SOLUTION DESCRIPTION

Energy Management System

Assignment 3

Distributed Systems 2023

Student: **Radu-Augustin Vele**

Group: **30442**

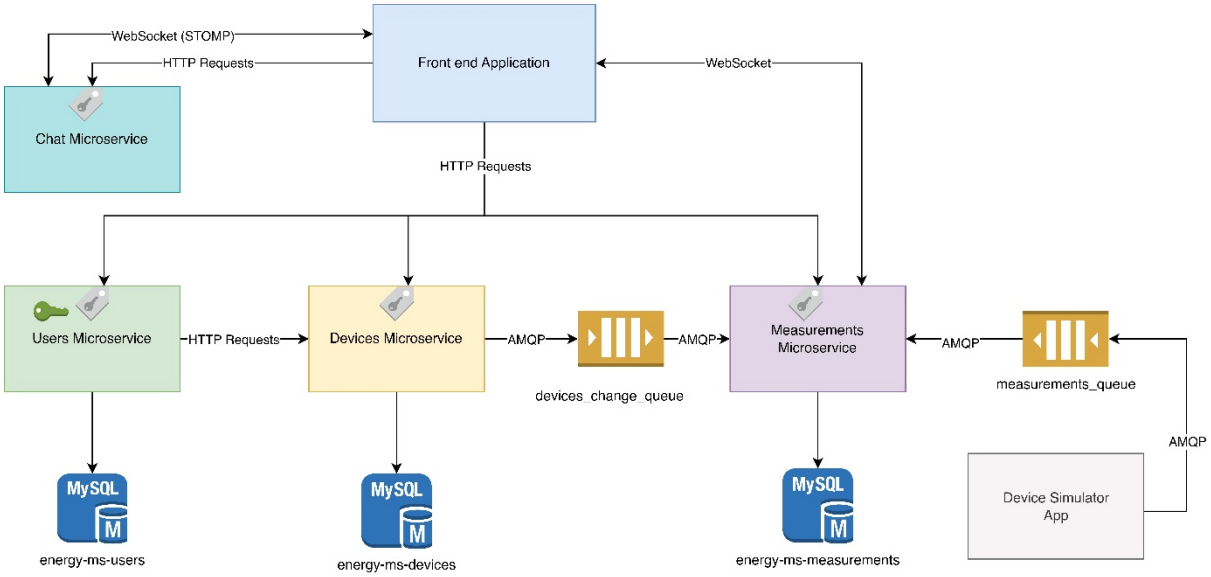
Table of Contents

1. Project objectives

The starting point for this stage of the development was a system enabling users to visualize real time data of energy consumed by their devices, and admins to manage the users and devices. But there is a missing link. How would the user reach out to an admin when something unexpected happens (without them picking up the phone and calling them)? To solve this problem, we introduce real time chat between admins and users.

Another problem solved in this stage is the security of the system. Before, anyone who knew the endpoint of the devices microservice could introduce new devices, delete devices, and so. Which brought a great vulnerability in the system, making it unreliable. To solve this, every REST endpoint of the application is secured. Note that, in this iteration the chat messages are not persisted, and both admin and users need to be logged in to be able to chat.

2. Conceptual architecture



Besides the components that were previously in the systems the Chat Microservice was added. Also, the diagram shows that each microservice protects its REST endpoints (from the implementation point of view we are using **J**son **W**eb **T**okens).

Next, we elaborate on the additions and changes in the system.

2.1 Components description

Chat microservice

The chat microservice has the purpose of setting up the web sockets endpoint and configuring the message broker used. It is organized as a subset of the layered architecture (with only the Controller, and Service layers as chat persistence is not yet included).

From the implementation point of view, the web sockets are included in the app using the STOMP protocol. The benefit of this is a higher-level usage of web sockets (with more functionality such as having the list of connected session already managed, using a default or custom message broker that already has the logic for broadcasting / subscribing to channels, and organizing the logic for responding to messages using controllers).

