Design of the Weather and Traffic Application

COMP.SE.110–2024–2025–1 Software Design – Group G4

Contents

1. Introduction 3
2. High-level description of the application 3
   2.1. Application Structure: back-end 3
   2.2. Application Structure: front-end 4
   2.3. Libraries and third-party components 5
3. Components and responsibilities 6
   3.1. Backend components 6
   3.2. Frontend components 6
   3.3. Detailed explanation of the structure of the components 7
      3.3.1. User interface components 7
      3.3.2. Controller layer components 8
      3.3.3. Service layer components 9

3.3.4. Model layer components 10
4. APIs 12
4.1. Transportation data: DigiTraffic API 12
4.2. Weather data: FMI Open Data 13
4.3. Other APIs considered 13
5. Design solutions and reasonings 13
6. Self-evaluation 14
6.1. Changes made from the prototype submission 14

1. Introduction

This document aims to provide an overview of the architecture and design of the *Weather and Traffic* application. This application is designed to help users make informed decisions about transportation by providing them with updates on weather conditions and how these may affect different services. The application provides weather and traffic updates based on location: predictions based on current weather conditions based on a state-of-the-art algorithm – will the traffic be delayed or not – and it also provides the current actual weather and traffic status.

2. High-level description of the application

- Key functionalities of front-end:
  - Real-time display of current traffic conditions.
  - Display of current weather conditions.
  - Make traffic predictions based on current weather and location or city.
- Key functionalities of back-end:
  - Process front-end requests to retrieve traffic and weather data.
  - Integrate and manage external APIs.
  - Apply business logic to determine traffic predictions based on the weather situation

2.1. Application Structure: back-end

The following figure describes the back-end structure of the application on a high level. The components *TransportController, PredictionController* and *WeatherController* handle requests from the client side and route them to the service layer. They also forward responses back to the client. *TransportService, PredictionService* and *WeatherService* are service layer components, which handle the business logic of the application. They fetch data from the external APIs and/or each other and map it to the data models *TransportStatus, PredictionStatus* and *WeatherStatus*, returning results back to the controller components.

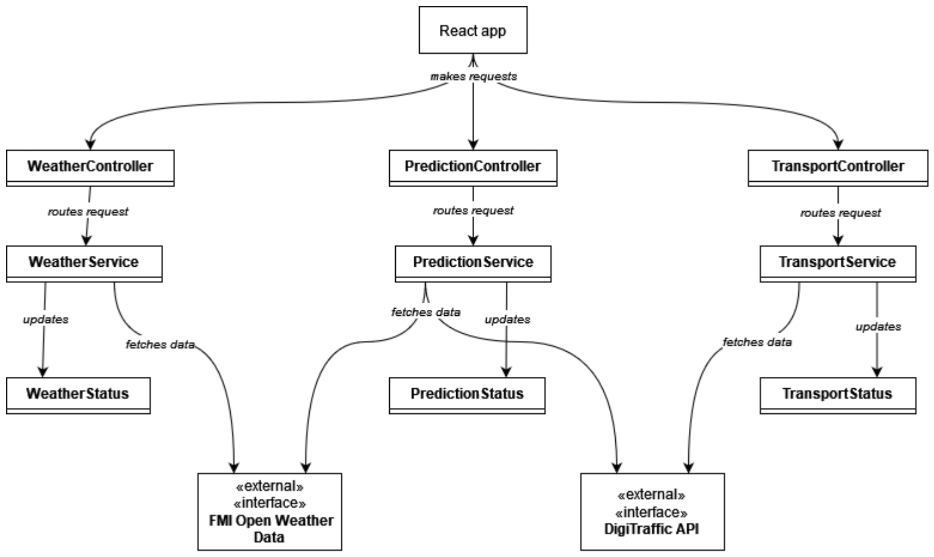

Figure 1: Back-end structure and interactions

2.2. Application Structure: front-end

The front-end of the application is implemented with React. *App* is the root component that manages the whole React application. *Dashboard* contains child components *SearchComponent*, *WeatherDisplay, TransportDisplay* and *PredictionDisplay*. The display components handle rendering data from the back-end on the user interface. *SearchComponent* provides a form for user input to query weather and transport data by city.

The application uses React Context (*DataContext*) to centralize data fetching from the back-end and manage shared state between components.

The figure below presents the component hierarchy and data flow in the front-end.

