Distributed Systems : Assignment 1

## Introduction

The goal of the project was to develop an Energy Management System. It consists of a front-end that should communicate with two other microservices: One designed for user management and one for the management of the energy metering devices associated with each user. The system has 2 user types: administrator and user/client. The administrator has higher rights, being able to perform CRUD operations on both users and devices and perform the mapping between them. The user can perform CRUD operations on their own devices and their own account.

The back-ends were developed using Java and its Spring Boot framework ,exposing a REST API to interact with the database via requests. The databases for both of them were implemented using PostgreSQL. These requests are made with the aid of a frontend developed using JavaScript and React+Vite. All these components were deployed in their own unique container, and are interconnected, being deployed on the same network using docker-compose.

## Architecture of the application

The back-end was split into 2 microservices: Device and User ,each with their own separate database. The user database contains logic for registering and authenticating users in the system using JWT. And all other requests require a JWT token in order to be performed. The 2 services are also synchronized, as in when a new user is created, an entry in the available user IDs table in the device database is also created.

A diagram of a cloud

Description automatically generated

Since I’ve used Spring Data JPA, the database will be automatically generated when the application is first run, and will then be only updated afterwards, and data will persist. There are some sql scripts in the microservices’ directories, should you want to manually run them in psql inside the docker containers.

