WeatherTrack - Final Report

Gian Luca Nediani
gianluca.nediani@studio.unibo.it

Wednesday 26th June, 2024

WeatherTrack is a distributed system that collects and processes real-time weather data from a network of simulated Internet of Things (IoT) sensors, and provides users with a web-based interface to visualize and analyze the collected data. The system is designed to be scalable, fault-tolerant, and extensible.

At the core of the system are the weather sensors, which simulate the behavior of real IoT devices by continuously generating and transmitting weather data, such as temperature, humidity and wind speed. A message broker acts as an intermediary, receiving the sensor data and routing it to the message consumers, which process the information and store it in a database.

The web application, built using a full-stack Javascript approach, provides users with a dashboard to interact with the collected weather data. Users can view real-time sensor readings, explore trends and statistics, and receive alerts for any detected weather anomalies.

The system's modular and containerized architecture, combined with the use of tools like Docker, GitLab CI/CD, and automated testing, ensures the project can be easily deployed, maintained and extended in the future with the integration of additional features.

Contents

List of Figures

1 Goal/Requirements

The primary goal of the WeatherTrack project is to develop a distributed system that collects and processes real-time weather data from a network of IoT sensors, and provides users with a web-based interface to visualize and analyze the collected data. The key requirements for the system are:

- Scalability and fault tolerance to handle large volumes of data from multiple sensors.
- User authentication to control access to the web application.
- Real-time data collection, processing and displaying from simulated IoT sensors in the web application.
- Storage of sensor data in a database for historical analysis.
- Web application functionality to display real-time weather data, aggregated statistics, weather anomaly detection.
- Possibility for administrators to manage user accounts and requests submitted by users. Such requests can be for both additional features and sensors in new locations.
- Possibility for users to manage their own account and to post requests. Requests can be anything from new sensor locations to new features.
- Displaying of real-time weather anomalies when detected by the system, as well as past anomalies as stored in the system.

The expected outcomes of the project are: a fully functional and (locally) deployable software system that meets the outlined requirements, as well as the consolidation of knowledge and skills related to building distributed, scalable, and fault-tolerant systems.

1.1 Scenarios

The WeatherTrack system is designed to provide users with real-time and historical weather data. Users can access this information **through a web-based interface**, which allows them to monitor weather conditions, analyze trends, and make informed decisions based on accurate and up-to-date data.

The following are the use scenarios for the system:

- **Registration and Login**: to access the features of the WeatherTrack system, users are required to register and create an account. Once registered, users can log in to their account using their credentials.

- **Viewing the Data Dashboard**: The primary purpose of the WeatherTrack system is to provide users with real-time weather and historical data from a network of IoT sensors. Users can view this data on the dashboard, which includes information such as temperature, humidity, wind speed.
- **Viewing Weather Anomalies**: The system is designed to detect and alert users of any weather anomalies, such as sudden changes in temperature or unexpected wind gusts. Users can view these anomalies on the dashboard and receive push notifications.
- **Managing Own Requests and Profiles**: Users can manage their own requests and profiles within the system. They can submit requests for new features or sensors in specific locations and track the status of their requests. Users can also update their profile information.
- **Managing All Requests and Profiles (ADMIN)**: In addition to the regular user functionalities, administrators have access to additional features for managing the system. Administrators can manage all user requests, including approving or denying requests. They can also manage user profiles, including creating, modifying, and deleting user accounts.

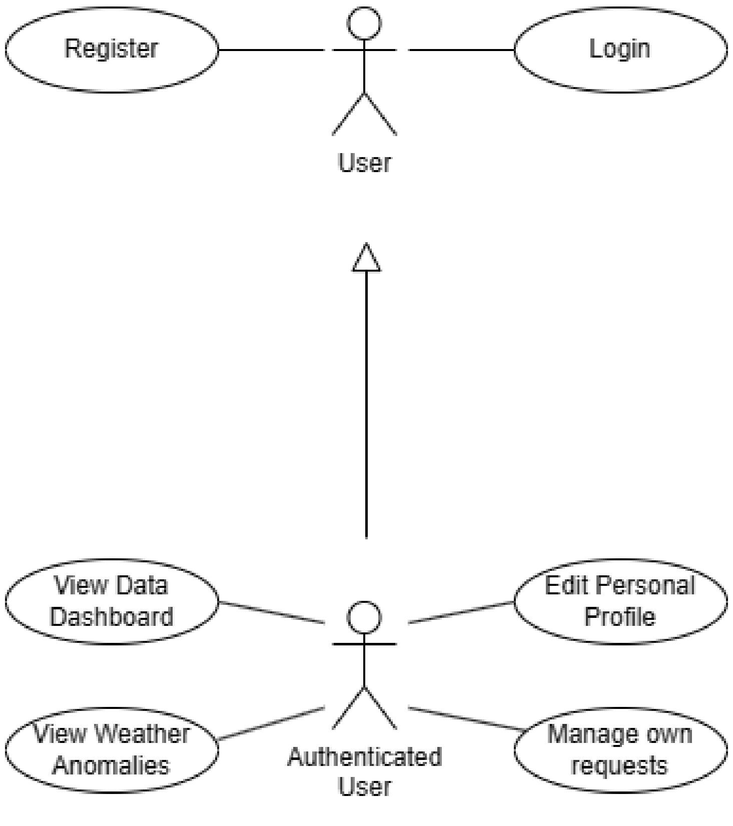


Figure 1: UML Use case diagram for the user

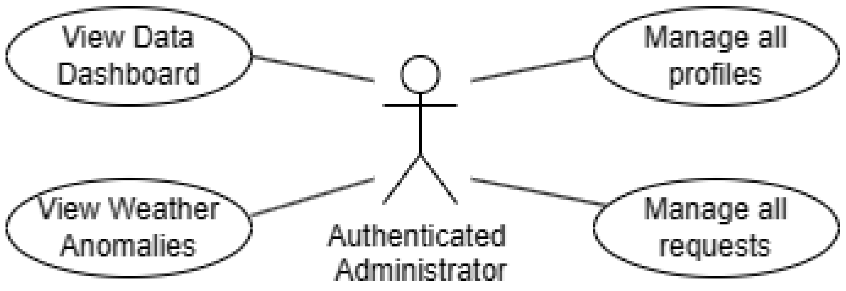


Figure 2: UML Use case diagram for the administrator

1.2 Self-assessment policy

To ensure that the software and system meet the desired requirements, several evaluation methods will be employed.

The following methods will be used to evaluate the software and system:

- **Automated Unit Tests**: these tests focus on individual components of the software, ensuring that each unit functions correctly and meets its intended purpose. By automating these tests, the evaluation process becomes more efficient and consistent.
- **Integration Tests**: in addition to automated unit tests, manual and automated integration tests are conducted to assess the interaction and integration of various software components. These tests will verify that the different parts of the software work together to produce the expected results.
- **Tests with Simulated Users**: to evaluate the system's performance under real-world conditions, tests with simulated users are be carried out. These tests provide insights into the system's usability, performance and responsiveness. These tests are especially aimed at empirically evaluating the web client.
- **Satisfaction of Functional and Non-Functional Requirements**: the software is evaluated based on its compliance with both the functional and non-functional requirements identified in this and the following section.

More specific information about software verification and self-assessment can be found in Section 5.

2 Requirements Analysis

2.1 Implicit Requirements

Upon analyzing the project requirements, some implicit requirements were identified:

- A simulated sensor that resembles a real one has to be implemented.
- The system should be able to handle multiple users simultaneously accessing the web application.
- The system should provide real-time updates to the web application interface as new sensor data is collected.
- The system should be able to handle different types of weather data, such as temperature, humidity and wind speed.
- The end-user functionalities for both regular users and admin should be provided via a web-based application interface.

2.2 Implicit Hypotheses

The project requirements also make some implicit assumptions:

- The sensors will be continuously collecting and transmitting weather data, but may fail and stop sending data at some point.
- The system will have a reliable network connection between the sensors, message broker, and web application.
- The web application users will have a basic understanding of weather data and terminology.

2.3 Non-Functional Requirements

In addition to the explicit functional requirements, there are also some non-functional requirements implied by the project which will have to be taken into account during the design phase:

- The system should be scalable, as mentioned in the project requirements.
- The system should be fault-tolerant, as mentioned in the project requirements.
- The system should have low latency and high availability to ensure real-time data processing and display.
- The system should have an authentication system with different roles: administrator and regular user.

2.4 Best Suited Technologies

Based on the project requirements and analysis, the following technologies and paradigms are well-suited for this project:

- The publish-subscribe model used by message brokers such as RabbitMQ or Apache Kafka is ideal for real-time data processing and communication between components.
- A NoSQL, document-based database such as MongoDB would be well-suited for storing and querying large amounts of unstructured weather data.
- For the frontend, a Single Page Application (SPA) framework such as React or Vue.js would be well-suited. These frameworks allow for the creation of dynamic, responsive web applications that can update in real-time, providing an optimal user experience.
- For the backend, Node.js, a JavaScript runtime built on Chrome's V8 JavaScript engine, would be a good choice. Node.js is non-blocking and event-driven, making it efficient and lightweight, perfect for real-time data-intensive applications. It also has a rich ecosystem of libraries that can simplify the development of the web backend functionalities.
- The use of containerization technologies such as Docker and orchestration tools such as Docker Compose or Kubernetes can help bridge the gap between development and deployment environments and simplify the management of the distributed system. The modularity provided by a containarized system can also make development easier should any single piece of the system change.

2.5 Abstraction Gap

The above-chosen technologies and models provide a high level of abstraction, allowing to focus on the individual components and their interactions rather than low-level implementation details.

3 Design

3.1 Structure

3.1.1 Entities

Based on the end goal and the requirements identified in the previous sections, the key components that need to be modeled in the system are:

- **WeatherSensor**: Represents a simulated IoT sensor that collects weather data, such as temperature, humidity, wind speed, and precipitation. Note that in a real-world deployment scenario, this would be an actual IoT sensor, with a slightly different implementation.
- **MessageBroker**: Responsible for receiving and routing messages (weather data) from the sensors to the consumers.
- **MessageConsumer**: Processes the weather data received from the message broker and stores it in the database, as well as looking for anomalies in the received data. Any anomaly will be stored both as a regular data point and as a separate document in the anomaly collection, to be processed in real-time to notify the end users.
- **Database**: Stores the collected weather data for historical analysis and retrieval. It also stores the user data and user request data.
- **WebServer**: Provides a RESTful API to retrieve weather data and information from the system. Also provides streams for real-time data updates.
- **WebClient**: Client for the web application that allows users to view and interact with the system. As shown in Section 1, regular users and administrators will have different interactions with the system.

