WeatherTrack

Distributed Systems course project

Gian Luca Nediani

Scenario

- Several IoT sensors located around the world continuously collect weather data.
- The data is collected and stored in a database.
- A web application shows to authenticated users both real-time data and historical data, along with notifications on weather anomalies.
- Users can request new features.
- Administrators can manage users and user requests.

Why the weather domain?

- This scenario, with many interacting components (IoT sensors, message broker, database, web app, ...), lends itself well to the nature and topics of the course.
- The requirement to display the data and have users interact with it lends itself well to the ASW course (project for both courses).
- Weather, and more generally climate, are very current and relevant issues for our society/civilization.
- I actually want to deploy this system with real sensors.

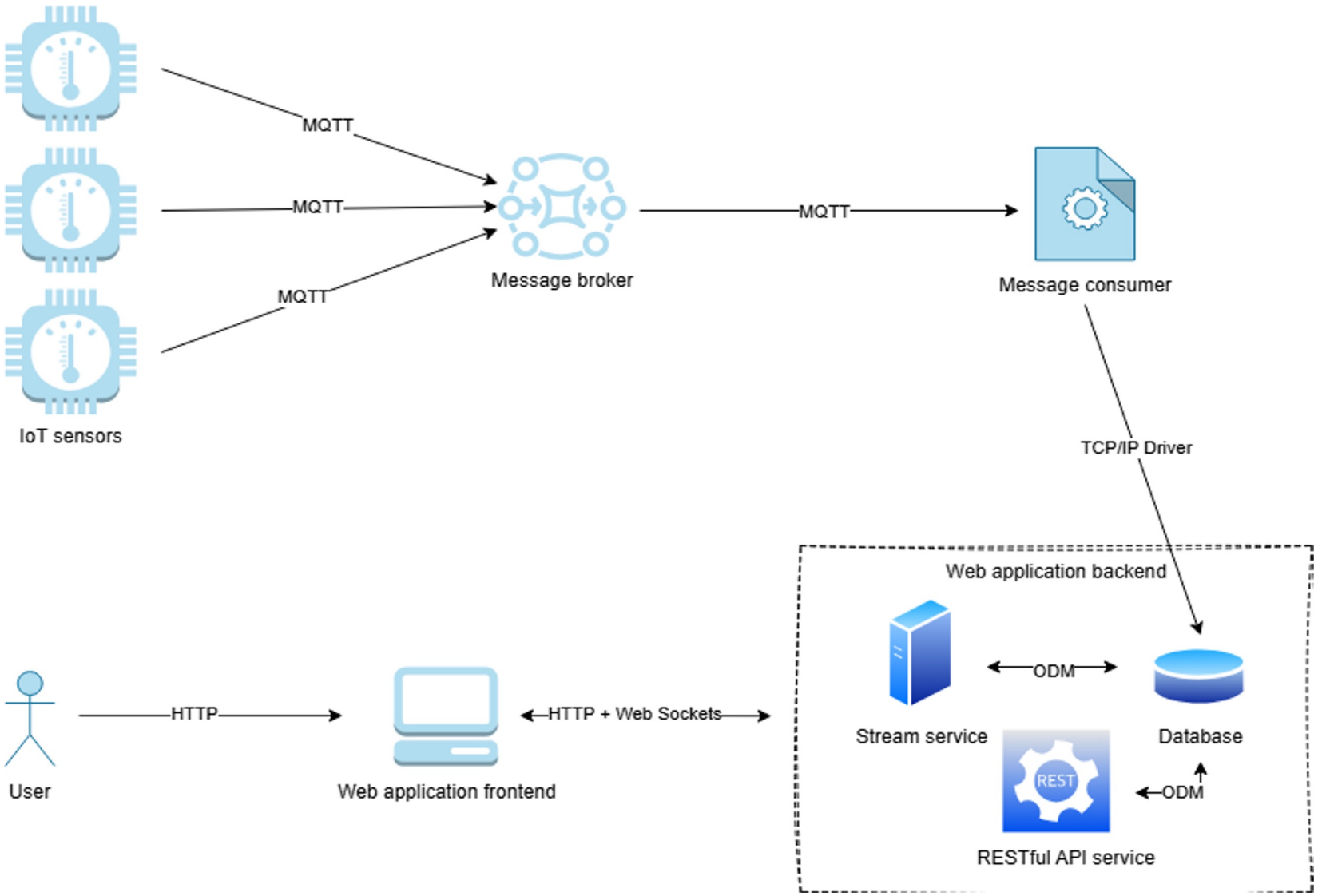
Functional requirements

- User authentication to control access to the web application.
- Real-time data collection, processing and displaying from simulated IoT sensors inthe 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.

Non-functional requirements

- The system should be scalable.
- The system should be fault-tolerant.
- 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.
- The system's interface for the end-users (i.e. the web application) should have high usability.
- The system should be modular to easily allow for the development of new features.