## Deployment :

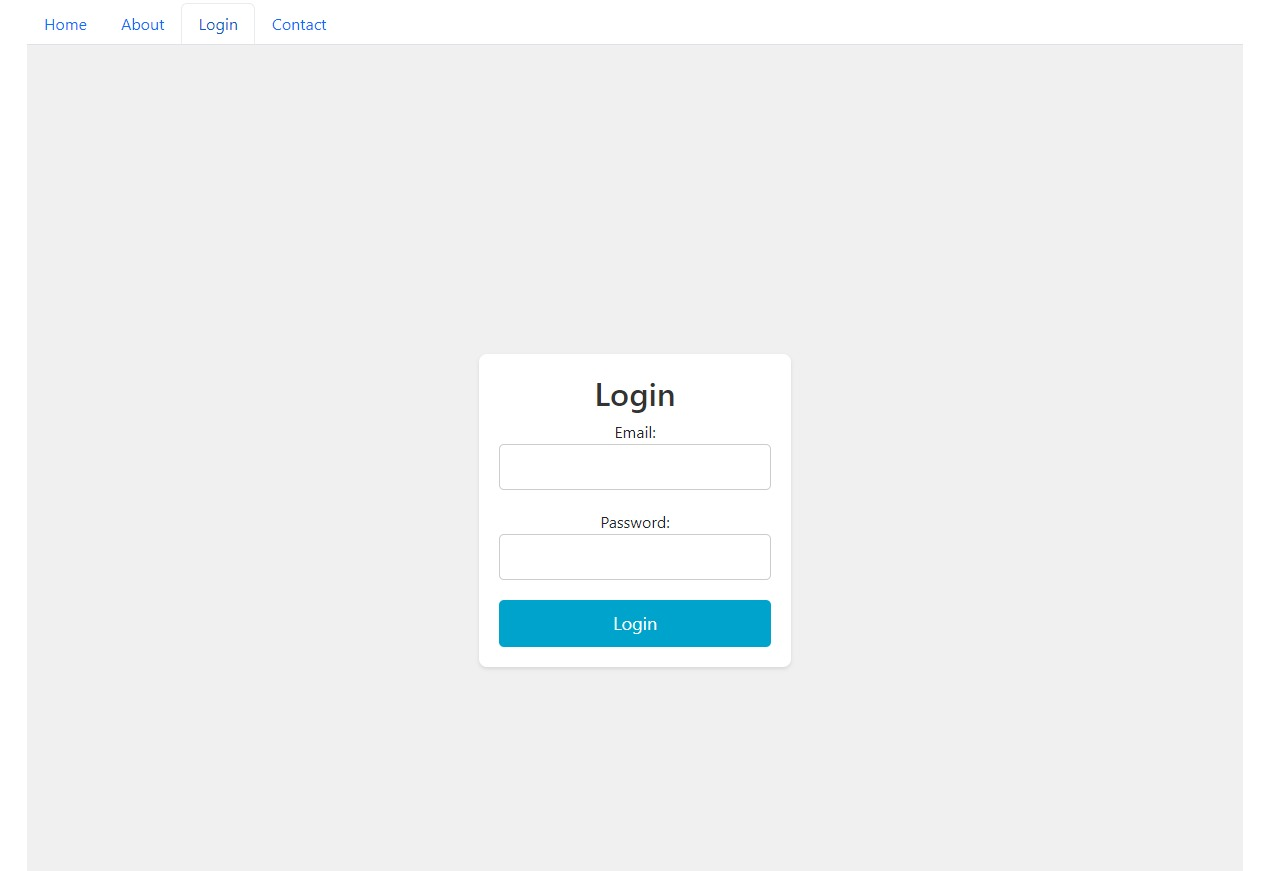
A screenshot of a computer

Description automatically generated

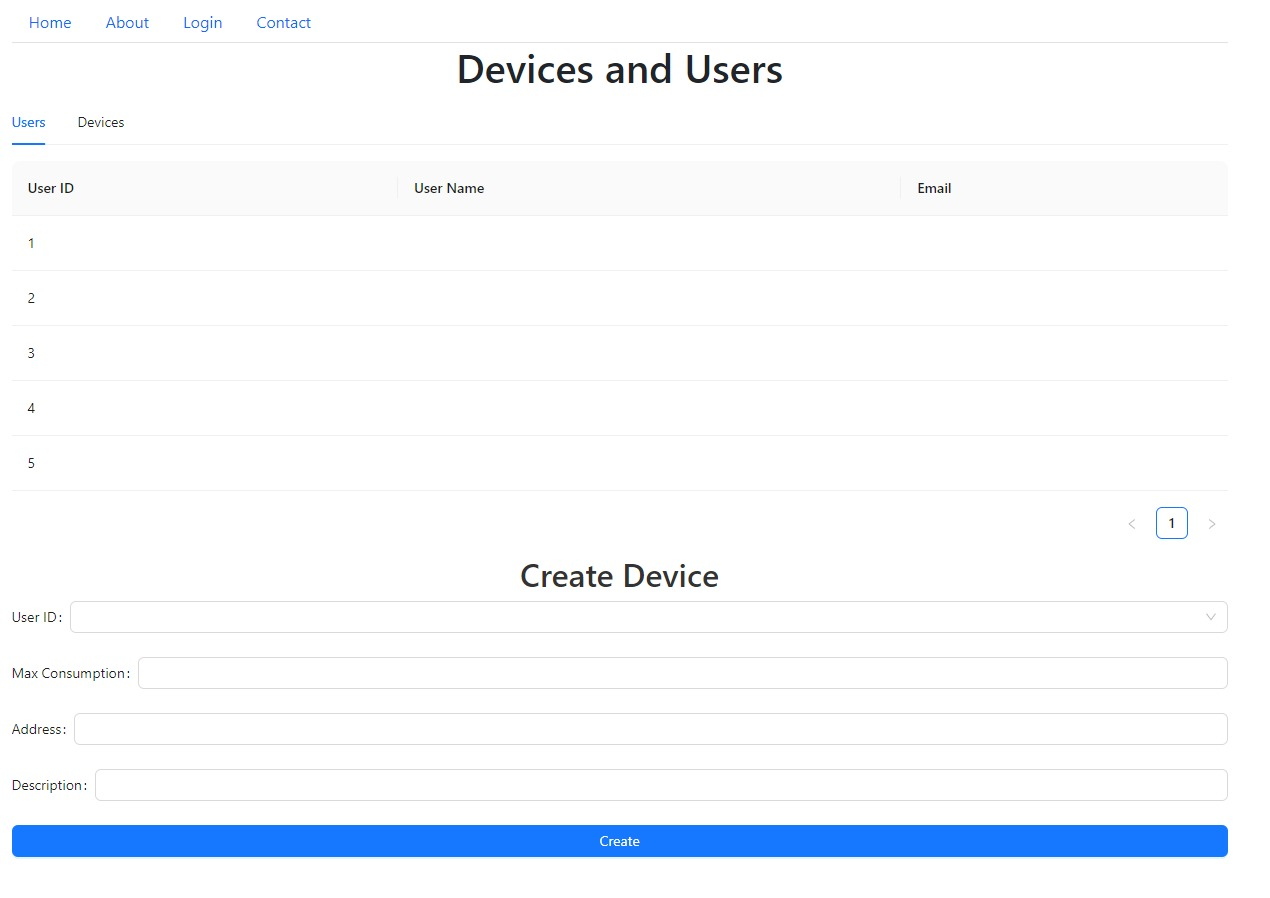
In order to run the application, I had to create the JAR files for the Java applications and the application properties file for the React frontend, then deploy them in images from inside docker. The final result contained a large container built using docker-compose, which contained the 5 smaller containers for every component of the whole application.

