Weather Guru - Group 5 Final Report

By Carolyn Owen, Geoffrey Vissers, Matthew Stewart, Sebastien Godbout
For CS4471A Software Design and Architecture
Prof. Nazim Madhavji
Computer Science Dept. - Western University

0 - Table of Contents

1 - Introduction

Weather Guru is a streamlined weather data and alert system hosted on IBM cloud using the publisher/subscriber model. It will provide clients with digestible qualitative and quantitative representations of past, present, and future weather data and predictions, as well as a severe weather event alert system to notify clients of potentially treacherous conditions. In this report, we will outline the requirements, documentation, implementation, and testing of our architecture. For our system, we are using a Service Oriented Architecture (SOA) consisting of three services: a weather email alert service, historical weather data comparison service, and general weather forecast service. These services together will provide users with a complete blanket of baseline weather data to use in their day-to-day lives.

The weather email alert service will provide subscribers with a formatted html email alert, warning of severe weather events such as rain, hail, tornados, or thunderstorms. The alerts will outline details of the event such as the event type, duration, severity, and location. The historical weather data comparison service provides users with the ability to compare weather data between two subscribed cities. The general weather forecast service provides users with a representation of present weather data for a user's subscribed cities.

2 - Requirements

When coming up with the requirements for the system, we brainstormed what users of the system would find most appealing and useful in a weather alert/forecast application. As we had no stakeholders for our project, all of our functional and quality attribute requirements came from inside of our group.

Our thought process for the functional requirements of our system consisted of us recounting personal experience in using weather apps and what features we found would be most important to have for our services. We decided on a few general, useful functional requirements per service as shown in the table below.

For our quality attribute requirements, we thought about user expectations for things such us performance (latency between requests and outputted data), security (password encryption), reliability (can the users rely on our application to be useful whenever they desire), and etc. We came up with a few quality attribute requirements for each service as shown in the table below.

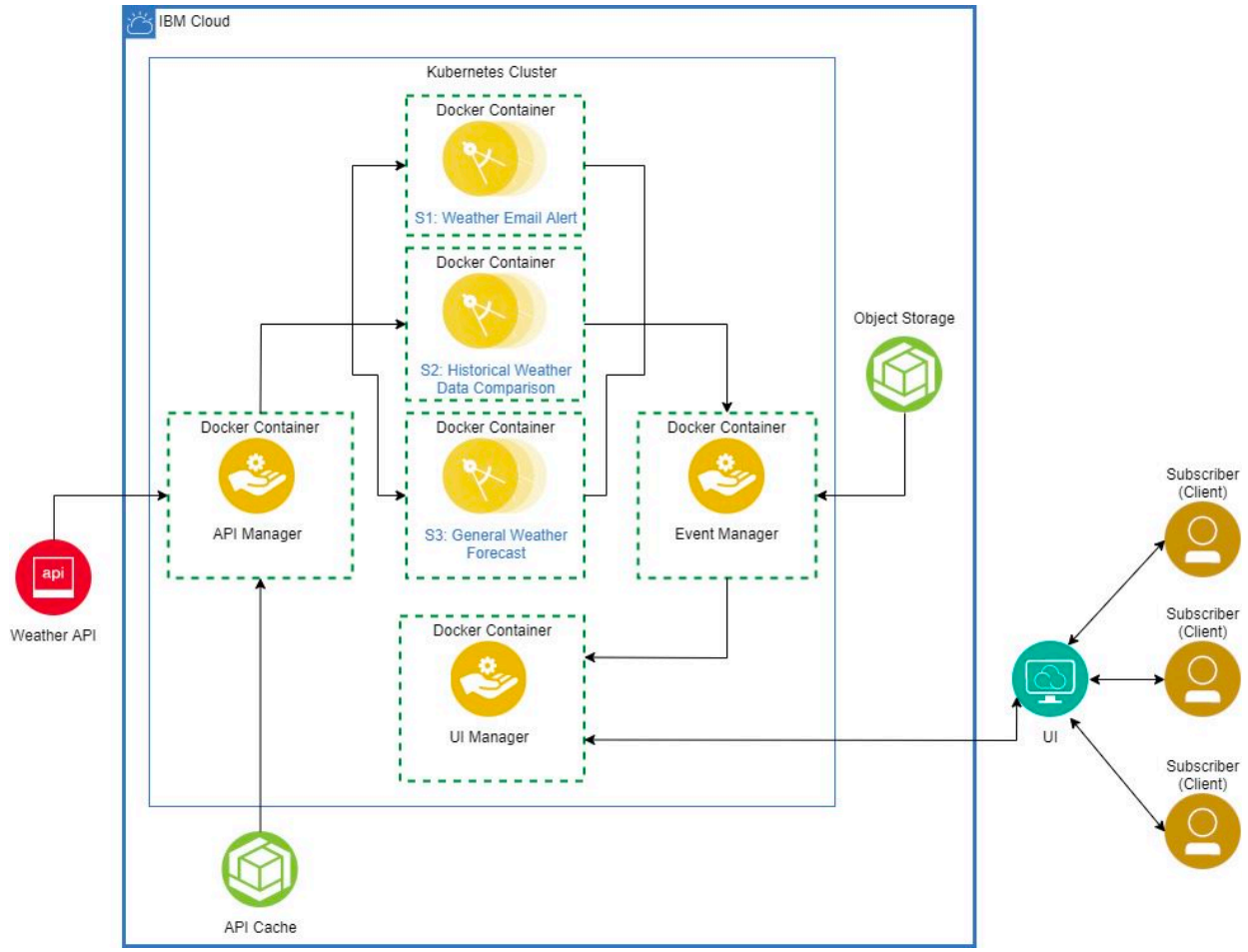
Service	Functional Requirements (FR)	Quality Attribute Requirements (QR)
S1: Weather Email Alert	FR1: User can subscribe to rain, hail, tornado, or thunderstorm weather events and receives alerts via e-mail when the chosen event is forecasted in their chosen location within the next 24h with over 50% probability.	QR1: Alerts are sent on a subscription priority basis (higher tier subscribers are sent alerts at a higher priority than lower tiers). QR1.1: An email to a gold subscriber will be sent within 5 minutes of the system becoming aware of the event. QR1.2: An email to a silver subscriber will be sent within 10 minutes of the system becoming aware of the event.
	FR2: E-mail will specify the following details of the forecasted event: Event Type, Duration, Severity, Location.	QR2: Alerts/Updates are sent on a more frequent basis for higher tier subscribers. QR2.1: The system will check for forecasted events and notify gold subscribers every 3 hours. QR1.2: An email to a silver subscriber will be sent within 10 minutes of the system becoming aware of the event.
	FR3: The user can subscribe and unsubscribe to events at any time.	QR3: The system will not send duplicate alerts to the same subscriber for the same weather event forecast.
	FR4: The user can subscribe and unsubscribe to alerts for different cities at any time.	QR4: The user will be able to subscribe to alerts for several cities determined by their subscription QR4.1: A gold subscriber can subscribe to events in up to 5 cities QR4.2: A silver subscriber can subscribe to events in 1 city
	FR5: The user can change their email address at any time.	QR5: The user can subscribe to several event alerts dependent on their subscription level QR5.1: A gold subscriber can subscribe to alerts for up to 10 events in total (across any cities that they are subscribed to) QR5.2: A silver subscriber can subscribe to alerts for up to 4 events in total
		QR6: Changes in subscriptions or email addresses will come into effect within 5 minutes QR6.1: If the changes are made within 5 minutes of the event polling and notification window for the user, the changes are not guaranteed to be in effect for the cycle.

		QR7: Emails and passwords will be encrypted at rest and in transit
S2: Historical Weather Data Comparison	FR1: The user can select two dates for a location within the past 5 days for weather comparison	QR1: Historical data will be projected to the user within 1 minute.
	FR2: The system will return quantitative data for the two dates, including the temperature, weather, humidity, pressure, and UV	QR2: The user can enter a date or select from a calendar to choose their dates for comparison
	FR3: The user can select any one of the cities that they are subscribed to for comparison	QR3: A user can select dates from the same or different cities for comparison QR3.1: Silver subscribers will have access to the one city that they are subscribed to for comparison QR3.2: Gold subscribers will have access to any two of the cities that they are subscribed to for comparison
S3: General Weather Forecast	FR1: When the user accesses the service, they will see a graphical representation of weather data for their provided location.	QR1: The number of locations available for subscription depends on the tier of their subscription. QR1.1: Gold subscribers can subscribe to up to 5 locations for daily forecasts QR1.2: Silver subscribers can subscribe to up to 1 location for daily forecast
	FR2: User can swap between present and future forecast predictions based on given time spans (3 hourly, daily, or 5 day).	QR2: Data will be projected to the user within 1 minute.
	FR3: User can subscribe to multiple locations/cities and swap between each locations weather data.	
	FR4: The service will display the temperature, weather, humidity, and pressure.	

