2023

WeatherAPP

DOCUMENTATION 1.2

PHILIPP SCHMID

PHILIPP SCHMID | Flühliweg 34 3624 Goldiwil

Table of Contents

Pitch

Frontend

The WeatherLocation app is there to conveniently display the weather in multiple locations on one single page. The user will be able to customize their list of locations to be displayed so that they can always have a quick glance at all their relevant travel points.

The customization of locations should include adding locations so that they can have as many locations where the weather interests them as they'd like. Of course, just as important as adding is the Possibility of deleting old Locations that are no longer of interest to the user. And for a third option the User will be able to filter their saved locations for various criteria entered by the User so that they can only display the information they truly need.

With this app searching the weather for multiple locations one after the other just to plan a single day's trip is a thing of the past, now you can easily see everything on one page.

Backend

Introducing the WeatherAPP backend—a robust foundation that goes beyond just storing location and weather data. Our secure database not only persists this information reliably but also seamlessly manages mutations requested by the front end, ensuring a dynamic and responsive user experience. Beyond data storage, our backend takes charge of API requests, effortlessly obtaining weather data while expertly navigating potential hiccups with a comprehensive error-handling system. Elevate your weather application with a backend that prioritizes both security and seamless communication.

Use Cases

In this section we will further explain the Use of this app. The reason why some one wants to sue it is to keep an eye on regional weather for daily cross region travel.

1. The user has to be able to add his preferred locations to the displayed list.
2. The user has to be able to delete a location once it is no longer necessary
3. The user must be able to see the Temperature and rainfall for the selected locations
4. The user must be able to search trough his list of cities by label or city name
5. The app needs to get the updated weather daily from the weatherstack API

Table 1 Use Cases

Use case	Actor	Description
Location Create	User	This Use case allows the user to add a new location entry to the location list in the weather app for future display. Once created the location will be stored with in the app for future display
Location Delete	User	This Use case allows the User to remove preciously added locations form their weather app, The user selects a location they want to delete, and the program will remove it from the list so it will no longer be displayed in the future.
Weather View	User	This is the main use case of the app and allows the user to see the weather for all the specified locations at once. The app shows detailed weather information for each location.
Location filter	User	This use case allows the User to only display specific locations from the list.

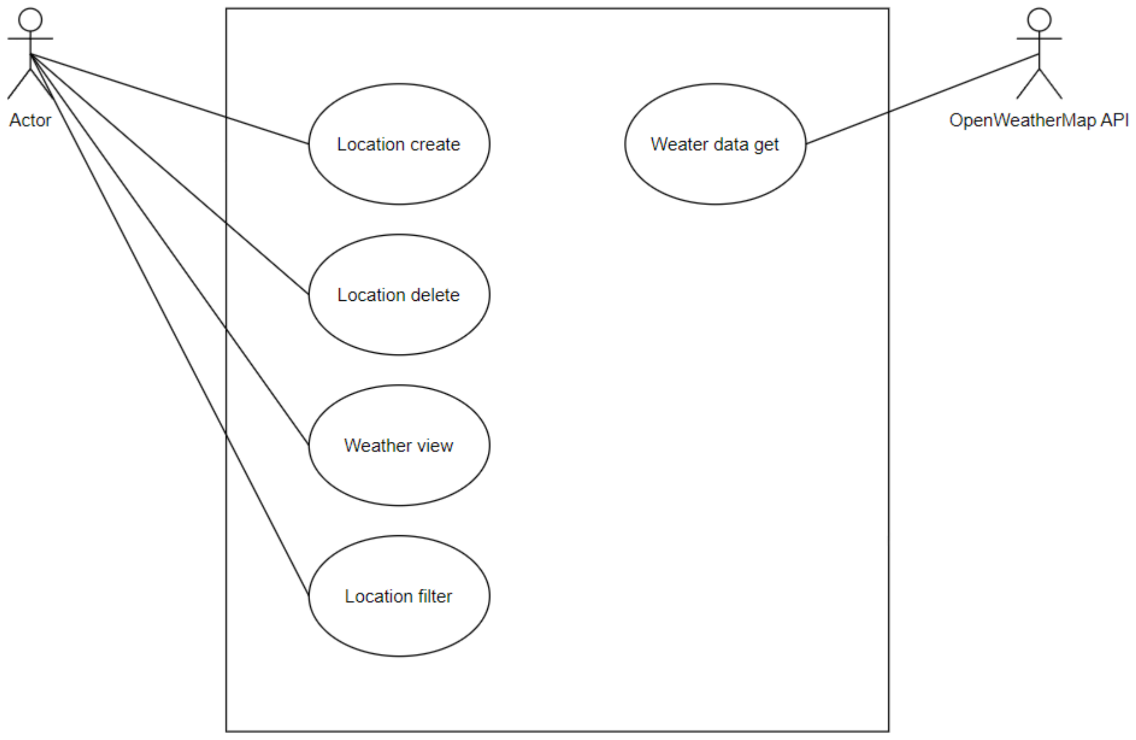


Figure 1 Use Cases

Requirements Catalogue

Functional Requirements

- The user must be able to create new Locations
- The user must be able to delete any and all locations from the list
- The program must be able to save locations
- The user must be able to filter locations by label or by name
- The user must be able to view the Temperature and precipitation in all the saved locations