In our case, we create a stomp endpoint used by the clients for connecting to the WebSocket. Also, we add the application destination prefix (used when sending all the messages), a message broker with multiple destination prefixes (such as seen, startTyping, etc.) that help us manage subscriptions. Also, a user destination prefix is set up to enable one-to-one communication. For example, the destination corresponding to the user `john@doe.com` is `/queue/john@doe.com`, therefore when someone wants

to talk to John, they send a message to its queue. John is subscribed to its own queue to receive its message.

For the seen notification, every time a user sends a message, they subscribe to the `seen/message-timestamp` channel until they receive a message on that channel.

For the typing notification, a user is subscribed to the `startTyping/partner-email-address` and `stopTyping/partner-email-address` channels and any message is interpreted as a command with the self-explanatory meaning.

Moreover, the chat microservice exposes a REST endpoint where the authenticated user can request the email address of an admin that is logged in the system.

In a future iteration it will be enhanced with the responsibility of persisting data into a database and providing means for retrieving past conversations and messages. In the controller, whenever messages are received, they are routed to the corresponding user or destination.

Frontend updates

The chat box functionality (except the part responsible for the HTTP requests) is added into a react functional component (`ChatBox.ts`). It is the same for admins and users and contains some conditional structures to make the difference between the desired behavior for the two roles.

The component has the purpose of keeping an in-memory chat history that lasts as long as the current user session is on (and the page is not refreshed). The messages are kept in two lists, for inbound and outbound messages, and contain information about whether is has been read, or whether the direction is in or out (to be able to interpolate the messages from the two lists with respect to the timestamp, for display). It also contains a stomp client (using the StompJs library). Another data structure keeps track of the paired users (for admins, or a single admin for the user). This data structure contains a `isTyping` field to help with the processing of start and stop typing notifications.

The flow of the chat communication work as follows:

1. User Flow
    a. The user opens the chat box.
    b. A WebSocket connection is opened between the browser and the chat microservice.
    c. The user subscribes to its own queue channel.
    d. A http request is issued and the chat microservice provides the email address of a logged in admin (if available).
    e. The user can send messages to the retrieved admin.
    f. Once a message is sent, the user subscribes to the channel corresponding to the seen notification about that message (when the admin clicks the message, the user will receive a message and unsubscribe from the channel)
    g. When the user starts typing a message, or when it clears the text field for messages it sends a message via the `startTyping` or `stopTyping` channel.
    h. When the user receives a startTyping or stopTyping notification it updates the state of the paired admin (and a text appears/disappears from the chat box)
2. Admin Flow
    a. The admin logs in and enters their home page.
    b. The WebSocket is open, and the admin subscribes to its own queue.
    c. When a message is received, if the sender is a new one, they are added to the paired users list.
    d. The rest of the communication happens as described in the user flow.

Limitations

- This is a temporary solution until the chat is persistent into a database. Once it is persistent, this component (or another one) will be added the responsibility of loading the chat history and

previous conversations. Also, the user needs to be able to send a message even if no admin is logged in (and the message will be answered to by an admin that picks it up).
- Once any user engaged in the chat disconnects, the other user is not notified.
- The functioning of the system is based on the continuity of the internet connection of both users and pages mustn't be refreshed.

Authentication & Authorization

In the previous iteration, the authentication component was only included in the Users Microservice. It has the responsibility of issuing JWTs and validating them against the user database (based on the username and password existing in the database).

For this implementation to work, all microservices need to share the same secret key. This key is used for validating the JWT (alongside an encryption algorithm chosen when creating the tokens). Every time an attempt to extract claims from the token is performed, this secret key is used to check if the body of the JWT has somehow been changed.

The workflow for authentication is the following:

1. A http request is sent from the client including the authorization header with the JWT.
2. The JWT authentication filter's `doFilter` method is called (as part of the Security Filter Chain configured)
3. Inside the filter there are a few operations.
    a. Check if the request contains the authorization header.
    b. Check get username claim from token (this one implies and throwing an exception if the JWT signature is not valid)
    c. Populate the security context holder with an object containing the username and authorities of the authenticated user (this info is extracted from the database for the Users Microservices, or from the JWT in the other microservices, also we need to include the same user roles in all microservices – an Enum is used to achieve this).

Also, different HTTP endpoints are configured to require ADMIN or USER authority.