The following class diagram provides a high-level overview of the components, the entities (which are described in section 3.2.4) and their interaction.

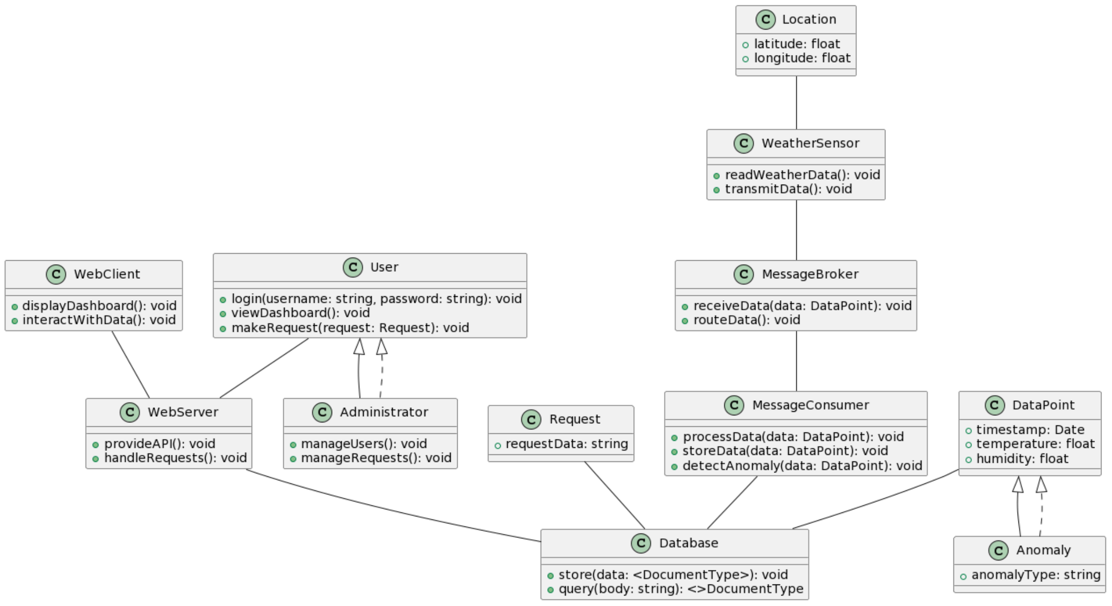


Figure 3: UML Class Diagram

3.1.2 Technology stack choices

The chosen technologies for the development of the system are the following:

- *RabbitMQ* as the message broker. RabbitMQ was chosen over *Apache Kafka* because it is easier to set up and manage, and its performance is sufficient for the project's needs. Additionally, RabbitMQ has better support for message routing and delivery guarantees, which are important for the system's reliability.

- *Node.js* + *Express.js* for the web application back-end development. Node.js was chosen because it is a lightweight and scalable platform for building server-side applications, and its non-blocking I/O model is ideal for handling multiple concurrent connections. Express.js was chosen as the web framework because it is simple, flexible, and widely used in the Node.js community.

- *Vue.js* for the web application front-end development. Vue.js was chosen because it is a lightweight and easy-to-learn framework for building dynamic user interfaces. It also has good performance and a growing ecosystem of plugins and tools.

- *MongoDB* as the database. MongoDB was chosen because it is a scalable and flexible NoSQL database that can handle large volumes of unstructured data. It also has good performance and supports rich querying capabilities. On top of that, its change streams feature is very suitable for delivering real-time updates.

- *Docker* as the Container Manager. Docker was chosen because it allows for easy deployment and scaling of applications in isolated containers. It also simplifies the development and testing process by providing a consistent environment across different machines.

- *Docker-compose* for the container orchestration. Docker-compose was chosen over *Kubernetes* because it is simpler to use and more suitable for small-scale deployments. It also integrates well with Docker and allows for easy management of multi-container applications.
- *Python* for the sensor and consumer scripts, as it provides a simple and efficient way to develop and manage the scripts and a large ecosystem of libraries for data and IoT systems.

3.1.3 Components

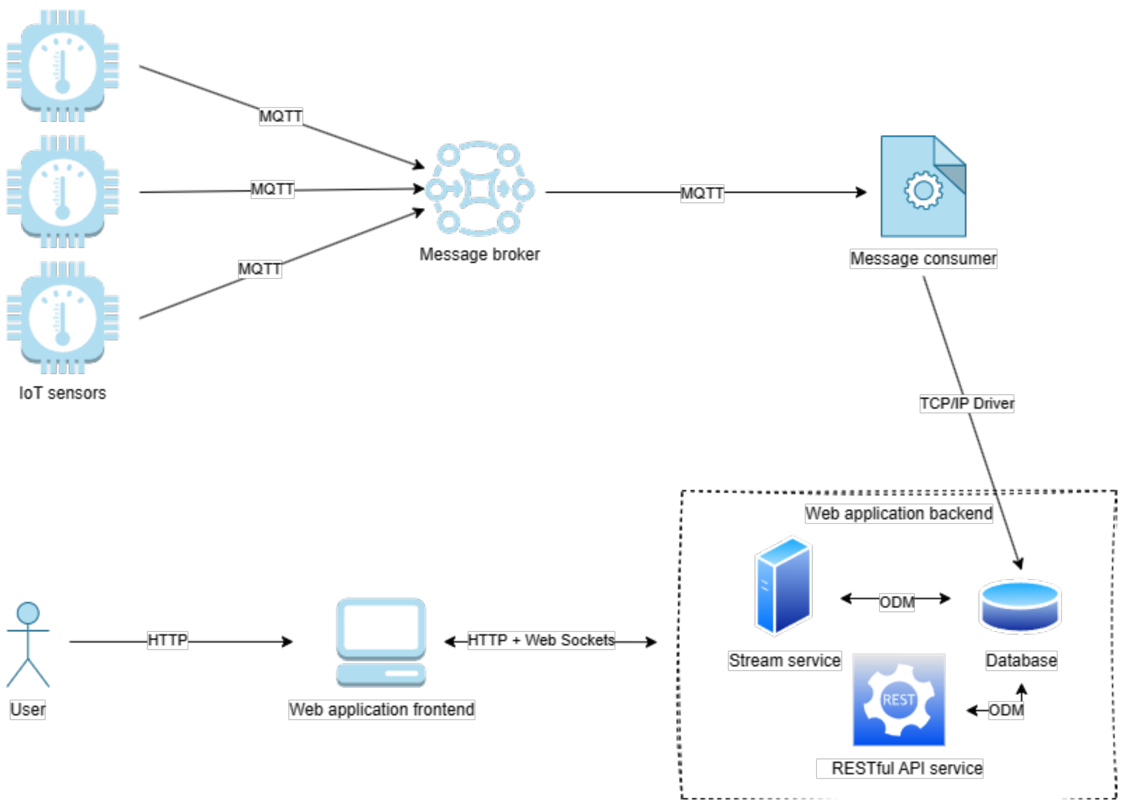


Figure 4: System architecture overview

The system architecture overview shown in Figure 4 illustrates the high-level components of the WeatherTrack system and their interaction protocols. Each of the components is deployed as an individual Docker container, allowing for easy scalability, development and management of the system.

The sensor and consumer scripts are implemented using *Python* and are responsible for collecting and processing weather data, respectively. The sensor script collects weather data from simulated IoT devices and sends it to the RabbitMQ message broker. The consumer script receives the data from the message broker, processes it, and stores it in the MongoDB database.

The web application is composed of a frontend and a backend, implemented using Vue.js and Node.js with Express.js, respectively. The frontend provides a user interface for visualizing and analyzing the collected weather data, while the backend provides a RESTful API for the frontend to interact with the system.

The RabbitMQ message broker acts as an intermediary between the sensor and consumer scripts, allowing for asynchronous communication and decoupling between the components. This enables the system to handle large volumes of data and provides fault tolerance in case of component failures.

The MongoDB database is used to store the collected weather data and provides a scalable and flexible data storage solution. The database is designed to handle large volumes of data and provides efficient querying and indexing capabilities.

The use of Docker containers for deploying the components provides a scalable and portable solution for the system. Each component is deployed as an individual container, allowing for easy management and scaling of the system. The containers are connected using a local network, providing a simple and efficient communication between the components.

3.1.4 Addressing the non-functional requirements

The designed system architecture provides a scalable and fault-tolerant solution for collecting and processing real-time weather data from a network of IoT sensors, addressing the non-functional requirements identified in sections 1 and 2 of this report.

With respect to the non-functional requirements of scalability, fault-tolerance and modularity the following design choices are here underlined:

- Each component of the system is deployed in an individual Docker container. This ensures that each component is isolated from the others, and any failures or issues within one container do not affect the others. If a container fails, it can be easily restarted or replaced without affecting the rest of the system.

- The use of a message broker such as RabbitMQ allows for asynchronous communication between the sensor and consumer components. This means that even if the consumer is down or overwhelmed with data, the sensor can continue to send data to the message broker, which will hold the messages until the consumer is ready to process them.

- The use of Docker-compose for container orchestration allows for easy scaling of the system by adding more instances of a particular component. For example, if the system is receiving too much data for a single consumer to handle, additional consumer instances can be added to distribute the load.

- The system can be made more fault-tolerant by adding redundancy to critical components, such as the weather sensors, message broker and database. This can be achieved by setting up multiple instances of these components and configuring them for high availability and failover.

3.2 Behaviour

3.2.1 Weather Sensor

The weather sensor component is responsible for simulating an actual physical weather sensor by collecting weather data from an external API and publishing it to a RabbitMQ queue (the message broker component). The behavior of the weather sensor component can be described using a state diagram, as shown in the following figure.

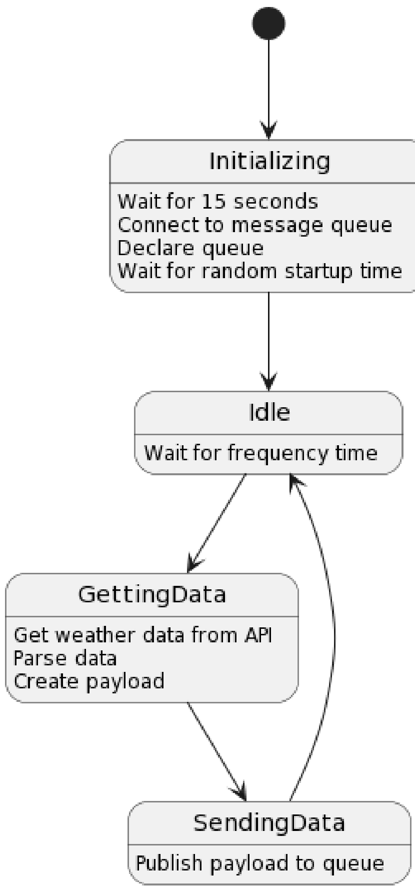


Figure 5: UML State Diagram for the weather sensor

The weather sensor component starts in the *Initializing* state, where it waits for an arbitrary amount of time (set in the configuration files) so that the broker is running, connects to the broker, declares a queue, and waits for a random startup time. After the initialization phase, the component transitions to the *Idle* state, where it waits for the frequency time specified in the configuration.