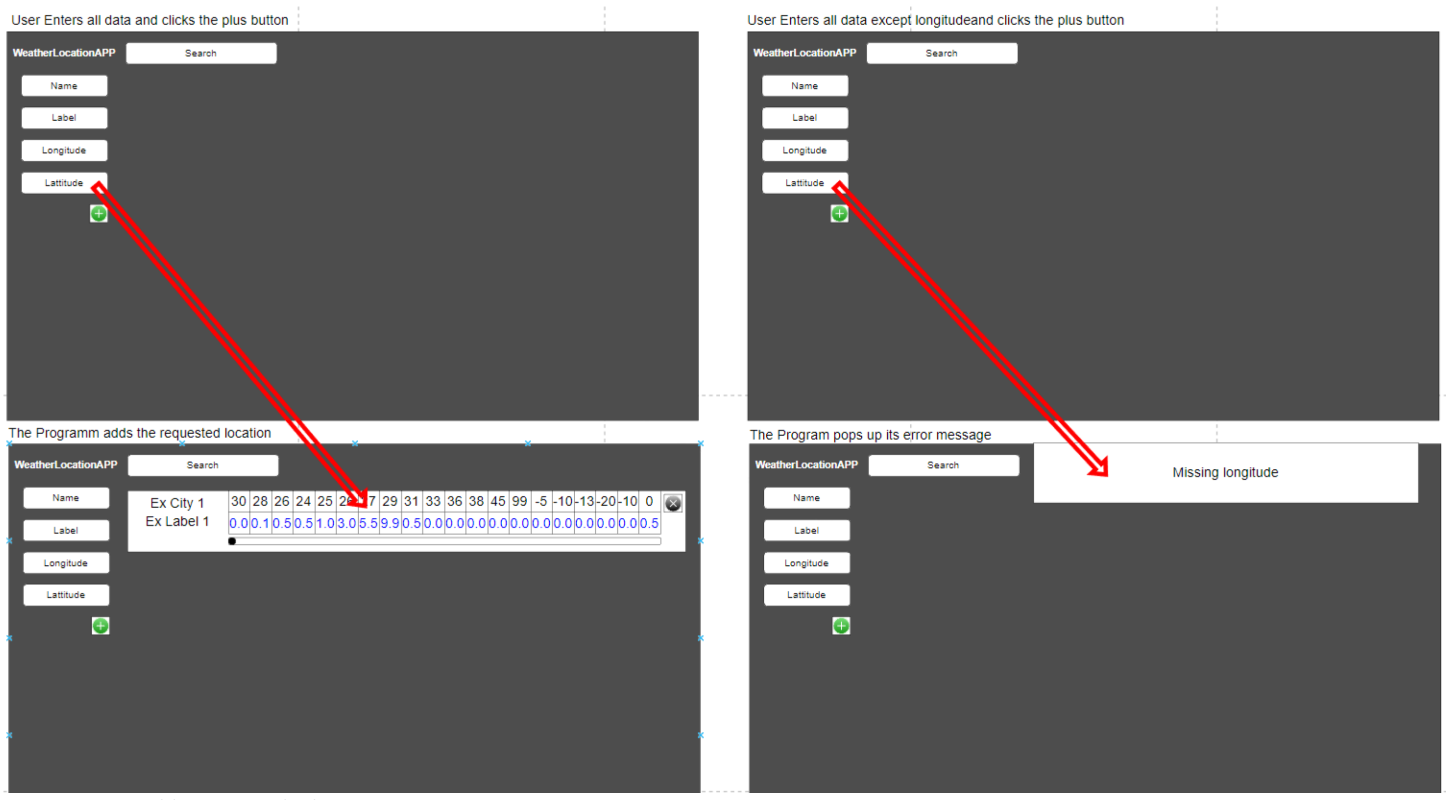
Technical Requirements

- The program must be capable of storing up to 50 Locations that the user can see
- The Program must run on a 1080p resolution
- The website must be displayed on a Firefox

