DISTRIBUTED SYSTEMS

Assignment 2

Asynchronous Communication

and Real-Time Notification

STUDENT NAME: Marin Andreea

GROUP: 30442/1

**CONTENTS**

1. Objectives
2. Requirements
3. Conceptual architecture of the distributed system
4. UML Deployment Diagram
5. Build and execution considerations
6. **Objectives**

Implement a Monitoring and Communication Microservice for the Energy Management System. The microservice is based on a message broker middleware that gathers data from the smart metering devices, processes the data to compute the hourly energy consumption and stores it in the database of the Monitoring and Communication Microservice.

The synchronization between the databases of Device Management Microservice and the new Monitoring and Communication Microservice is made through an event-based system that uses a topic for device changes (sends device information through a queue for the Monitoring and Communication Microservice).

A Smart Metering Device Simulator application will be implemented as the Message Producer. It will simulate a smart meter by reading energy data from a sensor.csv file (i.e., one value at every 10 minutes) and sends data in the form < timestamp, device\_id, measurement\_value > to the Message Broker (i.e., a queue). The timestamp is taken from the local clock, and the device\_id is unique to each instance of the Smart Metering Device Simulator and corresponds to the device\_id of a user from the database (as defined in Assignment 1). The device simulator should be developed as a standalone application (i.e., desktop application).

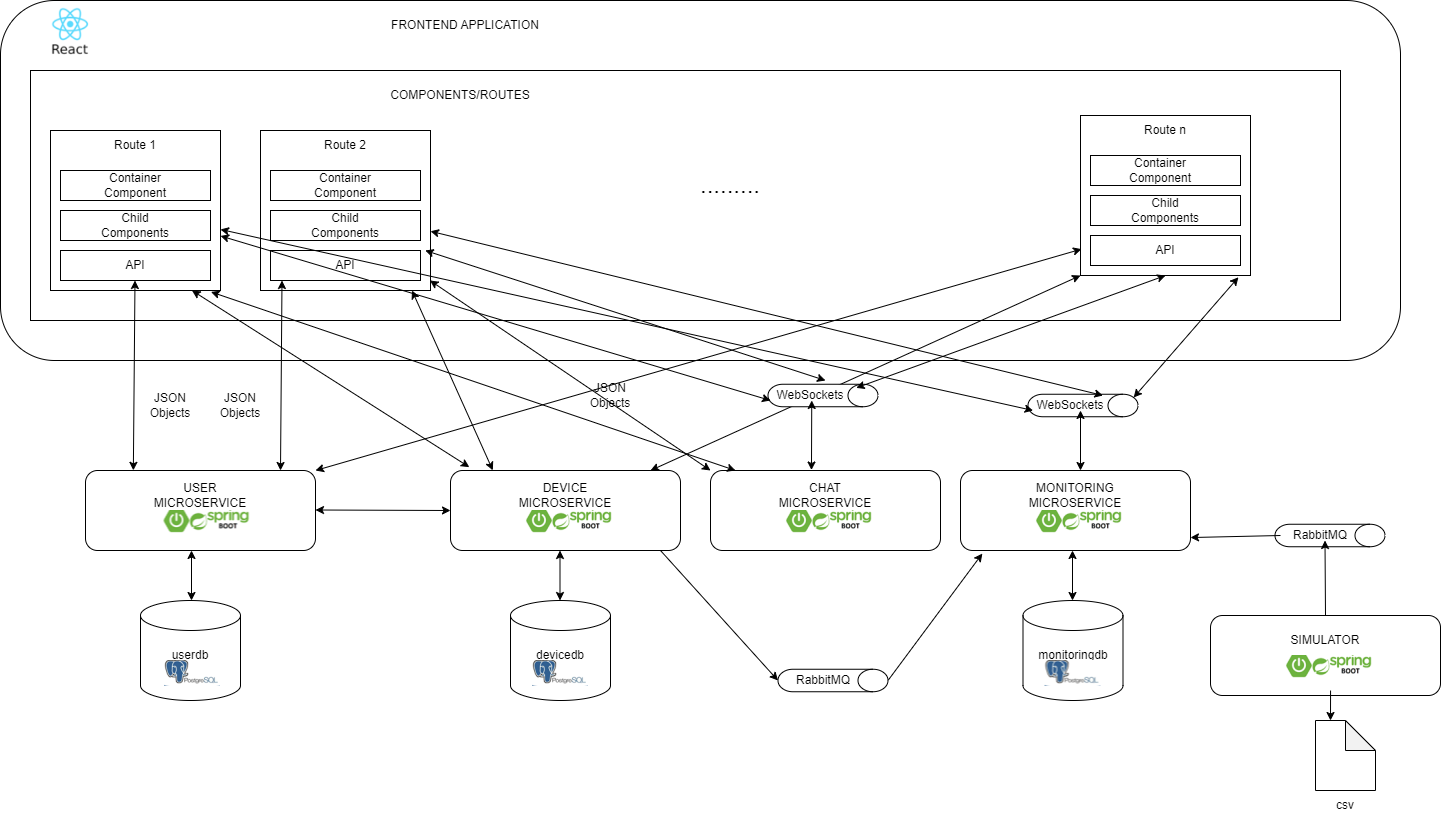
The Monitoring and Communication Microservice will have a Message Consumer component that will process the measurements to compute the total hourly energy consumption and store it in the database. If the computed total hourly energy consumption exceeds the device defined maximum value (as defined in Assignment 1) it notifies asynchronously the user on his/her web interface.

