COMP.SE.110 GROUP PROJECT: DESIGN DOCUMENT

Tourism planner web application

Group: Binary Brains
Aynur Rahman Talukdar, 150189473
Heidi Seppi, H252889
Lauri Nuutinen, H283219
Ville Kivioja, k424443

Table of contents

1 Introduction

Our group project consists of a web application that combines weather data with data from events happening in the near future. The purpose of the app is to give the user an option to filter events based on event location, event category and user's weather preferences but also give idea of how they should prepare for the event in clothing and otherwise. The original idea for this application was generated by Chat GPT by first giving it the project instructions file and following prompt "Please generate few ideas for the project" of which "Tourism Planner Based on Weather and Events" was selected by the group members.

2 Description of the application

When the user first arrives on the website, they can see different search criteria boxes and a calendar view. User can search for different events by selecting preferred search criteria for location, category and weather. If the user has previously saved their search criteria as a preference, then these will be automatically displayed for the user when they open the website. User needs to have selected something for each criteria and a date from the calendar so that the website automatically searches for events and displays them under the calendar. Users can click different dates on a calendar to display events that match the criteria as a list for that selected day. In case no events which match the criteria can be found for the selected day, information about this is given instead.

Users may open any of the event cards, which are shown as a list below the calendar, to get navigated to event page which displays information about the event and options to view weather information for the location. User can select to view different weather charts for temperature, feels like information, windspeed and precipitation. If any of the weather data isn't available for that date and location, "Couldn't find weather data for given date" is displayed instead of the chart. Users can return to the home page to do more searches using the return button.

3 High level description

First Tourism Planner's Figma prototype is introduced, and screenshots of the actual implementation is shown. After this, high level diagram of the whole application is introduced and frontend and backend components, modules and classes and their dependencies are described.

3.1 Figma Prototype

Before the implementation, we created a Figma prototype to plan out the user interface and how the application could behave.

Figma prototype can be found here:
https://www.figma.com/proto/3eTLZ5ACfWrHWrDT6vSlZb/Project-prototype-(final-version)?node-id=0-1&t=bv5hWxfP8ik0GI5I-1

Here is a screenshot of the prototype if the link above doesn't work.

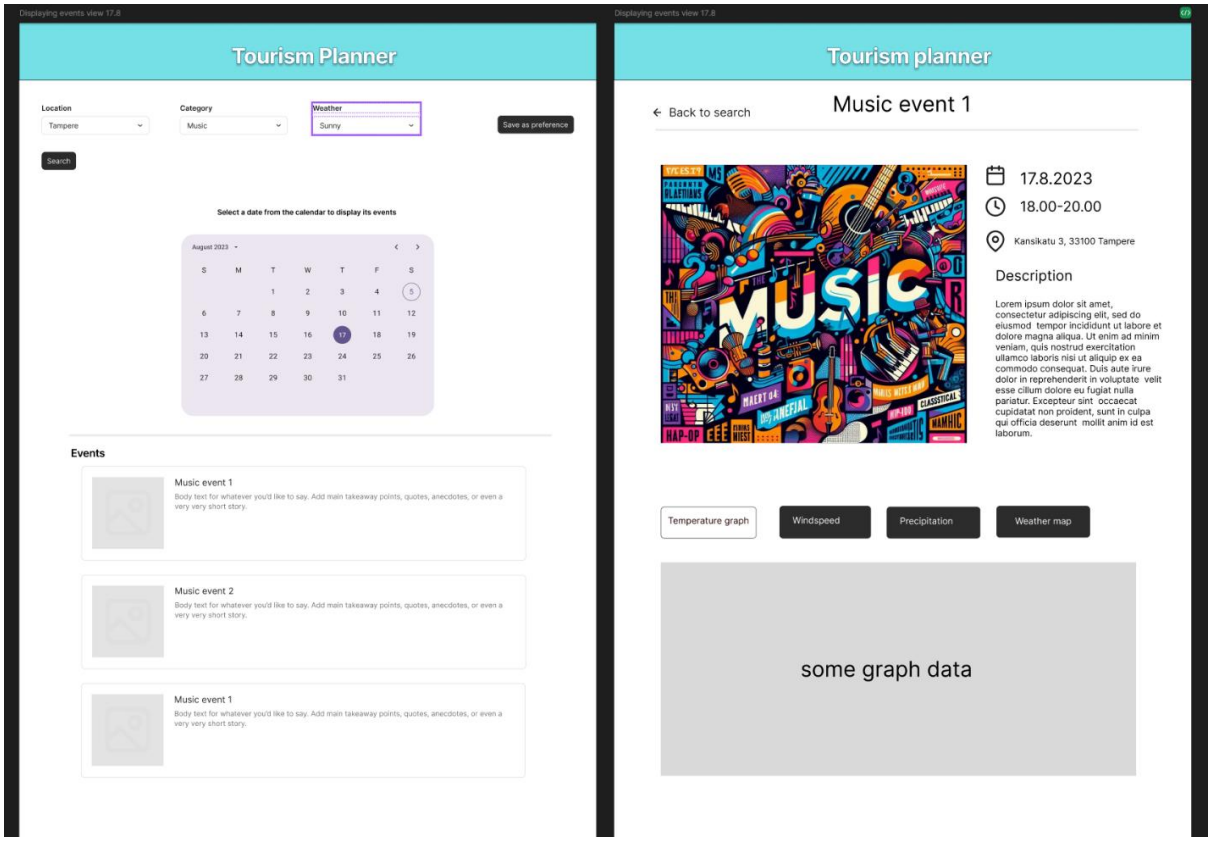

Figure 1: Screenshot of the Figma prototype.

