Sisteme Distribuite

Energy Management System

Autor: Sarkozi Lorand

Grupa: 30242

Contents:

1. Project Overview

This project involves a user and device management application designed to perform CRUD (Create, Read, Update, Delete) operations on user and device data.

The application utilizes Angular for the frontend and Spring for the backend, ensuring a modern, responsive interface and a secure, reliable server-side structure. For containerization, I used Docker as it enables each component (frontend, backends, and databases) to run in isolated environments, making the setup consistent across different systems and simplifying the deployment and scaling processes. Docker Compose coordinates these services, connecting them through a shared network to enable seamless inter-service communication. The application focuses on managing users and devices within a microservices architecture. Users and devices are handled through two interconnected services:

- A User Service, which manages detailed information on users, including attributes like name, contact information, and associated devices.

- A Device Service, which manages device-specific data and includes references to users for tracking device ownership and usage.

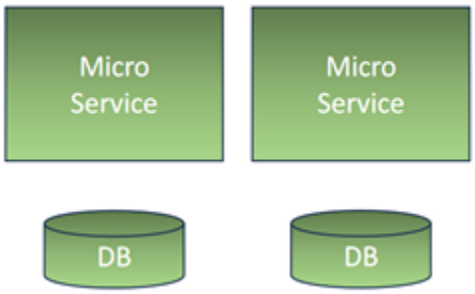
The application uses RESTful APIs for secure, standardized communication between services, with endpoints for creating, updating, and managing both users and devices. This microservices structure enhances scalability, flexibility, and facilitates handling each service independently.

2. Implementation Details

- Microservices Design: The project is divided into two backend services:

    - **User Backend**: Manages user data, such as name and contact details.

    - **Device Backend**: Primarily manages device-specific information, storing device details, and referencing users as needed.



- **Database and ORM**: Each backend connects to its own MySQL database instance, where:
    - **User Database** stores detailed user information.
    - **Device Database** handles device data and user relationships.
- **Session and State Management**: To optimize resource usage and improve response times, sessions are managed through server-side configurations, while LocalStorage is used on the client side to handle temporary data. Spring manages ORM tasks using Hibernate for seamless database interactions.
- **Technology Stack**:
    - **Frontend**: React, with state management, routing, and component-based architecture.
    - **Backends:** Spring Boot-based microservices, REST API, and database integration.
    - **Database**: MySQL, managed separately for each backend.
    - **Containerization**: Docker, enabling an isolated, scalable environment.

3. Justification for Chosen Approach

The microservices approach was selected for several reasons:

- *Scalability*: Each service can scale independently, ensuring that resources can be allocated efficiently, especially when usage varies between user management and device management.
- *Fault Isolation*: Issues in one service (e.g., device backend) do not necessarily impact the other (e.g., user backend), enhancing overall application stability.
- *Flexibility*: Services can be deployed, updated, and maintained independently, allowing continuous improvements to specific parts of the application without affecting the entire codebase.
- *Separation of Concerns*: Decoupling user and device management improves modularity and maintainability.

Compared to Monolithic Architecture:

While monolithic architecture might be simpler to deploy, it lacks the separation of concerns that microservices offer. In monolithic setups, even minor changes can require a full application redeployment, potentially increasing downtime and risk. Microservices architecture, on the other hand, provides modularity and encourages code reusability, despite increased complexity in managing inter-service communication.

Disadvantages:

- *Complexity*: Microservices introduce more complexity with inter-service communication and network overhead.
- *Increased Resource Use*: Running separate containers for each service may increase resource consumption.
- *Coordination Challenges*: Service dependencies need to be managed, which requires robust configuration.

4. Technology Stack and Integrations

Angular

- The Angular frontend serves as the interface for users to interact with both backends.
- It utilizes ng-router-dom for seamless navigation and state management for handling user and device data interactions.
- Forms, tables, and other UI components provide a user-friendly experience, while state and props manage data flow within the app.

Spring Framework

- The project uses Spring Boot in both backend services to handle business logic and RESTful endpoints.
- Controllers: These expose the REST API endpoints and serve as the entry points for client requests.
- Services: Business logic is handled here, encapsulating the functionality to interact with the data layer and manage user/device data.
- Repositories: These provide access to the databases, offering CRUD operations through Spring Data JPA.
- DTOs (Data Transfer Objects): These serve as intermediary objects to ensure that only necessary data is exchanged between the front and backends.
- Entities: These represent the data model for each backend, mapping the database tables to objects in the Java application.