3 - Architecture

3.1 - System Overview

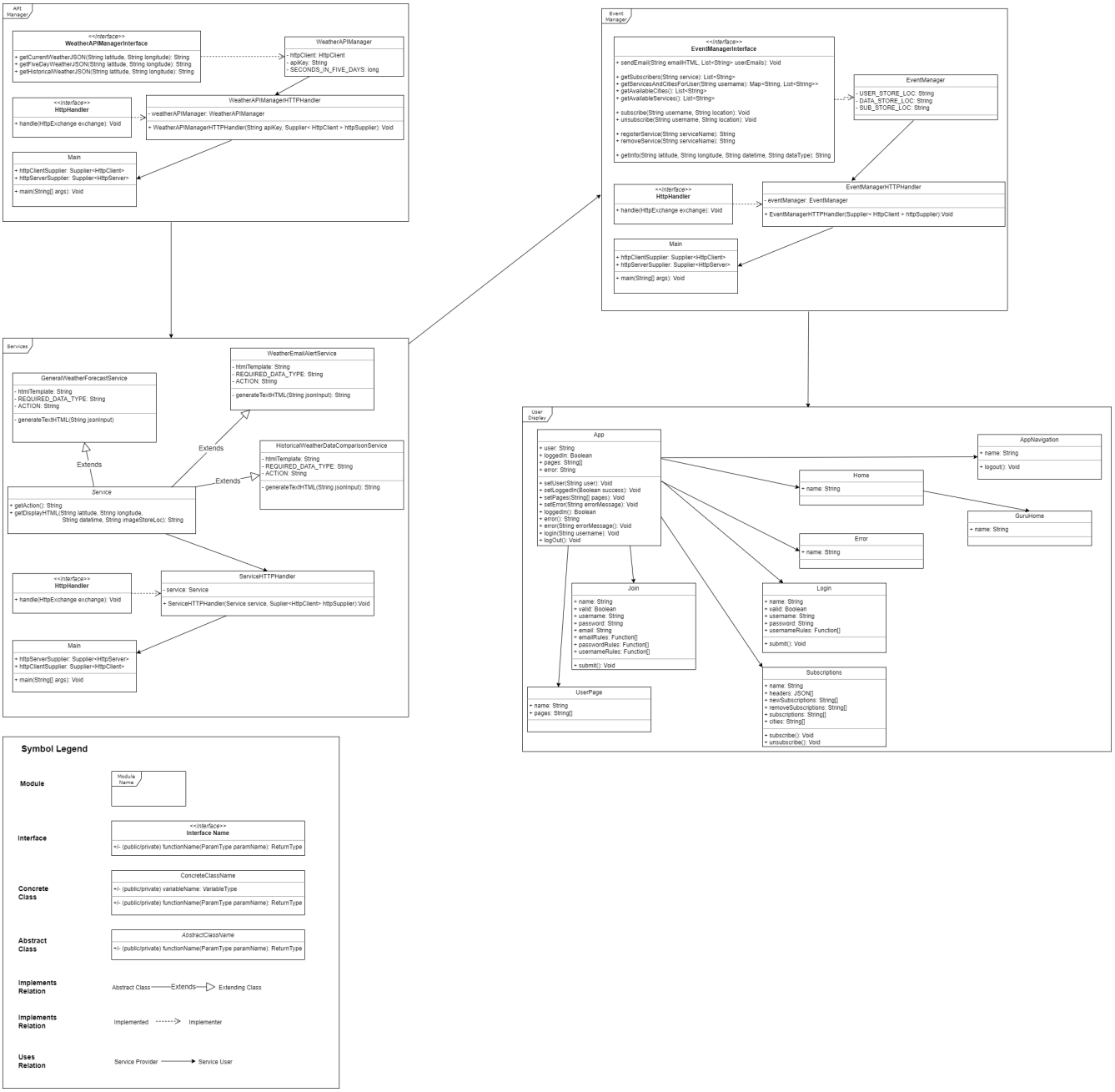
Below is our planned high-level service overview. This diagram depicts the general flow of data through our components from when a user requests one of our services to when the correct data is returned to the UI. Starting with the UI manager, it will provide the initial web pages for users to log in, subscribe and make requests to services. When a user makes a request for data or a subscription for a particular service the UI manager will take that request and then ask the event manager to provide it with a display page for that particular service. The event manager then is responsible for processing the various requests from the UI manager. Whether it's a request for data, where the event manager will relay the request for data to the correct service, or a change in subscription where the event manager will then update the subscription information in its object storage. If it is a request for data, the event manager will pass on that request to the corresponding service either the email alert system, the historical data comparison, or the general forecast. From there in order for the services to service the request, they need data from the external API. So the services will call the API manager to request for specific data. The API manager will then first look into its cache to see if that data has already been requested and stored in the cache, if not found then it will send a call to the external API for data which will then be stored in the cache and sent back to the service for processing. The service will then take that data and form it into a formatted HTML template to represent the data on the UI, and then send that formatted template back to the event manager and then the UI manager which will display the template for the user.



3.2 - Structural Views

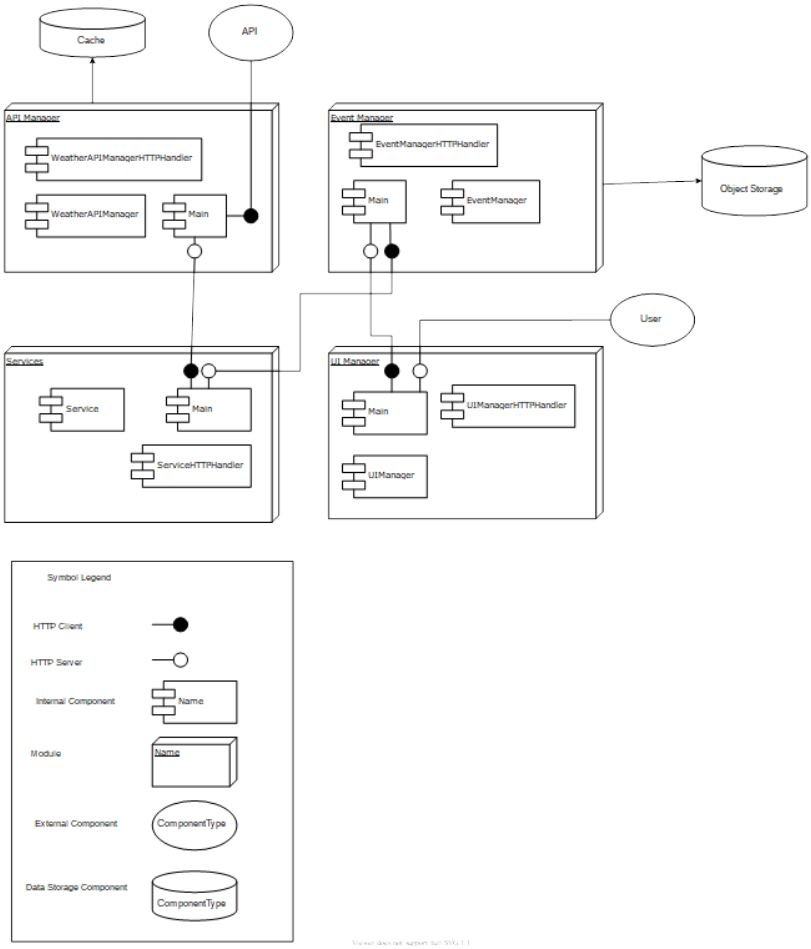
3.2.1 - Module View

This below design is intended to reflect a modularity in the services that are available and a flexibility to allow the services to provide a range of information and display it to the user. The use of interfaces and abstract classes allows for changes in the modules themselves without major re-work of the system as a whole, and ensures that the data contracts are met by any new services that are registered. The variation in this diagram is present in the object storage mechanisms that are employed by the Event Manager and API Manager. These may be a document store, a database, or a file system. The full Context Diagram for the design is provided in Appendix A.



3.2.2 - Component and Connecter View

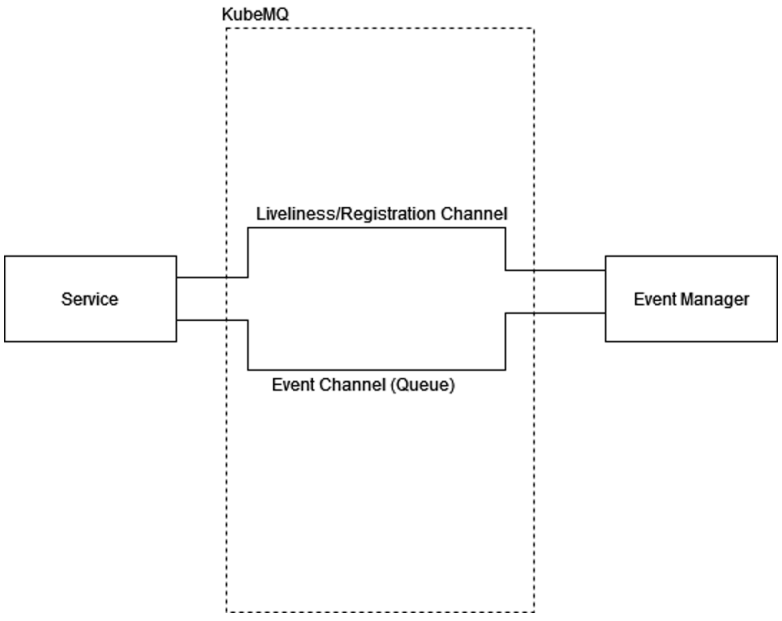
This design is intended to reflect the methods and lines of communication between all of our components. We are using a known standardized method of communicating between components (HTTP) as much as is feasible in order to maintain consistency in the application and more easily extend it to future uses. The use of HTTP communications also allows a lot of flexibility in case of changing requirements because it does not require knowledge of the internal functions other components on behalf of the client or server. The variation in this diagram is present in the request contracts between elements. The HTTP requests can be modified to include more or less information, which can be used to provide different responses.