3.2 Implemented user interface

The actual implementation is quite similar to the prototype but we ended up doing some minor changes to it. These changes have been described on the chapter 7 self-evaluation. Below in figure 2 is the home page in which the user can select their preferred search criteria and search for events for different dates. Website displays events which match the criteria under calendar.

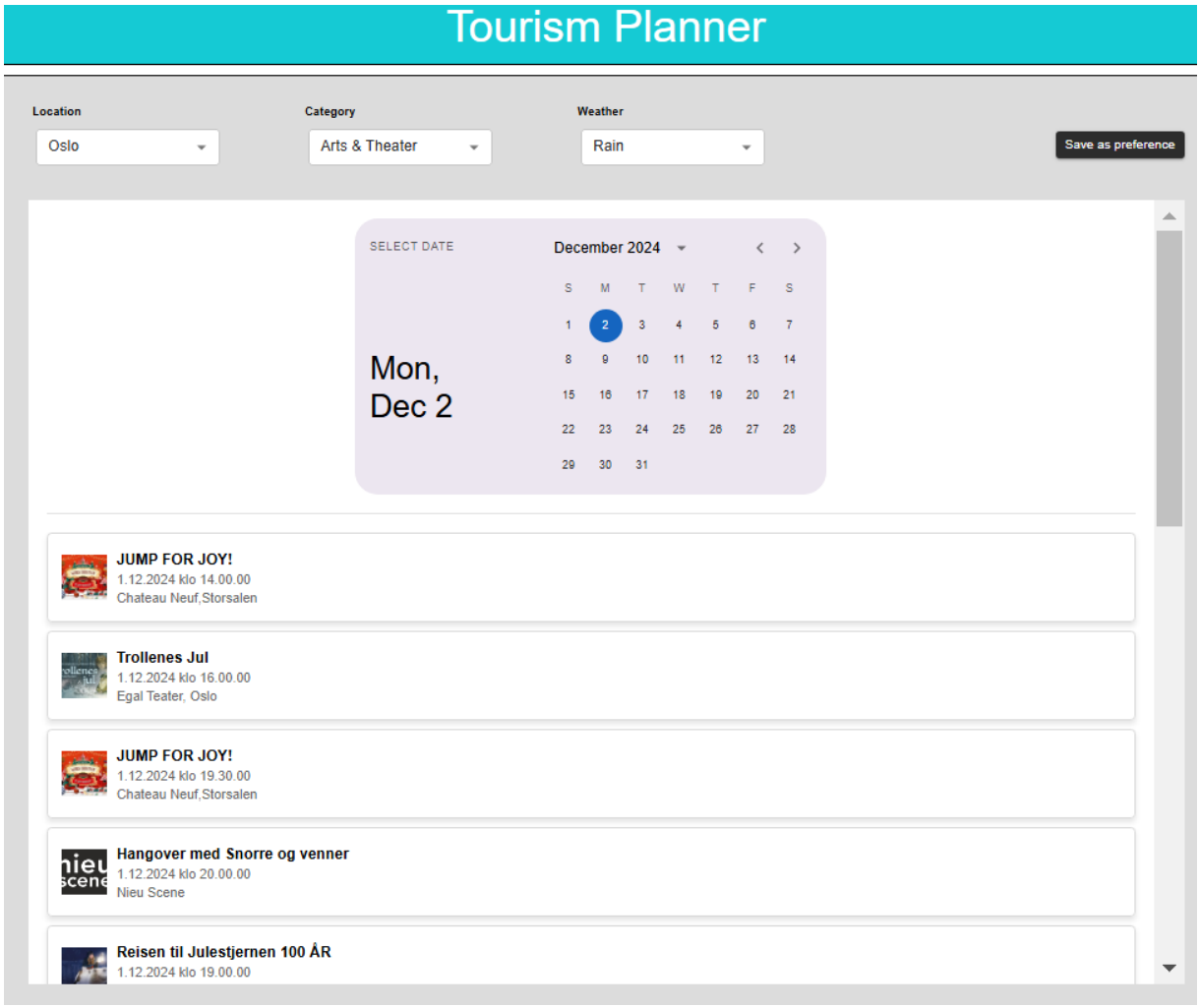

Figure 2: Tourism Planner home page

User can select any of the events to see more details about it. This opens an event page which can be seen below in the figure 3. User can see event information and look at the weather graphs to see what the weather is supposed to be like.