Once the frequency time has elapsed, the component transitions to the *GettingData* state, where it retrieves weather data from the external API, parses it, and creates a payload containing the relevant information. The payload includes the sensor's location, identifier and data points.

After creating the payload, the component transitions to the *SendingData* state, where it publishes the payload to the RabbitMQ queue. Once the message is sent, the component transitions back to the Idle state, where it waits for the next cycle.

To ensure fault-tolerance and maintain high availability of the weather monitoring service, Docker Compose's *scale* command is utilized. By scaling the sensor service, multiple instances of sensors are run for the same location, guaranteeing service availability even if one of the sensors fails.

To simulate independent operation, a random wait is introduced before each sensor starts sending data.

In the rare event that all sensors for a specific location fail, it's crucial to keep end-users informed about the freshness of the weather data. To achieve this, the user interface displays timestamps for the latest weather updates. By showing the timestamp of the last successful update, users can determine if the information is up to date or outdated.

3.2.2 Message Broker

The message broker is responsible for receiving and processing messages from the weather sensors and forwarding them to the appropriate consumers. In the WeatherTrack system, RabbitMQ is used as the message broker.

RabbitMQ is an open-source message broker software that accepts and forwards messages. It uses MQ Telemetry Transport (MQTT) to enable communication between different components of a distributed system. RabbitMQ is a popular choice because of its ease of use, reliabilty, scalability and the fact that it's open source.

The behavior of the message broker can be represented using a state diagram, as shown in the figure below:

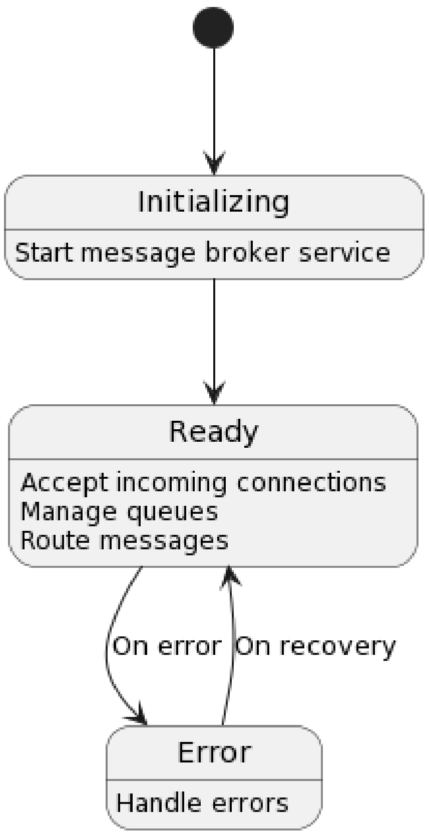


Figure 6: UML State Diagram for the message broker

The message broker starts in the *Initializing* state, where it starts the RabbitMQ service. Once the service is started, it moves to the Ready state, where it waits for incoming connections from the weather sensors and consumers. In the *Ready* state, it also manages queues and routes messages to the appropriate consumers.

If an error occurs while processing messages, the message broker moves to the Error state, where it handles the error. Once the error is handled, it moves back to the *Ready* state to continue processing messages.

RabbitMQ works by creating queues, which are used to store messages. Producers send messages to the queues, and consumers receive messages from the queues. RabbitMQ uses a routing mechanism to route messages from producers to consumers. The routing mechanism can be configured using exchanges, bindings, and routing keys. Exchanges are used to receive messages from producers and route them to queues. Bindings are used to define the relationship between exchanges and queues, and routing keys are used to specify which messages should be routed to which queues.

In WeatherTrack, the weather sensors act as producers, sending messages containing weather data to the RabbitMQ queues. The message broker then routes these messages to the appropriate consumers, which process and store the data in the database.

Because weather anomalies are by nature infrequent, they were manually tested. Utilizing the RabbitMQ web interface, anomalous weather events were added to the queue manually, verifying that the notification system correctly alerts users and handles the anomalies as expected.

3.2.3 Message Consumer

The consumer is responsible for receiving and processing messages from the RabbitMQ queue. The behavior of the consumer can be described using a state diagram as follows:

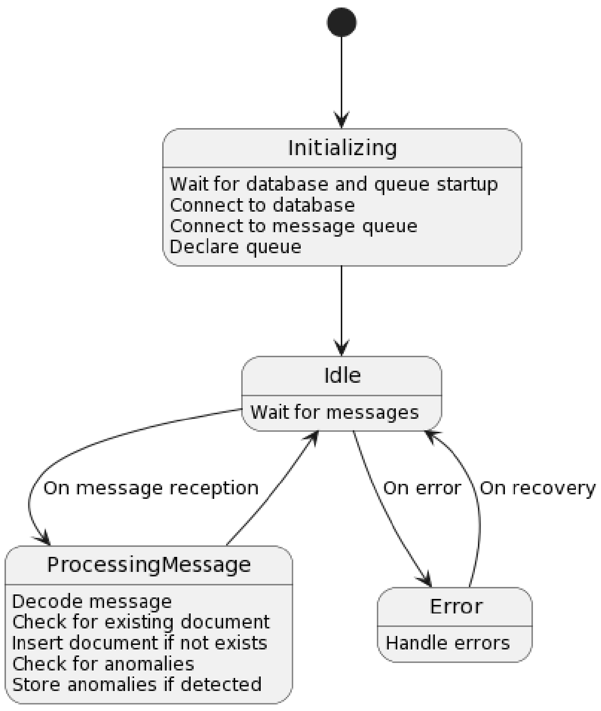


Figure 7: UML State Diagram for the message consumer

The consumer starts in the *Initializing* state, where it connects to the RabbitMQ broker and declares the queue. Once the initialization is complete, the consumer transitions to the *Idle* state, where it waits for messages to arrive.

When a message is received, the consumer transitions to the *ProcessingMessage* state. In this state, the consumer parses the message, checks if a document with the same timestamp and location already exists in the database, inserts the new document if it doesn't exist, checks for anomalies, and stores any detected anomalies in the database.

If an error occurs during the processing of a message, the consumer transitions to the *Error* state, where it handles the error. Once the error is handled, the consumer transitions back to the Idle state.

The consumer uses the *pika* library to connect to the RabbitMQ broker and consume messages from the queue. It also uses the *pymongo* library to interact with the MongoDB database. The consumer runs continuously, waiting for messages to arrive and processing them as they do.

3.2.4 Database

The database for the WeatherTrack system is implemented using MongoDB, a popular NoSQL document database. MongoDB was chosen for its flexibility, scalability, and performance in handling large volumes of unstructured data. The database schema consists of the following collections:

- **DataPoints**: This collection stores the weather data collected by the sensors. Each document in the collection represents a single data point, which includes the sensor ID, timestamp, environmental variables (like temperature and humidity).
- **Anomalies**: This collection stores the detected weather anomalies, such as extreme temperatures or sudden changes in humidity. Each document in the collection represents a single anomaly, which includes the sensor ID, timestamp, type of anomaly, and other relevant information. The schema for this collection is designed to support efficient querying and retrieval of anomalies based on different criteria, such as time range, location, and anomaly type.
- **Users**: This collection stores the user accounts for the WeatherTrack system. Each document in the collection represents a single user, which includes the user ID, username, password hash, email address, and role (admin or regular user). The schema for this collection is designed to support user authentication and authorization, as well as user management functions such as account creation and deletion.
- **Requests**: This collection stores the user requests for new sensors or features. Each document in the collection represents a single request, which includes the user ID, request type, request details, and other relevant information. The schema for this collection is designed to support efficient tracking and management of user requests, as well as communication between users and administrators.
- **Locations**: This collection stores the location information for each sensor. Each document in the collection represents a single location, which includes the sensor ID, latitude, longitude, and potentially other geographic information.

The database is deployed locally using Docker containers, which allows for easy deployment and scaling of the system. MongoDB replica sets are used to ensure scalability and availability of the database. A replica set is a group of MongoDB instances that maintain identical copies of the database. One instance acts as the primary node, which receives all write operations, while the secondary nodes replicate the data from the primary node. In case the primary node fails, one of the secondary nodes is automatically elected as the new primary node.

Replica sets are also used to set up change streams for real-time data. Change streams allow the system to listen to changes in the database and react to them in real-time. This is used to update the web client with new data points as they are collected by the sensors. The change streams are set up on the secondary nodes of the replica set, which ensures that the primary node is not overloaded with read operations.

The Mongoose framework was used to interact with the database in the web server and to map the database documents to objects.

3.2.5 Web server

The backend application is designed using Node.js and Express.js. The application is responsible for handling user requests, managing data, and providing real-time updates to connected clients through web sockets.

The state diagram below provides a high-level overview of the backend's lifecycle:

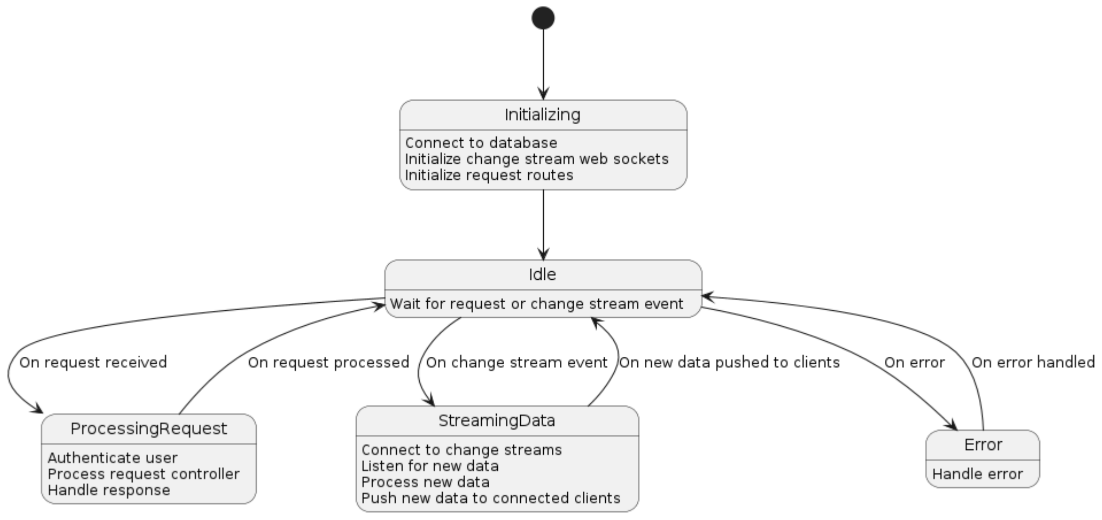


Figure 8: UML State Diagram for the web application backend

The backend starts in the *Initializing* state, where it connects to the MongoDB database, initializes change stream web sockets, and sets up request routes. Once the initialization is complete, the backend moves to the *Idle* state, where it waits for incoming requests or change stream events.

