**Distributed Systems Project**

**Assignment 3**

**Chat WebSockets**

By Sandu Bogdan-Emilian

30422

1. **Introduction**

This project aimed to create a Monitoring and Communication Microservice for the Energy Management System, consisting of a ReactJS front-end and two hybrid NestJS microservices. The microservice utilizes a message broker middleware to gather data from smart metering devices, calculates hourly energy consumption, and stores results in its database.

The architecture includes a front-end styled with MaterialUI and two hybrid NestJS microservices acting as HTTP servers and microservices. The Monitoring and Communication Microservice plays a key role in aggregating smart meter data, using a message broker for communication.

Synchronization between Device Management Microservice and Monitoring and Communication Microservice databases is achieved through an event-based system. A Smart Metering Device Simulator, a standalone desktop app, generates simulated data from a sensor.csv file, sending JSON-formatted measurements to the message broker.

The Monitoring and Communication Microservice's Message Consumer processes these measurements, computes total hourly energy consumption, and stores results. If the computed value exceeds the device's predefined maximum, asynchronous notifications are sent to users' web interfaces, as defined in Assignment 1.

In summary, this project establishes a comprehensive architecture for the Energy Management System, focusing on the Monitoring and Communication Microservice for effective data processing, storage, and user notifications.

1. **Conceptual architecture of the distributed system**

Distributed System Conceptual Architecture for Monitoring and Communication Microservice (Extended):

The architecture of the Monitoring and Communication Microservice within the distributed system involves several components distributed across different servers:

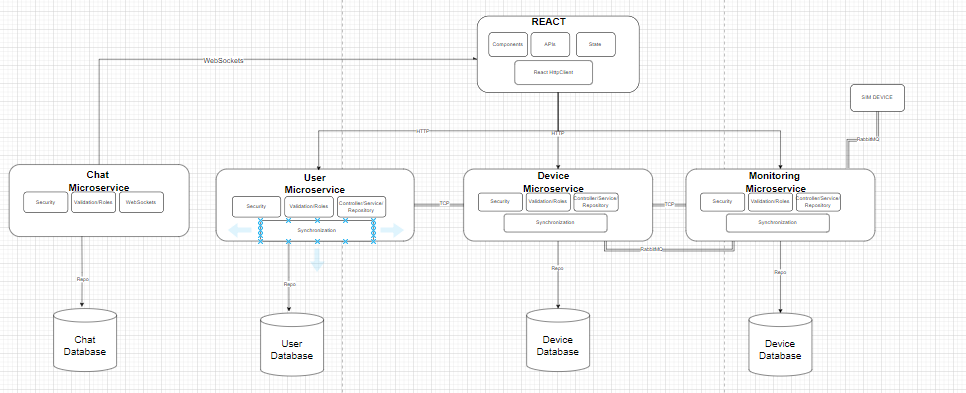
1. **React App Server:** Responsible for rendering pages and interfacing with the user.
2. **User Microservice Server:** Manages user-related functionalities such as authentication, CRUD operations, and security.
3. **Device Microservice Server:** Handles device-related operations, including CRUD operations and synchronization.
4. **User Database Server:** Dedicated to storing user-related data.
5. **Device Database Server:** Dedicated to storing device-related data.
6. **RabbitMQ Server:** Functions as a message broker middleware facilitating communication between components.
7. **Monitoring and Communication Microservice Server:** Orchestrates the core functionalities. Gathers data from smart metering devices via RabbitMQ, processes the data to compute hourly energy consumption, and stores the results in its dedicated database.

Requests from the React App trigger API calls to the User and Device Microservices. These calls follow a RESTful communication flow, with controllers on each microservice validating sessions using JWT and ensuring the necessary user roles for resource access.

Synchronization between the Device Management Microservice and the Monitoring and Communication Microservice databases is event-driven. A topic-based system emits events for device changes, sending device information through a RabbitMQ queue to the Monitoring and Communication Microservice.

The Monitoring and Communication Microservice includes a Message Consumer component, processing measurements to calculate total hourly energy consumption. If the computed value exceeds the device-defined maximum (as defined in Assignment 1), asynchronous notifications are sent to the user's web interface.

This architecture ensures efficient data flow, real-time processing, and seamless communication, creating a robust foundation for the Energy Management System's Monitoring and Communication Microservice



1. **Database design**

Energy Management System I used 3 databases, one for the user microservice and one for the device microservice and one for the monitoring microservice.