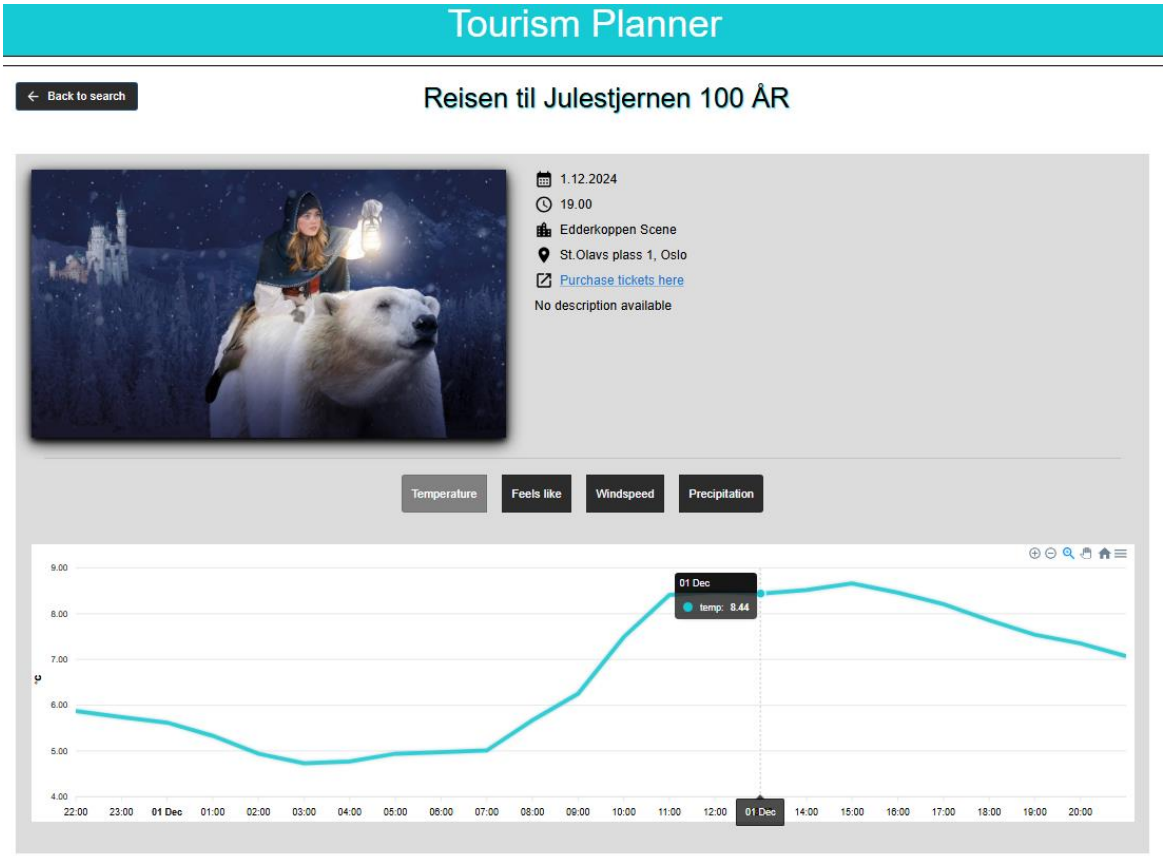

Figure 3: Tourism Planner event page. Example of how one event is displayed.

The sequence diagram below in the figure 4 describes how the user and frontend communicate and how the interaction happens in the application. From the sequence diagram you can clearly see the main use cases: searching for events with different criteria and viewing weather data for each event.

