**Energy Management System**

**Solution Description Document**

**Vlad Ursache**

1. **Introduction**

Efficient energy management has become increasingly crucial in recent years due to various factors. The growing awareness of environmental issues and the need to reduce carbon emissions have put a spotlight on sustainable energy practices. Additionally, rising energy costs and the finite nature of traditional energy sources have driven the push for better energy management.

An Energy Management System (EMS) plays a vital role in addressing these challenges. It optimizes energy consumption by monitoring, analyzing, and controlling energy usage in various settings, such as residential, commercial, or industrial.

1. **System Requirements:**

Users log in and are redirected to the corresponding role-based pages.

Administrator Role:

Perform CRUD operations on users.

Perform CRUD operations on devices.

Create mappings between users and devices.

User/Client Role:

Clients can view all their devices on their respective pages.

Role-based access control to restrict users from accessing pages intended for other roles.

The relationship between users and smart devices is many-to-many, meaning that multiple users can own multiple smart devices, and each device can belong to multiple users. This flexibility is crucial in scenarios where users may have devices in different locations and where devices may serve multiple users.

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 simulate a smart meter by reading energy data from a sensor.csv file (i.e., one value at every 20 seconds) and sends data. It is incorporated into the microservice.

**Data Model and Implementation:**

* Data Model: The data model for user-device mapping typically includes tables for "Users" and "Devices." Additionally, there is a third table, often referred to as a "User-Device Mapping" table. This mapping table contains records that associate users with their owned devices and vice versa.
* User-Device Mapping: The User-Device Mapping table includes foreign keys that link it to both the Users and Devices tables. These foreign keys establish relationships, allowing administrators and clients to associate devices with users. Each record in this mapping table represents a unique association between a user and a device.
* The measurements are sent to the queue using the following JSON format, from the sensor simulator to the Monitoring Microservice:

{

“timestamp": 1570654800000,

“device\_id”: “5c2494a3-1140-4c7a-991a-a1a2561c6bc2”

“measurement\_value”: 0.1,

}

1. **System Architecture**

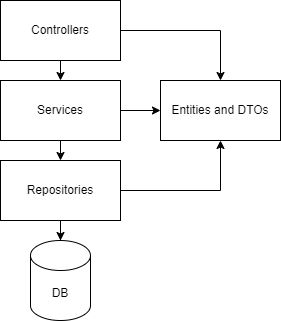
Both the Users and Devices microservices are structured around a layered architecture.

* Controller Layer: This is the entry point for external requests. Controllers handle incoming HTTP requests, route them to the appropriate service, and return responses. For the user and device management microservices, there are separate controllers.
* Service Layer: Here, the core business logic resides. It includes functionalities like user authentication, device management, and any other specific operations related to these microservices. These functionalities include inter-microservice communication (for authentication and CRUD delete propagation).
* Repository Layer: Repositories interact with the database or data storage. They encapsulate data access and management, ensuring separation from the business logic. Each microservice has dedicated repositories.
* DTOs and Models: Just like in the frontend, DTOs are used for data transfer between microservices and the frontend. Models represent the data structures used within the microservices.
* Security Filters and Configurations: Security is crucial. The system includes security filters and configurations to ensure that microservices are protected from unauthorized access and data breaches. The security is furtherly elevated by not storing any secrets in the source code.

The Monitoring microservice works in a similar fashion. Additionally, it reads data from the .csv file and sends it via WebSocket.

Regarding the frontend, it presents itself with a simpler structure that revolves around:

* Reusable Data Components: The frontend primarily consists of reusable components designed to request and display data. These components interact with the services to fetch data and display the responses.



1. **Microservices communication**

The communication and interaction between the two microservices, Users Microservice and Devices Microservice, to facilitate user-device mapping and ensure proper security are crucial aspects of the system. Here's how this communication works:

Facilitating User-Device Mapping:

* Client/User Interaction: When a user, whether an administrator or a client, interacts with the system to associate a device with a user or perform any related actions, they make requests to the Devices Microservice.
* Device Microservice Authentication: To ensure security, the Devices Microservice secures its endpoints by validating session tokens or authentication tokens. These tokens are usually issued to the client/user upon successful login and are included in subsequent requests to prove the user's identity.
* Request Validation: Before any device-related operation is executed within the Devices Microservice, it validates the session token to confirm the user's authorization. If the token is valid, the requested action, such as associating a device with a user, is carried out.
* User-Device Mapping: The Devices Microservice processes the request to establish or update the association between a device and a user. It also ensures that the operation adheres to the many-to-many relationship model between users and devices.
* Response: Upon successful execution of the request, the Devices Microservice responds to the user's request with a confirmation or relevant data, such as the updated user-device mapping.

Cascade Delete Query:

* User Deletion Request: When a user is deleted, whether by an administrator or due to other reasons, a request to delete the user is made to the Users Microservice.
* Cascade Delete Request: The Users Microservice recognizes the need for cascade deletion to maintain data integrity. It sends a request to the Devices Microservice, specifically requesting the deletion of any user-device mappings associated with the user being deleted.
* Validation and Execution: The Devices Microservice validates the request, ensuring it comes from a trusted source (Users Microservice). If the request is valid, the Devices Microservice executes the cascade delete query to remove the relevant user-device mappings.
* Response: After successfully executing the cascade delete, the Devices Microservice responds to the Users Microservice, confirming the removal of user-device mappings.
* This communication flow between the two microservices is essential for maintaining data consistency, securing device-related operations, and ensuring that users have the appropriate access to devices while preserving the many-to-many relationship between users and devices. It highlights the collaborative nature of microservices in achieving the system's overall functionality and security.