Storyboard

Use case 2

The user can add a location to have the weather displayed, the Program gives feed back if one of the required fields is not filled in. If everything is correct the Program will add dte described Location to the list and display its temperature



User Enters all data and clicks the plus button

User Enters all data except longitudeand clicks the plus button

The Programm adds the requested location

Ex City 1	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
Ex Label 1	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5

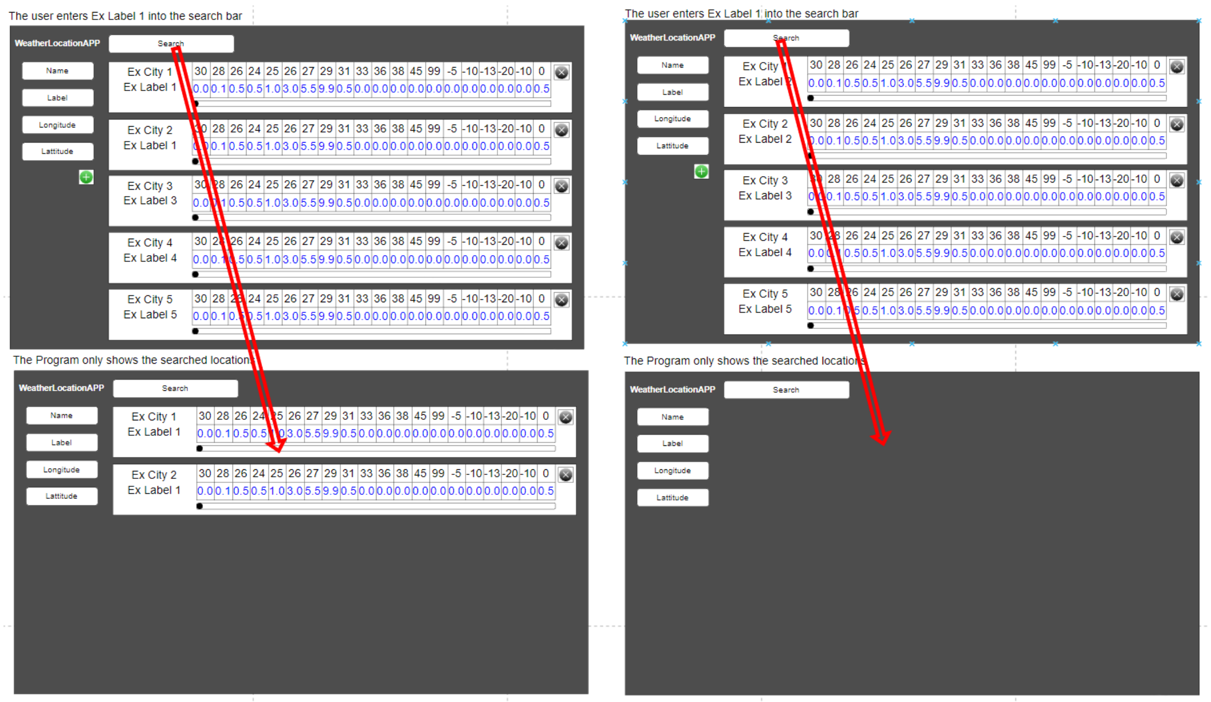
The Program pops up its error message

Missing longitude

Figure 2 User adds a city to the list

Use case 2

The User searches for “Ex Label 1”. The search bar Takes in a String and searches the City names and labels for a matching string regardless of capitalization, the results will be displayed live as the text is being entered. If no Label or city name matches the inserted string then the page should appear empty.



Ex City 1	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
Ex Label 1	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5
Ex City 2	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
Ex Label 1	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5
Ex City 3	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
Ex Label 3	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5
Ex City 4	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
Ex Label 4	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5
Ex City 5	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
Ex Label 5	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5

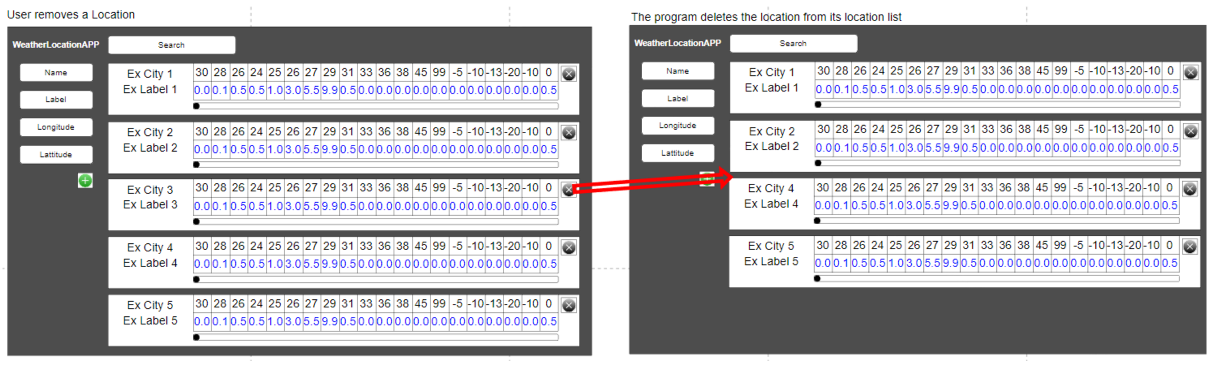
Ex City 1	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
Ex Label 1	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5
Ex City 2	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
Ex Label 1	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5