3.3 - Quality Views

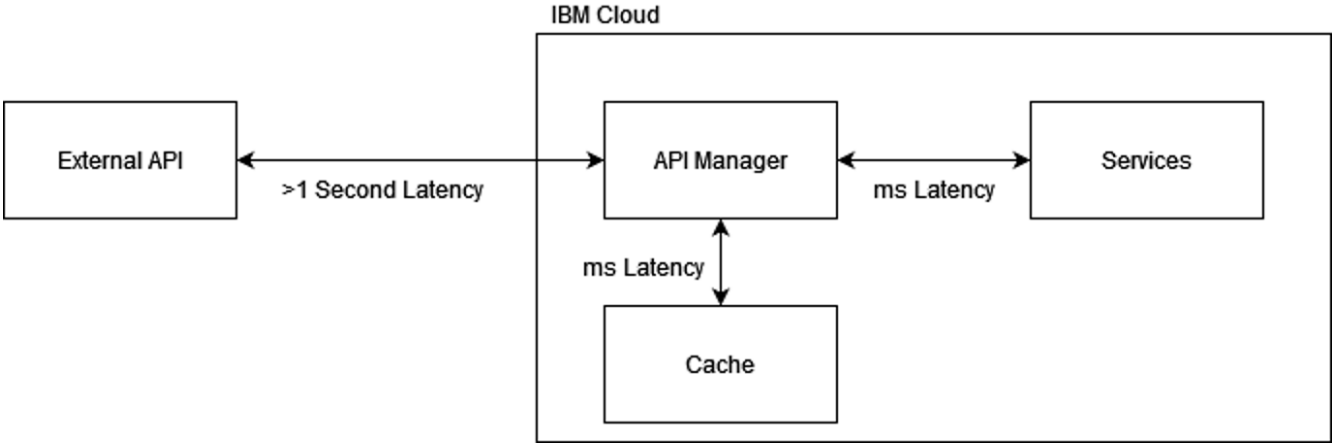
3.3.1 - Communication View

To allow the system to be as dynamic as possible, we are using this communication system to allow real-time addition and deletion of services. When developing new services, the base communication will always be the same, allowing us to re-use those components and only develop the business logic for the service. KubeMQ is the technology of choice as it is specifically developed for Kubernetes platforms, and requires minimal setup while being extremely performant. Services will vary in their output/functionality, but the base framework for communication will not differ between any of them. This will promote modifiability and extensibility. Please see Appendix B where a more detailed sequence diagram depicts the communication flow across the system.

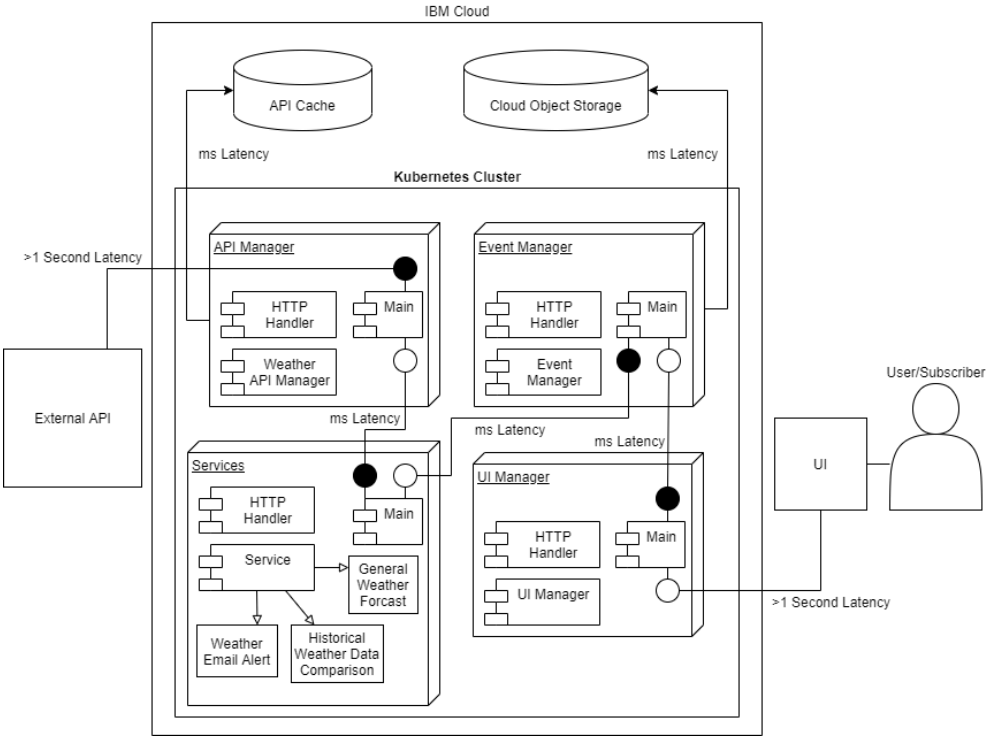


3.3.2 - Performance view

We need the system to perform as quickly as we can, especially for time sensitive services like the alerting system. At the same time, we require the system call the External API as infrequently as possible to reduce operating costs and stay withing the free usage range for it. Utilizing our API Manager with caching addresses both of these points. Since services use different parts of the same data returned from the API, caching the data rather than by calling the API each time (milliseconds instead of seconds), means we only need to call one location/timeframe once, and then that data can be used by any of our services. Some variability exists here with the implementation of the cache. Initial proposals are to use Cloud Object Storage but implementing it with the API Manager itself may be a more efficient option. Please see Appendix C which uses a more detailed activity diagram to outline the data flow for both scenarios of our data retrieval.

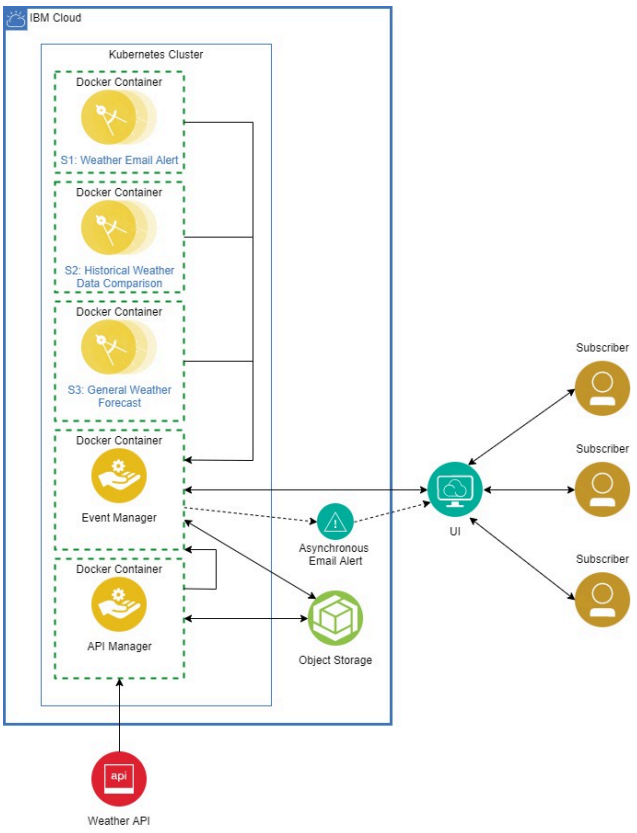


3.4 - Combined View



4 - Design Implementation

Our initial design for the system was similar to what we have now as shown above. In our initial design we didn't have a UI manager, and instead had our event manager handling all the UI functionality for our system. We thought that by decoupling the UI from the event management we could improve on the reusability and modifiability of the system as having them separated would make it easy for future changes. Shown on the right is our initial System Overview diagram for our system with no UI manager.