1. **Requirements**

Functional, non-functional requirements and implementation techniques were identified for this project.

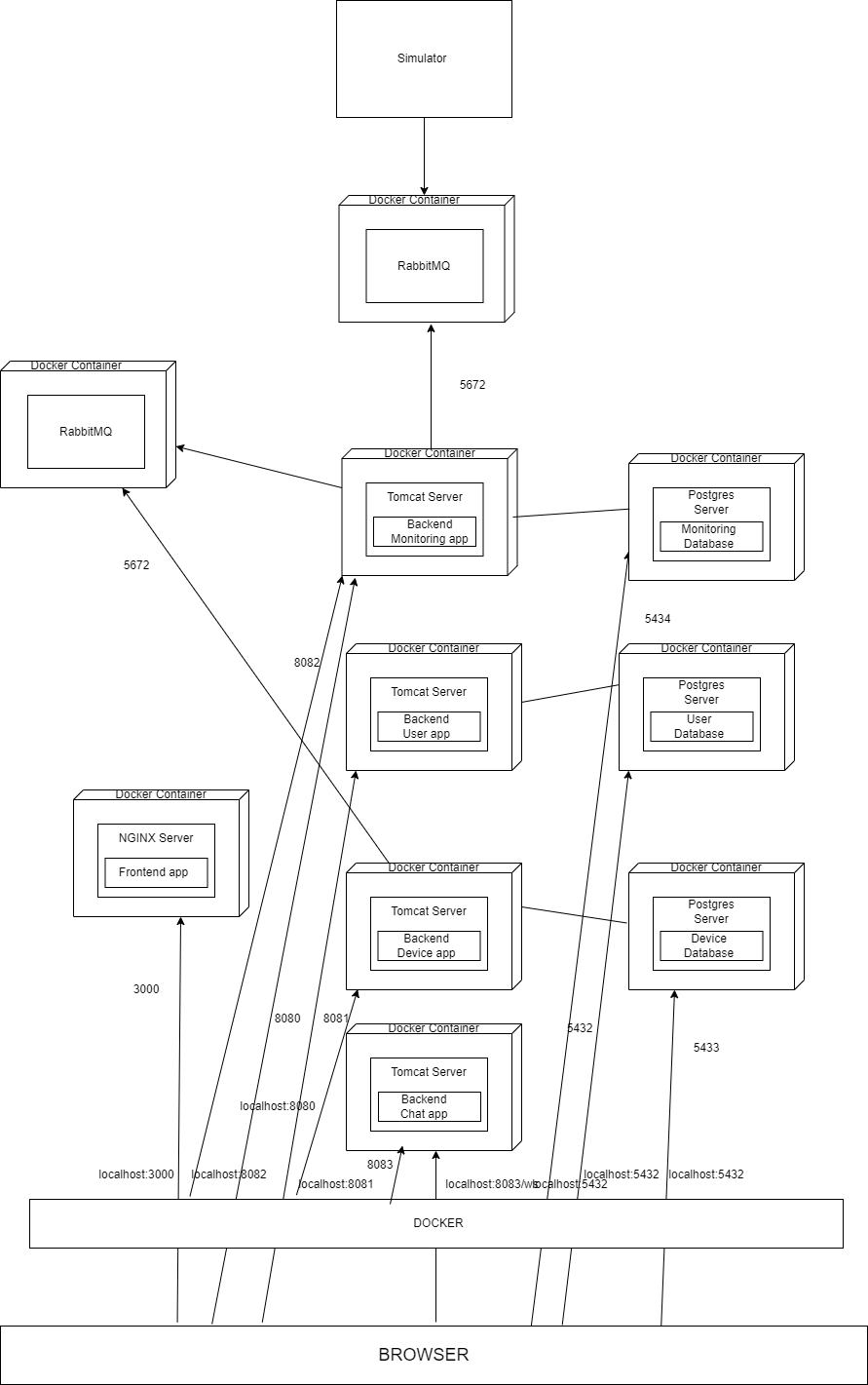
* **Functional requirements**
* The message-oriented middleware allows the Smart Metering Device Simulator to send data tuples in a JSON format.
* The message consumer component of the microservice processes each message and
* notifies asynchronously the client application using WebSocket
* The hourly energy values will be saved by the consumer component in the Monitoring database
* **Non-functional requirements**
* The application should be divided into four microservices: User Management Microservice, Device Management Microservice, Monitoring Microservice and Chat Microservice
* The application should use authentication to restrict users to access the administrator pages
* The Consumer component will be integrated into the Monitoring and Communication
* Microservice
* **Implementation techniques**
* Use the following technologies: RabbitMQ, WebSockets.