The user database contains the following entities: User, Role. We have the following 3 tables: user, role, user\_to\_role.

User Entity: id, email, password, firstName, lastName, phoneNumber, roles (ManyToMany).

Role Entity: id, name.

The device database contains the following entities: Device. We have the following table: device.

Device Entity: id, userId, description, maxHourlyCo nsumption, address.

The monitoring database coints the following enities: EnergyReading, HourlyConsumption.

EnegryReady: id, deviceId, timestamp, measurementValue.

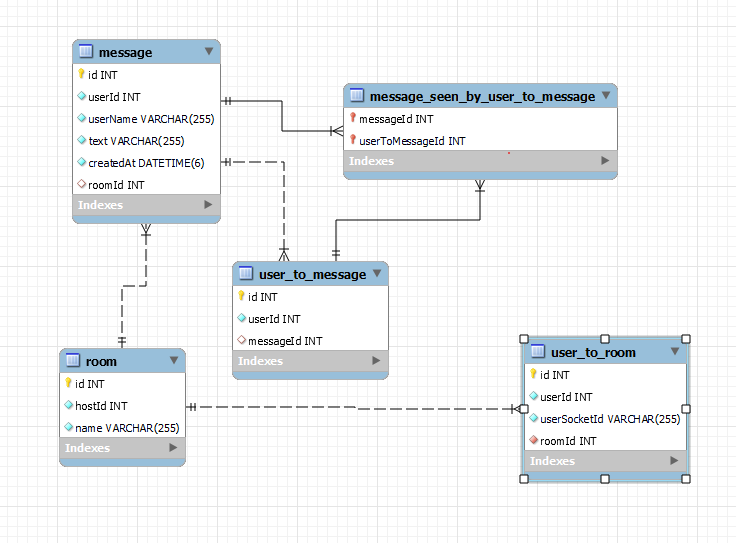
HourlyConsumption: id, deviceId, hourStart, totalConsumption.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

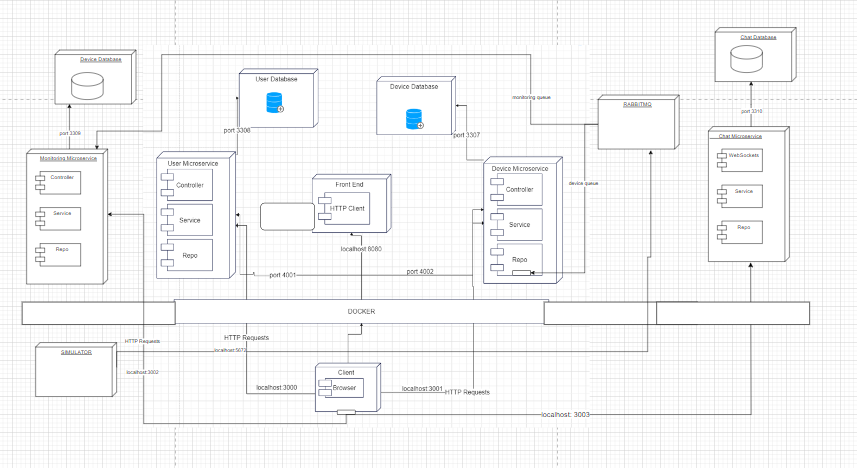
Description automatically generated

A screenshot of a computer program

Description automatically generated

1. **UML Deployment Diagram**

The main components are the React App, the 3 NestJS microservices,the 3 databases and the simulator, every one of these components are deployed independently. The communication is achieved by HTTP requests and TCP and RabbitMQ between the microservices. All the components run in their own docker container. The architecture is distributed because we could run every container on a different machine and not all on the same one, thus we can leverage the use of cloud platforms such as AWS for deploying our components.



1. **README considerations**

To run the deployed version of this application you need to configure your .env files with necessary fields (DB\_USERNAME, DB\_PASSWORD) for all microservices. The frontend doesn’t require any .env file to run although a change could be made such that we set up .env files for each application and branch in order to run the version of the application we want(the dockerized one or the classic one).

Additionally you need to add all 3 projects in the same directory and at the root add the docker-compose.yml file present in the front-end repository.

To setup multiple simulator create a “.env” file and add the port and device id to it.