When a request is received, the backend transitions to the *ProcessingRequest* state. In this state, the backend authenticates the user, processes the request using the appropriate controller, and sends a response back to the client. If an error occurs during the processing of a request, the backend transitions to the *Error* state.

When a change stream event occurs, the backend transitions to the *StreamingData* state. In this state, the backend connects to the change streams, listens for new data, processes the new data, and pushes real-time updates to connected clients through web sockets. If an error occurs during the processing of a change stream event, the backend transitions to the Error state.

3.2.6 Web client

The web frontend is a Single Page Application is built using Vue.js. It provides a dynamic and interactive user experience, allowing users to view and interact with weather data in real-time.

At a high level, the frontend can be divided into three main components: the navigation bar, the notification component, and the router view. The navigation bar provides links to different pages of the application, while the notification component displays alerts and messages to the user. The router view is responsible for rendering the different pages of the application based on the user's navigation in the same page, following the Single Page Application approach.

The frontend communicates with the backend API using *axios*, a promise-based HTTP client for the browser and *Node.js*. The API requests are intercepted by a middleware, which adds an authorization header with the user's token to each request.

3.3 Interaction

3.3.1 Data collection

In the data collection process, the sensor collects data from its environment and publishes a message with the payload containing the data to the broker. The broker then sends the message to the consumer, which processes the data and interacts with the database to store and retrieve information.

When the consumer receives the message from the broker, it first checks if a document with the same payload already exists in the database. If it does, the consumer skips the insertion process and returns the existing document. This is to prevent duplicate data from being stored in the database.

If the document does not exist, the consumer inserts the payload into the database as a new document. The database confirms the insertion, and the consumer then checks for any anomalies in the data based on predefined thresholds.

If an anomaly is detected, the consumer stores the anomaly in the database along with the relevant data. The database confirms the storage, and the consumer can then take any necessary actions based on the detected anomaly.

If no anomaly is detected, the consumer simply records that no anomaly was detected for the given payload.

The use of a broker allows for decoupling between the sensor and consumer, providing greater flexibility and scalability in the system.

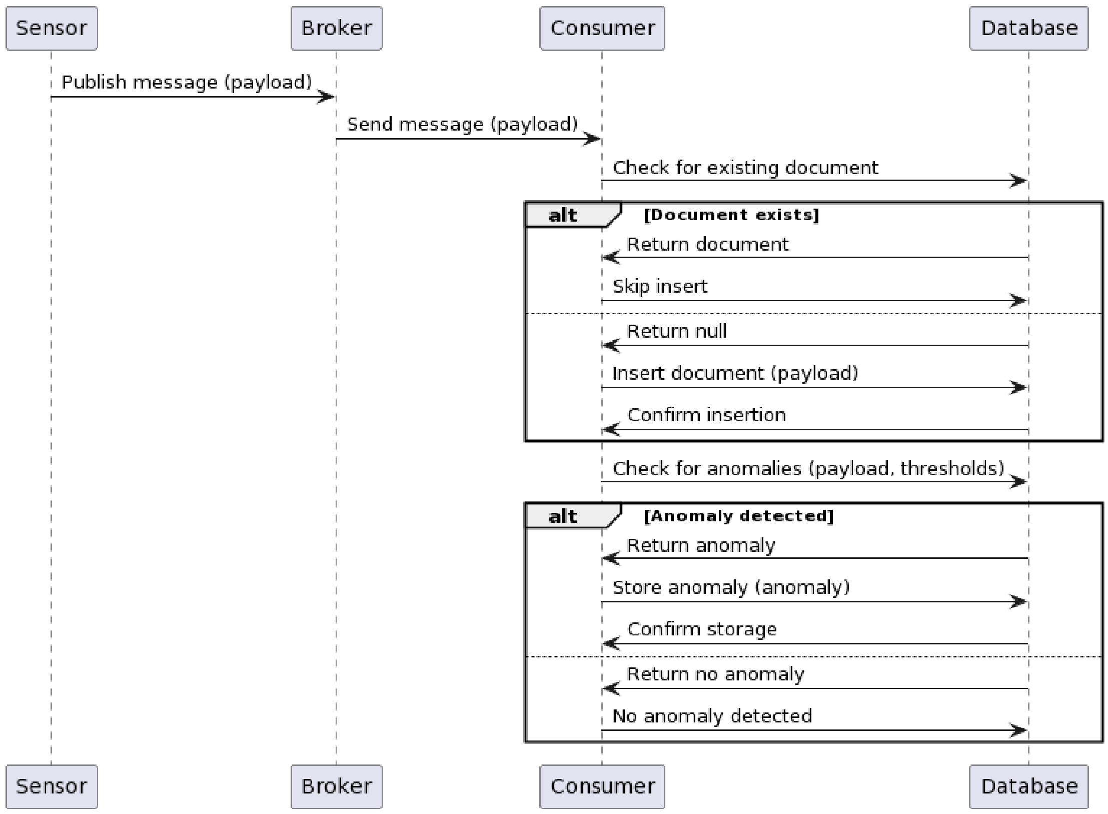


Figure 9: UML Sequence Diagram for data collection and storage

3.3.2 Authentication

For the registration, the process begins with the Client sending a registration request to the *AuthController* with user data, which includes the user's username, email, and password. If the user fails to enter all the required fields, the *UserModel* returns an error message to the AuthController, which in turn sends a response with an error message to the Client.

Assuming all the required fields are entered, the *AuthController* creates a new user object and sends it to the *UserModel* to be inserted into the database. Before inserting the new user, the *UserModel* checks if the username or email already exists in the database. If either the username or email already exists, the *UserModel* returns an error message to the *AuthController*, which then sends a response with an error message to the Client.

If the username and email do not already exist, the *UserModel* proceeds to insert the new user into the database. Once the user is successfully inserted, the Database sends a confirmation to the *UserModel*, which then sends a confirmation to the *AuthController*. Finally, the *AuthController* sends a response with a success message to the Client. The Client will now be redirected to the login page where they can access the service with their newly registered account.

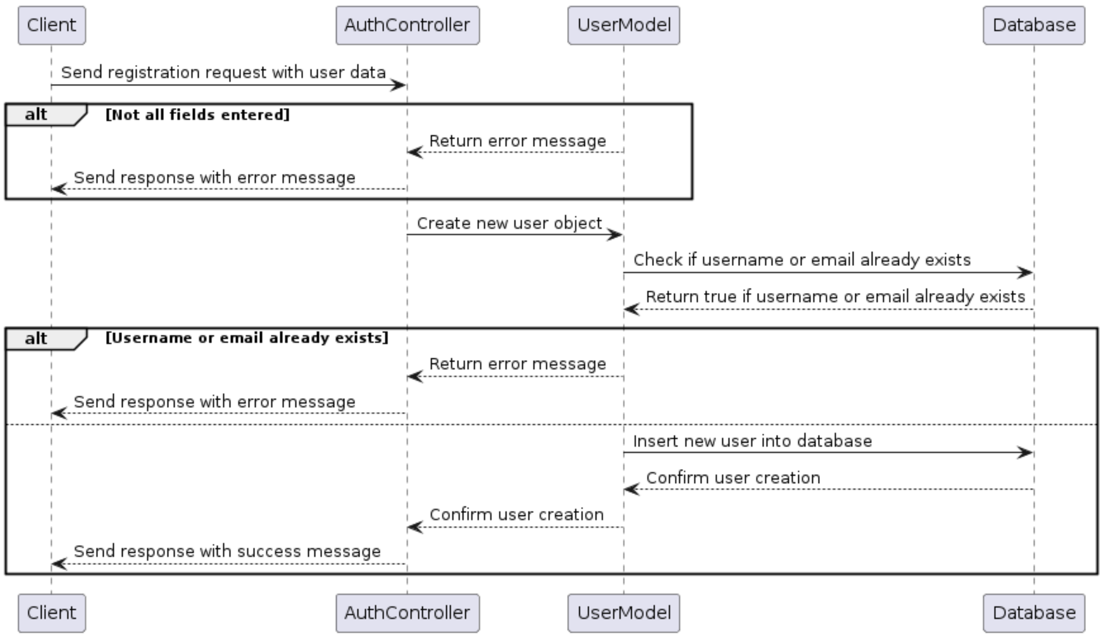

Client
AuthController
UserModel
Database
Send registration request with user data
alt
[Not all fields entered]
Return error message
Send response with error message
Create new user object
Check if username or email already exists
Return true if username or email already exists
alt
[Username or email already exists]
Return error message
Send response with error message
Insert new user into database
Confirm user creation
Confirm user creation
Send response with success message
Client
AuthController
UserModel
Database

Figure 10: UML Sequence Diagram for the registration process

For the login, when a user attempts to log in, the client sends a request to the authentication controller with their username and password. The authentication controller then communicates with the user model to find a user with the given username in the database. If a user is found, the user model returns the user object to the authentication controller.

Next, the authentication controller sends the user's password and the password stored in the user object to a password validator. The password validator compares the two passwords and returns a boolean value indicating whether they match or not.

If the passwords match, the authentication controller generates a *JSON Web Token* (*JWT*) with the user's data, including their role (either *admin* or *user*). JWT is a standard for securely transmitting information between parties, commonly used for authentication and authorization purposes. It consists of three parts: a header, a payload, and a signature. The header contains metadata about the token, such as the algorithm used to sign it. The payload contains the user's data in the form of key-value pairs, known as claims. The signature is used to verify the authenticity of the token.

In this case, the payload includes the user's ID, username, and role. The role field is used to distinguish between regular users and admin users. Regular users have limited permissions, while admin users have full access to all features of the application.

Once the JWT is generated, it is sent back to the client as a response. The client can then use the token to authenticate subsequent requests to the server. When a request is received with a valid JWT, the server can verify the user's identity and permissions based on the information contained in the token.

If the passwords do not match, the authentication controller sends an error message back to the client. The client can then display an error message to the user, indicating that their login attempt was unsuccessful.