DataContext holds the *city, transport, prediction* and *weather* states and provides the *updateCity* function, which the *SearchComponent* uses to update city state when the user submits a new query. *DataContext* also handles the API requests to the back-end. *Dashboard* is wrapped in the *DataContext,* which allows the *Dashboard* and its child components to access the data from the context directly.

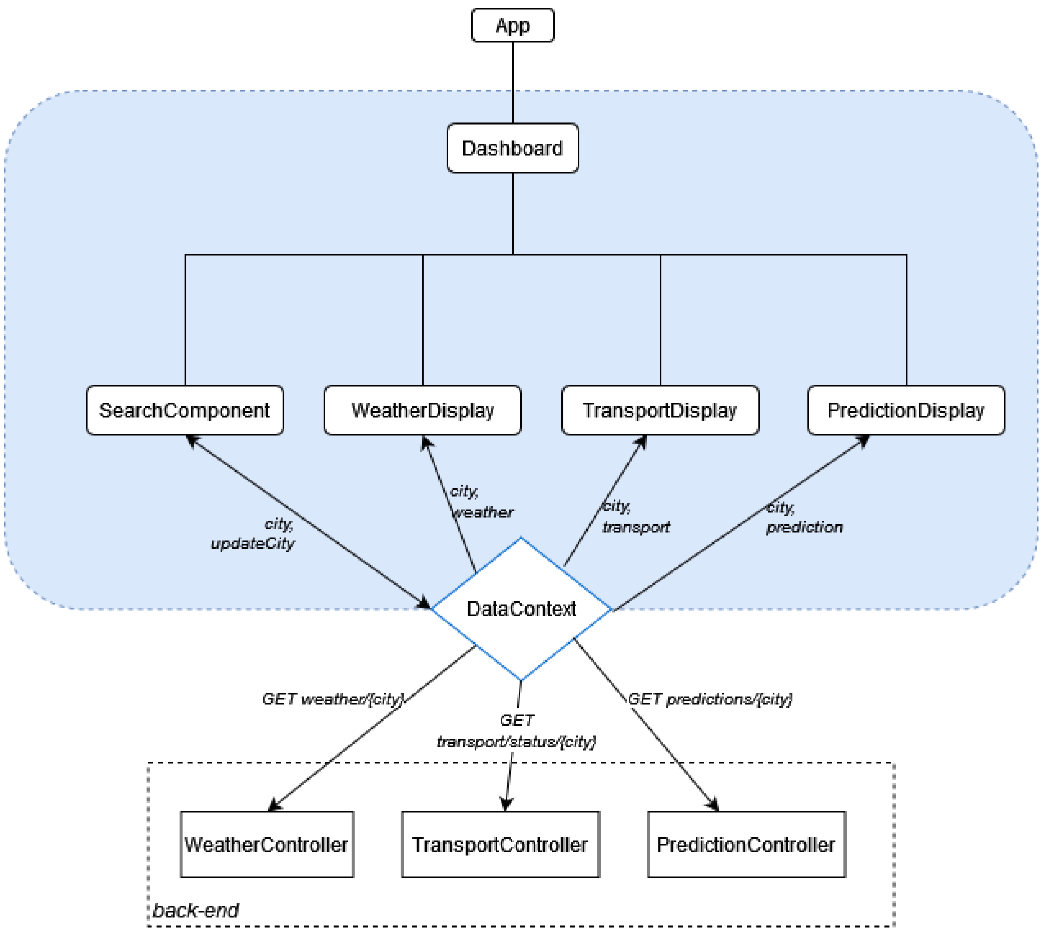

Figure *2*: Front-end structure and interactions

2.3. Libraries and third-party components

Axios is used within DataContext to fetch data from the back-end. It provides a promise-based client for making HTTP requests with easy syntax and response management.

React Chart.js 2, a React wrapper for Chart.js library, is used by the display components to visualize data in the application.

React Leaflet, a React wrapper for Leaflet.js library, is used to display transport and weather data on a map. It works well with open-source data and supports multiple plugins for additional functionality.

frontend-maven-plugin, a plugin that installs Node.js and NPM for the project, and install dependencies with command *npm istall*. Simplifies the building of front-end and back-end together.

maven-resources-plugin, a plugin that automatically copies project resources to the output folder, reducing the amount of manual work.