Ex City 1	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
Ex Label	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5
Ex City 2	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
Ex Label 2	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5
Ex City 3	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
Ex Label 3	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5
Ex City 4	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
Ex Label 4	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5
Ex City 5	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
Ex Label 5	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5

Figure 3 user searches the app

Use case 3

User deletes one of their previously saved locations. Every Location object has a delete button attached to it. This button specifically deletes that item immediately after being pressed. There is no additional query to ask for confirmation as the creation process is rather simple.



User removes a Location

Ex City 1	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
Ex Label 1	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5
Ex City 2	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
Ex Label 2	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5
Ex City 3	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
Ex Label 3	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5
Ex City 4	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
Ex Label 4	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5
Ex City 5	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
Ex Label 5	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5

The program deletes the location from its location list

Ex City 1	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
Ex Label 1	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5
Ex City 2	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
Ex Label 2	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5
Ex City 4	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
Ex Label 4	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5
Ex City 5	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
Ex Label 5	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5

Figure 4 User deletes a city

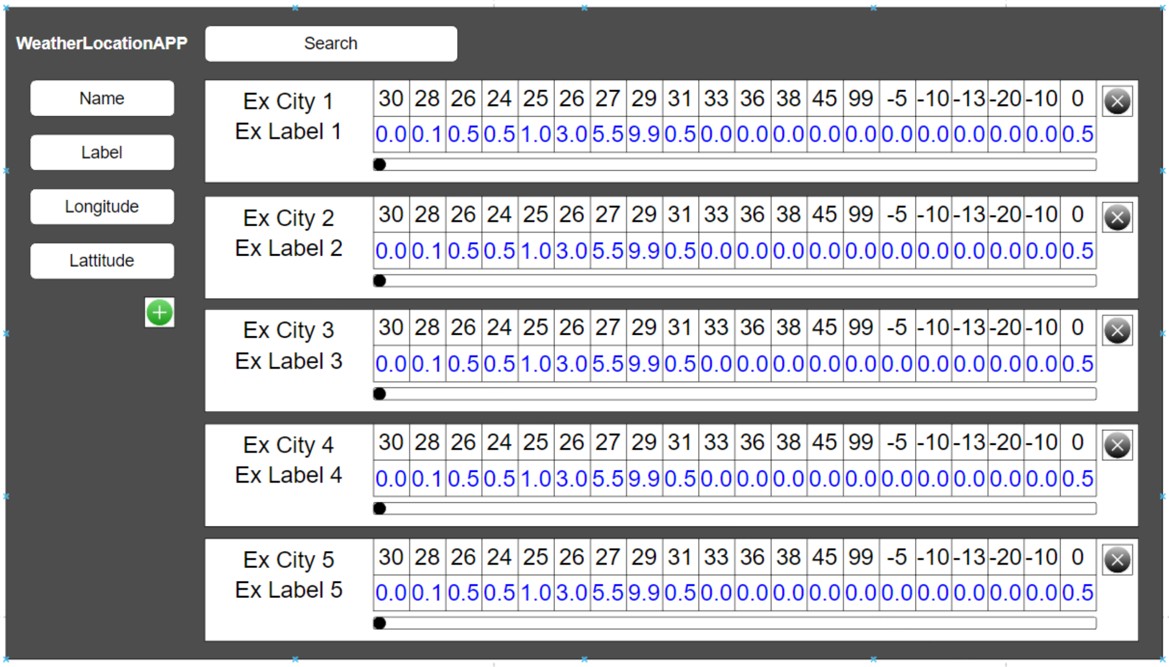
Screen-Mockup

The Mockup shows the completed Product as it has to be.

To the left are insert field that were Name,Label,Longitude,and Lattitude have to be entered This is one React component that then inserts an entry to the city list. All the fields are required. Latitude and longitude must be in the valid on earth existing range -90 to -90 and -180 to 180 respectively.