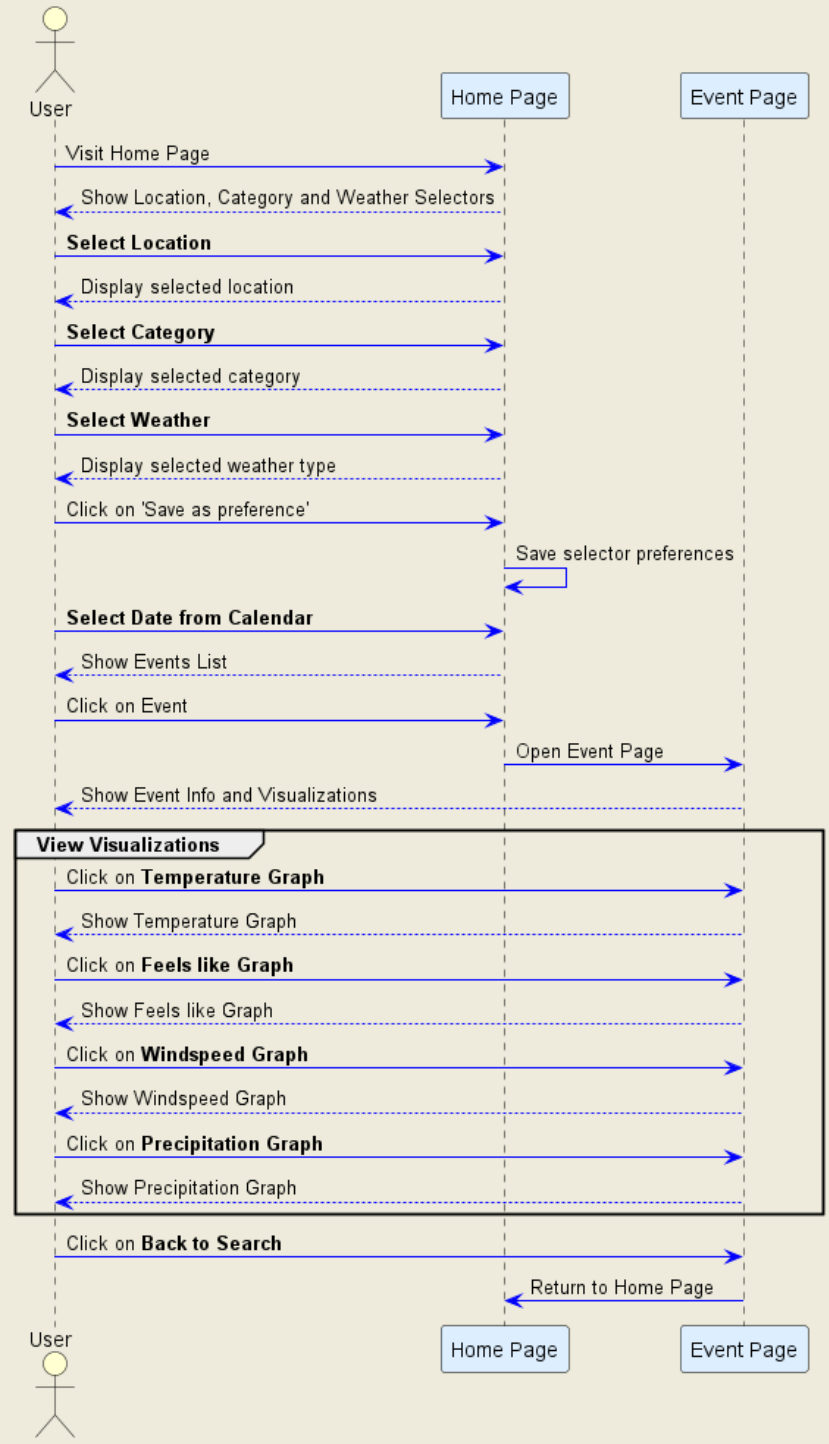

Figure 4: Sequence diagram of frontend.

3.3 General diagram of the whole application

The figure 5 shows generally how the Tourism Planner application works. The application is divided into frontend and backend and backend is responsible for querying data from the Open Apis.

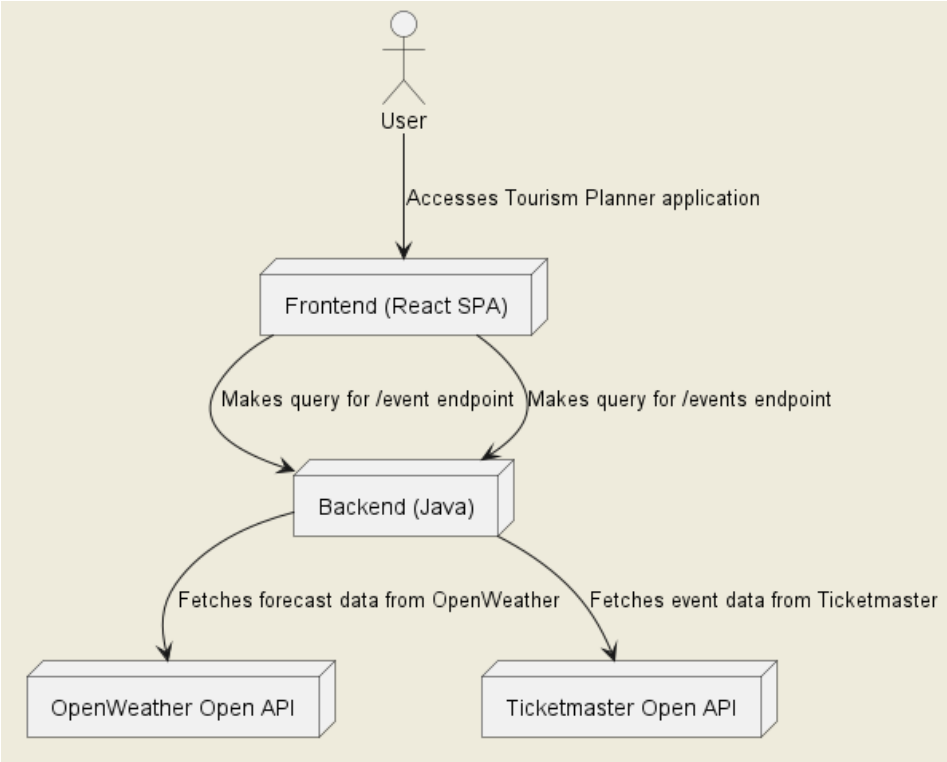

Figure 5: Diagram generally describing the whole application

3.4 Frontend

The frontend is divided into three main portions: user interface components, state management, and service layers.

User Interface Components:

The frontend consists of React components designed to create an interactive single-page application (SPA). The **App** component acts as the entry point, managing routing and navigation between the home page and the event details page. The **Home** component provides the main user interface, including subcomponents like **Select** for search criteria, **Calendar** for date selection, and **EventList** for displaying events that match user preferences. The **EventList** further uses **EventCard** components to represent individual events. The **Event** component handles detailed event views, integrating **EventDetails** for event-specific information and **Chart** for weather visualizations.

State Management:

Global state is managed using Redux Toolkit. The **Store** is configured with slices like **SelectionsSlice**, which manages user-selected criteria such as location, category, weather preferences, and dates. Middleware is used to enhance the state management layer, handling tasks like API logging, caching, and error handling, ensuring smooth interactions and efficient data flow. Saving the use preference is done with setting the preference into local storage so that it persists.

Service Layers:

The **EventsService** is responsible for interacting with the backend to fetch lists of events based on user-defined search criteria. Similarly, the **SingleEventService** retrieves detailed data for a specific event and its associated weather information. These services encapsulate API calls, ensuring that the communication with the backend is abstracted from the rest of the application.