Thymeleaf, a plugin that is used to allow the frontend of the application to be displayed, simply by sending a HTTP request to the URL http://localhost:8080 without adding anything to the address.

3. Components and responsibilities

3.1. Backend components

The architecture separates the responsibilities of different code components into distinct layers: Controller, Service, and Model - each with a specific role in the application.

Controller Layer:

The Controller layer (e.g., PredictionsController) handles HTTP requests from the client-side (React front-end) and returns the appropriate response. It acts as the interface between the user (via HTTP) and the backend.

For example, @RestController annotations are used to define REST endpoints that allow the frontend to interact with the backend.

Service Layer:

The Service layer encapsulates the business logic of the application. Services handle all interactions between controllers and repositories (or external APIs) to retrieve data and perform computations.

In our case, the PredictionService fetches data from WeatherService and TransportService to provide combined weather and traffic predictions to the frontend.

This layer ensures that the Controller remains focused on HTTP handling, while the business logic is centralized within the Service.

Model Layer:

The Model layer consists of data objects representing the entities in our system, such as PredictionStatus, WeatherStatus, and TrafficStatus.

These models are used to transfer data between the layers of the application. For instance, the service returns a PredictionStatus object that is constructed using the information fetched from both the weather and transport services.

The models are simple POJOs (Plain Old Java Objects) with attributes and getter/setter methods.

3.2. Frontend components

The frontend, built with React, communicates with the Spring Boot / Java backend via our REST API described above.

The frontend is able to display weather and traffic data for a given Finnish city. In addition, the front-end is able to retrieve accurate weather-based traffic condition predictions from the back-end.

3.3. Detailed explanation of the structure of the components

3.3.1. User interface components

App.js

App serves as the root component and entry point of the application. It is responsible for rendering the Dashboard component.

DataContext.js

DataContext is a context provider that fetches data from the back-end and manages shared state for the front-end of the application.

DataContext has state variables *city*, *weather*, *transport*, *prediction*, *loading* and *error*. *city* holds the name of the current city for which data is fetched, initially set to 'Helsinki'. *weather*, *transport* and *prediction* store the fetched weather, transport and prediction data for the specified city. *loading* is a boolean variable indicating if data is currently being fetched, and *error* holds any error messages encountered during the data-fetching.

The context defines and provides a function *updateCity(newCity*), which updates the *city* state with a new value. A *useEffect* hook is used to trigger an asynchronous function *fetchData()* whenever the city state changes. The function initiates simultaneus requests to fetch data for the current city and updates the data-holding states. It also manages *loading* and *error* states according to the success or failure of the requests.

The context passes *city*, *weather*, *transport*, *prediction*, *loading*, *error* and *updateCity* to the subscribing components.

Dashboard.js

Dashboard serves as the main interface for displaying weather, transportation and prediction data based on the city search by the user. It renders child components *SearchComponent*, *WeatherDisplay*, *TransportDisplay* and *PredictionDisplay*.

SearchComponent.js

SearchComponent allows the user to write a search query and submit it to fetch relevant data. It renders an input form where the input field updates as the user types and triggers a search function when the form is submitted.

State variable *query* holds the current value entered in the search input field. When the form is submitted, the *handleSearch* function calls *updateCity*, received from the DataContext. It receives the current *query* as an argument, so the *city* state in the context is updated with the search term.

WeatherDisplay.js

WeatherDisplay is responsible for displaying weather data for a specified city. It uses states *city*, *weather*, *loading* and *error* from the DataContext. The data is displayed both verbally and visually.

The component conditionally renders the weather data, a loading message or an error message, depending on the success of the data retrieval.

TransportDisplay.js

TransportDisplay is responsible for displaying transportation data for a specified city. It uses states *city*, *transport*, *loading* and *error* from the DataContext.

PredictionDisplay.js

PredictionDisplay displays prediction data for weather and traffic situations. It uses states *city*, *prediction*, *loading* and *error* from the DataContext.

3.3.2. Controller layer components

The code for Weather, Transport and Prediction controllers was taken from Chat GPT's example and modified from that.