The city list is the React component next to the insert/ city creation component. This includes a search function that sifts trough all the cities inserted into the City list.

City is its own React component that sits inside the CityList. This element has all the attributes given by the Creation component as well as the weather data for that location delivered by the weatherstackAPI.



Ex City 1 Ex Label 1	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5
Ex City 2 Ex Label 2	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5
Ex City 3 Ex Label 3	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5
Ex City 4 Ex Label 4	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5
Ex City 5 Ex Label 5	30	28	26	24	25	26	27	29	31	33	36	38	45	99	-5	-10	-13	-20	-10	0
	0.0	0.1	0.5	0.5	1.0	3.0	5.5	9.9	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5

Figure 5 Mockup

Interfaces

API-Interaction (old)

This code is a React component called CityList that is designed to display weather data for a list of cities. It utilizes the weatherstackAPI to fetch weather information for each city and allows users to search for cities. Let's break down how it works:

```
 const fetchWeatherData = async (city) => {
      try {
        const response = await axios.get(
          `http://api.weatherstack.com/current?access_key=f2c7de1ae140db82f7d9
cc8898514296&query=${city.latitude},${city.longitude}`
        );

        // Extract relevant weather data from the API response
        const temperature = response.data.current.temperature;
        const rainfall = response.data.current.precip;

        // Update weatherData state with the new data
        setWeatherData((prevWeatherData) => ({
          ...prevWeatherData,
          [city.location]: { temperature, rainfall },
        }));
      } catch (error) {
        console.error(`Error fetching weather data for ${city.location}:`,
error);
      }
    };
```

State Variables:

weatherData:	This state variable is used to store weather data for each city. It's an object where the keys are city locations, and the values are objects containing temperature and rainfall data.
searchTerm:	This state variable is used to store the user's search query for filtering cities.
filteredCities:	This state variable holds the list of cities to display after applying the search filter. Initially, it's set to the same list of cities passed as a prop.

useEffect for Fetching Weather Data:

The first useEffect is responsible for fetching weather data for each city. It runs whenever the cities prop changes (when the component mounts or when the list of cities is updated). Inside this effect, a function called fetchWeatherData is defined. It makes an asynchronous request to the Weatherstack API for the current weather data of a specific city using Axios.

The weather data, including temperature and rainfall, is extracted from the API response and then added to the weatherData state object using the setWeatherData function.

```
// Update weatherData state with the new data
        setWeatherData((prevWeatherData) => ({
          ...prevWeatherData,
          [city.location]: { temperature, rainfall },
        }));
```

If there is an error during the API request, it is caught, and an error message is logged to the console.

```
      } catch (error) {
        console.error(`Error fetching weather data for ${city.location}:`,
error);
```

useEffect for Filtering Cities:

```
  useEffect(() => {
    // Filter the cities based on the search term for location or label
    const filtered = cities.filter(
      (city) =>
        city.location.toLowerCase().includes(searchTerm.toLowerCase()) ||
        city.label.toLowerCase().includes(searchTerm.toLowerCase())
    );
    setFilteredCities(filtered);
  }, [cities, searchTerm]);

  const handleSearchChange = (event) => {
    setSearchTerm(event.target.value);
  };
```

The second useEffect is responsible for filtering the list of cities based on the searchTerm. It runs whenever the cities prop or searchTerm state variable changes.

It filters the list of cities based on whether the city's location or label contains the search term (case-insensitive), and the filtered list is stored in the filteredCities state variable using setFilteredCities.

In summary, this CityList component fetches weather data for a list of cities when it mounts and updates it when the list of cities changes. The weather data for each city is stored in the weatherData state variable, and the filtered city list is displayed based on the search term in the filteredCities state variable.

Front end Requests

Base URL

The backend is standard listening to this address:

```
http://localhost:8080
```

Endpoints

Add a Location

Request

Method: Post

Endpoint: /locations

Example Request Body:

```
{
    "location": "CityName",
    "label": "CityLabel",
    "latitude": 40.7831,
    "longitude": -73.9712
  }
```

Response

Status Code: 200 OK/500 internal Server Error

Body: The newly created location object.

```
{
    "id": 4,
    "location": "CityName",
    "label": "CityLabel",
    "latitude": 40.7831,
    "longitude": -73.9712
}
```

/

```
  Error: WeatherStack API response does not contain valid weather data.
Location entry deleted.
```

Retrieve All Locations

Request
Method: Get
Endpoint: /locations

Response
Status Code: 200 OK
Body: A list of all locations.

```
  [
    {
        "id": 4,
        "location": "CityName",
        "label": "CityLabel",
        "latitude": 40.7831,
        "longitude": -73.9712
    }
]
```