5 - Testing

5.1 - API Manager Component

We decided to implement extensive automated testing for this module because it was interacting with an external resource (the Open Weather Map API). We used Junit as a framework to create both unit and integration tests for all methods in the component. We tested both happy path executions as well as failure cases. See below test reports:

Class TestWeatherAPIManagerHTTPHandler

all > default-package > TestWeatherAPIManagerHTTPHandler

8	0	0	0.028s
tests	failures	ignored	duration

100% successful

Tests

Test	Duration	Result
handle_APIManagerFailure_GetCurrent()	0.008s	passed
handle_APIManagerFailure_GetFuture()	0.002s	passed
handle_APIManagerFailure_GetPast()	0.002s	passed
handle_FailureAPIBadURI()	0.003s	passed
handle_FailureAPIRequestType()	0.001s	passed
handle_HappyPath_GetCurrent()	0.003s	passed
handle_HappyPath_GetFuture()	0.006s	passed
handle_HappyPath_GetPast()	0.003s	passed

Generated by Gradle 7.1 at Dec 2, 2021, 9:05:53 AM

Class TestWeatherAPIManager

all > default-package > TestWeatherAPIManager

37	0	0	0.120s
tests	failures	ignored	duration

100% successful

Tests

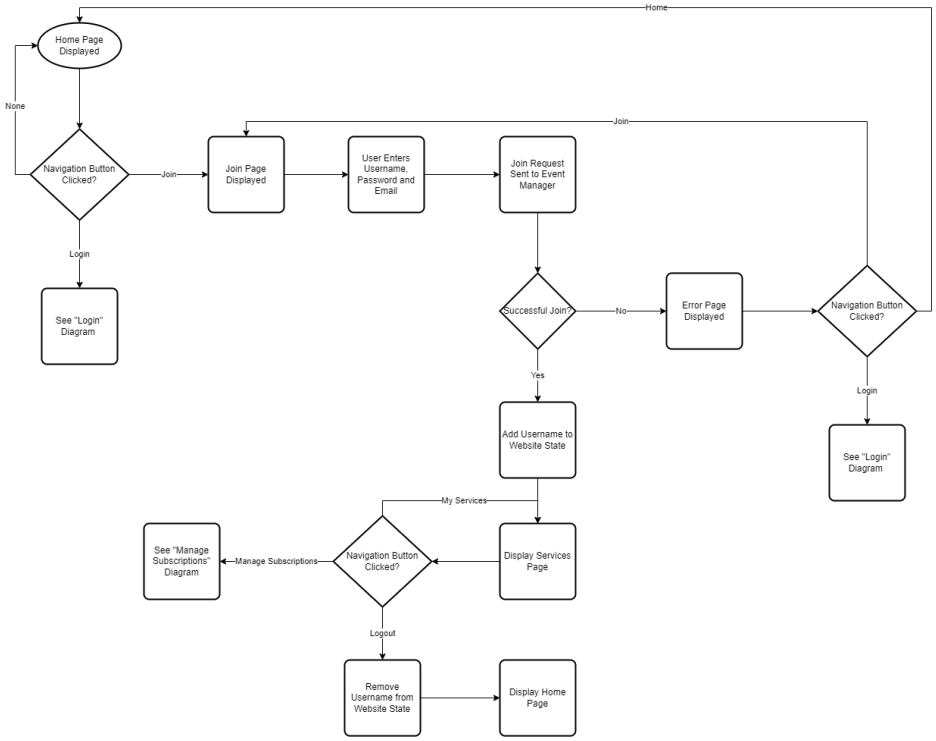
Test	Duration	Result
getCurrentWeatherJSON_badConnectionToAPI()	0.001s	passed
getCurrentWeatherJSON_badReturnFromAPI()	0.002s	passed
getCurrentWeatherJSON_happyPath()	0.014s	passed
getCurrentWeatherJSON_highLatitude()	0.001s	passed
getCurrentWeatherJSON_highLongitude()	0.004s	passed
getCurrentWeatherJSON_lowLatitude()	0.001s	passed
getCurrentWeatherJSON_lowLongitude()	0.001s	passed
getCurrentWeatherJSON_nonNumberLatitude()	0.001s	passed
getCurrentWeatherJSON_nonNumberLongitude()	0.003s	passed
getCurrentWeatherJSON_nullLatitude()	0s	passed
getCurrentWeatherJSON_nullLongitude()	0.002s	passed
getFiveDayWeatherJSON_badConnectionToAPI()	0.044s	passed
getFiveDayWeatherJSON_badReturnFromAPI()	0.001s	passed
getFiveDayWeatherJSON_happyPath()	0.005s	passed
getFiveDayWeatherJSON_highLatitude()	0.001s	passed
getFiveDayWeatherJSON_highLongitude()	0.001s	passed
getFiveDayWeatherJSON_lowLatitude()	0.001s	passed
getFiveDayWeatherJSON_lowLongitude()	0.001s	passed
getFiveDayWeatherJSON_nonNumberLatitude()	0.001s	passed
getFiveDayWeatherJSON_nonNumberLongitude()	0.001s	passed
getFiveDayWeatherJSON_nullLatitude()	0.001s	passed
getFiveDayWeatherJSON_nullLongitude()	0.001s	passed
getHistoricalWeatherJSON_badConnectionToAPI()	0.001s	passed
getHistoricalWeatherJSON_badReturnFromAPI()	0.001s	passed
getHistoricalWeatherJSON_happyPath()	0.014s	passed
getHistoricalWeatherJSON_highDateTime()	0.001s	passed
getHistoricalWeatherJSON_highLatitude()	0.002s	passed
getHistoricalWeatherJSON_highLongitude()	0.004s	passed
getHistoricalWeatherJSON_lowDateTime()	0.001s	passed
getHistoricalWeatherJSON_lowLatitude()	0.001s	passed
getHistoricalWeatherJSON_lowLongitude()	0.002s	passed
getHistoricalWeatherJSON_nonNumberLatitude()	0.001s	passed
getHistoricalWeatherJSON_nonNumberLongitude()	0s	passed
getHistoricalWeatherJSON_nonNumericDateTime()	0s	passed
getHistoricalWeatherJSON_nullDateTime()	0.002s	passed
getHistoricalWeatherJSON_nullLatitude()	0s	passed
getHistoricalWeatherJSON_nullLongitude()	0.002s	passed

Generated by Gradle 7.1 at Dec 2, 2021, 9:05:53 AM

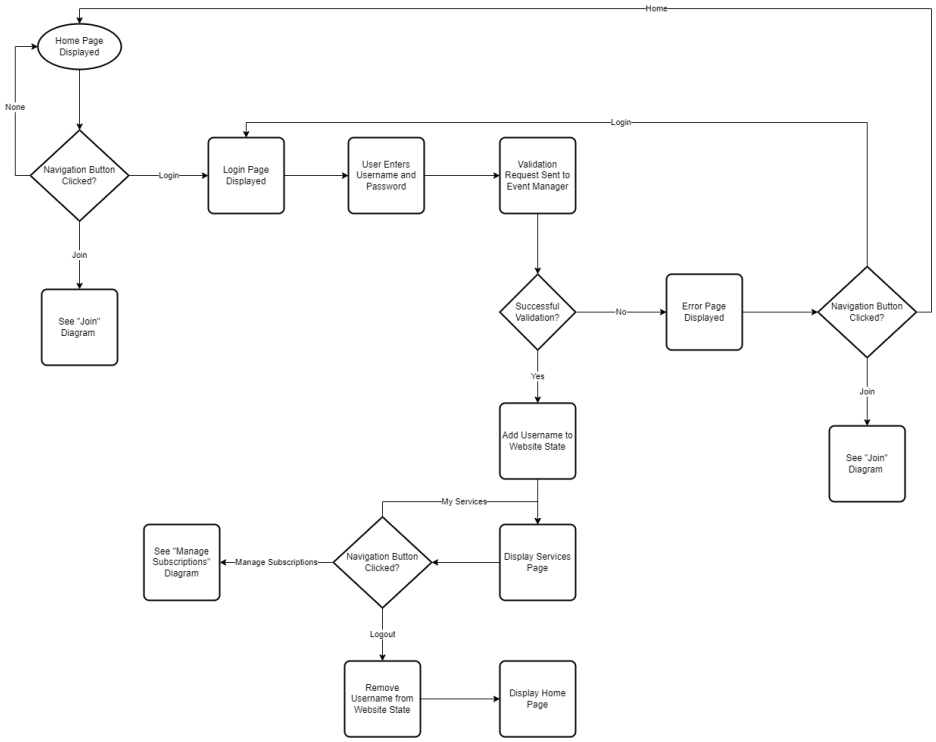
5.2 - UI Component

This component was tested manually using the following execution flows:

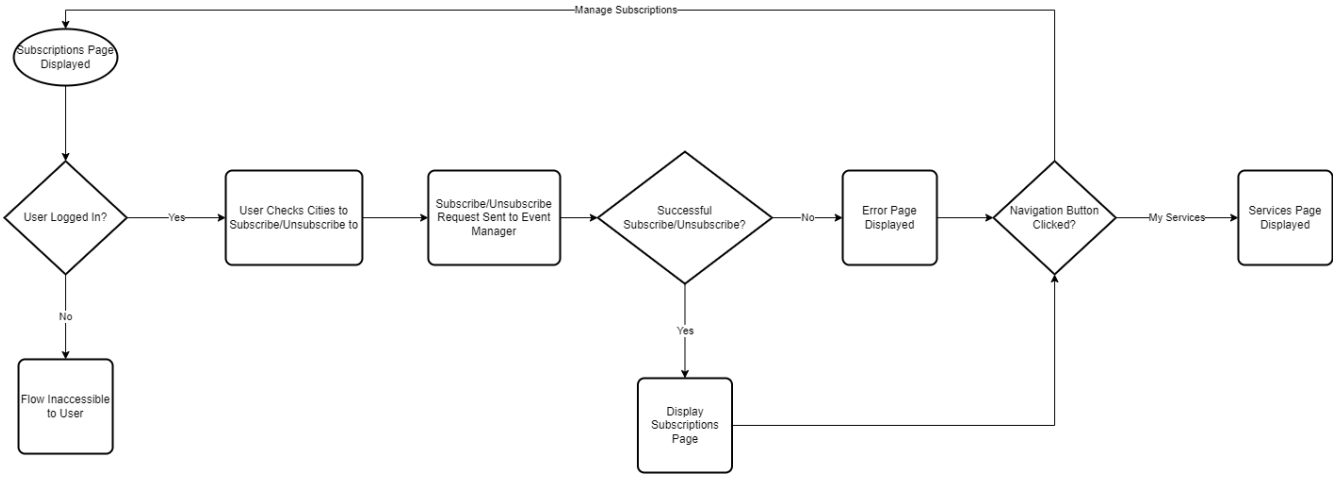
For Login:



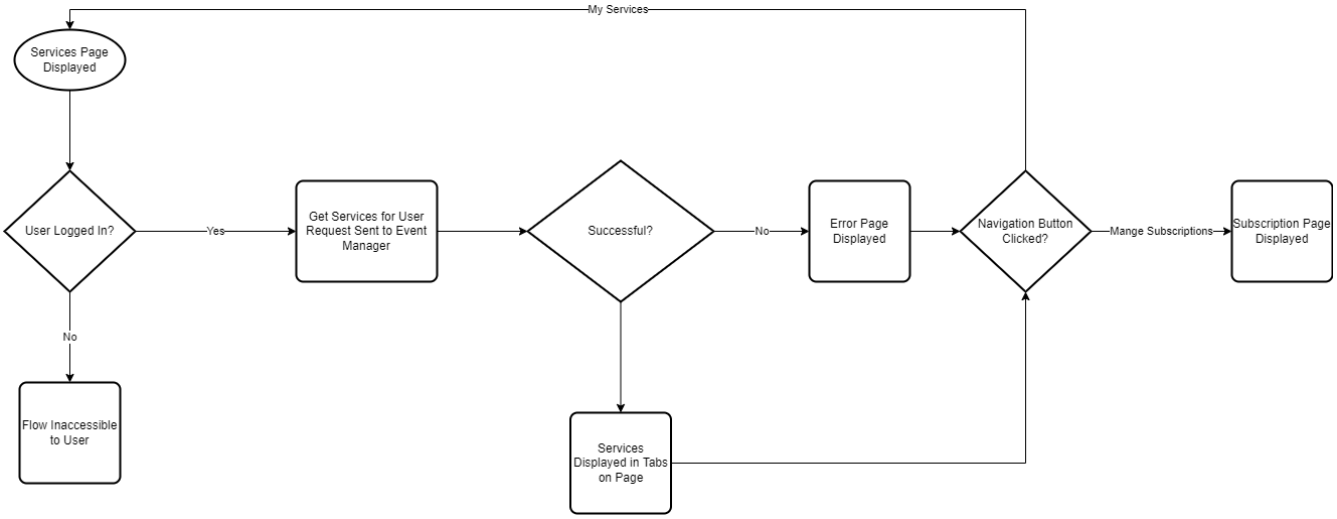
For Joining:



For Subscribing/Unsubscribing:



For Service Use:



The execution paths were tested with single and multiple inputs where available (eg. multiple simultaneous subscriptions and unsubscriptions).

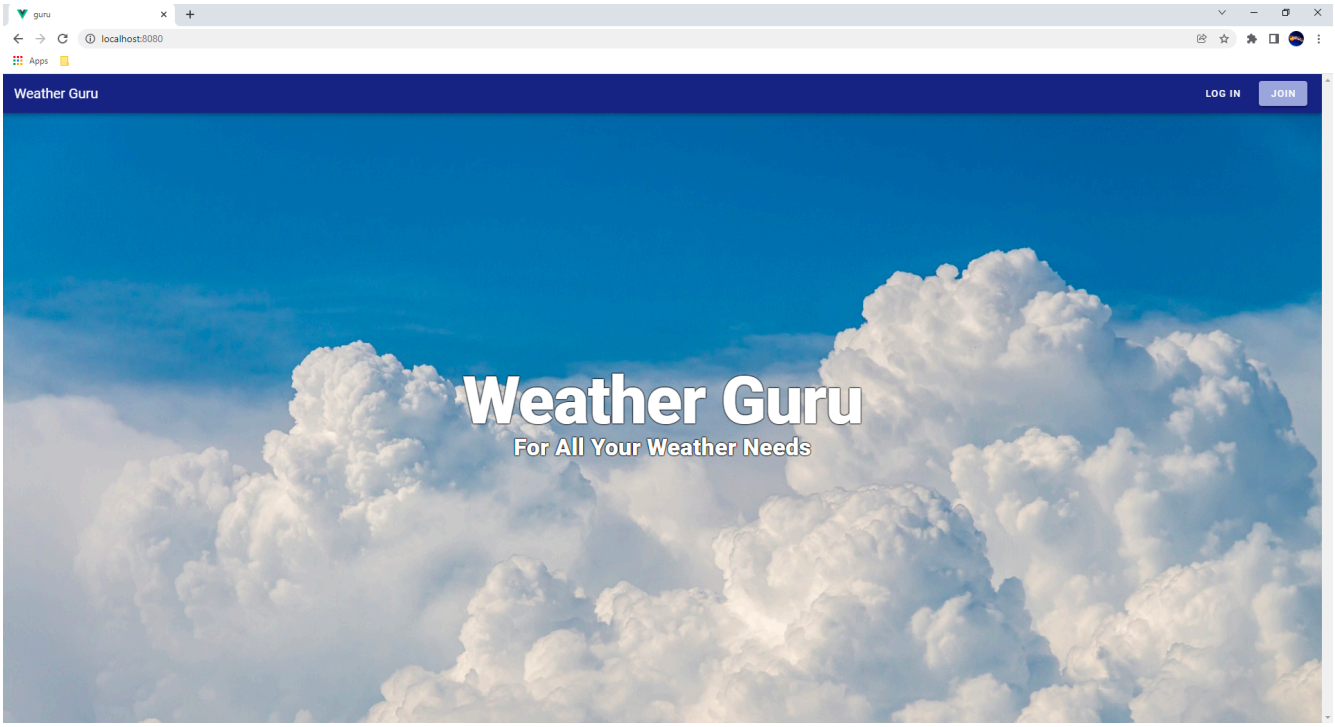
5.3 - Event Manager and Services

Due to the various connections made by the event-manager (KubeMQ channels, Object storage entities), mainly integration testing was performed. Using diagnostic endpoints, various functions of the manager would be triggered and through a mix of logged information and checks of storage buckets/messaging queues, the behaviour was validated. Services were built upon the same blueprint, which contains behaviour for connecting to the event-manager. Business logic was tested independently, using local tests run against a sample API response. Once this behaviour was correct, it was added into the blueprint, and end-user flows were tested to ensure the correct data was being returned.

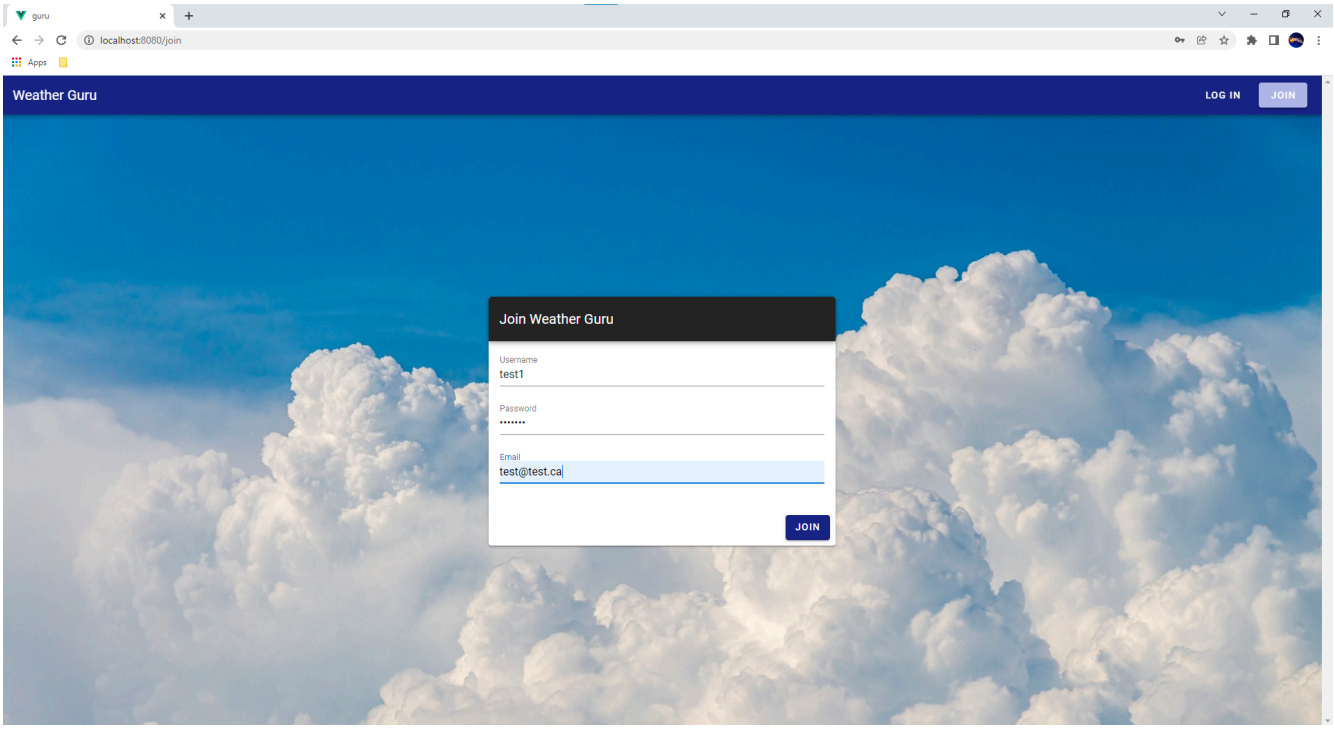
6 - Screen Dumps as Operational Evidence

