**Distributed Systems Project**

**Assignment 1**

**Request-Reply Communication**

By Sandu Bogdan-Emilian

30422

1. **Introduction**

The focus of this assignment was to build the main functionality and architecture of the Energy Management System. It consists of a front-end which is built using ReactJS and 2 microservices developed in NestJS.

For the front-end part the main styling was developed with the help of MaterialUI.

The 2 micro-services are hybrid NestJS applications since they act as a HTTP server and as a micro-service at the same time. Both microservices accept HTTP requests and they are connected using as protocol TCP for communication and synchronization.

1. **Conceptual architecture of the distributed system**

Each microservices has its own database. The synchronization between the 2 databases is performed with the help of a TCP protocol which connects the two. Whenever a change which requests synchronization with the other database is performed an event is emitted which is catched by the other microservice which performs the needed synchronization.

The architecture consists of a front end application, 2 microservices(user\_microservice, device\_microservice) and 2 databases one for each microservice(user\_database, device\_database).

A diagram of a device database

Description automatically generated

The application uses 5 servers. One for the React App responsible for rendering the pages, one for the User Microservice responsible with everything user related (auth, CRUD, security, synchronization etc), one for the Device Microservice responsible with everything device related (CRUD, synchronization etc), one for the User Database and one for the Device Database. Requests are being made to the microservices through a series of API calls made in the React App. These requests are intercepted by their corresponding controllers, checked if the session is valid using JWT and checking if the user has the necessary role to access the resource. If the request is allowed then corresponding services will take care of the business logic and necessary operations will be performed on the corresponding databases.

The flow is the following: user accesses page, the component makes the specific requests to the backend APIs setting the needed data for it (header, body etc.). The corresponding microservice responds to the request and sends back a response. That response is then processed by the frontend. The whole communication flow is performed using RESTful APIs.

1. **Database design**

Energy Management System I used 2 databases, one for the user microservice and one for the device microservice.

The user database contains the following entities: User, Role. We have the following 3 tables: user, role, user\_to\_role.

User Entity: id, email, password, firstName, lastName, phoneNumber, roles (ManyToMany).

Role Entity: id, name.

The device database contains the following entities: Device. We have the following table: device.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedDevice Entity: id, userId, description, maxHourlyCo nsumption, address.

1. **UML Deployment Diagram**

The main components are the React App, the 2 NestJS microservices and the 2 databases, every one of these components are deployed independently. The communication is achieved by HTTP requests and TCP between the microservices. All the components run in their own docker container. The architecture is distributed because we could run every container on a different machine and not all on the same one, thus we can leverage the use of cloud platforms such as AWS for deploying our components.

A diagram of a computer

Description automatically generated

1. **README considerations**

To run the deployed version of this application you need to configure your .env files with necessary fields (DB\_USERNAME, DB\_PASSWORD) for both microservices. The frontend doesn’t require any .env file to run although a change could be made such that we set up .env files for each application and branch in order to run the version of the application we want(the dockerized one or the classic one).

Additionally you need to add all 3 projects in the same directory and at the root add the docker-compose.yml file present in the front-end repository.