DISTRIBUTED SYSTEMS

ENERGY MANAGEMENT SYSTEM

Laboratory assignment 3

Student: Muntean Cristian

Group: 30441

Contents

[1. Conceptual architecture of the system 3](#_Toc149401647)

[1.1. Overview 3](#_Toc149401648)

[1.2. Components 3](#_Toc149401649)

[1.3. Conceptual architecture diagram 4](#_Toc149401650)

[2. Deployment 5](#_Toc149401651)

[2.1. Deployment diagram 5](#_Toc149401652)

[2.2. Build and execution considerations 5](#_Toc149401653)

# Conceptual architecture of the system

## Overview

This documentation provides an in-depth insight into the architecture of an Energy Management System, designed to manage and connect users with smart devices to monitor the energy consumption of the devices linked to each user as well as interpreting and displaying monitoring data for every device.

The system's front end is driven by a React application, serving as the user interface. Users interact with this interface, triggering HTTP requests that initiate operations within the microservices. There are two types of users that can interact with the application: clients and administrators. The clients can only see the devices that are associated with themselves, whereas the administrators can create, update, and remove clients and devices, as well as linking devices with clients, making them visible to said clients. The clients receive notifications when a device has passed the maximum hourly energy consumption for their device. Also, they can access a special page that shows them a graph containing data about hourly energy consumption for every device that is linked to them which is updated live with new data coming from their devices.

Throughout this documentation, the focus will be on the technical intricacies of this architecture. Key aspects such as modularity, scalability, and data isolation will be explored in detail, providing a comprehensive understanding of the system's robust foundation.

## Components

The main components of the application are:

* The **user microservice**
  + Back-end microservice developed in **Spring Boot** designed to receive HTTP requests from the front-end application regarding operations related to the users of the whole application.
  + Every user creation/deletion is propagated to the device microservice to facilitate the user-device linkage feature.
  + Connects to two IBM MQ queues that provide user data on demand to the monitoring microservice.
  + The **security** aspect of the application is implemented with the help of **Spring Security** and **JWT** **tokens**, which facilitate authorization on certain API endpoints.
* The **device microservice**
  + Back-end microservice developed in **Spring Boot** designed to receive HTTP requests from the front-end application regarding operations related to devices.
  + Connects to two IBM MQ queues that provide device data on demand to the monitoring microservice.
  + This microservice is not implementing any security, because it is only accessed after the login operation has been performed.
* The **device simulator**
  + Console application designed to read data from a csv, read a device id from a config file, connect to a designated IBM MQ queue, and send a message containing a reading, the device id and a timestamp once every 10 minutes on the queue.
  + Created such that multiple instances of the same app can be run at the same time.
* The **monitoring microservice**
  + Back-end microservice developed in **Spring Boot** designed to receive HTTP requests from the front-end application regarding the request of monitoring data for a specific device and to send notifications and live data back to the front-end app for display purposes.
  + Connects to five IBM MQ queues: two for each of the other microservices and one that connects to the device simulator (from which it consumes monitoring data for the current devices).
  + This microservice is not implementing any security, being accessed only after the login has been performed.
* The **chat microservice**
  + Is responsible for redirecting messages coming from different users to facilitate the chat functionality.
  + Has 2 kinds of topics: 1 for the messages coming from clients and 1 for the messages coming from administrators. Each of the messages are redirected to the other type of topic. The client topics are “dynamic”, having the following structure: **“/topic/message/<client\_username>”**
  + There are 4 types of messages that are sent from the front-end to this microservice and are redirected towards the desired user:
    - Actual text messages. This represents a message sent by a user towards another user
    - Typing messages. This type of message is sent whenever a user starts typing inside a chat component. Using this type of message, the other user can see whenever his/her conversation partner is typing something.
    - Stopped typing messages. This is the opposite of the previous message type.
    - Seen messages. This type of message is sent whenever the user focuses on the chat component. Using this type of message, the other user can see whenever his/her conversation partner saw the last message.
* The **front-end application**
  + Front-end application implemented in **React JS**, using **Axios** to create HTTP calls to the back end microservices, using **Web sockets** to connect to the topics on which data is send from the monitoring microservice.
  + Uses JSCharting to display the monitoring data for each user.
  + Implements all the logic behind sending and receiving messages via the Chat function.
* The **user database**
  + Database that stores information about the users of the application, such as username, password (which is stored encrypted) and role (which can be either CLIENT or ADMIN).
* The **device database**
  + Database that stores information about devices such as description, address, maximum hourly energy consumption.
  + Additional tables are required to keep client-device linkage information. These are a table containing user ids that only holds the ids of the current users and a table that contains client-device linking, which contains the id of the device and the id of the user for each linkage.
* The **microservice database**
  + Database that stores information about readings such as device id, reading value, timestamp.
  + Additional tables are required to keep the client-device linkage information and the usernames for each client. These are a table containing user ids and their username, and a table containing the device id, max hourly consumption, and linked client.
* The **IBM MQ service**
  + Runs 5 queues managed by a queue manager. The queues are used by the other microservices to communicate with each other.
  + Runs a web console such that the admin is able to see and manage the queues from a user interface.