UI Screenshots

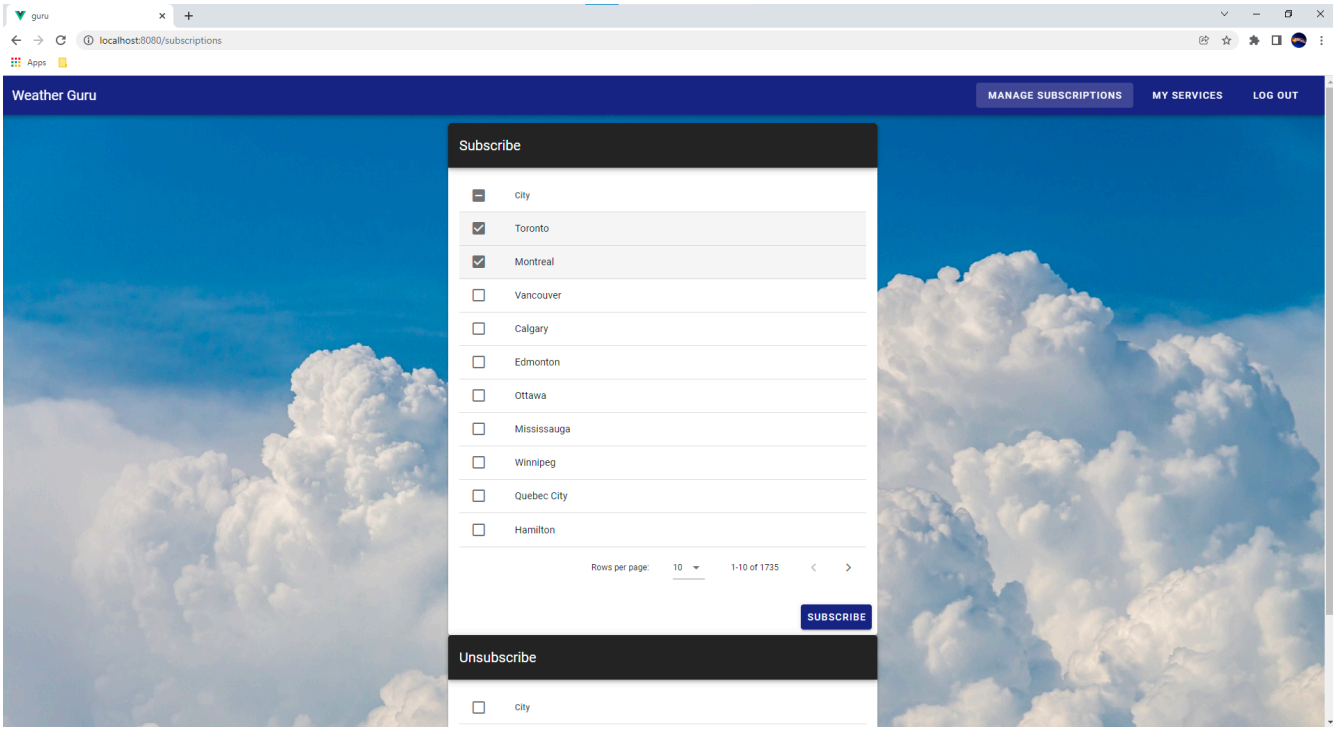
User view upon initial visit:

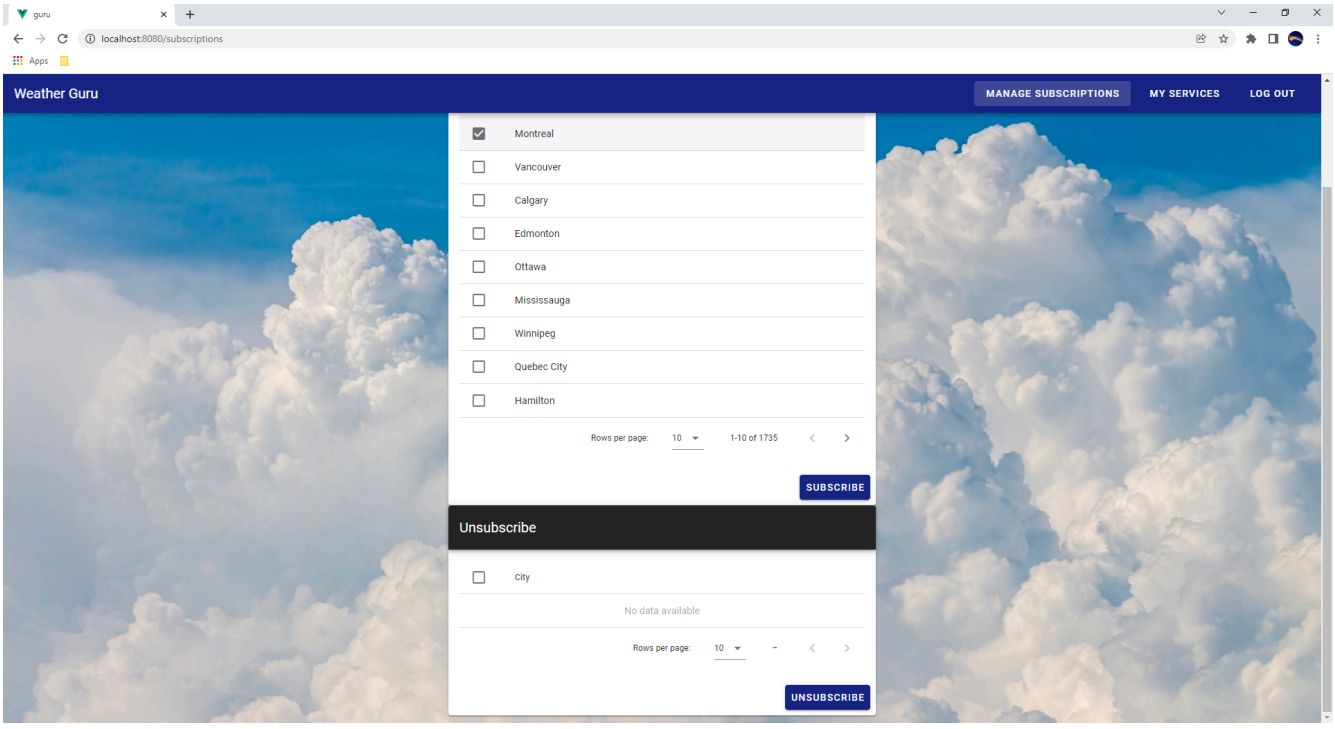


When the Join button is clicked, the user is prompted to join the website:

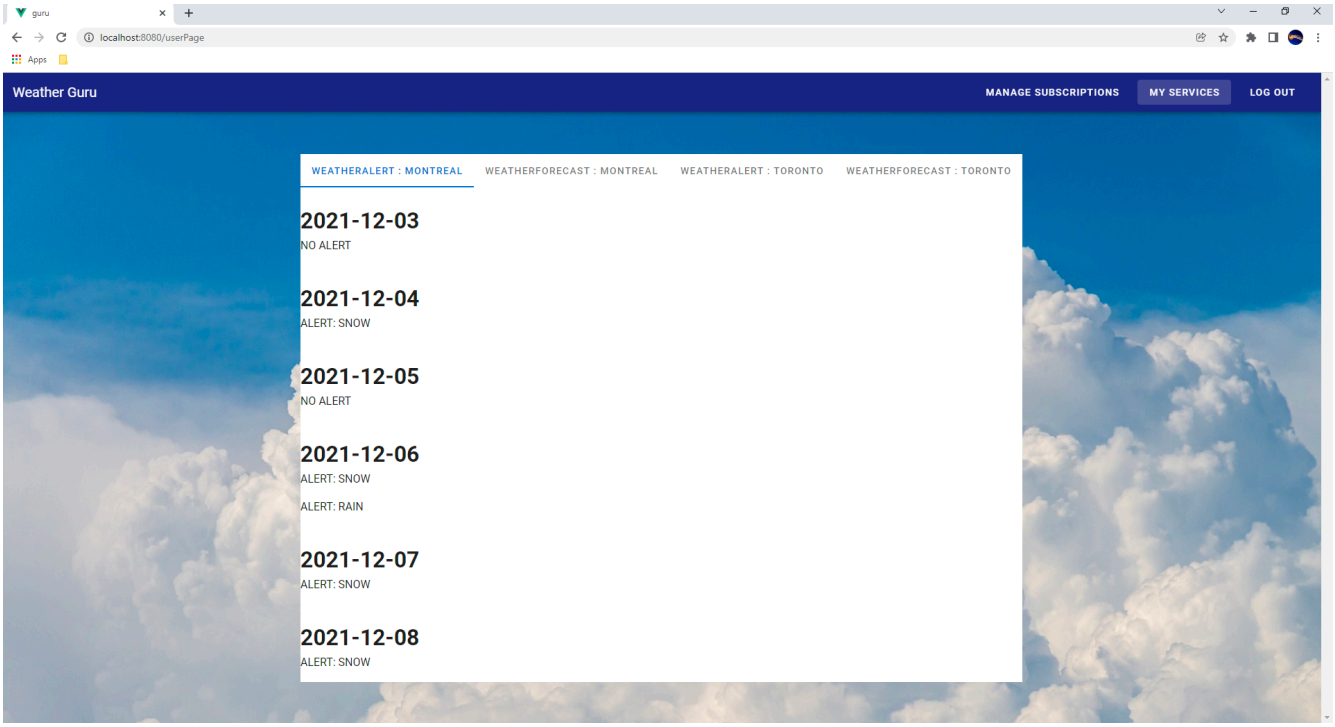


User is then shown their services page, which is empty:
When the Manage Subscriptions button is clicked, the user is shown a list of possible cities to subscribe to and a list of their current subscriptions that can be removed:

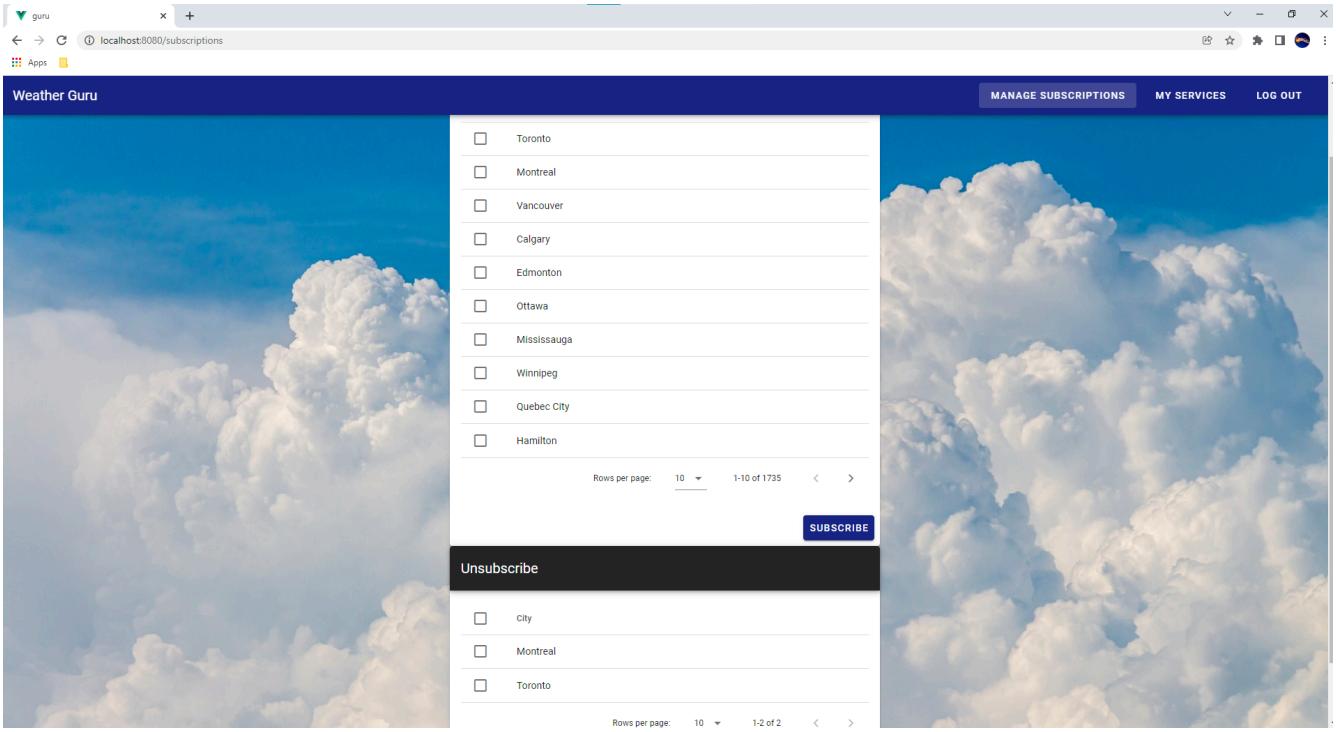


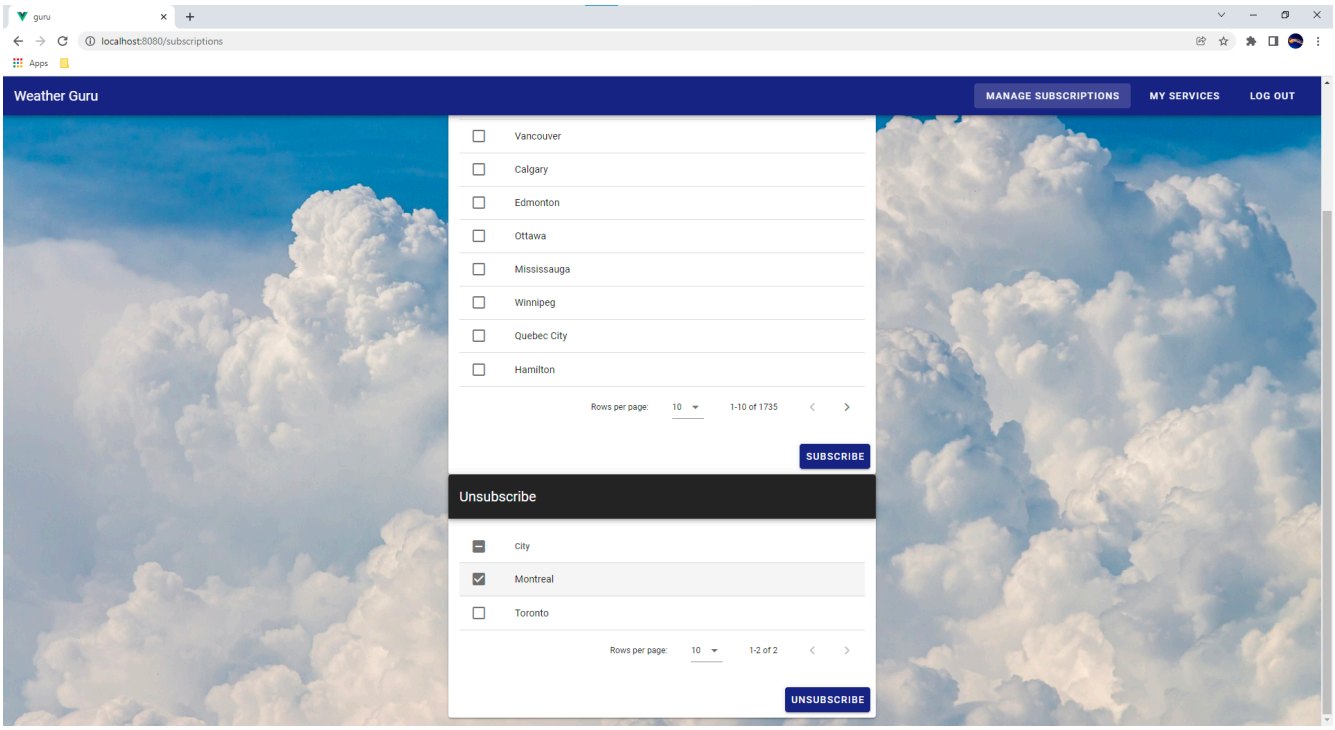


When the My Services button is clicked after subscribing to Toronto and Montreal, the user is shown the services for those cities:

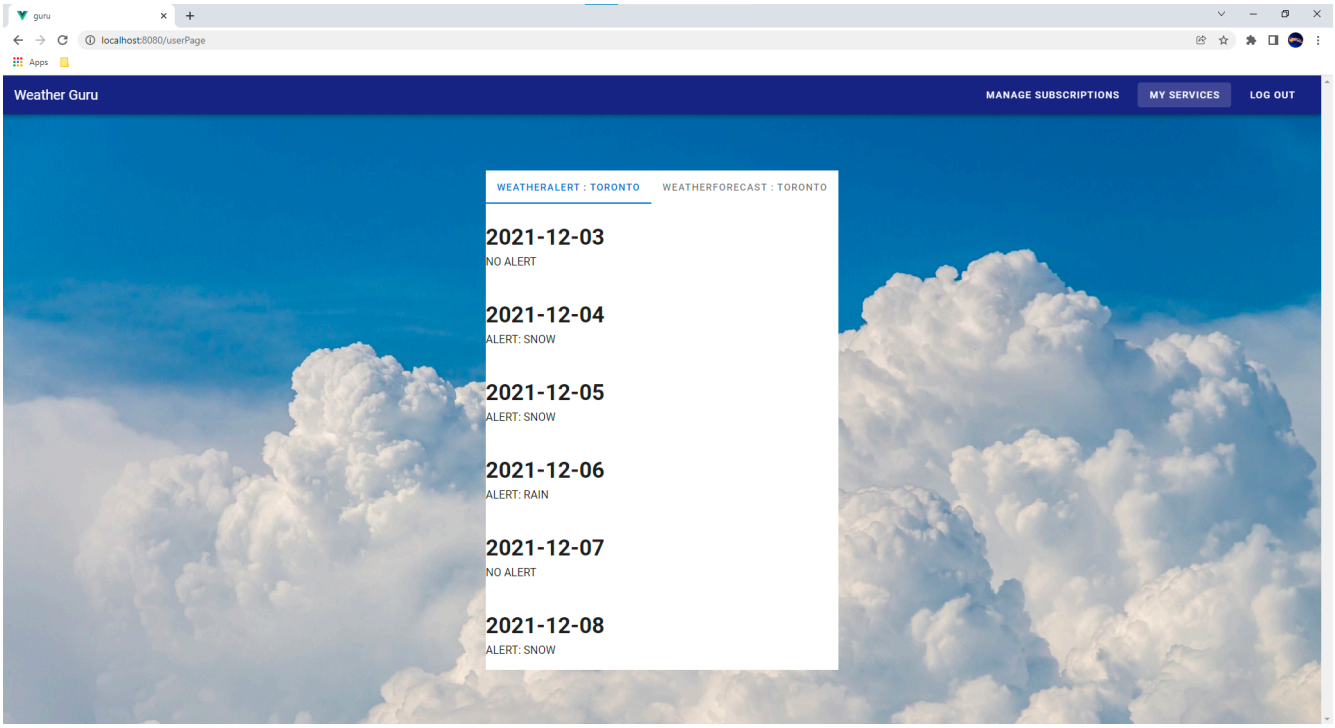


When the Manage Subscriptions button is clicked, the user's new subscriptions are shown and can be unsubscribed from:



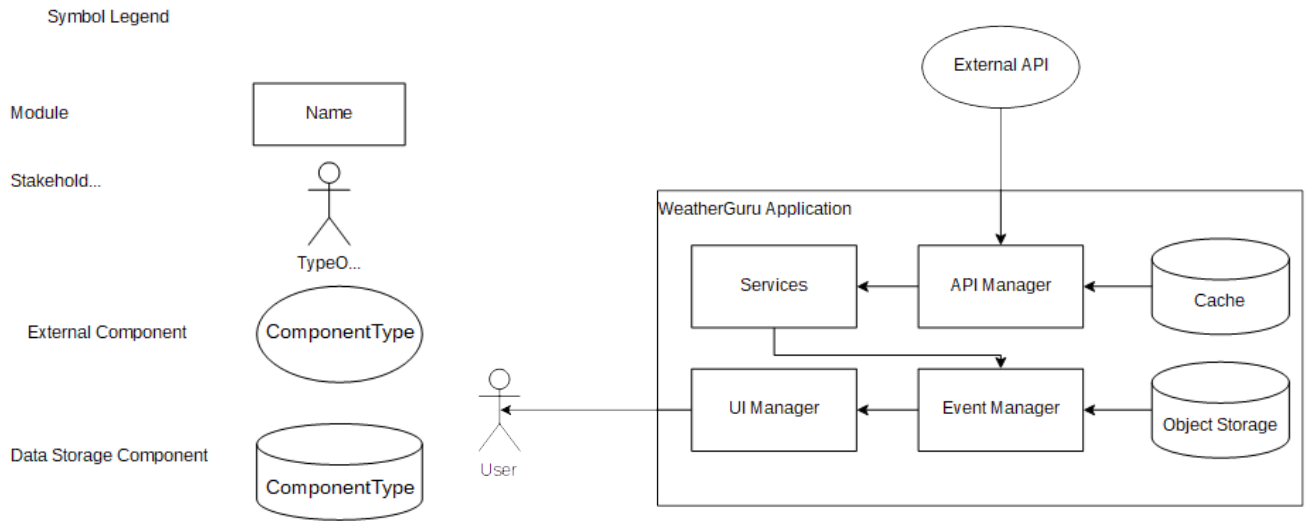


When the user has unsubscribed to Montreal and clicks the My Services page, services for Montreal are no longer displayed:

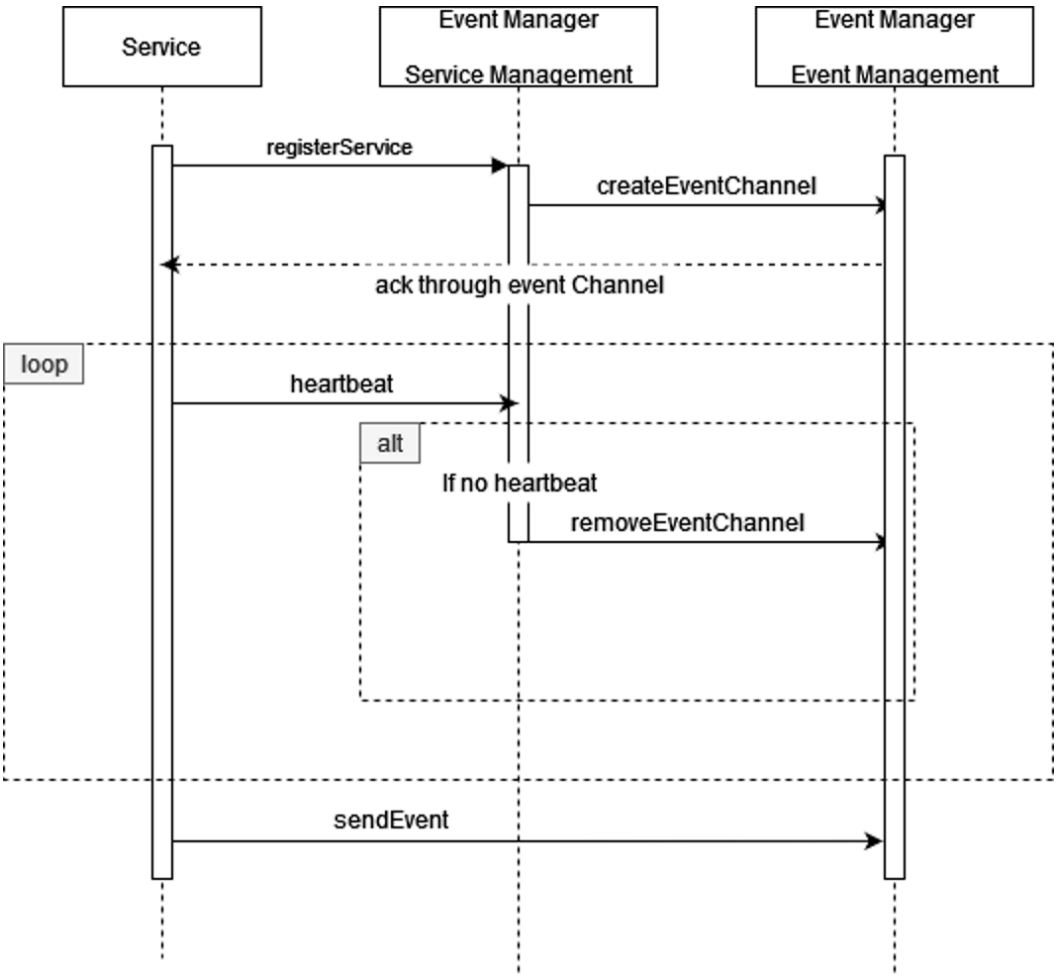


7 - Appendix

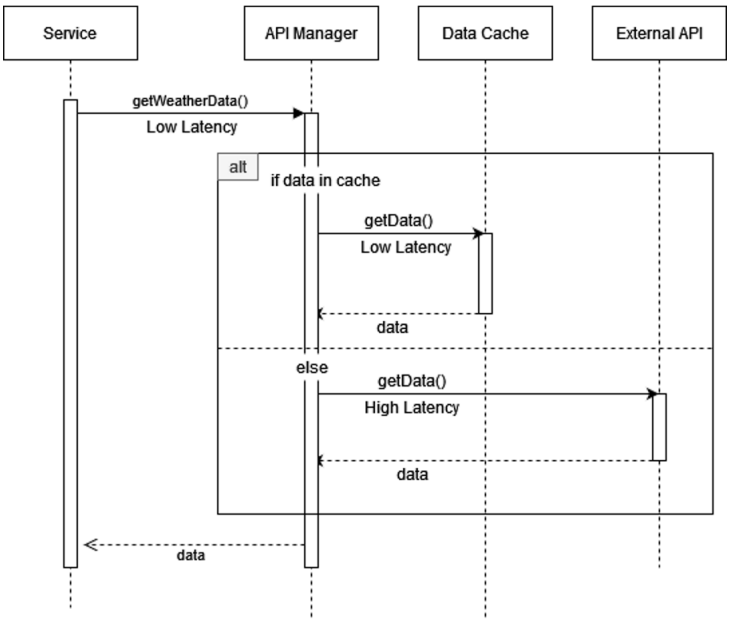
Appendix A - Context Diagram:



Appendix B - Sequence Diagram for System Communication:



Appendix C - Sequence Diagram Outlining Data Retrieval:



8 - Running the Service on Windows:

Step 1: in your console, Unzip /dist folder (attached with report and code)
(if you don't have NPM and Node.js installed, please follow download steps at https://docs.npmjs.com/downloading-and-installing-node-js-and-npm)

Step 2: execute "npm install -g serve" to install the file server for the UI

Step 3: execute "serve -s dist" to start the UI locally

Step 4: Open a web browser using the IP or URL provided after execution of "serve -s dist"

9 - References

APA Formatted:

"Clouds-Nature-Sky-Cumulus-Mood." *Pixabay*, https://pixabay.com/photos/clouds-nature-sky-cumulus-mood-4215608/. Accessed 29 Nov. 2021.

"Open Weather Map API Documentation (Community)." *RapidAPI*, https://rapidapi.com/community/api/open-weather-map/.