UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Distributed Systems

Assignment 1

Morar Timotei Cristian
Grupa: 30241

Facultatea de Automatica
si Calculatoare

30 Noiembrie 2024

Cuprins

1 System Requirements

1.1 Functional Requirements

Develop an Energy Management System that consists of a frontend and two microservices designed to manage users and their associated smart energy metering devices. The system can be accessed by two types of users after a login process: administrator (manager), and clients. The administrator can perform CRUD (Create-Read-Update-Delete) operations on user accounts (defined by ID, name, role: admin/client), smart energy metering devices (defined by ID, description, address, maximum hourly energy consumption), and on the mapping of users to devices (each user can own one or more smart devices in different locations).

1.1.1 Authentication

- Users must log in to access the system.
- Upon login, users are redirected to a dashboard based on their role.
- Role-based access restriction ensures that users cannot access pages not associated with their role.

1.1.2 Administrator (Manager) Role

The administrator has the ability to manage users, devices, and associations between users and devices.

- **User Management:** Create, Read, Update, Delete (CRUD) operations on users (defined by ID, name, and role).
- **Device Management:** CRUD operations on devices (defined by ID, description, address, and maximum hourly energy consumption).
- **Mapping Users to Devices:** Assign devices to users, with each user potentially having multiple devices.

1.1.3 Client Role

Clients can only view their assigned devices. Each client page displays all devices that the client owns or has access to.

1.2 Non-Functional Requirements

1.2.1 Microservices

The application is designed using a microservices architecture, featuring:

- **User Management Microservice:** Handles CRUD operations and authentication for user accounts.
- **Device Management Microservice:** Manages device-related operations and user-device associations.

1.2.2 Security

The system employs authentication measures to prevent unauthorized access. Cookies or sessions are utilized to ensure users can only access their respective role-based pages.

1.3 Implementation Technologies

The following technologies have been chosen for the EMS:

- **Frontend:** ReactJS
- **Backend Microservices:** Java Spring REST with Maven for dependency management
- **Data Storage:** PostgreSQL
- **Deployment:** Docker was used to create container for each individual service

2 System Architecture

2.1 BACKEND

The backend architecture is organized into two distinct microservices: **User Management** and **Device Management**. This architecture is designed using a layered approach, allowing for a clear separation of responsibilities within the application. To enhance code organization and maintainability, each microservice is divided into several packages, including **Controllers**, **DTOs**, **Entities**, **Services**, and **Repositories**.

- **Controllers:** The controller components play a critical role in managing incoming HTTP requests. They are responsible for interpreting these requests and routing them to the appropriate service methods for further processing. Acting as the primary interface between the client-side application and the backend logic. In addition to routing, controllers typically perform input data validation to ensure that all required parameters meet specified formats and standards. After processing, they also prepare and structure the responses based on the outputs received from the services, ensuring that clients receive the correct and expected information.
- **Data Transfer Objects (DTOs):** DTOs are specialized objects that facilitate efficient data transfer between different layers of the application. By using DTOs, the system minimizes the exposure of internal data structures, thereby enhancing security and performance. DTOs carry the necessary data in a well-defined, structured, and serializable format, making them suitable for both requests and responses.
- **Entities:** Entities are fundamental components that represent the core data models within the system, directly mapping to the corresponding tables in the database. By encapsulating the data structure, entities provide a clear representation of the data without embedding any business logic, which allows them to focus solely on data representation.
- **Services:** The service layer contains the essential business logic that orchestrates operations across multiple entities and layers within the application. Services are responsible for validating business rules and managing complex interactions between various data objects, thereby ensuring data consistency and integrity throughout the application. By acting as intermediaries, services coordinate the flow of data between controllers and repositories, enabling smooth communication and efficient processing of requests.
- **Repositories:** The repository layer abstracts the data access logic, providing a clear and reusable interface for managing CRUD (Create, Read, Update, Delete) operations for entities. Repositories encapsulate the details of how data is retrieved and stored in the database, allowing services to interact with the data layer without needing to know the specifics of the underlying database technology. The use of repositories ensures that the data access logic is centralized, making it easier to modify or extend in the future as application requirements evolve.

2.2 FRONTEND

For the frontend of the Energy Management System, I opted to use React, a popular JavaScript library known for its flexibility and efficiency in building user interfaces. One of the key advantages of React is its component-based architecture, which promotes reusability and makes it easier to manage and scale the application as it grows.

Within the application, multiple modules are implemented to handle different aspects of the user experience. These include separate pages for administrators and regular users, as well as a main landing page. Each of these modules contains various components tailored to their specific functionalities. For example, the admin page may include components for managing users and devices, while the client page focuses on displaying relevant data about the user's smart energy devices.

To facilitate user authentication, I developed a custom Session Storage mechanism that securely stores data related to the currently logged-in user, functioning similarly to a cookie. This approach allows for easy retrieval of user information across different components and sessions. Additionally, the application employs a RoleGuard to manage access to specific pages based on user roles. This security feature prevents regular users from accessing administrator-only pages, ensuring that sensitive functionalities are protected and accessible only to authorized personnel.