WeatherController.java

Weather Controller listens to weather related HTTP requests, expecting city as a path variable and time as an optional parameter. After parsing the information, it directs the request to the service layer. The format of the HTTP request is */weather/{city}?time=YYYY-MM-DDThh:mm:ssZ* where {city} is a city in Finland, and in the optional time parameter *YYYY-MM-DD* part tells the date and *hh:mm:ss* part tells the time in UTC time. The time is expected to be around 10 minutes. An example request would be */weather/Helsinki?time=2024-09-27T12:00:00Z*.

3.3.2.1 Unit Test: WeatherControllerTest.java

Purpose: It tests the WeatherController class which provides the weather data through its endpoints.

What is Tested:

`getCurrentWeather_Success()`: Checks that the controller correctly returns weather data for the current time.

`getWeatherAtTime_Success()`: Checks that the controller correctly returns weather data for a specific time.

Mocks Used: It uses a fake "WeatherService" to simulate weather data.

Logging: Logs steps and outcomes.

TransportController.java

Transport Controller handles HTTP requests related to transport status, requiring the city name as a path variable. It forwards requests to the service layer to retrieve the relevant data. The HTTP request format is */transport/status/{city},* where *{city}* specifies the city for which the transport status is being requested. For example, a request to check the transport status for “Tampere” would be: *transport/status/Tampere*. The controller processes the request and returns the transport status data as a JSON response.

PredictionController.java

The PredictionController handles HTTP requests related to traffic and weather predictions, requiring the city name as a path variable. It delegates requests to the service layer to retrieve combined weather and traffic prediction data.

The HTTP request format is /predictions/status/{city}, where {city} specifies the city for which the prediction is being requested. Optionally, a time query parameter can be provided to specify the prediction time (e.g., */predictions/status/Helsinki?time=2024-10-11T14:00:00*).

For example, a request to check the prediction status for "Helsinki" at a specified time would be: */predictions/status/Helsinki?time=2024-10-11T14:00:00*. If the time parameter is omitted, the current time is assumed.

The controller processes the request and returns a JSON response containing the combined weather and traffic prediction data for the specified city and time.

3.3.3. Service layer components

TransportService.java

Transport Service has one public method: *getTransport*, which provides the latest transport status data for a specified city. This data is retrieved from the Digitraffic API, which supplies real-time traffic measurements across various stations.

To gather transport information for a city, the program first requests all available station data from the Digitraffic API. Once the response is received, it filters the station names to find matches with the specified city name. If any stations match, the program requests additional sensor data for each relevant station. This data includes measurements defined in the *TransportStatus* and *SensorData* class, such as sensor ID, name, unit, value, and measurement times.

After receiving and parsing the response from Digitraffic, the transport data is returned as a T*ransportStatus* object, consolidating all relevant information from multiple stations in the city.

Much of the code handling HTTP requests and parsing was adapted from example code and expanded to ensure robust handling of all required data points.

WeatherService.java

Weather Service has two public methods: *getCurrentWeather*, which gives you the latest weather data in a given city, and *getWeatherAtTime* which gives you the weather information in a given city at a given time. The weather is requested from the API of The Finnish Meteorological Institute (FMI) Open Data.

To get the latest weather information, the program requests FMI’s API for all the measurements from the past day, of the things that Weather Status class

(*WeatherStatus.java*) includes. Then it picks the latest valid measurements if there are any. If the time is specified, the API is only asked for a single measurement of each thing in a Weather Status object.

After getting a response from FMI, the response is parsed, and the weather information is returned as a Weather Status object.

A big part of the code for making a call and parsing a response was taken from Chat GPT's example code, after which the code was cleaned up and expanded for all the required measurements.

3.3.3.1: Unit Test: WeatherServiceTest.java

Purpose: Tests the `WeatherService` class which, this time, gets weather data from an external API.

What's Tested:

> getCurrentWeather_ReturnsWeatherStatus(): Checks that weather data from a sample API response is converted correctly into a `WeatherStatus` object.

Mocks Used: Uses a fake `RestTemplate` to simulate API responses.

PredictionService.java

Prediction Service has one public method: *getCurrentPredictions*, which makes a prediction of the upcoming traffic situation based on the given city's current weather conditions and also compares this to real-time traffic data. The service uses both Transport and Weather services to retrieve needed data from external API's.

The prediction algorithm is still under development as of writing this and is subject to changes. Currently if any of the following weather conditions are met, the prediction marks a predicted delay:

- Wind Speed: Greater than 10 m/s.
- Rain Amount: Greater than 10 mm in 1 hour.
- Visibility: Less than 2 km.
- Snow Amount: Greater than 20 mm.