## Functionality examples:

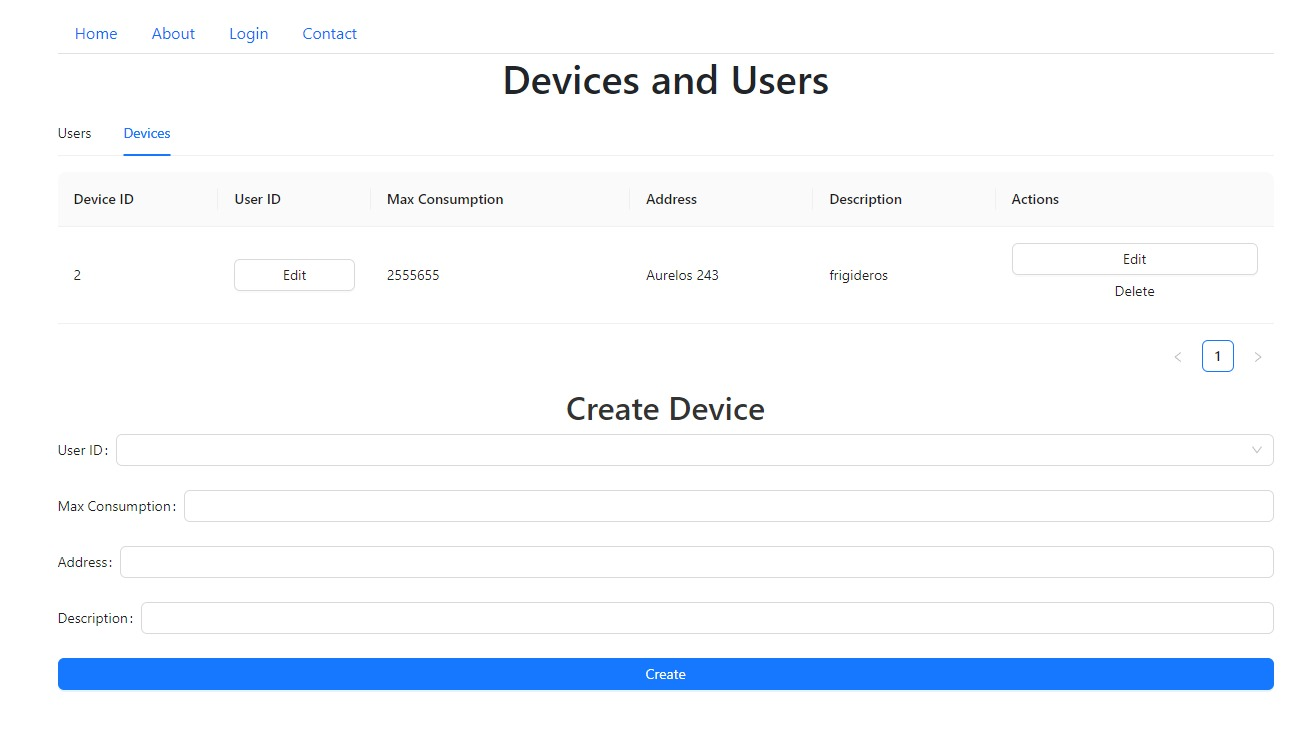
Login Page:



Administrator Dashboard to see user Id’s and device creation form:



Administrator dashboard for device management:



Assignment 2:RabbitMQ

This system is designed to facilitate the seamless exchange of data between applications, allowing for efficient communication in a distributed environment. The core of our architecture leverages RabbitMQ, a robust message broker that enables the decoupling of data producers and consumers.

1. Data Producer:

Functionality: The data producer, implemented in Python, reads consumption data from a CSV file. This data is then sent to the RabbitMQ exchange for further processing.

2. RabbitMQ Message Broker:

Role: Serves as the central hub for data exchange. Manages the routing of messages from the data producer to the Java microservice consumer.

3. Java Microservice Consumer:

Functionality: The Java microservice consumes data from the RabbitMQ queue. It processes the consumption data and determines whether it exceeds a predefined threshold.

Communication: Uses WebSockets to push a message to the React frontend when consumption surpasses the specified threshold.

Finally, the frontend from the previous assignment will display an alert when on the UserDashboard when it receives a message through the socket.

Deployment Overview:

Data Flow: The Python data producer sends consumption data to the RabbitMQ exchange. The Java microservice consumes this data from the queue and communicates with the React frontend via WebSockets.

Real-time Alerting: WebSockets enable instantaneous communication between the Java microservice and the React frontend, ensuring that alerts are promptly displayed when consumption exceeds the threshold.

Scalability: The decoupled nature of components allows for the scalability of the system. Additional producers or consumers can be added without affecting the overall architecture.

WebSockets Integration:

In the implemented architecture, WebSockets play a crucial role in enabling real-time communication between the Java microservice and the React frontend. WebSockets provide a bidirectional, full-duplex communication channel over a single, long-lived connection. This contrasts with traditional HTTP, which is stateless and follows a request-response paradigm.

How WebSockets are Used:

Java Microservice to React Frontend:

Alerting Mechanism: The Java microservice, responsible for processing consumption data from the RabbitMQ queue, utilizes WebSockets to push real-time alerts to the React frontend.

Immediate Notification: When the microservice determines that consumption has exceeded the defined threshold, it initiates a WebSocket connection to transmit an alert message to the React frontend instantaneously.

React Frontend Connection:

Establishment: The React frontend establishes a WebSocket connection with the Java microservice to receive real-time updates.

Subscription to Events: The frontend subscribes to WebSocket events, allowing it to listen for alerts triggered by the Java microservice.

Simple Diagram of the architecture of this specific assignment:

A diagram of a data processing process

Description automatically generated

Updated deployment Diagram:

A computer screen shot of a diagram

Description automatically generated