Docker

• Docker simplifies the deployment process by containerizing each component, creating an isolated environment for each backend and database.

• Containers enable consistent deployment across environments and simplify scaling and updating of individual services.

• A shared network (via Docker Compose) allows seamless communication between containers, while environment variables control database credentials, IPs, and ports.

5. Architecture and Components

Controllers:

• Each controller class acts as an API layer, responsible for receiving HTTP requests from the frontend and returning responses.

• For example, the UserController might handle all /user routes, while DeviceController handles /device routes. Services:

• The service layer contains business logic that processes data before it's sent to or retrieved from the repository layer.

• It ensures clean separation from controllers, helping with code maintainability and testability.

Repositories:

• Repositories interact directly with the database. They extend JpaRepository for convenient access to CRUD methods.

• For instance, UserRepository and DeviceRepository manage database operations for user and device tables, respectively.

DTOs (Data Transfer Objects):

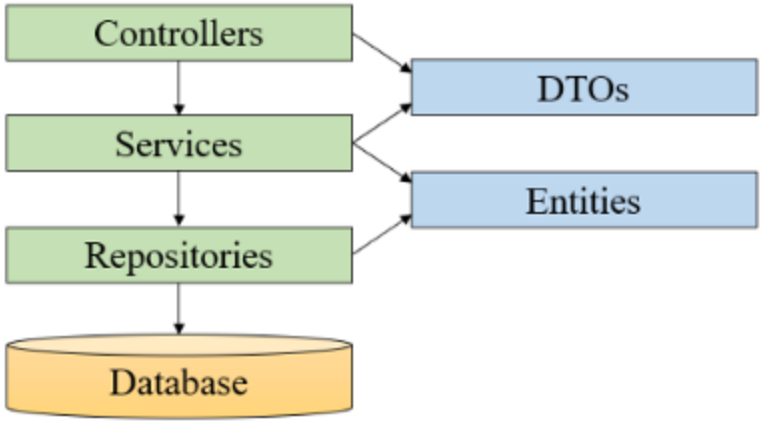
• DTOs facilitate communication between layers, allowing only the necessary data to be transferred, enhancing security and performance.

• For example, a UserDTO might contain only specific fields like name and email, while the UserEntity contains complete data.

Entities:

• These represent the core data model. Entities define table structure and relationships using annotations such as @Entity, @Table, and @ManyToOne.

• They form the foundation for the database layer, enabling ORM through Hibernate to interact with MySQL databases seamlessly.