The base of the code was taken from Chat GPT's example code, but the state-of-the-art algorithm used to predict weather conditions has been developed in-house.

3.3.4. Model layer components

TransportStatus.java

Transport Status object describes the status of transport conditions in a specified city. Supports building, organizing, and displaying the transport data retrieved from the Digitraffic API.

- description (String): A readable description of the transport data, often including the city name and general context, mainly for debugging.
- sensors (List<SensorData>): A list of *SensorData* objects representing individual sensor measurements from stations in the city.

To support data organization, *SensorData* is a helper class designed to encapsulate detailed information for each sensor measurement, allowing *TransportStatus* to manage transport-related data effectively. Each *SensorData* instance within sensors includes:

- sensorId (int): The unique ID of the sensor.
- name (String): The name of the sensor, which describes the type of data it measures.
- shortName (String): A short name or abbreviation for the sensor.
- unit (String): The unit of measurement for the sensor data (e.g., km/h, °C).
- value (double): The actual measurement value recorded by the sensor.
- timeWindowStart (String): The start time of the measurement window.
- timeWindowEnd (String): The end time of the measurement window.
- measuredTime (String): The exact timestamp when the measurement was taken.

WeatherStatus.java

Weather Status object describes the status of weather conditions. It includes getters and setters for the following members:

- description (String): A description of a weather measurement in a readable form, which also includes information about the time of each measurement. Mainly useful for debugging.
- cloudAmount (Double): The amount of clouds
- rainAmount (Double): The amount of rain in 1 hour (mm)
- rainIntensity (Double): The intensity of rain in 10 minutes (mm/h)
- temperature (Double): The temperature (C°)
- snowAmount (Double): The amount of snow (mm)
- visibility (Double): Visibility
- windSpeed (Double): Wind speed (m/s)

3.3.4.1: Unit Test for WeatherStatusTest.java

Purpose: Tests the `WeatherStatus` class which holds weather details.

What is Tested: testWeatherStatusSettersAndGetters(): Confirms that data like description, temperature, and wind speed can be controlled and retrieved correctly.

PredictionStatus.java

The PredictionStatus object represents the status of traffic delay predictions for a specific city and time. It includes getters and setters for the following members:

- city (String): The city for which traffic delay data is provided.
- time (String): The time of the prediction or status check, useful for both current and historical data. Optional.
- isDelayedPrediction (boolean): Indicates whether a delay is predicted in the future based on our in-house delay prediction algorithm.
- isDelayed (boolean): The actual current delay status, showing whether a delay is currently occurring based on our in-house delay algorithm.

Test Frameworks: JUnit 5 and Mockito

The Framework **JUnit 5** is used for its simplicity and modularity and because it offers clear annotations "(`@Test`", "`@BeforeEach`", etc.). It is also well-suited for organizing and running tests efficiently.

Mockito allows us to mock dependencies like `WeatherService` and `RestTemplate` lows us helped us to isolte tests by simulating external interactions. Basically, the smooth integration with JUNIT 5 and its use of annotations (@Mock, @InjectMocksts) was the main reason for choosing these test frameworks.

So far, the unit tests cover the controller (handling user requests), model (storing data), and service (getting data from the API) layers to make sure that the main parts of the app work as expected.

4. APIs

For the development of our application, we selected APIs based on their ability to provide real-time traffic and weather data. After testing several options, we chose the **DigiTraffic API** and **Finnish Meteorological Institute (FMI) Open Data** for their robust capabilities in Finland. Below is a summary of the key features and reasons behind our choices, along with brief mentions of the APIs that were not selected.

4.1. Transportation data: DigiTraffic API

The DigiTraffic API was selected to deliver real-time traffic data in Finland. It provides comprehensive information, including traffic incidents, roadworks, and current traffic conditions. Real-time traffic updates and detailed information on traffic events, such as road closures and traffic density, help users to plan their route effectively.

4.2. Weather data: FMI Open Data

The Finnish Meteorological Institute (FMI) Open Data was chosen for weather data. It provides highly accurate and official weather forecasts, particularly useful for users within Finland. The key features affecting our decision were the accuracy and free access for weather data in Finland.

4.3. Other APIs considered

TransportAPI: Provides detailed British public transport data with real-time updates and flexible data formats, including historical and aggregated data. However, documentation limitations and the necessity of paid plans for most endpoints influenced our decision not to use it.