The combination of React's powerful features, along with a well-organized project structure and robust authentication methods, provides a solid foundation for the frontend of the Energy Management System.

3 UML Deployment Diagram

Docker is a powerful platform that enables developers to automate the deployment of applications inside lightweight, portable containers. These containers encapsulate all the dependencies and configurations necessary for the application to run consistently across different environments. The use of Docker simplifies the development and deployment processes, promotes resource efficiency, and enhances scalability by allowing applications to be easily packaged and moved between development.

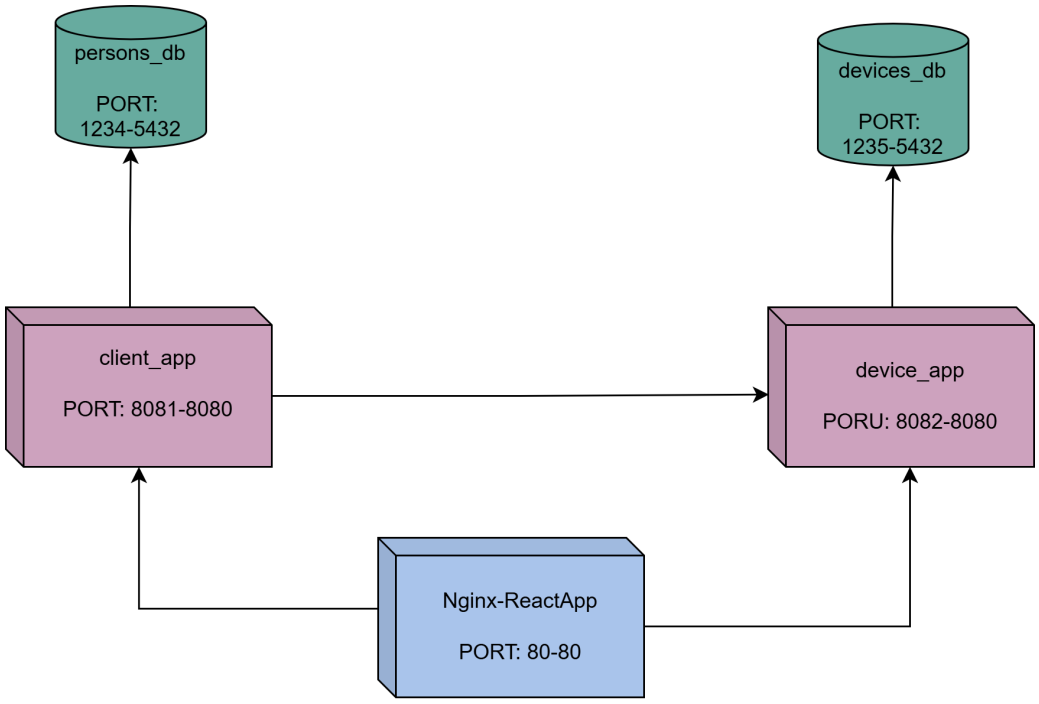

Figura 1: Containers interaction diagram

- **Network Configuration:** The deployment defines a custom bridge network named backend, allowing the services to communicate with each other efficiently. The network is configured with a specific subnet (172.18.0.0/24), ensuring that all containers can communicate while being isolated from other networks.
- **Persons DB:** Mapped host port 1234 to container port 5432, allowing access to the PostgreSQL database from outside the container.
- **Devices DB:** Mapped host port 1235 to container port 5432, providing access to the second PostgreSQL database.
- **Client APP:** Built from the client-App/ directory, which contains the necessary files for the client application. The environment variable *OTHER-SERVICE-URL* is set to communicate with the device application running at *http://device-app:8080*. Maps host port 8081 to container port 8080, allowing external access to the client application. nsures that the persons db service is started before this service, providing the necessary database connectivity. Assigned a static IP address (172.18.0.4) within the backend network, facilitating direct communication with other services.
- **Device APP:** Built from the device-App/ directory. Maps host port 8082 to container port 8080, allowing external access to the device management application. Ensures that the devices db service is up and running before this service starts. Assigned a static IP address (172.18.0.5) within the backend network for direct service-to-service communication.
- **Nginx React:** Built from the react-demo/ directory, which contains the necessary files for the React application. Maps host port 80 to container port 80, allowing external access to the Nginx server serving the React application. The NODE-ENV variable is set to production, optimizing the application for deployment. Assigned a static IP address (172.18.0.10) within the backend network, facilitating communication with other services if necessary.