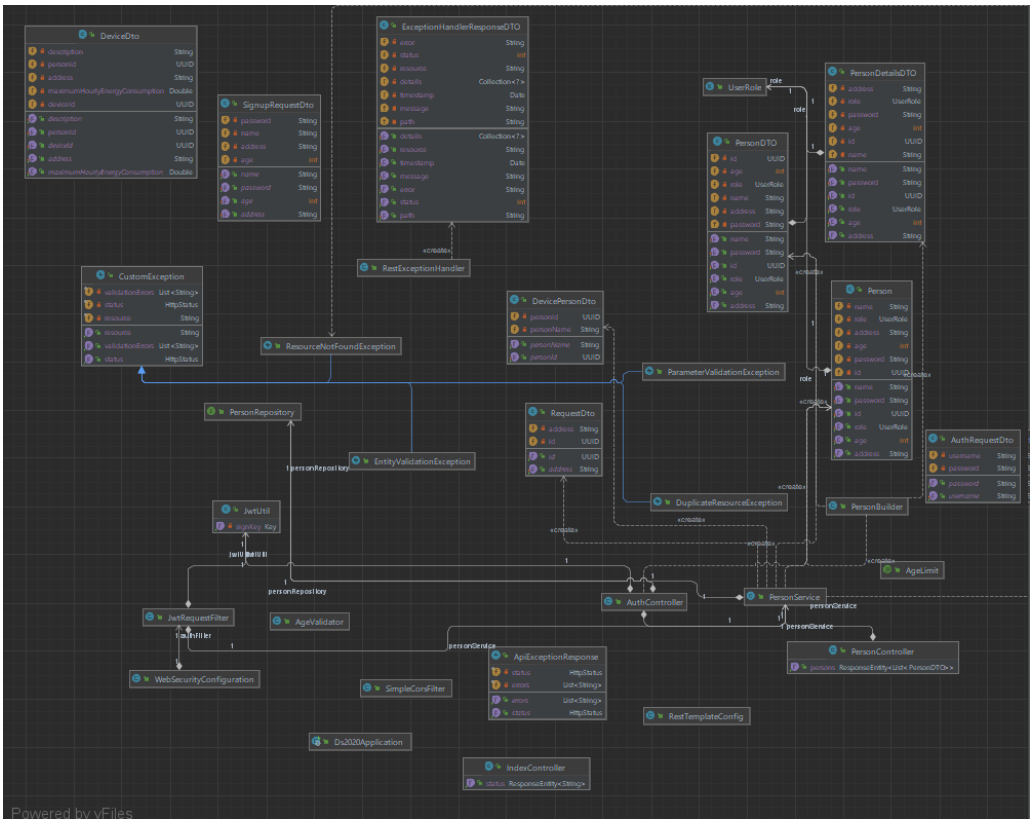


6. Conclusion and Future Development

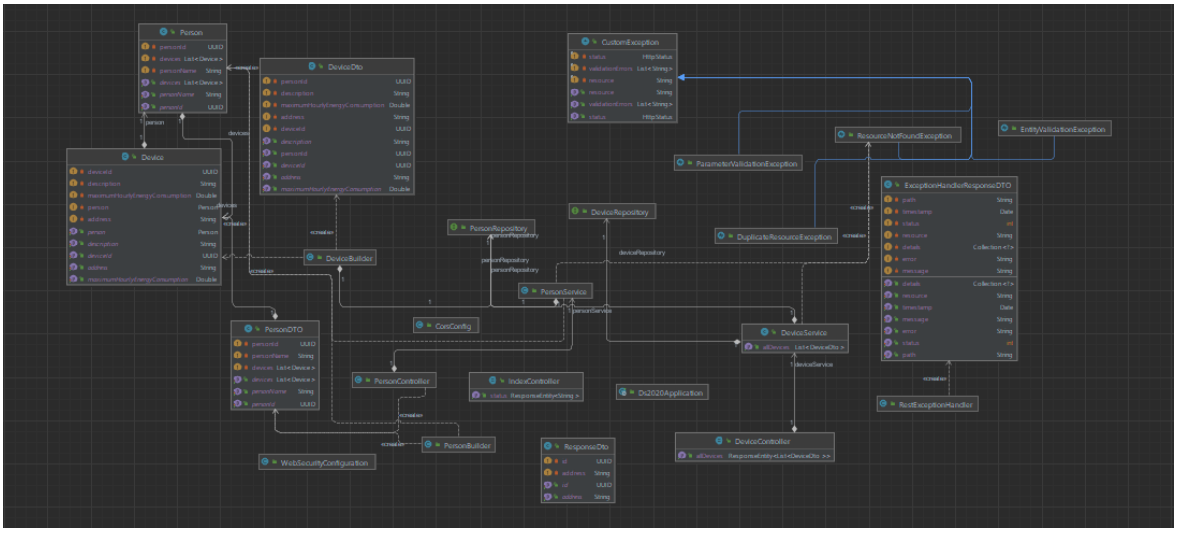
This application effectively demonstrates the benefits of a microservices architecture for managing users and devices, where each module can evolve independently. Future developments could include:

- **Enhanced Security**: Implementing advanced authentication and authorization measures, such as OAuth for user access control and security.
- **Logging and Monitoring**: Adding centralized logging and monitoring to detect performance bottlenecks or faults in specific services, improving maintainability and user experience.
- **Horizontal Scaling**: Expanding individual services to support larger user and device databases through load balancing and service replication.
- **Integration with Additional Services**: Extending functionality with analytics or IoT integrations to monitor device health and usage patterns in real time.

7. User Backend UML diagram



8. Device Backend UML diagram



9. Docker Configurations :

Docker Configuration Overview

1. *Docker Compose*: Manages the setup of multiple services and networking in a single configuration file. Each component in the application—frontend, user backend, device backend, and databases—has its own service in the Docker Compose file, which simplifies configuration and deployment.

2. *Dockerfiles*: Each backend and frontend component has a specific Dockerfile detailing the instructions to build its Docker image.

Network Configuration:

The my_custom_network network enables communication between services without exposing internal ports to the external network, making the setup secure. Each service refers to other services (like db for databases) within the Docker network, ensuring seamless internal connectivity.

Network Subnet: 172.18.0.0/24 with Gateway: 172.18.0.1

Summary of Ports Mapping

Service	Internal Port	Host Port
`db-user`	3306	3307
`db-device`	3306	3308
`user-backend`	8080	8080
`device-backend`	8081	8081
`frontend`	3000	3000