This structured approach ensures a modular, scalable, and responsive frontend, facilitating seamless user interactions and efficient communication with the backend.

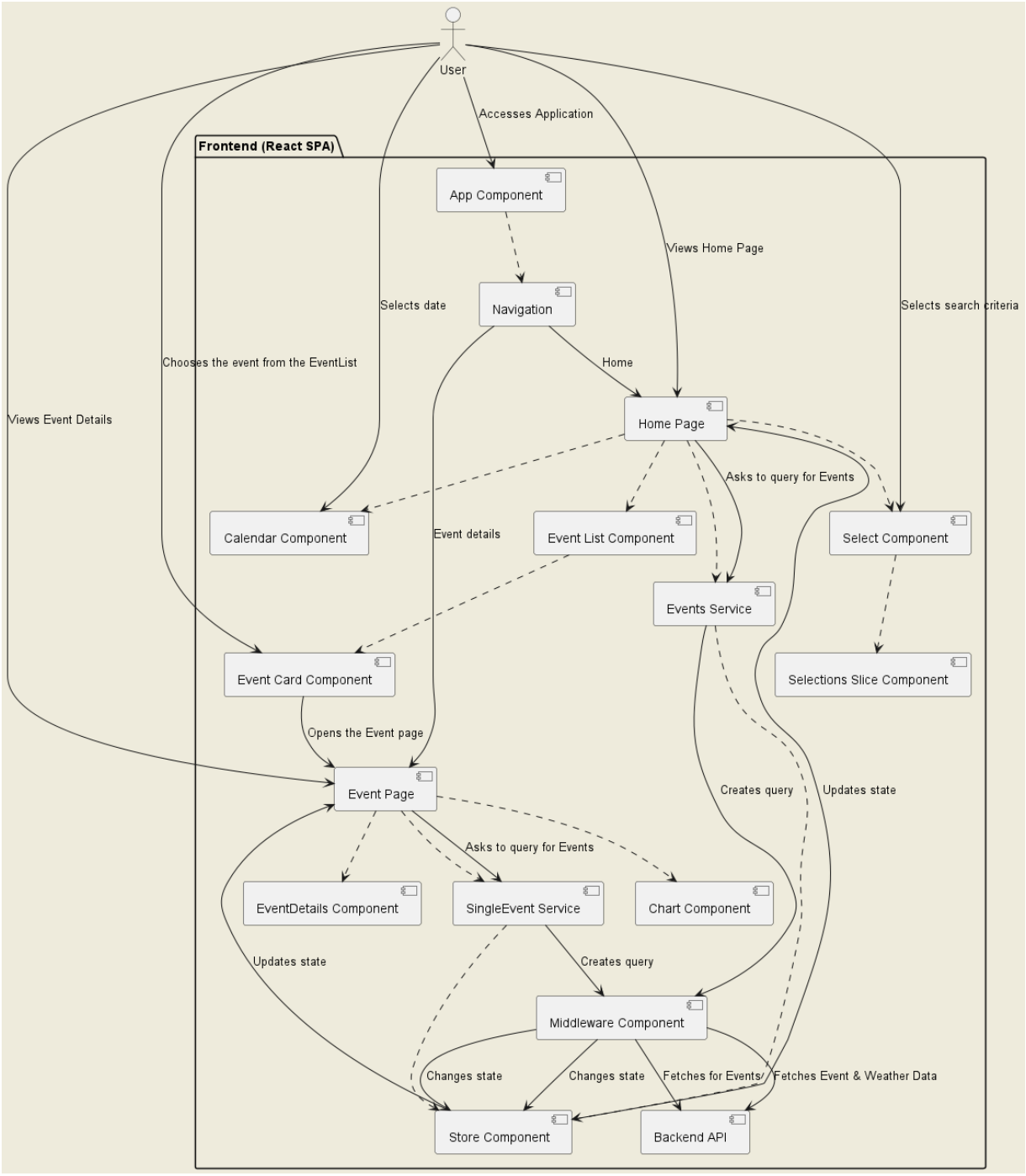

Figure 6: Frontend component diagram

3.5 Backend

The back end is split to three main portions: controller, events and weather.

Controller:

The controller is responsible for receiving requests from the frontend and gathering the requested data by utilizing other backend classes. It uses Spring Boot to listen for GET requests to the endpoints /events and /event on port 8080. Additionally, the controller reads the API keys from the API key files and stores them for use by the EventService and WeatherService.

EventService:

The EventService handles generating a request to the ticketmaster API according to the parameters received from the controller. The EventService class also handles conversions from Json to event objects and back.

WeatherService:

The WeatherService handles generating the requests to the OpenWeather API according to the parameters received from the controller. The API provides both geocoding endpoint and weather data endpoints, first of which is used to make conversion from location names into coordinates that are then used for the second to retrieved weather data. After fetching the data, it is then saved into caches where it can then be retrieved, reducing the need for additional API-calls.

WeatherController:

WeatherController is responsible for aggregation of the needed weather data and providing tools for Controller to filter the events based on weather criteria selected by the user.