Refresh Weather Data

Request
Method: Get
Endpoint: /locations/refresh

Response
Status Code: 200 OK
Body:

```
Refresh Successful
```

Retrieve Weather Data for a Location

Request
Method: Get
Endpoint: /weatherdata/*locationId*
Parameters: locationId

Response:
Status Code: 200 OK/
Body:

```
[
    {
        "id": 2,
        "location": {
            "id": 4,
            "location": "CityName",
            "label": "CityLabel",
            "latitude": 40.7831,
            "longitude": -73.9712
        },
        "temperature": 13.0,
        "rainfall": 0.0
    }
]
```

Delete a Location (and associated data)

Request
Method: Delete
Endpoint: /locations/*locationId*

Parameters: *locationId*

Response
Status Code: 200 OK/404 not found
Body:

```
Deletion successful for location with ID 4
```
/
```
Error: Location with ID 2 not found.
```

Backend Request API Interaction

WeatherService Class

The WeatherService class is responsible for interacting with the WeatherStack API to retrieve current weather data for specified locations. It uses the Spring Framework's `RestTemplate` for making HTTP requests.

Dependencies

- **LocationRepository**: A repository for managing location data.
- **WeatherDataRepository**: A repository for managing weather data.
- **RestTemplate**: A Spring framework class for making HTTP requests.

Properties

- **weatherStackAccessKey**: An API key for accessing the WeatherStack API. It is injected using the `@Value` annotation.

Methods

`updateWeatherData(Location location)`

This method updates weather data for a given location.

1. Constructs the WeatherStack API URL with the provided API key and location coordinates.
2. Uses `RestTemplate` to make a GET request to the WeatherStack API.
3. Handles different HTTP response scenarios:
   - If the response is successful (2xx) and contains valid data, it saves the weather data to the database.
   - If the response is successful but does not contain valid data, it deletes the location entry and throws a `WeatherDataRetrievalException`.
   - If the response indicates a client error (4xx) or server error (5xx), it deletes the location entry and throws a `WeatherDataRetrievalException`.
   - If the request to the WeatherStack API fails for any other reason, it deletes the location entry and throws a `WeatherDataRetrievalException`.

`refreshAllWeatherData()`

This method refreshes weather data for all locations in the database.

1. Retrieves all locations from the database.
2. Iterates over each location and attempts to update its weather data using the `updateWeatherData` method.
3. If an exception occurs during the update, it is caught, and the process continues to the next location.

Exception Handling

- **WeatherDataRetrievalException**: A custom exception class extending `RuntimeException`. It is thrown when there are issues retrieving weather data, and it includes a descriptive error message.

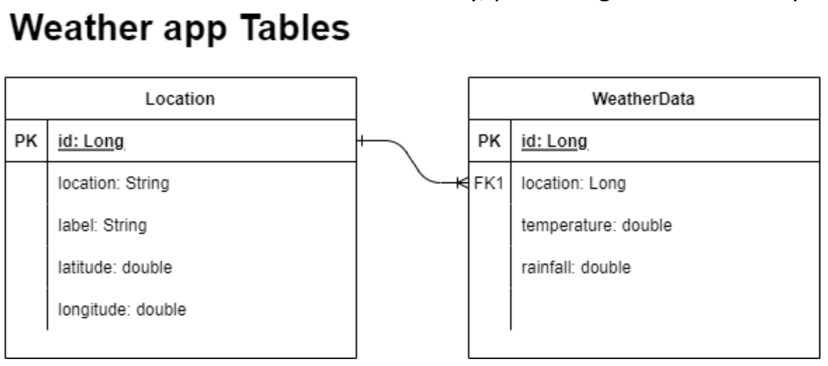
Note

- The code follows a defensive strategy, deleting the location entry in case of any issues with the WeatherStack API to maintain data integrity.
- The use of Spring's `RestTemplate` simplifies the process of making HTTP requests to the WeatherStack API.

Code Structure

Entity Relationship Diagram:

The WeatherAPP backend operates on a dual-table system, managing the relationships between entities. The first table records various locations for which users seek WeatherData. Each entry in this Location Table becomes a reference point for the application's weather insights. The second table is populated by drawing on the locations archived in the Location Table and serves as the repository for WeatherData acquired through requests to the weather stack API. Each entry in the Weather Data Table is linked to a singular location. However, the system is designed with the foresight that each location may have multiple weather data entries, a feature yet to be fully realized due to the constraints of our current free API approach. The intricacies of these entity relations form the backbone of the WeatherAPP's functionality, promising a robust and dynamic user experience.