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



* **Simulator**
* The simulator reads data from the csv file for the given device id and produces the message that is sent to the RabbitMQ queue to be consumed by the Monitoring Microservice
* Every 10 seconds a new value is read from the csv
* **Frontend application**
* The frontend application was modified in order to be able to display the notifications sent through the WebSocket from the Monitoring Microservice
* Another modification was added in order to allow the user see the historical energy consumption of his device as a line chart
* **Monitoring Microservice**
* This microservice is the one managing the monitoring for devices. It is divided in the following packages:
* Model package: This package contains the entity classes for Device and Measurements. The entity Measurements contains the id of the monitored device, the timestamp and the measurement\_value that is read from the csv
* Repository package: contains interfaces DeviceRepository and Measurements Repository
* Dtos package: contains dto classes for the entity classes
* Controller package: this package contains classes that receive and handle incoming requests from the user interface or external systems
* Config package: contains class RabbitMQConfig that configures the connection to RabbitMq
* Consumer package: contains class RabbitMQConsumer that has methods that handle the consumption of messages from the queue
* Security package: contains classes that handle the security part of the project
* **Device Microservice:**
* The Device Microservice was modified such that to configure the producer part for the RabbitMQ in order to communicate with the Monitoring Device when an update occurs on devices
* **Database:** A new database was introduced for the Monitoring Microservice. Contains two tables: device and measurements, corresponding to the entities from the Model package

1. **UML Deployment Diagram**

****

On the host computer runs the docker runtime that will host ten containers, one for each application:

• Docker container for frontend application – runs a NGINX server and maps local port 3000 to host computer port 3000

• Docker container for backend application – runs four TOMCAT servers and maps local port 8080 to host computer port 8080, local post 8081 to host computer port 8081, local post 8082 to host computer port 8082, local post 8083 to host computer port 8083

• Database container for database server – runs three Postgres servers and maps local port 5432 to host computer port 5432, host computer port 5433 and host computer port 5434. A new container is introduced for RabbitMQ.

This means that from the host computer we can access the servers within the containers as follows:

• Frontend application: localhost: 3000

• Backend application: localhost: 8080, 8081, 8082, 8083

• Postgres server: localhost: 5432; RabbitMQ: 5672

1. **Build and execution considerations**

* When running the application locally, the following steps need to be followed:
* Open microservice applications in IntelliJ
* Start running the applications simultaneously
* Open Frontend application, open a terminal and run the command: “npm start”
* After running the command “npm start” in frontend, a web page will appear. Note that, at the beginning, the userdb contains a table “userr” with a single record (user with admin role) and the devicedb contains two tables: device and user, which are empty.
* The first step is to log in as admin, add new users with client role, add new devices and map devices with the new users added and then the remain functionalities can also be tested
* The simulator is started and the device\_id is introduced from the keyboard. The messages are placed in the queue and can be read by the Monitoring Microservice. While the messages are consumed the user gets a notification if the hourly measurements are greater than the value set by the user. After all messages are consumed, the user can see his historical energy consumption.
* The credentials for login are: username: “andre”, password: “andre”
* For testing the functionality assigned to users with client role, the logout is necessary and then login with the credentials of a newly introduced client user
* When running the application in docker, the following steps need to be followed:
* Enter the docker desktop application
* Start the three containers
* Open the application in browser by introducing the URL : “http://localhost:3000”
* Note that, at the beginning, the userdb contains a table “userr” with a single record (user with admin role) and the devicedb contains two tables: device and user, which are empty.
* The first step is to log in as admin, add new users with client role, add new devices and map devices with the new users added and then the remain functionalities can also be tested
* The simulator is started and the device\_id is introduced from the keyboard. The messages are placed in the queue and can be read by the Monitoring Microservice. While the messages are consumed the user gets a notification if the hourly measurements are greater than the value set by the user. After all messages are consumed, the user can see his historical energy consumption.
* The credentials for login are: username: “andre”, password: “andre
* For testing the functionality assigned to users with client role, the logout is necessary and then login with the credentials of a newly introduced client user