1. **Implemented Security Measures**

JWT Tokens for Authentication: The system employs JSON Web Tokens (JWT) for secure user authentication. When a user logs in, they receive a JWT token that contains information about their identity and role. This token is signed with a secret key on the server, ensuring its authenticity. Subsequent requests include this token, allowing the system to verify the user's identity and role without the need to store session data on the server.

A diagram of a computer

Description automatically generated

Role-Based Access Control (RBAC): Role-based access control is enforced to ensure that only authorized users can access specific functionalities. The system recognizes two primary roles, administrators/managers and clients/users. Each role has a set of permissions, and the system checks the user's role in the JWT token before allowing access to particular endpoints or functionalities. This way, administrators can perform CRUD operations on users and devices, while clients can view their devices but not modify other user-related data.

1. **Chat Functionality**

In our WebSocket-based chat system, we have defined five distinct message types to facilitate communication and provide various functionalities. This documentation outlines these message types and their purposes.

**1. Assigned Message**

**Purpose:** The "assigned" message is used to assign two contacts to engage in a conversation.

**Usage:** When a chat session is initiated between two users, an "assigned" message is sent to inform both parties about their assigned conversation partner.

**2. Content Message**

**Purpose:** The "content" message is used to transmit the content of text messages between users.

**Usage:** "Content" messages carry the actual text content of user messages exchanged during the conversation.

**3. Seen Message**

**Purpose:** The "seen" message is used to indicate that a message has been seen by the recipient.

**Usage:** When a user views a message, a "seen" message is sent to the sender to confirm that the message has been read.

**4. Typing Message**

**Purpose:** The "typing" message is used to notify the other user that the sender is currently typing a message.

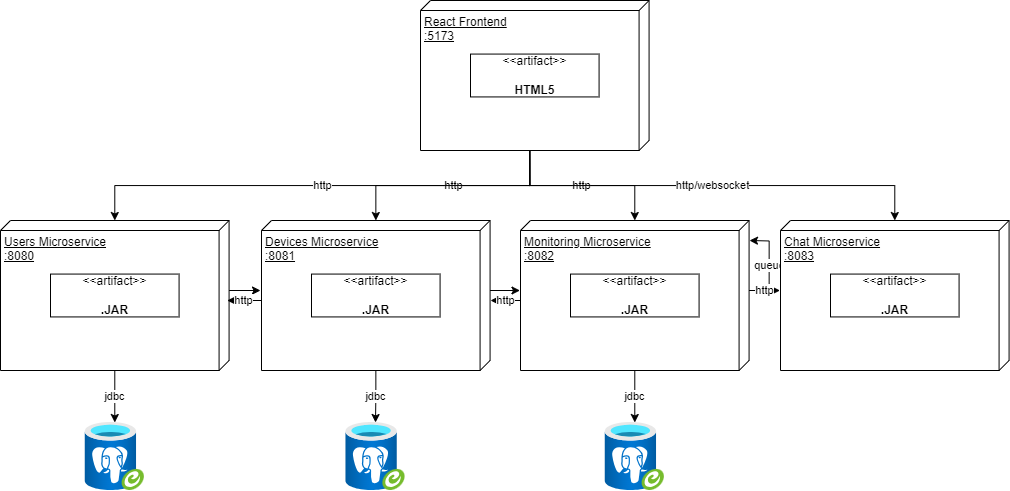
**Usage:** "Typing" messages provide real-time feedback to the recipient, indicating that the sender is actively composing a message.

**5. Untyping Message**

**Purpose:** The "untyping" message is used to notify the other user that the sender has stopped typing a message.

**Usage:** "Untyping" messages inform the recipient when the sender has ceased typing without sending a message.

1. **Deployment**

****

**Frontend**: The frontend service is responsible for serving the user interface of the application. It listens on port 5173 and communicates with the users and devices microservices.

**Users** **DB**: A PostgreSQL database for storing user-related data. It listens on port 5432 and is used by the Users Microservice.

**Devices** **DB**: Another PostgreSQL database for storing device-related data. It listens on port 5433 and is used by both the Users and Devices Microservices.

**Users** **Microservice**: A microservice responsible for handling user-related operations. It communicates with the Users DB and the Devices Microservice.

**Devices** **Microservice**: A microservice responsible for handling device-related operations. It communicates with both the Devices DB and the Users Microservice.

**Monitoring Microservice**: A microservice responsible for simulating sensor data and reporting high (abnormal) power consumption. The threads (sensors simulated in the monitoring microservice) read from the .csv file directly and send the data through the rabbitmq queue to the monitoring microservice. Then the frontend takes that data from the monitoring/sensors database. If the consumption is too high, a message is sent through a websocket to the frontend. It is also responsible for the chat feature.

**Chat Microservice:** A microservice responsible for the communication between administrators and multiple clients via websocket. Multiple message types are in place in order to facilitate the basic chat functionality, including the typind and seen indicators.

The communication between the components is facilitated by the Docker Compose networking. The containers are connected to the same bridge network allowing them to communicate with each other using their service names as hostnames (e.g., "frontend," "users-microservice," etc.).

This configuration provides a scalable and isolated environment for running the application with separate microservices and databases. The frontend can be accessed from the internet, and the microservices interact with their respective databases as needed.