DISTRIBUTED SYSTEMS

Assignment 1

Request-Reply Communication Paradigm

Online Energy Utility Platform

Andrei Dorofte

Group 30641

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

**Backend**

The distributed system must consist of an online platform designed to manage clients, smart devices equipped with energy consumption sensors and monitored data from sensors. This application was implemented using a Spring REST API, for backend functionality. The conceptual arhitecture of the system is the following:



- controller (for the request mappings)

- repository (for persisting Java objects into the database)

- entity (an entity corresponds to a table from the relational database)

- service (for business logic and applying more complex operations and validations before accessing the repository layer)

- dto (objects exposed outside the application -> to the UI)

- webSecurity (for securing the application authentication)

The application is based on the request-response communication paradigm, and it is secured with the Spring provided security solution, using JWT token for user authentication. A token cache is also used in order to mantain current stored tokens in a Map. On logout, current token is removed from the map. Also, there are two roles available for every user: ROLE\_USER and ROLE\_ADMIN. Provided this, every type of user can access certain endpoints provided by the application.

The user must sign-up in order to use the application. After that, he can login, and the backend application will return the generated token for the current session, along with the usename and user’s roles. Then, he will be redirected to the page corresponding to his role. Basic users cannot access admin management pages. Admins can perform CRUD operations on the tables in the database: users, devices and sensors. Also, he can assign devices to users and view user’s devices, as well as assign sensor to device and view device’s sensor.

**Frontend**

The user interface of the distributed system was implemented using React.js library. React components were used, along with functions, for pages. The calls to the backend are managed by the axios component. Async and await promises are used for managing the responses from the backend, instead of callback functions, in order to solve the problem of slow backend response.

The homepage is common for both users and administrators. It displays a welcome message, along with the username.

The administrator page contains links to the 3 pages, where he can perform CRUD operations on the users, devices and sensors, which are displayed using a table. The buttons for editing or deleting a row, either open a new component, or a modal, in case of client devices or assignation of a device.

Text

Description automatically generated

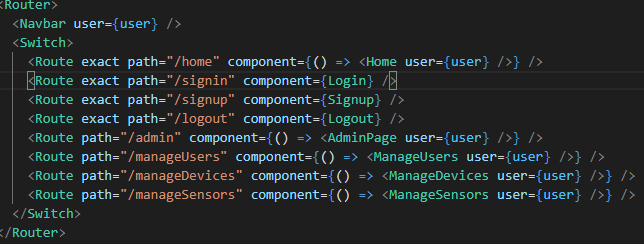
The package structure relies on having a separate package for every page, or set of pages (e.g. Admin management pages).

The requests to the backend endpoints are made using axios, in a separate class, HttpRequestController.js. I used the async and await promises, in order to pass the slow backend responses. The login token is stored in localStorage, and it is added to the header of every request, using the const function auth-header.js.

Text

Description automatically generated

Routing is done using BrowserRouter, and every Route is defined inside the main Switch component.



1. **Database Design**

The ER Diagram of the database:

Diagram

Description automatically generated

User represents the entity that is used on login. It has a one-to-one relationship with a Client entity, which represents the user’s details.

Role entity represents the user’s roles. It only has a field with the name of the role, which is actually a enumeration.

I found it easier to think that between an user and a role is a many-to-many realtionship, mainly because an user has one or more roles, and a role can point to one or more users. This lead to the creation of the user\_roles table, in which every entry corresponds to a role of a user.

The Client entity contains a list of his own devices, which leads to a one-to-many relationship between a client and its devices.

The Device entity contains a back refernce to the client it is assigned to, as well as a refernce to the sensor which is attached to it. This means it is a one-to-one relationship between a device and its sensor.

Lastly, the sensor entity contains a refernce to a list of measurements that are taken during a period of time, which leads to a one-to-many relationship.

1. **UML Deployment diagram**

Diagram

Description automatically generated

Here is the UML Deployment diagram as I could understand from snooping over Google. User accesses the link provided by Heroku, from the deployment of the frontend application. The deployed frontend app then calls the endpoints of the deployed backend application, over the Heroku provided endpoint. Then, the Heroku database of the backend container is used to store all the objects used.

The deployment was done by adding a pipeline worker into the backend and frontend repository on GitLab, along with the .gitlab-ci.yml configuration for it.

In order for the Heroku backend app to access the postgreSQL database, an add-on was needed. It created a new container with a blank database with all the app tables, and the authentication credentials for the database were defined in the .gitlab-ci.yml file.

Because the new database was all empty, I exposed an endpoint from the backend, which creates the roles for admin and user, and adds the predefined admin user. In order to use the app, the database must be configured if it is empty, as every user that signs up, it’s created as a basic user, and if the roles does not exist in the database, the app will not work.

The apps were deployed on two separate containers on my Heroku profile, and can be accessed freely over the two URLs:

* Frontend: <https://online-energy-monitor-frontend.herokuapp.com/>
* Backend: <https://online-energy-monitor.herokuapp.com/>