# Energy Management System Solution Description

# Vijoli Ioana

# Conceptual architecture of the distributed system.

The system is architectured as a collection of loosely coupled microservices, each catering to specific functionalities:

1. User Microservice:
   * Responsible for user management, authentication, and authorization.
   * Incorporates JWT (JSON Web Token) for secure user authentication across all microservices.
2. DeviceManagement Microservice:
   * Handles device-related operations such as device registration, monitoring, and data retrieval.
3. Sensors Microservice:
   * Manages notifications, alerting users based on predefined criteria or events.
4. Chat Microservice (New Addition):
   * Handles real-time messaging and communication among users.
   * Utilizes WebSocket protocol for establishing bidirectional, low-latency communication channels.

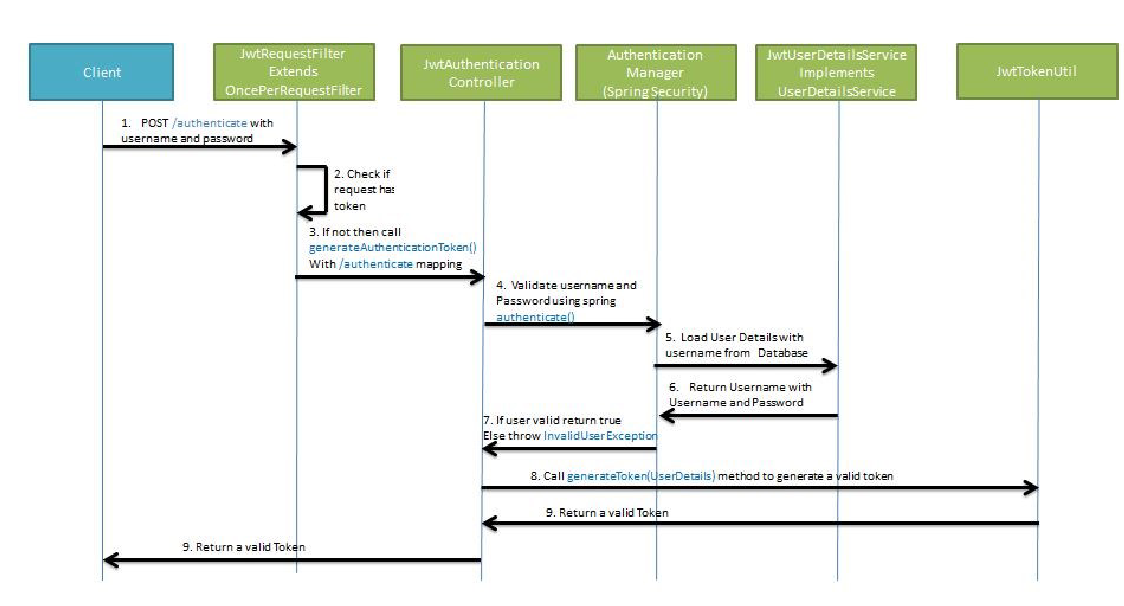
#### Communication:

Inter-service communication is facilitated through RESTful APIs and message brokers like RabbitMQ, ensuring seamless data exchange between microservices.

### JWT Token Authentication:

JWT tokens are used to ensure secure authentication across microservices:

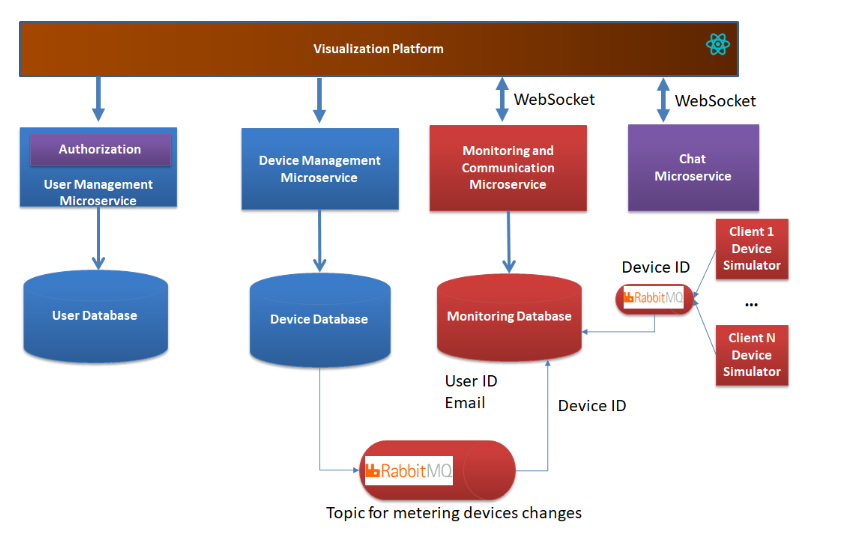
1. User Authentication:
   * Upon user login, the User Microservice generates a JWT token containing user information and a signature based on a shared secret key.
2. Token Transmission:
   * This token is then transmitted to other microservices by the client (e.g., React frontend) with subsequent requests.
3. Token Verification:
   * Microservices receiving the token can verify its authenticity using the shared secret key, ensuring the user's identity and access rights.
4. Enhanced Security:
   * JWT tokens provide a secure mechanism for authentication and authorization without the need for sessions, reducing server-side storage and enhancing scalability.