## Conceptual architecture diagram

A screenshot of a computer screen

Description automatically generated

Figure

As it can be seen from Figure 1, the user microservice is responsible with everything that has to do with user operations and authentication, whereas the device microservice is responsible with devices and mapping users to devices. To maintain the synchronization of the existing users between the user microservice and the device microservice, the communicates with the device microservice by accessing different endpoints (present in the UserId Controller). The monitoring microservice is responsible for maintaining readings from every device and push notifications to the front-end app.

Also, the monitoring microservice communicates with the user microservice and device microservice through queues to collect data about the clients and devices present in the application and keep them updated. The monitoring microservice communicates with an indefinite number of device simulators from which it receives energy reading data.

Also, the chat microservice communicates bidirectionally with the front-end application to supply messages to the app’s users.

## Security

The security part of the application is implemented using Spring Security in some microservices, and a simple JWT check in others.

The main microservice that handles security is the user one, that being responsible for generating a JWT token whenever a login or register happens. The JWT token has as claims inside it the username and password of the user, as well as a Boolean field that signals to the other microservices that the received JWT token is a valid one.

The other microservices check JWT token that is received in any request they get, either by implementing Spring Security and having a custom JWT filter that authorizes the request, or by having a service that checks the authenticity of the JWT, without implementing Spring Security. In order to check whether a given JWT token is valid or not, the secret key is shared among all microservices, being present inside the “application.yaml” file.

# Deployment

The whole application is deployed on Docker, using a total of 5 different containers to run every component individually. These can be categorized in the following:

* Containers that run MySQL Servers: There are three different containers that run MySQL Servers, which are used to run the user, device, and monitoring databases.
* Containers that run Tomcat Servers: There are also four different containers that run Tomcat Servers, which are used by the spring boot microservices. Each microservice accesses their respective database, except the chat microservice, which doesn’t require a container.
* Container that runs a NGINX Server: Used to run the React application that provides the user interface which the user can use to interact with the whole application. The NGINX Server also catches and reroutes every request to its respective IP:Port location for the microservices to be used.
* Container that runs a IBM MQ Server: Used to run the IBM MQ queue manager, which is responsible for managing the 5 queues used by the whole distributed system.

## Deployment diagram

A diagram of a computer network

Description automatically generated

Figure

As it can be seen from the deployment diagram from above, the browser can access the front-end application by using for example the <http://localhost:3000> URL. Whenever a request needs to be made towards the microservices, given the code is ran in the local browser, it needs to access APIs present at localhost:8081 or localhost:8082 or localhost:8083, but those requests first reach the NGINX server which runs the front-end react application. Whenever a request towards localhost:8081 or localhost:8082 or localhost:8083 is caught by NGINX, it forwards that request towards 172.16.0.54:8081 or 172.16.0.55:8082 or 172.16.0.58:8083 respectively, where the containers that have the Spring boot microservices are running. In turn, whenever a microservice receives a request, it needs to use its respective database, which is in a different container. So, each microservice connects to a database present at 172.16.0.53:3306 (location where the docker container which runs the MySQL server that has the user database is) or 172.16.0.52:3306 (location where the docker container which runs the MySQL server that has the device database is) or 172.16.0.57:3306 (location where the docker container which runs the MySQL server that has the monitoring database is).

## Build and execution considerations

### Docker compose file

The whole application can be deployed on Docker by running the “docker-compose.yml” file present at the root of the microservice directories. The structure of the file can be split up in 4 main parts:

* Internal network configuration: A bridge network is configured that is used to connect all containers together.

A computer screen shot of a black background

Description automatically generated

Figure

* Database containers configuration: Three containers are created that contain the user, device and monitoring databases are configured to run on the 172.16.0.52:3306, 172.16.0.53:3306 and 172.16.0.57:3306 ip:port pairs, with the exposed ports being 3307, 3308 and 3309 respectively.

A screenshot of a computer program

Description automatically generated

Figure

* Microservice containers configurations: Four containers are configured to run the Spring boot microservices that run the user, device, monitoring and chat microservices on 172.16.0.54:8081, 172.16.0.55:8082, 172.16.0.58:8083, and 172.16.0.60:8084. Environment variables are set in order to connect to the desired database, and each of them wait on the creation and execution of their respective database container before they are created and ran.