Another aspect that needs to be treated in the context of spring security is including the CORS policy. Therefore, all microservices need to establish a list of allowed origins (IP : port combinations) for the HTTP requests received. In our case we need to include the IP address that is configured for accessing the front-end application from the browser.

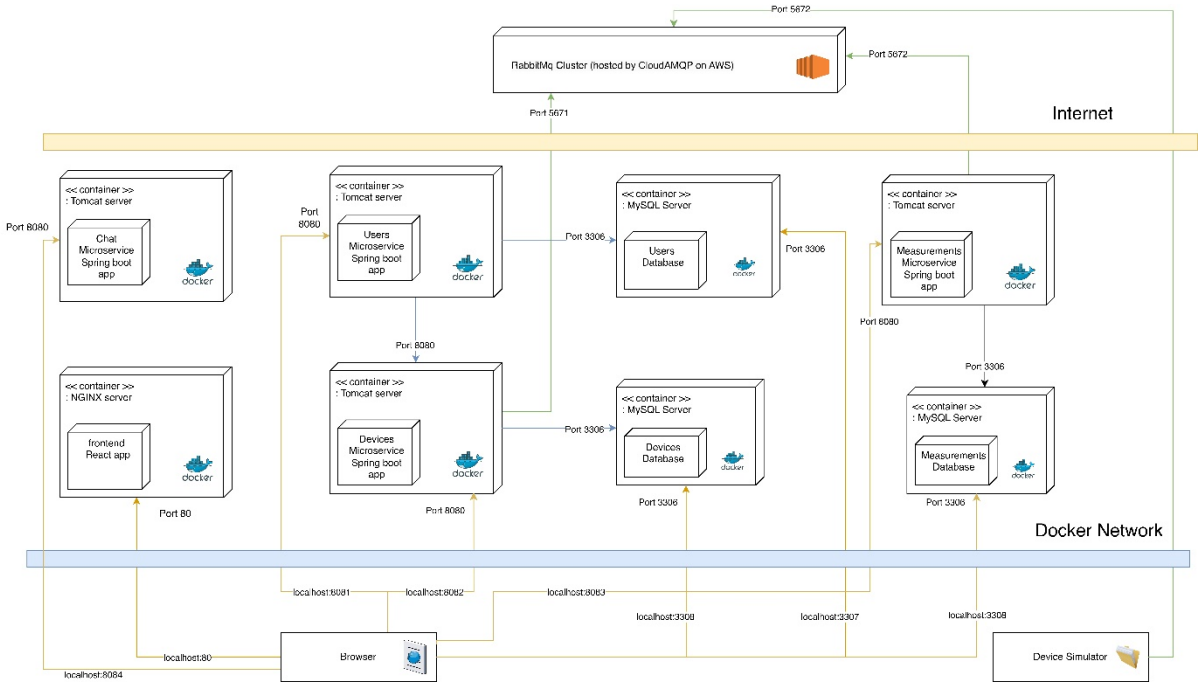
Limitation:

- If a user role or password are changed, the previous JWTs may still be used in other services but the user one (as that one is the only one connected to the user database)

3. UML Deployment Diagram

The system is deployed using Docker containers, which facilitates their inclusion in a network of interconnected Docker containers. At this stage, the Docker containers are executed locally and are made accessible through specific local host ports.

The newly added chat spring application is deployed at the address 172.20.0.9 and is accessible from localhost through the port 8084. The only environment variable it needs is the port it runs on in its container.



4. Readme – build and execution considerations.

Spring + React Setup

- For instructions related to running the microservices locally, and dockerization check out the assignment 1 notes. The prerequisites remain the same.
- Make sure to fetch the code from the correct branch on GitLab.

RabbitMQ Setup

- Check out the Assignment 2 documentation.

Flutter App setup

- Check out the Assignment 2 documentation.

5. Useful Links

Spring web sockets (stomp)

- https://docs.spring.io/spring-framework/reference/web/websocket/stomp.html

React web sockets (stomp)

- https://stomp-js.github.io/guide/stompjs/using-stompjs-v5.html

- https://jwt.io/