoDto weatherInfoDto, Long userId);
UserServiceImpl
+ UserDto logIn(String username, String password)
+ void updateUserInfo(Long id, UserDto userDto)
<<Interface>>
FavoriteRepository
+ List<FavoriteEntity> findAllById(Long userId)
+ Optional<FavoriteEntity> findByUserIdAndCityName(Long userId, String cityName)
<<Interface>>
UserRepository
+Optional<UserEntity> findByUsernameAndPassword(String username, String password)
extends
JpaRepository
Database