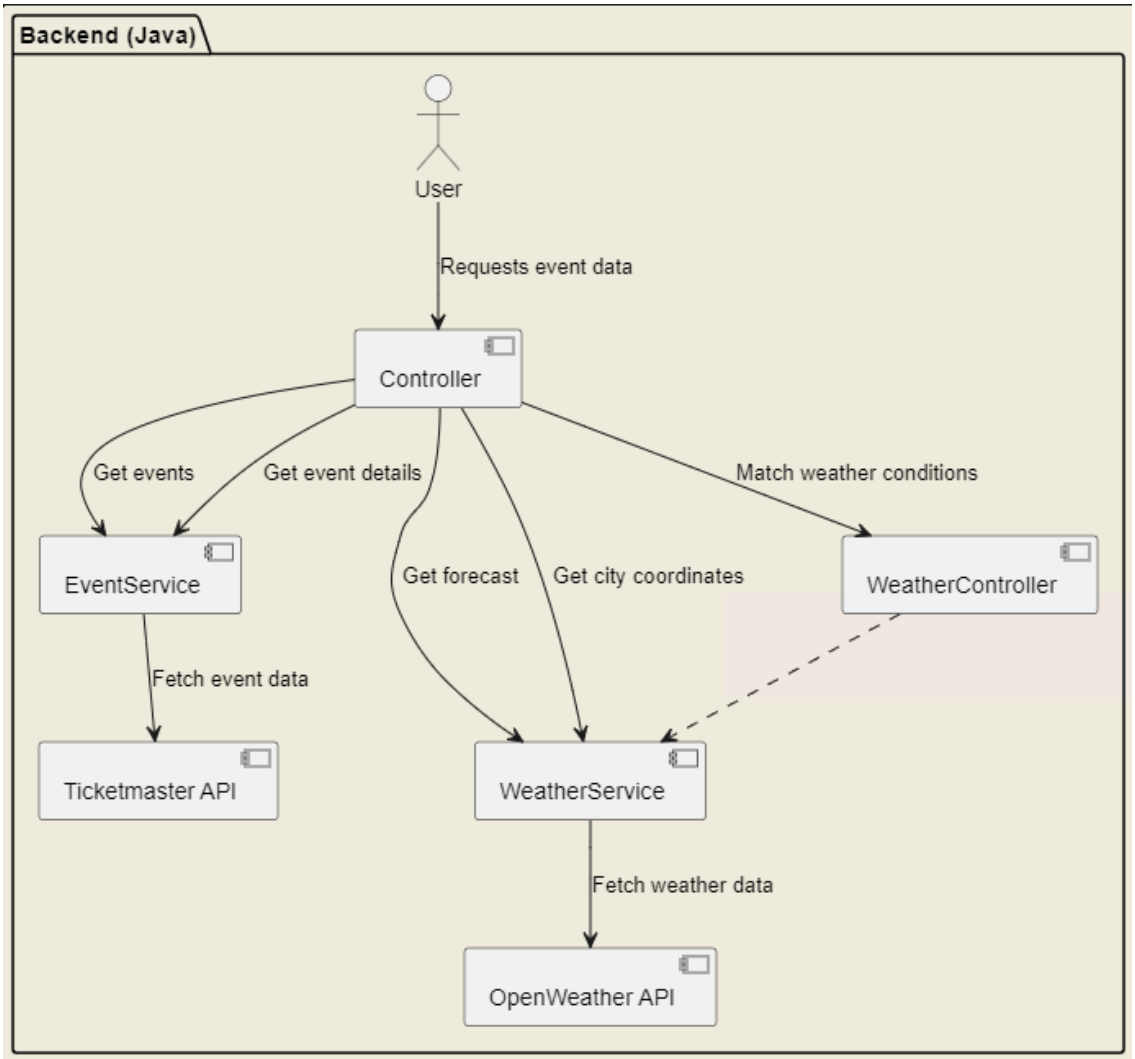

Figure 7: Backend component diagram.