ERD 6

Classes

1. **Application**

   1.1. This is the main class with the main method that serves as the entry point for the application.

2. **AppConfig**

   2.1. Configuration class responsible for defining beans, such as the RestTemplate bean.

3. **Location**
    3.1. Represents a location entity with attributes like id, location, label, latitude, and longitude.
    3.2. It is annotated with @Entity, indicating it is a JPA entity.
4. **LocationController**
    4.1. RestController responsible for handling HTTP requests related to locations.
    4.2. Has dependencies on LocationRepository, WeatherDataRepository, and WeatherService.
5. **LocationRepository**
    5.1. Extends JpaRepository for CRUD operations on Location entities.
6. **WeatherData**
    6.1. Represents weather data associated with a location.
    6.2. It has a Many-to-One relationship with Location (annotated with @ManyToOne and @JoinColumn).
7. **WeatherDataRepository**
    7.1. Extends JpaRepository for CRUD operations on WeatherData entities.
    7.2. Provides custom query methods to find weather data by location and location ID.
8. **WeatherDataController**
    8.1. RestController responsible for handling HTTP requests related to weather data.
    8.2. Depends on WeatherDataRepository.
9. **WeatherService**
    9.1. Service class responsible for updating and refreshing weather data.
    9.2. Depends on LocationRepository, WeatherDataRepository, and RestTemplate.

Dependencies

spring-boot-starter-data-jpa: Provides a set of dependencies to work with Spring Data JPA, which simplifies the implementation of data access layers.

spring-boot-starter-web: Includes dependencies to set up a basic web application with Spring Boot, including the Spring MVC framework.

mysql-connector-java: Enables connectivity to a MySQL database by providing the JDBC driver (mysql-connector-java). This dependency is essential for Spring Boot applications that interact with MySQL databases.

spring-boot-starter-test: Includes dependencies for testing Spring Boot applications. It provides testing tools, such as JUnit and other testing libraries, to facilitate unit and integration testing.

Test Plan

This Test plan is for white box testing the finished product, To ensure functionality and handling of edge cases.

Frontend

Test 1

ID / Name	T-001	All ok
Description	In this Test the User enters correct data.	
Test requirements	The program must be installed Npm start has been successful The program is displayed in Firefox	
Test Steps	1. Enter the name “Thun” 2. Enter the tag “Work” 3. Enter coordinates of Thun copied from google maps 46.764051849039205, 7.624559011672853 4. Click submit	
Expected Result	The Program should add the Location to the list and lets you filter them by either.	

Test 2

ID / Name	T-002	Edge Case
Description	The test user enters valid coordinate values at the edge of the acceptable range	
Test requirements	The program must be installed Npm start has been successful The program is displayed in Firefox	
Test Steps	1. Enter the name “Edge” 2. Enter the tag “Case” 3. Enter in the coordinates 90,180 4. Click submit	
Expected Result	Weather data the app lets this entry trough with out alert	

Test 3

ID / Name	T-003	Out of bounds
Description	The test user enters invalid coordinates values	
Test requirements	The program must be installed Npm start has been successful The program is displayed in Firefox	
Test Steps	1. Enter the name “Edge” 2. Enter the tag “Case” 3. Enter in the coordinates 91,180 4. Click submit	
Expected Result	The program will send out an alert to the user giving information as to the proper formatting of the coordinates. As well as log an entry in the browser console	

Test 4

ID / Name	T-004	Filter Name
Description	The user utilizes the powerful Filter function to sort trough their locations	
Test requirements	The program must be installed Npm start has been successful The program is displayed in Firefox T- 001 Completed successfully	
Test Steps	1. The user enters letters into the search field 2. T 3. H 4. O	
Expected Result	Step 2 and 3 should still display the previously created location THUN but once step 4 is fulfilled the location must disappear from the list.	

Test 5

ID / Name	T-005	Filter Label
Description	The user utilizes the powerful Filter function to sort trough their locations	
Test requirements	The program must be installed Npm start has been successful The program is displayed in Firefox T- 001 Completed successfully	
Test Steps	1. The user enters letters into the search field 2. W 3. O 4. K	
Expected Result	Step 2 and 3 should still display the previously created location THUN but once step 4 is fulfilled the location must disappear from the list.	

Test 6

ID / Name	T-006	Delete
Description	The user Deletes an entry in the city list	
Test requirements	The Program must be installed Npm start has been successful The program is displayed in Firefox T- 001 Completed successfully	
Test Steps	1. Click delete on the Thun Location	
Expected Result	The program removes Thun from the list of displayed places with out any error messages.	

