DS Assignment: Microservice Application with simple Docker deployment

Robotin George Cătălin, 30441

Table of Contents

[1.Overview 2](#_Toc183366511)

[2. Microservices 2](#_Toc183366512)

[2.1 User microservice: 2](#_Toc183366513)

[2.2 Device microservice 3](#_Toc183366514)

[3. Frontend 4](#_Toc183366515)

[4. Docker deployment 4](#_Toc183366516)

[5. Deployment diagram 4](#_Toc183366517)

[2.1 Monitoring microservice 5](#_Toc183366518)

[2.1.1 Measurement simulator 6](#_Toc183366519)

[2.2 Load balancing and reverse proxy 7](#_Toc183366520)

# 1.Overview

This project consists of developing a microservice based application, used for an Energy Management System. We should have independent microservices for users and devices that can be associated with them, as well as separate databases for each and a frontend where the two can be put together.

# 2. Microservices

Each service must include a REST api controller that should be able to perform CRUD operations on the given entites, as well as provide the extra logic to facilitate the ability to keep consistency of data. Each are written using the Java Spring Boot framework and are based on a layered architecture, with separate layers for the entity, service, repository and controller.

## 2.1 User microservice:

The entity is simple, each user having only an id field, name and password. The corresponding endpoints are:

* GET /api/user/getAll
* POST /api/user/addUser
* GET /api/user/getById/{id}
* PUT /api/user/updateUser
* DELETE /api/user/deleteUser/{id}

For this microservice, all operations are standalone, i.e. rely only on the user database and don’t need extra checks for consistency.

Here is the top view of the architecture:

A screenshot of a computer screen

Description automatically generated

Figure . User class diagram

## 2.2 Device microservice

Again, the entity itself is quite simple, only having fields for id, description, address, maximum hourly consumption, as well as an extra field for the mapping with the user database, namely userId, used as a sort of “foreign key”, in order to to be able to track the 1:many relationship between the user and device entities, as such: when a user is deleted, all their devices are also deleted. Also, for non-admin users, when a user is fetched through the login page, also a list of their devices is fetched. This interaction is put together in the frontend. Also, for data consistency, the repository for the users is marked as transactional, such that when the corresponding device operations are being performed, all other transactions have already been processed. For this, the REST endpoints are:

* GET /api/devices/getAllByUserId/{userId}
* DELETE /api/devices/deleteAllByUserId/{userId}
* GET /api/devices/getAll
* POST /api/devices/addDevice
* PUT /api/devices/editDevice
* DELETE /api/devices/deleteDevice/{id}

The top level view is very similar to the one for the user service:

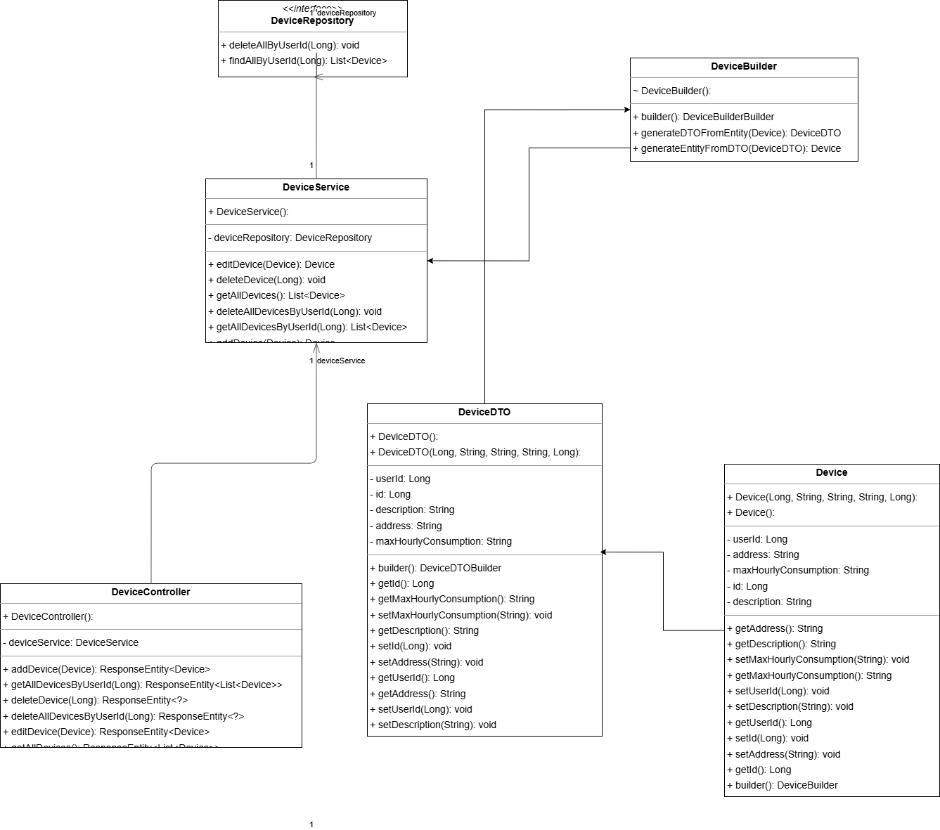


Figure . Device microservice class diagram

# 3. Frontend

The presentation level for both the microservices is a frontend application, using the Angular framework. It has simple views for logging in, seeing a list of devices (i.e. the ones you own) for the normal users, and a view that allows admins to perform CRUD operations on both the devices and users. It uses the built-in httpService from Angular to communicate with the backend controllers through the corresponding api’s, and I used 1:1 models with the backend DTO’s, to have a seamless mapping between them.

# 4. Docker deployment

All the independent components have separate containers, i.e. one for each database, backend and the frontend. Each of the microservices are grouped into networks, which encapsulate the backend and corresponding database containers, to facilitate direct connection and communication between them.

# 5. Deployment diagram

Docker

A diagram of a application

Description automatically generated

Figure . Top level architecture

Assignment 2: Monitoring Microservice and Reverse Proxy + Load balancing

# 2.1 Monitoring microservice

Similar to the previous microservices, this is a standalone Spring Boot application, with a layered architecture and is used as a consumer for a RabbitMQ queue which sends consumption data for different devices, with their corresponding timestamp. There is also a synchronization aspect, as whenever a device is added, its maximum consumption value is also stored in a separate table, in the form of id and value, in order to then be able to push notifications whenever the maximum value is exceeded. The only endpoint used in frontend retrieves all the data for a given device id, and then the processing based on the date of collection is done in the frontend, due to type mismatches when it comes to how dates are represented in both Java and Javascript.

A diagram of a company

Description automatically generated with medium confidence

Figure . Monitoring Class Diagram

## 2.1.1 Measurement simulator

This is used as a producer for the monitoring, sending measurement data read from a csv through the same RabbitMQ queue which the monitoring listens to. It is a standalone desktop application, more specifically a simple maven project, which can be run in multiple instances, a requirement for the last item of the assignment. I used CloudAMQP for the queue in order to not have extra containers in my docker, since space is limited.

# 2.2 Load balancing and reverse proxy

I have done this using Traefik, which is a tool that provides the ability to create replicas for the backend microsevices and handle the routing of the requests automatically, such that the load is properly managed between them.

A diagram of a computer

Description automatically generated

Figure . Final architecture [1]

Bibliography:

1.DSRL website