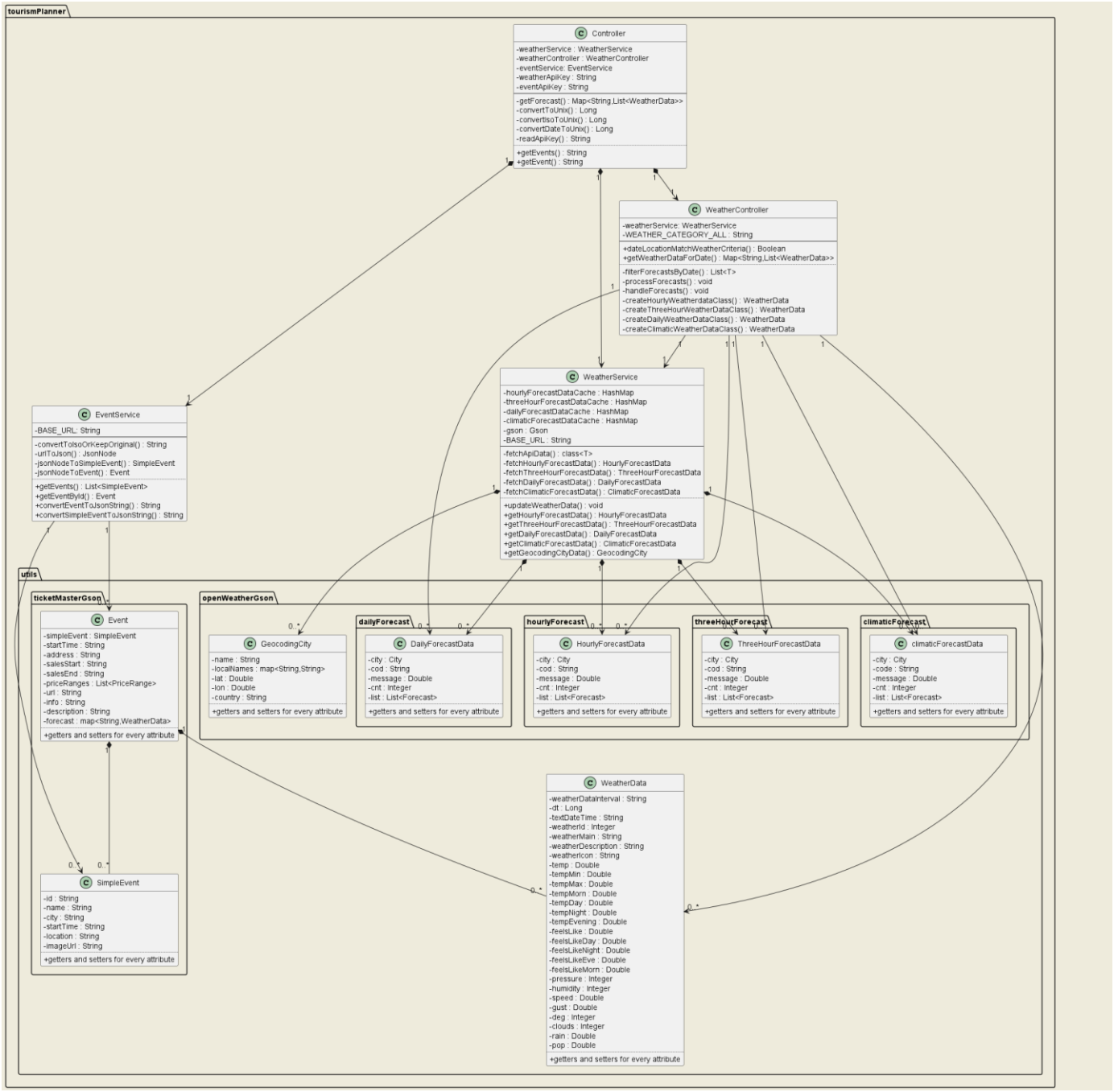

Figure 8: Backend class diagram.

4 Internal interfaces

4.1 Frontend

The frontend consists of various components and services that work together to deliver an intuitive user experience. The **App** component serves as the root, managing routing and navigation between the home and event details pages.

The **Home** component provides the main user interface, including subcomponents like **Select** for search criteria, **Calendar** for date selection, and **EventList** for displaying matching events.

The **Event** component displays detailed information about a selected event, using the **EventDetails** component for event-specific data and the **Chart** component for weather-related visuals. It relies on the **SingleEventService** to fetch event details and associated weather data.

The **EventsService** and **SingleEventService** handle API calls to the backend for retrieving event lists and specific event details, respectively. Global state is managed using Redux Toolkit's **Store**, with slices like **SelectionsSlice** for user criteria and date management. Middleware optimizes performance by handling logging, caching, and error management.

4.2 Backend:

The controller class exposes two interfaces to the frontend, one for getting a list of events matching criteria and one for providing details about a single event.

EventService class has the public methods getEvents, getEventById, convertSimpleEventsToJsonString and convertEventToJsonString. These are called by the controller class to handle operations regarding events.

Weathervice has public methods updateWeatherData, getHourlyForecastData, getThreeHourForecastData, getDailyForecastData, getClimaticForecastData and getGeocodingCityData. WeatherController has public methods dateLocationMatchWeatherCriteria and getWeatherDataForDate. These are called by the controller to handle operations regarding weather data.

Utility classes such as Event and WeatherData are used for holding data and making conversion from data collected from different endpoints into Json that is finally provided to front end as seamless as possible.

5 Design decisions and used patterns

MVC:

In our application we structured our application based on Model-View-Controller-pattern. In our application, the front end is responsible from almost exclusively for handling the changes in the view and taking user input. The input is then sent to backend. Backend consists of Controller classes (Controller and WeatherController) and Service-classes (WeatherService and EventService). The latter of these hold and request the weather-information and event-information from the APIs, while the former manipulate the said data to handle requests from front end and give commands for updating the data when needed. In strict sense it is debatable whether our application currently follows MVC as our Service-classes are also responsible for making the API-requests. However, we believe this adjustment allowed for clearer information flow as now data doesn't have to go through controller classes and then be handed to service classes but allow one-way transfer. The use of MVC allowed for clear separation of concerns between the components, clear mental image of how the data should flow when designing components and is commonly used in web applications.

6 Chosen technologies

Spring boot-framework

The spring boot-framework was chosen on recommendation of Aynur. After taking a short look at the framework it seemed like a good decision. Having used flask with python previously, using the framework felt natural.

React-framework

React framework was selected to develop an interactive single-page application (SPA) where users can seamlessly engage with a Java-based backend. After a brief exploration of React's capabilities, it became clear that its component-based architecture and dynamic rendering would be well-suited for the project. Having prior experience with similar frameworks, working with React felt intuitive and efficient, further reinforcing its suitability for this use case.