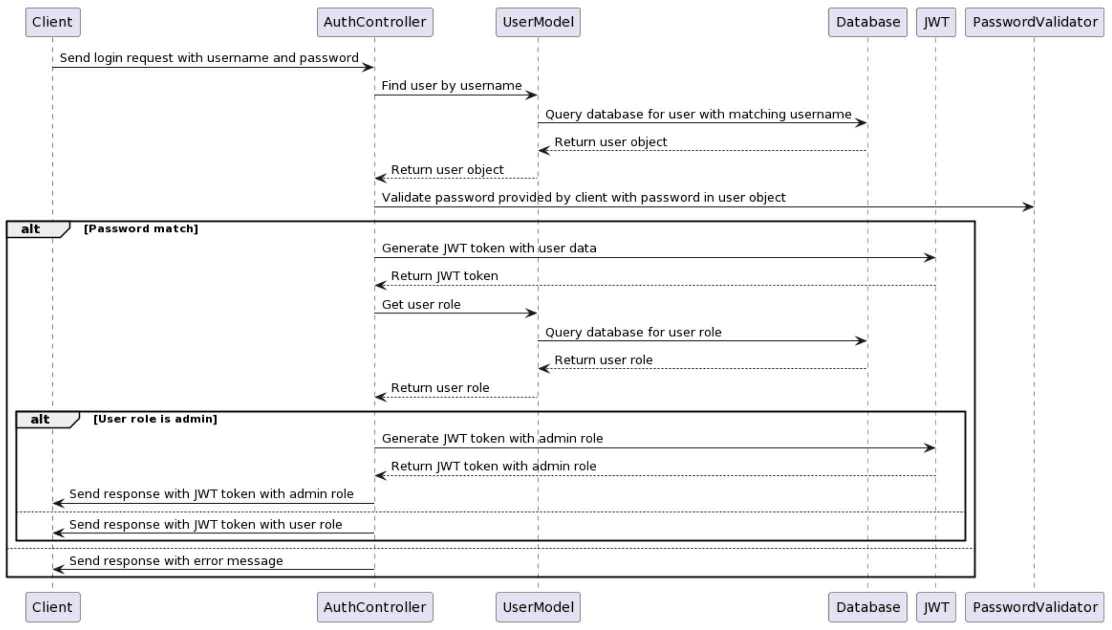

Client
AuthController
UserModel
Database
JWT
PasswordValidator
Send login request with username and password
Find user by username
Query database for user with matching username
Return user object
Return user object
Validate password provided by client with password in user object
alt [Password match]
Generate JWT token with user data
Return JWT token
Get user role
Query database for user role
Return user role
Return user role
alt [User role is admin]
Generate JWT token with admin role
Return JWT token with admin role
Send response with JWT token with admin role
Send response with JWT token with user role
Send response with error message

Figure 11: UML Sequence Diagram for the login process

3.3.3 Client-server interactions

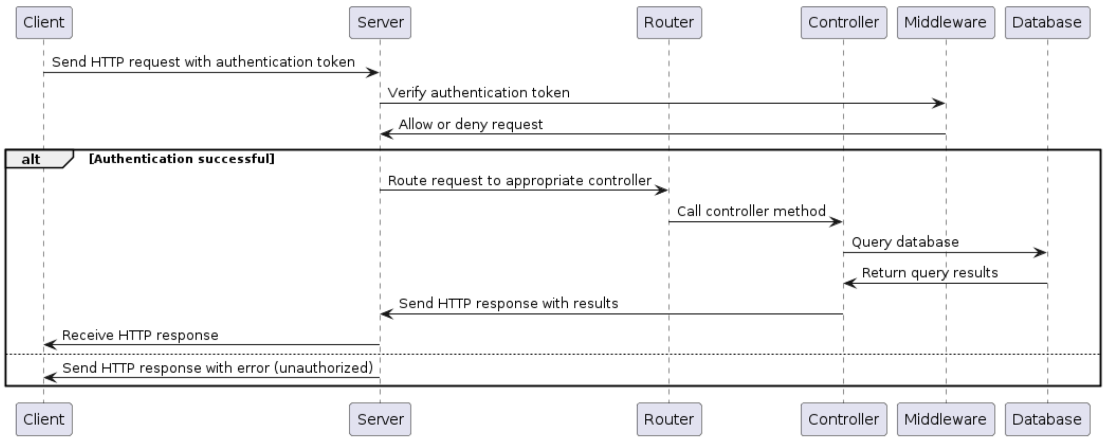

Client
Server
Router
Controller
Middleware
Database
Send HTTP request with authentication token
Verify authentication token
Allow or deny request
alt [Authentication successful]
Route request to appropriate controller
Call controller method
Query database
Return query results
Send HTTP response with results
Receive HTTP response
Send HTTP response with error (unauthorized)

Figure 12: UML Sequence Diagram for HTTP requests

This sequence diagram illustrates the interaction between the server and the client for HTTP API requests. The client sends a request to the server, which is routed to the appropriate controller through the server's router. The controller processes the request and interacts with the database as necessary. If the request requires authentication, the server's middleware is used to validate the client's credentials before processing the request. The server then sends a response back to the client with the requested data or an error message if the request could not be fulfilled.

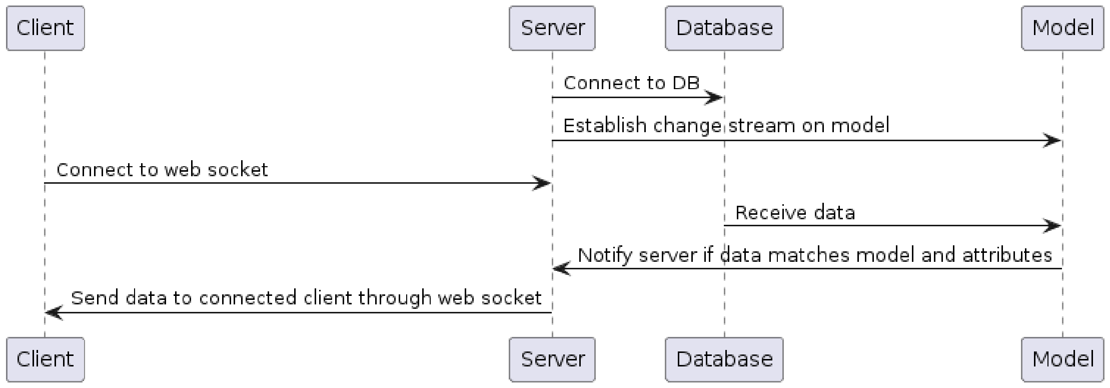

Client
Server
Database
Model
Connect to DB
Establish change stream on model
Connect to web socket
Receive data
Notify server if data matches model and attributes
Send data to connected client through web socket

Figure 13: UML Sequence Diagram for real-time updates

This sequence diagram illustrates the interaction between the server and the client for real-time (which also includes anomaly notifications). The server connects to the database and establishes a change stream on a specific model. The client connects to a web socket to receive real-time updates. When the database receives new data that matches the model and attributes, the server is notified and sends the data to the connected client through the web socket

4 Implementation Details

4.1 Web server

4.1.1 RESTful APIs

The WeatherTrack system's web server provides a set of RESTful APIs for interacting with the system's backend.

The WeatherTrack system uses Swagger to document its APIs, making it easier to understand and interact with the system. Swagger is an open-source software framework used to document and test RESTful APIs. The Swagger specification defines the API's endpoints, request and response formats, and authentication requirements. The Swagger documentation for the WeatherTrack system is generated automatically from the code using Swagger annotations. The generated documentation is included in the repository and can also be generated by running the following command (requires *node*):

```
node app-backend/swagger.js
```

The following listing shows an example of an API route, defined in a router file in the *routes* module of the backend code, along with its swagger annotation and the corresponding controller function, implemented in the *controller* module of the backend code.

```
const express = require("express");
const authController = require("../controllers/
authController");
const router = express.Router();

 /**
 * @swagger
 * /auth/register:
 *   post:
 *     summary: Register a new user
 *     requestBody:
 *       required: true
 *       content:
 *         application/json:
 *           schema:
 *             $ref: '#/components/schemas/User'
 *     responses:
 *       '200':
 *         description: User registered successfully
 *       '400':
 *         description: Username or email already exists
 *       '500':
 *         description: Server error
 */
router.route("/auth/register").post(authController.register);

    exports.register = (req, res) => {
  if (!req.body.username ||
     !req.body.email ||
     !req.body.password) {
    return res.status(500)
    .send({
      message: "All fields are required"
  });
  }

  User.findOne({ $or: [{ username: req.body.username },
     { email: req.body.email }] })
    .then((existingUser) => {
      if (existingUser) {
        return res.status(400).send({
          message: "Username or email already exists"
        });
      }

      const user = new User({
        username: req.body.username,
        email: req.body.email,
        password: bcrypt.hashSync(req.body.password, 8),
        role: req.body.role ? req.body.role : "user",
      });

      user
        .save()
        .then(() => {
          res.send({
            message: "User was registered successfully!"
          });
        })
        .catch((err) => {
          res.status(500).send({ message: err });
        });
    })
    .catch((err) => {
      res.status(500).send({ message: err });
    });
};
```

4.1.2 Real-time Events

The WeatherTrack system uses MongoDB Change Streams and WebSockets to provide real-time updates to the web client. MongoDB Change Streams is a feature of MongoDB that allows applications to listen to changes in the database in real-time. WebSockets provide a bi-directional communication channel between the web server and the web client, allowing the server to push real-time updates to the client.

The following code snippet shows an example of how MongoDB Change Streams and WebSockets are used in the WeatherTrack backend to provide real-time updates to the web client. specifically, it is used to dinamically instantiate a change stream and websocket for a given location: once a document is added to the dataPoints collection for that location, it will be emitted to the connected clients.

```
async function setupDataStream(io, location) {
  console.log(`Setting up data point stream
  for location: ${location}`);

  const dataSocket = io.of(`/data-${location}`);

  dataSocket.on("connection", (socket) => {
    console.log(`A user connected to the data stream
    for location: ${location}`);

    const pipeline = [
      {
        $match: {
          operationType: "insert",
          "fullDocument.location": location,
        },
      },
    ];

    const changeStream = Data.watch(pipeline);

    changeStream.on("change", (change) => {
      console.log("Change in dataPoints collection:", change);

      if (change.operationType === "insert") {
        dataSocket.emit("newData", change.fullDocument);
        console.log("Emitting new data point to clients: ", +
         change.fullDocument);
      }
    });
  });
}
```

4.2 Web client

4.2.1 Vue Router

The WeatherTrack system's web client uses the Vue.js Router to handle client-side routing, allowing users to navigate between different pages of the application without requiring a full page reload. The routing configuration defines the different routes of the application, their corresponding components, and any necessary meta information, such as authentication and authorization requirements.