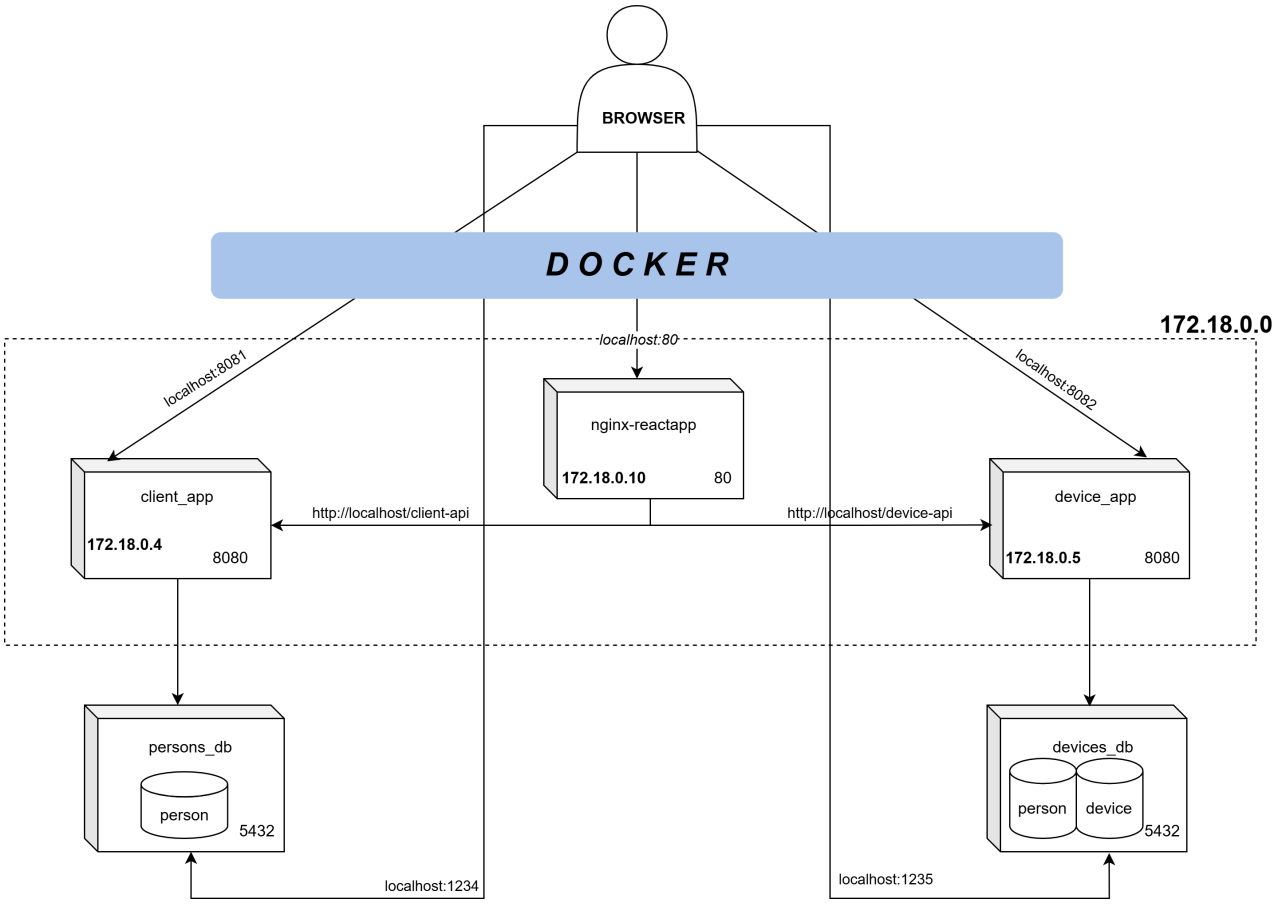

Figura 2: Deployment diagram

4 Security Implementation

To secure the EMS, the following authentication and authorization methods are implemented:

SessionStorage Class

The `SessionStorage` class encapsulates functionality for managing user session data in the browser's `sessionStorage`. It includes three static methods:

- `setUserInfo(id, username, location, role)`: Stores user information such as ID, username, location, and role in `sessionStorage`.
- `getUserInfo()`: Retrieves the stored user information as an object, allowing easy access to user details.
- `clearUserInfo()`: Removes user information from `sessionStorage`, effectively logging out the user and clearing their session data.

This class serves as a utility for managing user session state, ensuring that critical user details are readily accessible and maintainable throughout the application.

ProtectedRoute Component

The `ProtectedRoute` component is a higher-order component designed to enforce role-based access control in a React application. It takes in a component to render, along with the allowed roles for that route. The component uses React Router's `Route` and `Redirect` components to manage routing:

- It checks the user's role retrieved from the `SessionStorage` class.
- If the user's role is included in the `allowedRoles` array, the specified component is rendered; otherwise, the user is redirected to an error page (`/error`).

This component enhances security by ensuring that only authorized users can access certain routes based on their roles, providing a streamlined way to manage protected routes in the application.

Together, these components facilitate user session management and role-based routing, contributing to a secure and user-friendly application experience.

5 Conclusion

The backend architecture of the Energy Management System employs a microservices approach with a layered architecture, promoting scalability, maintainability, and a clear separation of concerns. This design enables each microservice to effectively manage its specific functionalities while ensuring robust interactions across the system.