Redux RTK

Redux Toolkit (RTK) was chosen to streamline state management and facilitate efficient API integrations for the application. Its built-in tools, such as RTK Query, made it particularly suitable for creating service hooks that manage backend API calls seamlessly. Additionally, its ability to handle global state in a scalable and maintainable manner aligned perfectly with the requirements of the project. With its straightforward

setup and developer-friendly features, Redux Toolkit proved to be an excellent choice for enhancing the application's functionality and maintainability.

Gson

Gson was chosen for implementing weather-classes for its versatility. As the different weather endpoints give different kind of weather information, converting those into Gson-objects that are then saved into maps for any possible future use was seen as more clear option than having different maps for WeatherData -objects, that while may have same object-type offer different kinds of weather data. Another decision affecting factor was the familiarity with Gson-library from the previous courses.

7 Self-evaluation

If you compare the prototype and the actual implementation of the user interface, we did some small adjustments and changes which are described below. Overall, we were able to follow the prototype quite nicely and the application ended up looking and behaving as we planned in the beginning.

- There was no need for a separate search button. Website searches automatically for events when all the search criteria and a date have been selected. This made the user interface much smoother than having the separate search button.
- The home page calendar doesn't highlight days which have events that match the search criteria. This feature was left out because it would have made the event search process for different criteria much slower. Instead, the user can now click the preferred date on the calendar for which they want to search for events.
- Event page has similar information as the prototype describes but the Ticketmaster Open Api didn't offer description text value for most of the events so that's why the description is mostly not available. Similarly, it was also noticed that most of the time there are no end-times for the events so those are left out in the implementation.
- The implemented charts are a bit different than on the prototype. We have now temperature, feels like, windspeed and precipitation charts which were chosen because the OpenWeather Api offered data for these.

8 Responsibilities

We decided to divide project responsibilities so that two people would work on the backend and two people on the frontend. Below is generally listed what each group member was responsible for.

Aynur (frontend): Home page UI, retrieving and modifying data from backend, charts

Heidi (frontend): Event page UI, retrieving and modifying data from backend, charts

Lauri (backend): Controller basic structure, EventService, geocoding and event related utility classes.

Ville (backend): WeatherService and WeatherController –classes and weather related utility classes.

9 Use of AI

The project idea was generated using Chat GPT using the project instructions file and using prompt “Please generate few ideas for the project”. Chat GPT gave loads of different ideas and example open APIs to use and from those we selected “Tourism Planner Based on Weather and Events”. Originally Chat GPT suggested Eventbrite API for the application but after some research, we found that Ticketmaster open API was more useful for us. OpenWeather API was also one of the Chat GPT’s suggestions for getting weather data.

Lauri:

I used Chat GPT for various purposes in this project. Spring boot was a new framework for me, and I started learning it by asking Chat GPT what it does and give some simple examples on how to host a backend using it. GeocodingCity class and reading data into it was made with help from ChatGPT. The Javadoc comments of controller and EventService were written by ChatGPT.

I usually don’t use this much AI in my programming but in this case, it felt useful. I think usage of AI can be very advantageous for troubleshooting or getting very simple portions of code written but when generating more complicated code, the time it takes to properly understand what the AI wrote is usually about the same it would take to just write the code yourself and you’d feel way less shame.

Ville:

The tool I used: Github Copilot

How I used it:

The most common way of use was to get code completions. Especially in parts where there would be similar blocks of code, this would speed up the process greatly. Another thing where completions helped the most was when writing test prints as most of the time Copilot’s best guess was good enough. The second way of use was when I would ask for simple functions such as implementation for converting ISO time into Epoch time or asking equivalent functions or methods to those in other programming

languages, I'm more familiar with such as python. Third use case was finding syntax errors or asking what possible reasons are, why some methods could fail. Finally, I used it to refactor a few functions into more manageable sized ones using java's generics and generating Gson classes and generating test for WeatherController. The first one I did by first selecting the code and then writing prompt "Make the following codeblock modular by using Java generics" and I used this approach on creating generic version for fetching API data from different endpoints (hourly, 3-hour, daily and climatic) and in splitting originally monstrously sized getWeatherDataForDate that I had written myself into multiple smaller parts (filterForecastByDate, processForecasts and handleForecasts). In conversing Json responses into Gson classes I would first give prompt "Generate java gson class based on following Json file". As the result would be all classes being in one file, I would then split this into separate ones manually. Next, I would give prompt "Generate gson Serialized names" for each class-file and finally after that prompt "generate getters and setters". The test for WeatherController was generated by prompt "Generate Unit testss using juniper for WeatherController" and providing the class code. The code generated was the modified as there was error in how copilot tried to initialize mock-version of Weather-class.

I believe that AI tools increase the speed of completing simple or repetitive tasks and therefore allow developers to focus on the larger picture of design. At the same time, they demand a lot of attention from the developer to ensure that the code produced matches the use case and works correctly.

It is paramount that the developer can understand what each line generated will do precisely. Something I find somewhat challenge in using these tools that they create quite convincing illusion of true intelligence, and one can quickly fall into false sense of security, that they can rely on the tool to solve all the issues they may face, which is false. At best I feel these tools help when the use case is comparable to doing quick web searches, giving small snippets of code where it's easy to check any possible errors or on auto correct in word processor programs.

Frontend team: AI was not used while implementing the frontend.