A screenshot of a computer program

Description automatically generated

Figure

* Front-end container configuration: The front-end application is configured to run on port 3000, having the same port exposed externally, in order to be reached at <http://localhost:3000>. It also depends on the containers that run the spring-boot microservices so it is built and ran after the creation of both.

A screen shot of a computer

Description automatically generated

Figure

* IBM MQ container configuration: The container is configured to run on ports 1414 (the IBM MQ server) and 9443 (the IBM MQ web console), having the Queue Manager name set to “Monitoring\_Queue\_Manager” and running on 172.16.0.59 IP address on the local custom network.

A screen shot of a computer

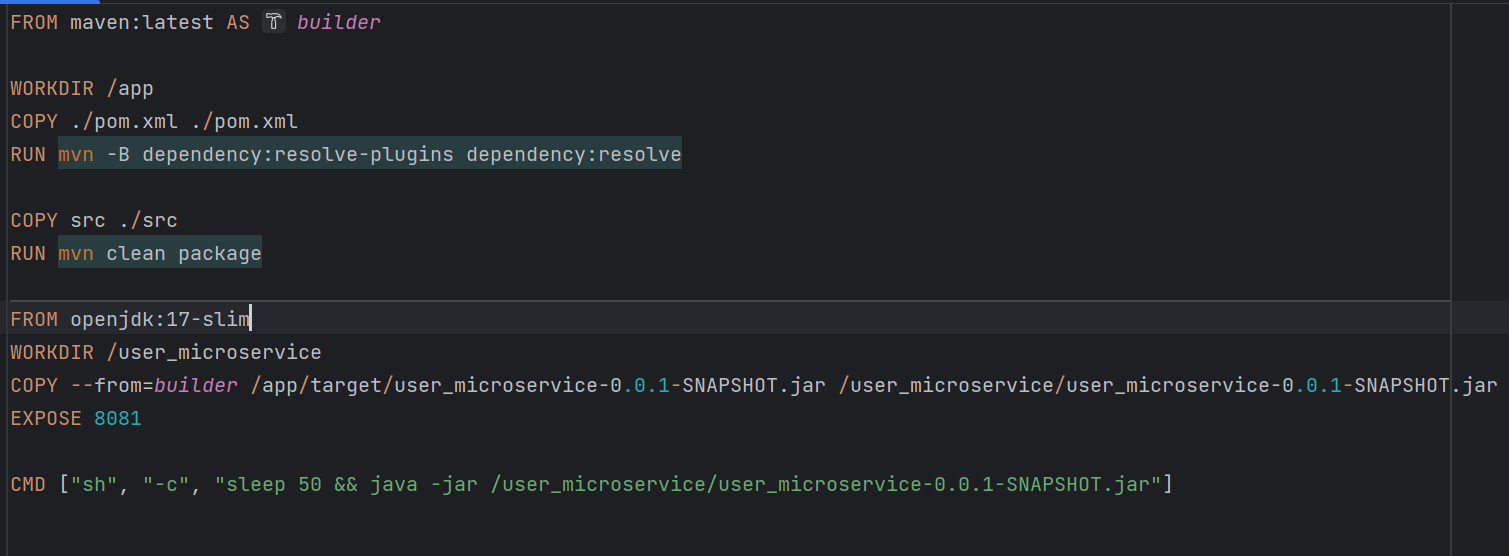
Description automatically generated

Figure

### Dockerfiles

Every docker container contains a dockerfile that is responsible for building the building the docker images that should run on the docker containers. The main idea behind all of the dockerfiles is that the source code is copied on the container, followed by a build of the code in order to make it runnable, then the formed executable is ran.

The dockerfiles for the microservice apps are similar and look like the one in Figure .8



Figure

The dockerfile for the front-end app can be seen in Figure 9.

A screen shot of a computer program

Description automatically generated

Figure

The dockerfiles for the database images creation are similar and looks like the one present in Figure 10.

A screenshot of a computer

Description automatically generated

Figure

The dockerfile for the creation of the IBM MQ container looks like the one presented in figure 11.

A screenshot of a computer

Description automatically generated

Figure

This copies the queues present in the queues.mqsc file into the /etc/mqm/queues.mqsc file present in the docker container. The queues.mqsc file has the following content presented in Figure 12.

A screen shot of a computer program

Description automatically generated

Figure

As it can be see, there are 5 queues created: Device\_Monitoring\_Queue, Device\_Request\_Queue, Device\_Send\_Queue, User\_Request\_Queue, User\_Send\_Queue. Also, there are 4 queues deleted. These represent the default queues that get created for development purposes.

### Batch file for docker cleanup

In order to make the cleanup of the docker containers, images and volumes faster, I also created a batch file that removes all running containers, all images and all volumes and it can be seen in Figure 13.

A screenshot of a computer program

Description automatically generated

Figure