### WebSocket Usage in Chat:

The recently implemented Chat Microservice utilizes WebSocket technology for real-time communication:

1. Real-time Messaging:
   * WebSocket enables bidirectional, low-latency communication between clients (e.g., React frontend) and the server (Chat Microservice).
2. Establishing Persistent Connection:
   * Upon user interaction (e.g., sending a message), WebSocket facilitates a persistent, full-duplex communication channel, eliminating the need for frequent HTTP polling.
3. Efficient, Real-time Updates:
   * Chat messages are instantly transmitted and received, ensuring a real-time chat experience for users interacting through the React frontend.
4. Scalability and Performance:
   * WebSocket's lightweight nature and efficient communication mechanisms contribute to system scalability and performance.

By integrating JWT tokens for secure authentication and leveraging WebSocket technology for real-time communication within the chat functionality, the Smart Metering Devices Application ensures robust security and seamless, responsive user interactions across its microservices-based architecture.

# UML Deployment diagram

# On the host computer runs the docker runtime that will host three containers, one for each

# application:

# • Docker container for frontend application – runs a server and maps

# local port 3000 to host computer port 3000

# • Docker container for users microservice (backend application) – runs a TOMCAT server and maps local port 8080 to host computer port 8080

# • Docker container for devices microservice (backend application) – runs a TOMCAT server and maps local port 8081 to host computer port 8081

# • Docker container for sensors microservice (backend application) – runs a TOMCAT server and maps local port 8082 to host computer port 8082

# • Docker container for chat microservice (backend application) – runs a TOMCAT server and maps local port 8083 to host computer port 8083

# • Database container for the first database server – runs a PostgreSQL server and maps local

# port 5434 to host computer port 5432

# • Database container for the second database server – runs a PostgreSQL server and maps local

# port 5433 to host computer port 5432

A diagram of a user interface

Description automatically generated

# Readme

The Smart Metering Devices Application is a microservices-based project designed to manage users and devices and notifications. It leverages a modern technology stack, including Spring Boot for microservices and React for the frontend.

Before launching the Monitoring Spring Boot Application, it's crucial to ensure the seamless operation and setup of interconnected components that form the backbone of the system. The React and Spring Boot applications play pivotal roles in this ecosystem and warrant thorough validation.

Firstly, it's imperative to confirm the proper configuration and functionality of the React application, which serves as the frontend interface. This involves validating the installation of all necessary dependencies and verifying that the React app effectively establishes robust WebSocket connections with the Monitoring Spring Boot Application.

Additionally, the correct setup of the DeviceManagement and Smart Metering Device Simulator Spring Boot applications is essential. These applications act as vital sources of critical data, making their operational readiness a prerequisite for the smooth functioning of the Monitoring Spring Boot Application.

Ensuring the correct configuration and effective utilization of RabbitMQ holds paramount importance for the efficient operation of the message-oriented middleware. It's vital to verify that the RabbitMQ image is appropriately configured, and the requisite queues are established. These queues facilitate seamless communication among the Smart Metering Device Simulator, DeviceManagement Application, and the Monitoring Spring Boot Application.

Before initializing the Monitoring Spring Boot Application, it's advised to execute a provided Python script dedicated to data preparation. This script populates the data.json file with information that complies with the specific format mandated by the Monitoring Spring Boot Application. The accurate execution of this script is crucial as it ensures data integrity, enabling precise computation of energy consumption within the application. This preparatory step significantly contributes to maintaining accurate data analysis and functionality within the system.

## Technologies Used

**- \*\*Microservices Architecture:\*\***

- User Microservice: Developed using Spring Boot and Java, manages user-related operations, and authentification with JWT TOKEN.

- Device Microservice: Also built with Spring Boot and Java, oversees device-related functionalities.

-Sensors Microservice: Uses WebSocket and RabbitMq

-Chat Microservice: Uses WebSocket

**- \*\*Frontend:\*\***

- Developed with React for an interactive and engaging user interface.

**- \*\*Communication:\*\***

- Microservices communicate synchronously via RESTful APIs.

The project follows a microservices architecture, facilitating independent deployment and scalability.

**- \*\*User Microservice:\*\***

- Endpoints:

- `/users` (GET, POST): Retrieve and create user information.

- `/users/{id}` (GET, PUT, DELETE): Manage individual user details.

Also endpoints for LOGIN and LOGOUT

**- \*\*Device Microservice:\*\***

- Endpoints:

- `/devices` (GET, POST): Retrieve and create device information.

- `/devices/{id}` (GET, PUT, DELETE): Manage individual device details.

**- \*\*Chat Microservice:\*\***

- Websocket for managing group chat and individual chats between users and admins.

**- \*\*Frontend:\*\***

- Built with React for a dynamic and responsive user experience.