Here is the example of a route:

```
const routes = [
  .
  .
  .
  {
    path: "/register",
    component: Registration,
    meta: { requiresAuth: false, requiresAdmin: false },
  },
  .
  .
  .
];
```

4.2.2 Vue Store

The WeatherTrack system's web client uses *Vuex*, a state management library for Vue.js, to manage the application's state. Vuex provides a centralized store for all the components in an application, with a strict unidirectional data flow. This makes it easier to manage the application's state. The Vuex store in the WeatherTrack system is used to store the user's authentication state, the selected location, and the real-time weather data received from the web server. The store is also used to handle actions such as user login and logout and role.

4.2.3 WebSockets

The WeatherTrack system's uses client-side WebSockets to receive real-time updates from the web server. The web client establishes a WebSocket connection with the web server when the application is loaded, and the connection is kept open until the user closes the page. The web server sends real-time weather data to the web client through the WebSocket connection, and the web client updates the user interface accordingly.

4.2.4 API middleware

The frontend API middleware is implemented using Axios. It specifies once the URL for the server and sets up an interceptor for all outgoing API requests. The interceptor checks if a user is currently authenticated by looking for a token stored in the Vuex state management system. If a token is present, it is added to the request headers as an Authorization bearer token. This ensures that all subsequent API requests will be authenticated and authorized by the server, providing an additional layer of security. By centralizing the token handling in the middleware, it also simplifies the code in other parts of the application (DRY principle).

The following listing shows the API middleware implementation:

```
    const api = axios.create({
  baseURL: "http://localhost:3000",
});

api.interceptors.request.use(
  (config) => {
    const token =
    store.state.authModule.token;
    if (token) {
      config.headers.Authorization = token;
    }
    return config;
  },
  (error) => {
    return Promise.reject(error);
  }
);

export default api;
```

5 Self-assessment / Validation

5.1 Evaluation Criteria

The following criteria have been chosen to assess the software's compliance with the requirements:

1. **Functional Correctness**: The software should successfully perform all specified tasks and operations as outlined in the functional requirements.
2. **Non-Functional Correctness**: the software should meet the implicit non-functional requirements; those are scalability, fault tolerance, low latency, high availability, and security.
3. **Usability:** The software should provide an intuitive and user-friendly interface, making it easy for all the envisioned users to accomplish their tasks.

5.2 Evaluation

5.2.1 Unit testing

Unit tests have been created for both the components that collect data (Python) using the Python *unittest* module and unit tests for the web application backend (JavaScript) using Jest. This section explains how unittest and Jest work, and discusses the integration of these tests into the GitLab CI/CD pipeline, along with the benefits of doing so.

The Python *unittest* module is a built-in testing framework for writing and running unit tests in Python. It provides a set of tools for creating test cases, test suites, and test runners, making it easy to organize and execute tests. The *unittest* module follows the *xUnit* testing pattern, which is a popular testing framework structure used in many programming languages.

To create a test case using *unittest*, you need to define a class that inherits from the *unittest.TestCase* class. Within this class, you can create individual test methods, each starting with the prefix test_. The *unittest* module provides several assertion methods, such as *assertEqual()*, *assertTrue()*, and *assertRaises()*, to verify the expected behavior of the code under test. The following listing shows an example of such Python unit test:

```
import unittest
import os
from sensor import sensor

class TestWeatherDataProcessor(unittest.TestCase):

    def test_data(self):

        payload = sensor.getData()
        self.assertIn("timestamp", payload)
        self.assertIn("temperature", payload)
        self.assertIn("humidity", payload)
        self.assertEqual(payload["location"],
                         os.getenv("SENSOR_LOCATION"))

if __name__ == '__main__':
    unittest.main()
```

Jest is a popular JavaScript testing framework developed by Facebook. Jest provides features like snapshot testing, mocking, and code coverage analysis, making it a comprehensive solution for testing JavaScript applications. It was used to create unit tests to automatically test the backend functionalities. Below is an example of a testing function for one of the APIs.

```
describe("GET /data/latest", () => {
  it("should return the latest data", async () => {
    const res = await api
      .get("/data/latest")
      .set("Authorization", token)
      .query(data);

    expect(res.statusCode).toBe(200);
    expect(res.body).toHaveProperty("temperature");
  });

  it("should not return the latest data without" +
  "required query parameters",
   async () => {
    const res = await api.get("/data/latest")
    .set("Authorization", token);
    expect(res.statusCode).toBe(400);
    expect(res.body.error)
    .toMatch(/Missing required query parameter/);
  });
```

A Gitlab CI/CD pipeline has been setup to run these tests automatically anytime new code is pushed to the project's repository. The following is the pipeline configuration file:

```
test-js:
  stage: test
  image: node:18.0
  services:
    - mongo:latest
  variables:
    MONGODB_HOST: "test-mongo"
  script:
    - cd app-backend
    - npm ci
    - npm test

test-py:
  stage: test
  image: python:3.9
  services:
    - mongo:latest
  variables:
    MONGODB_HOST: "mongo"
    SENSOR_LOCATION: "roma"
    SENSOR_LONGITUDE: "11.342616"
    SENSOR_LATITUDE: "44.494887"
    FREQUENCY: "10"
    API_KEY: $OPENWEATHER_API_KEY
  before_script:
    - pip install --upgrade pip
    - pip install --trusted-host pypi.python.org pika pymongo
    requests coverage
  script:
    - coverage run -m unittest discover -s ./tests
    -p '*_test.py'
    - coverage report -m
  coverage: '/TOTAL.\* (\[0-9\]{1,3}%)$/'
```

It can be noted that the API key is stored as a protected CI/CD variable in the GitLab pipeline. This approach helps keep sensitive information, such as API keys, secure, separate from the codebase and not visible to malicious actors.

The backend testing shows a code coverage of 85% for the Node backend, while the Python code has a coverage of 83%.

5.2.2 Integration testing

Integration tests were carried out incrementally as the system was being developed. Some integration tests are also bundled with the automatic tests in the pipeline. The final tests preceding deployment did not reveal any critical issues with respect to the requirements and the previously established criteria. In the future, it would be useful to make these tests completely automatic and have end-to-end integration tests.

5.2.3 User testing

The simulated user test was conducted with the help of volunteers who tested the system in the role of potential users. Two users, with limited computer skills and domain knowledge, tested the system as regular users. Two more experienced users, on the other hand, tested the system in the role of administrators.

The user tests performed with volunteers produced positive outcomes. Although these outcomes were not quantitatively measured, using formal tools like questionnaires, all users reported good usability, low latency and the ability to easily perform the tasks through the web interface.

5.2.4 Qualitative evaluation of the functional requirements

The qualitative assessment evaluates the software's compliance with the functional requirements. This involves testing the system's ability to fulfill the key functionalities outlined in the initial requirements, that is:

1. Real-time data collection and processing from simulated IoT sensors:
   - The system successfully collects and processes real-time weather data from the simulated sensor scripts, with data being published to the message broker and consumed by the message consumers.
   - The data is correctly stored in the MongoDB database, with the consumer scripts handling duplicate data detection and anomaly identification as specified.
2. Web application functionality for data visualization and analysis:
   - The *Vue.js* web application frontend is able to retrieve and display the collected weather data in real-time, allowing users to view current conditions, historical trends, and identified anomalies.
   - The web application provides the expected user interactions, such as the ability to filter and sort data, view detailed information about specific data points, and receive alerts for detected weather anomalies.
3. User management and request handling:
   - Thanks to *JWT*, the system successfully implements user authentication and authorization, with separate interfaces and permissions for regular users and administrators.
   - Administrators are able to manage user accounts and requests submitted by users for additional sensors or features, as per the requirements.
   - Users are able to submit requests and view the status of their requests through the web application.
4. Anomaly detection and notification:

- The message consumer scripts are able to successfully detect weather anomalies based on the predefined method and store the anomaly information in the database.
- The web application is able to retrieve and display the detected anomalies, allowing users to view real-time and historical weather events.

Overall, the qualitative assessment shows that the system is able to meet the key functional requirements identified for the WeatherTrack project.

5.2.5 Qualitative evaluation of the non-functional requirements

In the requirement analysis the following non-functional requirements were identified: scalability, fault-tolerance, low latency, high-availability and security.

- The scalability of the system is guaranteed by the technology stack in addition to the use of docker-compose.
- The fault-tolerance of the system is guaranteed by the system design with redundant components and a replicated database.
- Low latency and high-availability were observed in empirical testing and reported by the testers, though no quantitative measurement has been employed.
- The role-based system and JWT middleware provide the right authentication framework given the system's requirements although it should be noted that JWT alone is not enough to guarantee security as usually it is combined with other security measures such as HTTPS for secure transmission and proper server-side validation to prevent attacks like JWT tampering or replay attacks. These additional security measures were not implemented as they are outside of the scope of the project but they could be added in the future (see Future Works 8.1 below)

6 Deployment Instructions

The system was developed with a component-based logic. The orchestration of the components and the management of their interactions was handled with the *docker-compose* tool from *Docker*. With regard to build automation, *pip* was used as the package management system for the parts of the system developed in *Python* (sensor and message consumer), while *npm* was used for the parts developed in *JavaScript* (backend and frontend of the web application).

The repository also contains database dumps in JSON form which are imported running building the database image. These contain demonstration data along with some pre-registered users to test the system. The database Dockerfile, along with importing the demonstration data also initializes the replica set, which is required both for the real-time updates feature and the fault tolerance of the system.

To test the regular user scenario, use:

- Username: *user*.
- Password: *user*.

To test the administrator scenario, use:

- Username: *admin*.
- Passowrd: *admin*.

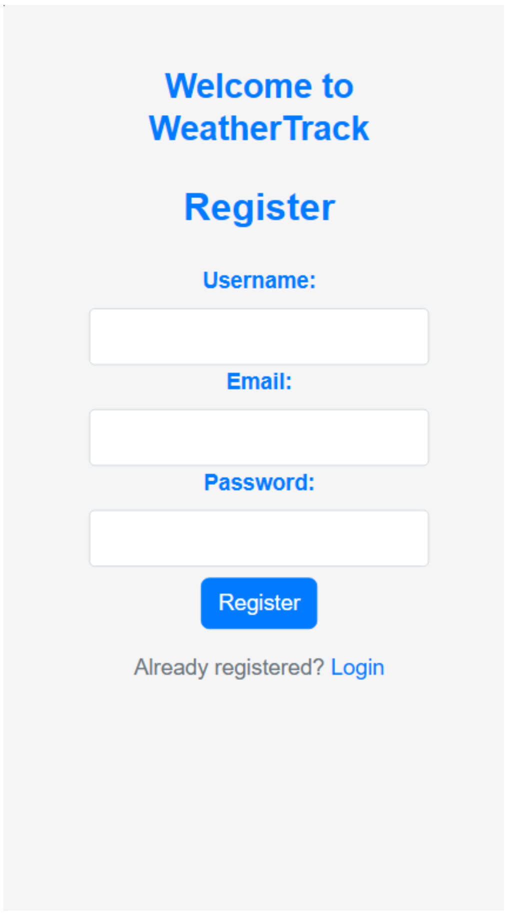
Also note that becauase the sensors are simulated through API calls, an OpenWeatherMap API key is required. The follwing API key can be used for testing the system and has to be passed as an argument upon running the system: *KEY*.