System design: high-level overview



System design: tech choices

- IoT sensors: simulated with Python scripts and API calls
- Message broker: RabbitMQ
- Message consumer: Python script
- Database: MongoDB
- Web backend: Node.js + Express.js (event-driven architecture)
- Web frontend: Vue.js (single-page application)
- Containerization: Docker
- Container orchestration: Docker-compose

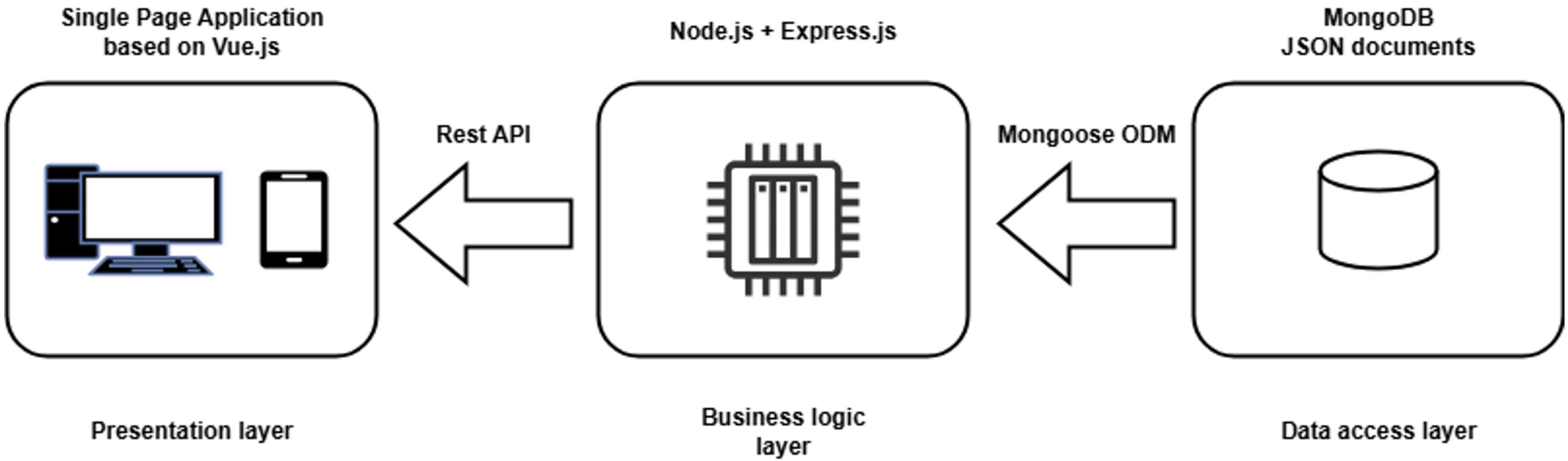
System design: addressing the non-functional requirements

Choice of technologies and paradigms that are well suited not only for the functional requirements, but also for the non-functional requirements of the system:

- **Containers:** modularity and fault-tolerance.
- **Docker-compose:** scalability and availability.
- **RabbitMQ:** scalability, fault-tolerance and low-latency.
- **MongoDB:** scalability, flexibility, and real-time data handling.
- **Redundant Critical Components:** fault-tolerance and availability.
- **Event-driven web server runtime:** low-latency.
- **Single-page minimal web client:** usability.

Web App Design + Tech Stack

MEVN Stack



Implementation details

- RESTful API with automatically generated Swagger documentation.
- JWT-based backend middleware for authentication and role management.
- JWT-based middleware for HTTP requests and token management with interceptor (DRY principle).
- Socket.io and MongoDB Change Streams for real-time weather updates.

Evaluation criteria

The following criteria were chosen:

- **Functional correctness**: the software should successfully perform all specified tasks and operations as outlined in the functional requirements.
- **Non-Functional Correctness**: the software should meet the implicit non-functional requirements; those are scalability, fault tolerance, low latency, high availability, and authentication.
- **Usability**: the software should provide an intuitive and user-friendly interface.

Evaluation process

Five-step process:

1. 1. Unit testing (automated with CI/CD pipeline).
2. 2. Integration testing (partially automated with CI/CD pipeline).
3. 3. User testing (informal approach with volunteers).
4. 4. Qualitative assessment of the functional requirements.
5. 5. Qualitative assessment of the non-functional requirements.

What was learned?

- Practical experience in designing and implementing a fault-tolerant and scalable distributed system.
- Better understanding of the benefits of containerization for managing and deploying multi-component applications.
- Insights into the benefits of build automation practices.
- Better understanding of the importance of modularity for creating maintainable and extensible software.
- Improved writing and communication skills through documentation (initial/final reports, Swagger docs, UML diagrams).

Future WeatherTrack developments

The project provides a foundation for future developments, enabling experimentation with different data processing technologies, especially in weather-related applications. Some ideas:

- Deployment of the system in a real world scenario with **actual sensors** instead of simulated ones.
- 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.** Both unit tests and end-to-end tests that holistically verify the whole web app could be implemented.

Thanks for your attention!