Backend Testing

The Backend of this application can be tested Separately, first of all there are automated Unit tests available that ensure the functionality of the responses given to valid and invalid requests sent by the frontend. This same functionality and also be simulated by Insomnia for which these manual Tests are. All of the Tests requite to have imported “FrontEndSim.json” to Insomnia

Test 1

ID / Name	T-001	BadLocation
Description	The User attempts to add a location to the weather app for which no data exists in the API (too remote)	
Test requirements	Application Running on Localhost:8080	
Test Steps	Run the AddLocationBad post request	
Expected Result	API Response Error: WeatherStack API response does not contain valid weather data. Location entry deleted.	

Test 2

ID / Name	T-002	AddLocation
Description	The Test User enters a valid location	
Test requirements	Application Running on Localhost:8080 T-001	
Test Steps	Run the AddLocation post request	

Expected Result	200 OK
	{
	"id": N,<-number depending on DB Prior use (2 if new)
	"location": "CityName",
	"label": "CityLabel",
	"latitude": 40.7831,
	"longitude": -73.9712
	}
	Both Tables have added a corresponding entry	

Test 3

ID / Name	T-003	getAllLocations
Description	The Test simulates the front end sending a request for information and getting an answer	
Test requirements	Application Running on Localhost:8080 T-002	
Test Steps	Run the getAllLocations get request	

Expected Result	[
	{
	"id": 2,
	"location": "CityName",
	"label": "CityLabel",
	"latitude": 40.7831,
	"longitude": -73.9712
	}
	]
	The backend sends back the full data set in the location table	

Test 4

ID / Name	T-004	refreshWeather Data
Description	The Front end Application offers the user the possibility to initiate an update of the current weather data	
Test requirements	Application Running on Localhost:8080 T-003	
Test Steps	Run the refreshWeatherData request	
Expected Result	Refresh Successful The backend application has resent information requests to the Used API and updated the weather data table corresponding to the saved locations	

Test 5

ID / Name	T-005	getWeatherDataForLocation
Description	The front end needs to be able to read out the weather data for each of the locations chosen by the User	
Test requirements	Application Running on Localhost:8080 T-004	
Test Steps	Run the getWeatherDataForLocation get request	
Expected Result	`[ { "id": 1, "location": { "id": 2, "location": "CityName", "label": "CityLabel", "latitude": 40.7831, "longitude": -73.9712 }, "temperature": 15.0, "rainfall": 0.0 }]`	

Test 6

ID / Name	T-006	deleteLocation
Description	The user Deletes an entry in both tables of the weatherapp DB	
Test requirements	Application Running on Localhost:8080 T-005	
Test Steps	Run the deleteLocation delete request	
Expected Result	Deletion successful for location with ID 2 The program has removed the location with the specified ID from the list	

Test 6

ID / Name	T-007	deleteLocation2
Description	The user Deletes an entry in both tables of the weatherapp DB	
Test requirements	Application Running on Localhost:8080 T-006	
Test Steps	Run the deleteLocation delete request again as in T-006	
Expected Result	Error: Location with ID 2 not found. The Program has sent back an error indicating that no such ID exists within the DB.	

Tested System Requirements

The system has been tested on the following configuration:

- Operating System: Windows 11 Home
- Browser: Firefox 117.0.1
- Database: mySQL
- Backend Tool: Insomnia (standalone)

Installation instructions

1. Download WeatherApp:
   1.1. Download the WeatherApp zip file.
2. Unpack the Zip File:
   2.1. Unpack the downloaded zip file into your preferred folder.
3. Frontend Setup:
   3.1. Navigate to the 'src' folder.
   3.2. Open the Windows Terminal in that folder.
   3.3. Run `npm install` to ensure all relevant files are up to date.
   3.4. Wait for the installation to complete.
   3.5. Run `npm install axios`.
   3.6. Wait for the installation to complete.
   3.7. Type `npm run` to start the website.
   3.8. The website should now be open in your preferred browser.
4. Backend Configuration:
   4.1. Execute the SQL script provided in the 'APP' folder with mySQL.
   4.2. Use the command `source /path/to/create_weatherapp.sql`.
   4.3. After successful implementation of the User and DB, import the Maven project into your preferred IDE (Developed in Eclipse EE).
   4.4. Run the ClassApplication, Spring boot should now run on Port 8080 and is ready to be interacted with
5. Insomnia Setup:
   5.1. Import the file 'FrontEndSim.json' into your Insomnia.
   5.2. Start emulating the requests that could be sent by the frontend.

Figure Index

Table Index

References

W3School – Used for code references and to iron out details.

ChatGPT 3.5 – Used for code references and analysis and refinement.

Hugo Lucca – Teacher of the class.