OpenWeatherMap/OpenMeteo: These APIs offer global weather forecasts with good geographical coverage. *OpenWeatherMap* has higher accuracy but requires a paid plan for advanced features, while *OpenMeteo* is easier to integrate but provides less detailed data.

5. Design solutions and reasonings

The initial design phase of the project included the creation of several preliminary sketches that served as the basis for the development of the application. These first sketches allowed us to visualize the overall structure of the system and facilitated the identification of the key components required. From these sketches, we were able to make more informed decisions about the architecture and functionalities of the software, allowing us to start the project with a clear and well-defined direction. The basic project structure of the backend, as well as a lot of the code on each layer was taken from ChatGPT's example.

We evaluated several front-end frameworks to build the application's user interface. After reviewing the options, we selected the **React** JavaScript library due to its ease of use, strong community support, and ability to integrate various libraries and components. We also had earlier experience with React, and ChatGPT recommended it as a strong choice for the front-end.

For the back-end development, we are using the programming language assigned in the course, which is **Java**. The choice of Java allows it to take advantage of its robustness, portability and extensive support community. In addition, its extensive ecosystem of libraries and frameworks facilitates the implementation of various system functionalities.

We use **Spring Boot** as the framework for the backend of the application, as it simplifies building scalable and maintainable applications.

We use **Maven** as the build and dependency management tool for the project. It automates the management of libraries and dependencies, so that all required packages are easily installed. This facilitates the deployment and management of the application in different environments.

The application architecture follows the Model-View-Controller and Service Layer design patterns.

The *TransportStatus*, *WeatherStatus* and *PredictionStatus* classes represent the data in the backend, constituting the Model component. The React front-end application serves as the View, presenting information to users, while the *TransportController*, *WeatherController* and *PredictionController* function as the Controller, managing the interaction between the user interface and the service layer.

This approach establishes a clear separation of concerns through modular design, enhancing maintainability, scalability, and testability. Dividing back-end code into Controller, Service, and Model layers allows for the encapsulation of business logic (in *WeatherService*, *TransportService*, and *PredictionService*), supporting flexibility for future extensions, such as new API integrations. This layered structure promotes reusability, collaboration, and secure back-end architecture, ensuring a clean, scalable system.

6. Self-evaluation

6.1. Changes made from the prototype submission

During the prototype phase we were planning to do an application using public transport and weather API's. We pivoted away from this idea because we found the general traffic API to be a better one, with more comprehensive information. This did not hinder our progress, as we were already looking into multiple different options during the prototyping phase and immediately after we decided not to use the public transport idea.

The Model-View-Controller design pattern of the application did not change from the prototype phase. Some changes have been made to the front end to accommodate the change from a public transport plus weather application to a general traffic and weather application. We did not end up using the AI-made sketches for the front-end except as a general baseline for how the application could look from a visual point of view. For instance, the lack of historical data from transport APIs ruled out correlation graphs between transport and weather data, which were considered in the prototype phase. A big part of weather data is unavailable in some cities. For example, in Tampere the only pieces of weather data available are temperature and wind speed, which limits the capabilities of showing and using the information.

The MVC design pattern has supported implementation of the application by providing a clear separation between application components and their responsibilities. This has also facilitated the distribution of development tasks between group members. Using the MVC model should make the application expandable, easier to debug and more versatile for external changes. For example, changing the individual external APIs should be easy and shouldn't affect the other components of the application.

The initial solution for front-end structure followed a Lifting State Up pattern common in React applications. In this version, Dashboard component held a shared state *city*, which was passed to its child components as props. The interactions between Dashboard and its children also resembled the Mediator pattern; SearchComponent updated the *city* via a callback to the Dashboard, and it forwarded the updated city to the other children which handled the data-fetching for themselves. However, it was noted that some components would need both weather and transport data in the future, and React Context was considered as a better option for the components to access the data more directly. This centralizes the data-fetching and leads to a simpler data flow between components. The new solution aligns with the Observer pattern, where the DataContext is the subject, and the display and search components act as observers that re-render when the data changes.

Mostly the application should be ready to be finished based on our design documentation and the codebase. The base for most of the code is already there and to gain an idea of how the application works, and what it should do when finished should be possible from the documentation.