Here are the instructions for starting the system:

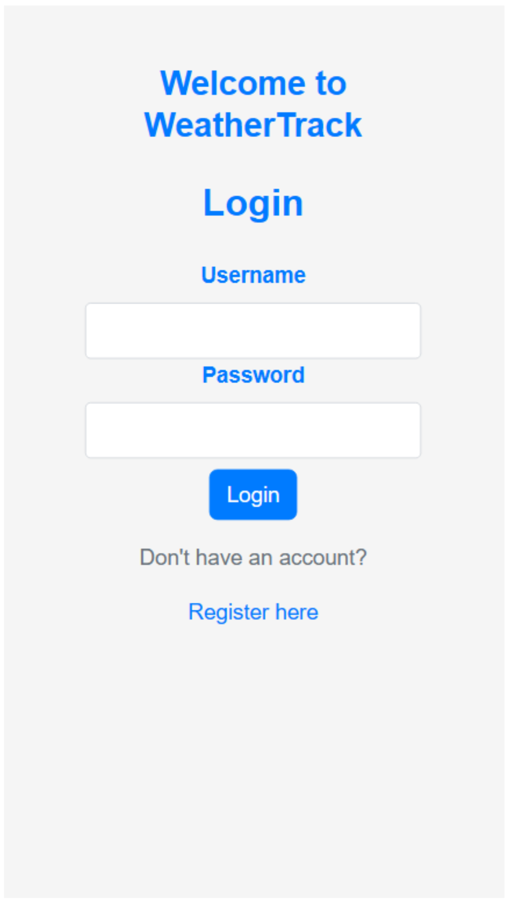
1. Install *Docker* and *docker-compose* on your machine.
2. Clone the project repository.
3. Navigate to *config/secrets.txt*, open and edit the file adding the OpenWeatherMap API key between the quotation marks (API_KEY="*your_key*").
4. Navigate to the root directory of the repository.
5. Run the command *docker-compose up –build*, this will build and start all of the system's containers.
6. Once the containers are up and running, you can access the web application by navigating to *http://localhost:5173* in your web browser.
7. To stop the system, run the command *docker-compose down* in the root directory of the repository.

7 Usage Examples

7.1 Registration and login



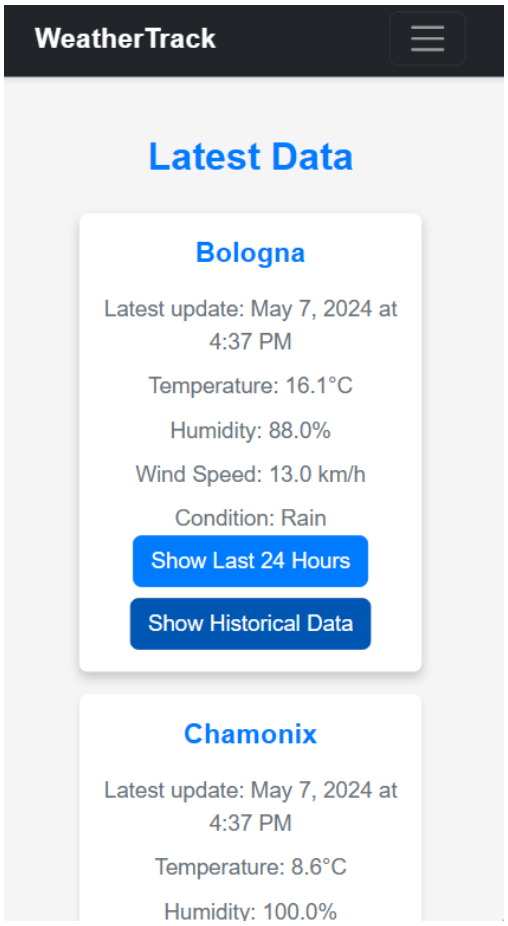
(a) Registration example view



(b) Login example view

Figure 14: Example views for authentication

7.2 Viewing weather data



WeatherTrack

Latest Data

Bologna

Latest update: May 7, 2024 at 4:37 PM

Temperature: 16.1°C

Humidity: 88.0%

Wind Speed: 13.0 km/h

Condition: Rain

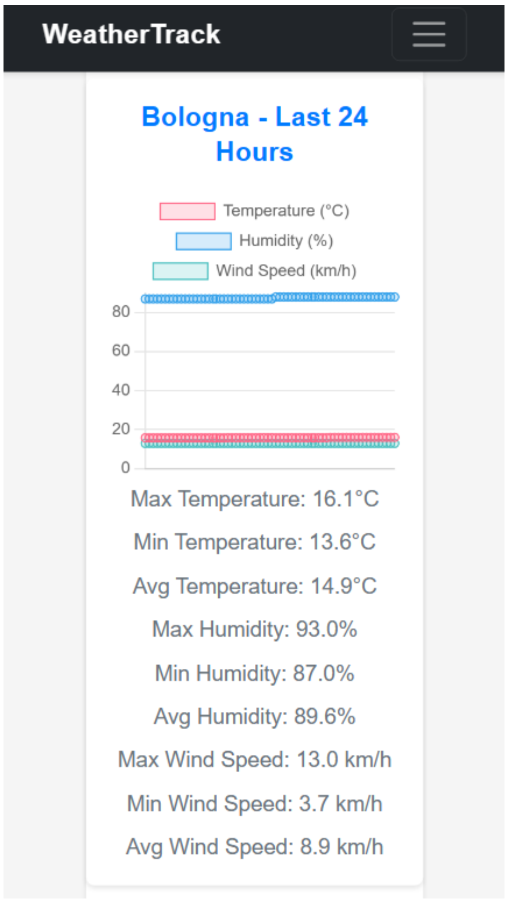
Chamonix

Latest update: May 7, 2024 at 4:37 PM

Temperature: 8.6°C

Humidity: 100.0%

(a) Data sheets example view



WeatherTrack

Bologna - Last 24 Hours

Max Temperature: 16.1°C

Min Temperature: 13.6°C

Avg Temperature: 14.9°C

Max Humidity: 93.0%

Min Humidity: 87.0%

Avg Humidity: 89.6%

Max Wind Speed: 13.0 km/h

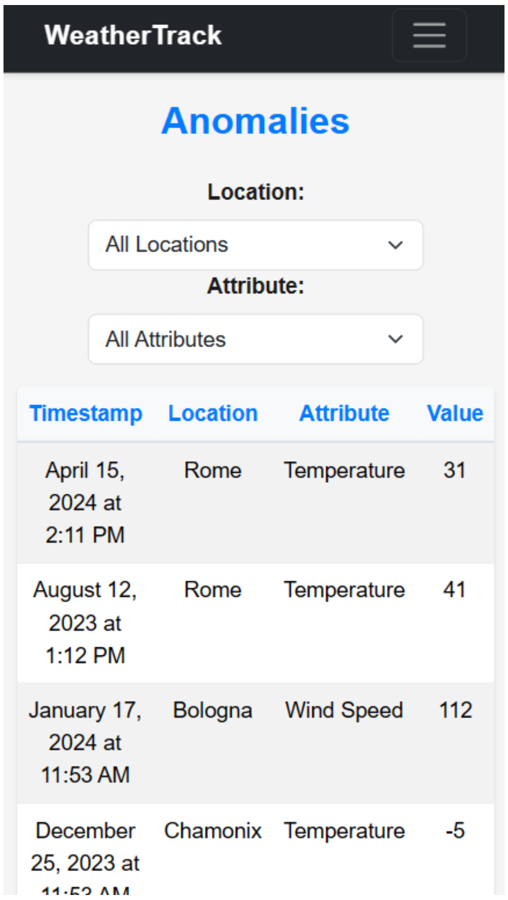
Min Wind Speed: 3.7 km/h

Avg Wind Speed: 8.9 km/h

(b) Last 24 hours example view

Figure 15: Data dashboard example views

7.3 Viewing weather anomalies



WeatherTrack

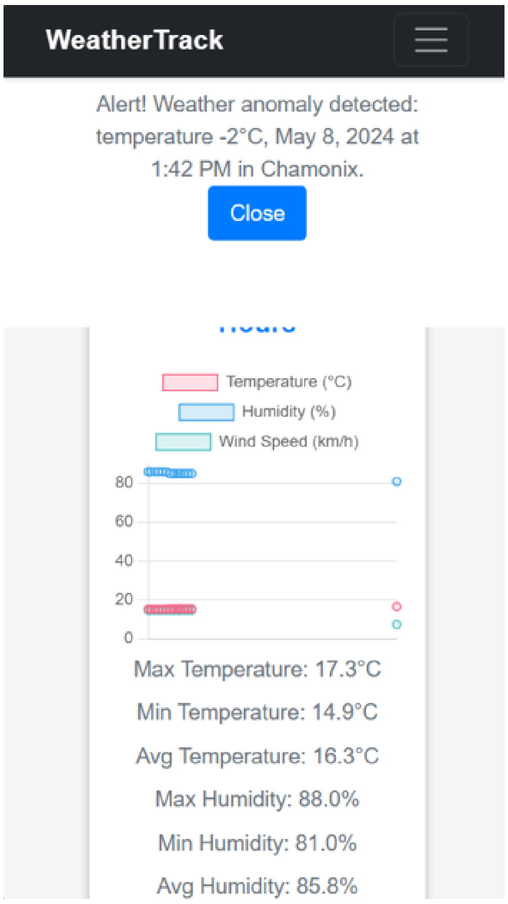
Anomalies

Location: All Locations

Attribute: All Attributes

Timestamp	Location	Attribute	Value
April 15, 2024 at 2:11 PM	Rome	Temperature	31
August 12, 2023 at 1:12 PM	Rome	Temperature	41
January 17, 2024 at 11:53 AM	Bologna	Wind Speed	112
December 25, 2023 at 11:53 AM	Chamonix	Temperature	-5

(a) Historical anomalies example view



WeatherTrack

Alert! Weather anomaly detected: temperature -2°C, May 8, 2024 at 1:42 PM in Chamonix.

Max Temperature: 17.3°C

Min Temperature: 14.9°C

Avg Temperature: 16.3°C

Max Humidity: 88.0%

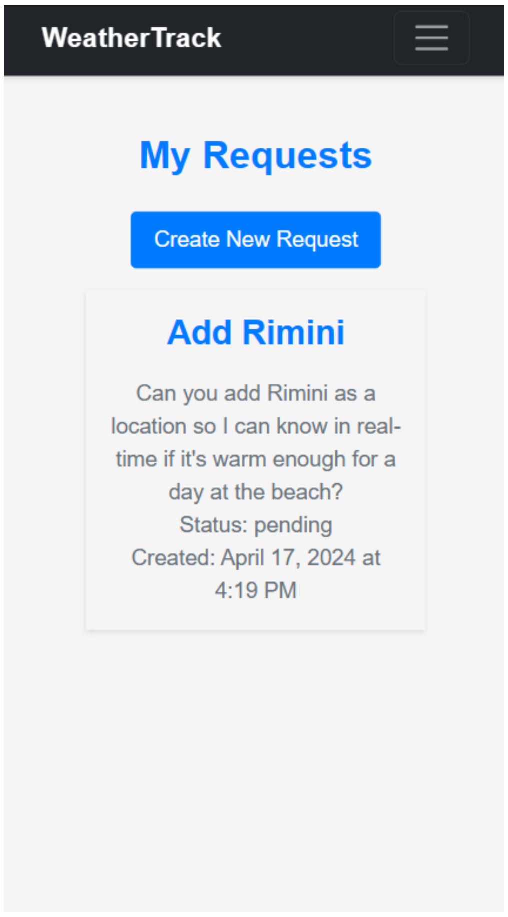
Min Humidity: 81.0%

Avg Humidity: 85.8%

(b) Anomaly notification example view

Figure 16: Anomaly example views

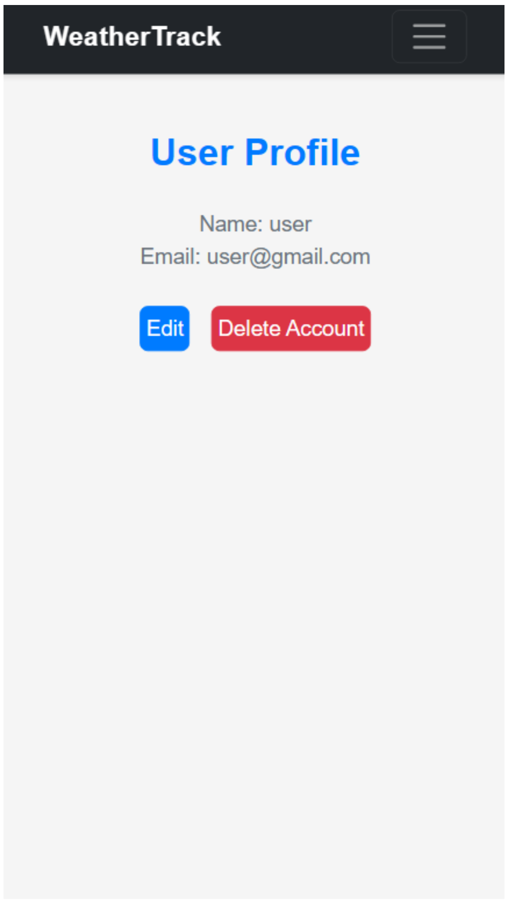
7.4 Manage own requests and profiles



Add Rimini

Can you add Rimini as a location so I can know in real-time if it's warm enough for a day at the beach?
Status: pending
Created: April 17, 2024 at 4:19 PM

(a) Manage requests view

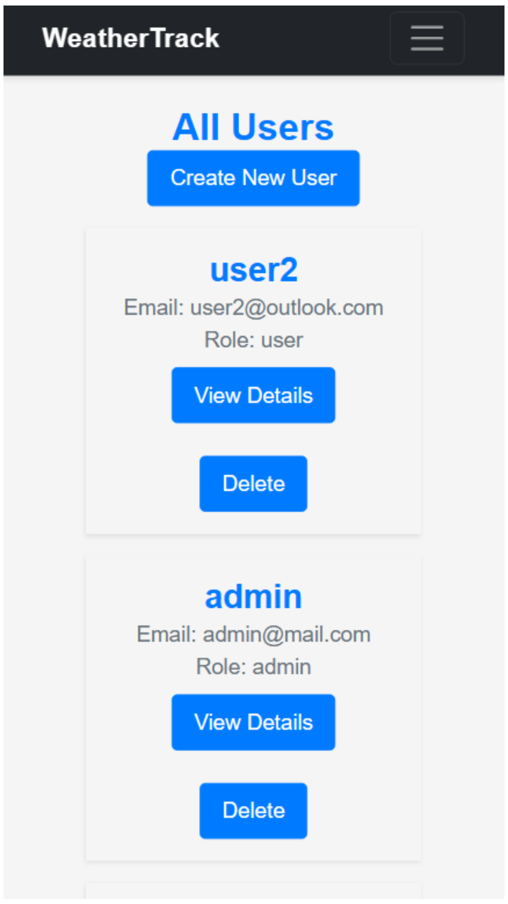


Name: user
Email: user@gmail.com

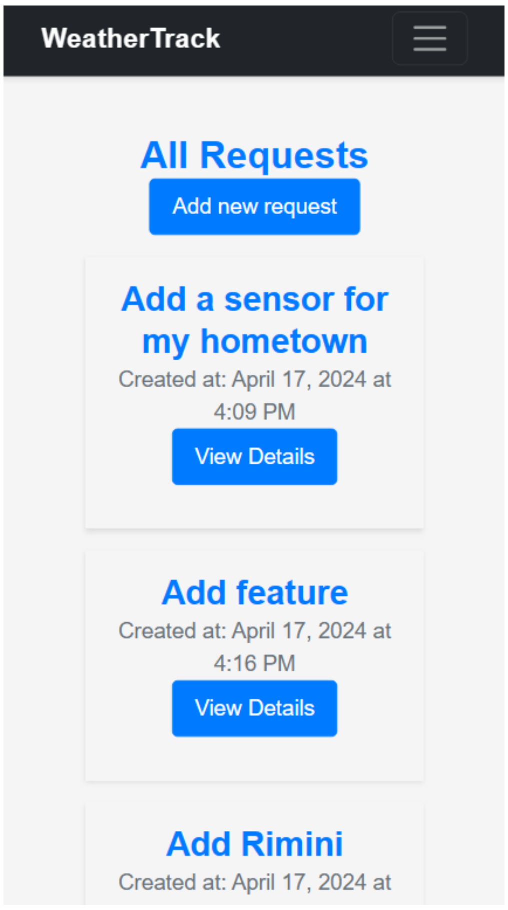
(b) Manage profile view

Figure 17: Example views for own management

7.5 Manage all requests and profiles (ADMIN)



(a) Manage requests view


WeatherTrack
All Requests
Add new request
Add a sensor for my hometown
Created at: April 17, 2024 at 4:09 PM
View Details
Add feature
Created at: April 17, 2024 at 4:16 PM
View Details
Add Rimini
Created at: April 17, 2024 at

(b) Manage profile view

Figure 18: Example views for all users management

8 Conclusions

In the WeatherTrack project, a distributed system was developed to collect and process real-time weather data from a network of simulated IoT sensors. The system provides users with a web-based interface to visualize and analyze the collected data. The core components of the system include weather sensors, a message broker, message consumers, a database, a web server, and a web client.

The system was designed to be scalable, fault-tolerant, and extensible, leveraging containerization to ensure reliable and efficient operation. The project employed tools like Docker, GitLab CI/CD, and automated testing to facilitate deployment, maintenance, and future feature integration.

The web application, built using a full-stack Javascript approach, offers users a dashboard to interact with the collected weather data, a role-based authentication, a request feature and an anomaly notification functionality.

Throughout the project, key distributed systems concepts such as designing scalable and fault-tolerant distributed systems, utilizing message brokers, asynchronous processing, real-time communication, containerization, and build automation were explored and implemented.

8.1 Future Works

The system developed in this project provides a platform for future developments and can therefore be useful for experimenting with various technologies applied to data processing, particularly in the domain of weather.

The modularity of a containerized component system allows experiments and upgrades to be performed on individual components without affecting the rest of the system. This means that changes can be made to one component without impacting the functionality of other components, making it easier to test and implement new features or improvements.

Some aspects that would be interesting to explore and implement are:

- Deployment of the system in a real world scenario with actual sensors instead of simulated ones (e.g. *Arduino Board* with modules for weather data gathering).
- Development and deployment of a machine learning model that uses the data collected from the sensors to generate weather forecasts for users. It would be interesting to explore the concept of continual learning, as new data is constantly being collected.
- Development and deployment of a machine learning model for anomaly detection, which would replace the simpler model currently in use.
- Implementation of additional security measures such as HTTPS and server-side validation.

- Automation of frontend testing, which is currently performed manually. Both unit tests (e.g. with *Vitest* or *Jest*, which is also used for backend unit testing) and end-to-end tests that holistically verify the whole the web app could be implemented.

Implementing these features would increase the reliability of the system, as well as provide interesting additional features to users.

8.2 What was learned

Through the development of the WeatherTrack system, several key concepts and skills have been learned and/or refined:

- The development process has provided practical experience in designing a scalable and fault-tolerant distributed system, including the use of message brokers, asynchronous processing, and real-time communication.
- The use of Docker and Docker-compose has helped understand the benefits of containerization for deploying and managing complex, multi-component applications.
- The project has involved the use of a GitLab CI/CD pipeline, automated testing strategies, and package management tools (such as pip and npm), which have provided insights into the benefits of utilizing build automation approaches.
- Implementing a modular system with clearly defined entities and responsibilities, has underlined the importance of creating maintainable and extensible software architectures.
- The need to document the system's architecture, design decisions, and implementation details through an initial and final report, as well as additional documentation (such as Swagger docs and UML diagrams) was valuable to improve writing and communication skills and understand the importance of documenting the code base.

These skills and concepts will prove valuable in the development of future projects or in the possible extension of this one based on the aspects identified in the *Future Works* section.

References

[1] Docker, Inc. Docker. `https://www.docker.com/`.

[2] Docker, Inc. Docker Compose. `https://docs.docker.com/compose/`.

[3] Evan You. Vue.js - a progressive JavaScript framework. `https://vuejs.org/`.

[4] MongoDB, Inc. MongoDB. `https://www.mongodb.com/`.

[5] npm, Inc. npm - the Node.js package manager. `https://www.npmjs.com/`.

[6] OpenWeatherMap. OpenWeatherMap - Weather API. `https://openweathermap.org/`.

[7] Python Software Foundation. PyPI - the Python Package Index. `https://pypi.org/`.

[8] RabbitMQ. RabbitMQ - a message broker. `https://www.rabbitmq.com/`.