

Assignment 1: Energy Management System

Distributed Systems 2024

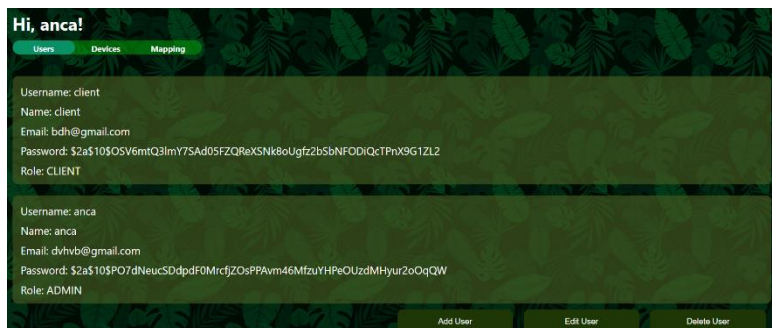
Giurgiu Anca-Maria

Conceptual Architecture

The energy management system is designed with a microservices architecture, comprising two distinct microservices—User and Device—as well as a frontend built with React. Communication between components relies on RESTful APIs, ensuring a decoupled, modular system. Each microservice addresses a specific business domain and operates independently. The React-based frontend provides the user interface, managing user interactions, while the backend enforces business rules and data handling.

The system supports two user roles with different permissions:

- **Admin:** Full permissions to create, update, view, and delete users and devices.



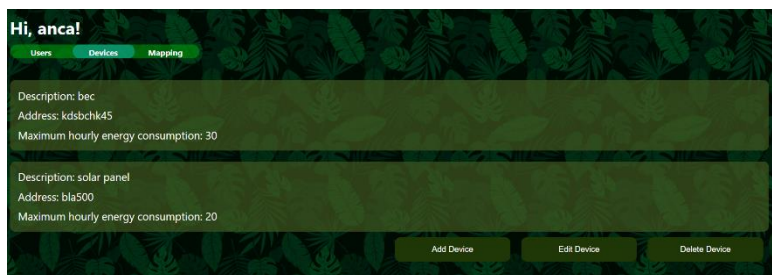
Hi, anca!

Users Devices Mapping

Username: client
Name: client
Email: bdh@gmail.com
Password: \$2a\$10\$OSV6mtQ3lmY7SA05FZQReXSNk8oUgfz2b5bNFODIQcTPhX9G1ZL2
Role: CLIENT

Username: anca
Name: anca
Email: dvhvb@gmail.com
Password: \$2a\$10\$PO7dNeucSDdpdF0MrcfZOsPPAvn46MfzuYHFeOUzdMHyr2oQqQIW
Role: ADMIN

Add User Edit User Delete User



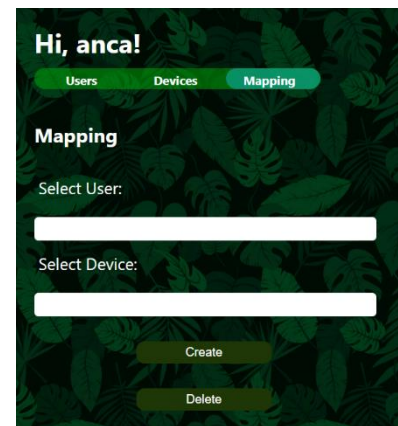
Hi, anca!

Users Devices Mapping

Description: bec
Address: kdsbchk45
Maximum hourly energy consumption: 30

Description: solar panel
Address: bla500
Maximum hourly energy consumption: 20

Add Device Edit Device Delete Device



Hi, anca!

Users Devices Mapping

Mapping

Select User:

Select Device:

Create Delete

- **User:** Restricted to viewing devices assigned to them, enhancing both data security and personalized access.



Energy Management

Hi, client!

Your devices:

Description: bec
Address: kdsbchk45
Maximum hourly energy consumption: 30

Microservices Architecture

The backend is divided into two independent Spring Boot microservices, each responsible for managing specific types of data and functionalities:

User Microservice

Manages all user-related information and actions, offering CRUD operations for user management and handling authentication.

- **Controller Layer:**
Provides endpoints for user management (/user), login (/auth/sign_in) and register (/auth/sign_up).
- **Service Layer:**
The UserService class provides core user management functionalities within the application. It facilitates operations like deleting a user by username, finding users by ID, retrieving all users, and updating user details. This service interacts with the UserRepository to perform database operations and employs custom exceptions to handle cases where a user is not found.
- **Persistence Layer:**
The UserRepository interfaces with the database, executing CRUD operations for user data storage and retrieval.

Device Microservice

Handles all device-related data, with capabilities to create, update, view, and delete device records.

- **Controller Layer:**
Exposes API endpoints for device management (/device).
- **Service Layer:**
The DeviceService class implements business logic for managing devices, such as creation, updates, and deletion.
- **Persistence Layer:**
The DeviceRepository communicates with the database for CRUD operations, ensuring devices are managed and stored correctly.

Each microservice operates as an autonomous Spring Boot application, promoting modularity and scalability, with independent deployment capabilities.

Frontend (React Application)

The frontend application is structured using React components and services, with the following main elements:

1. App Component:

- The App component sets up the main routing structure using react-router-dom.
- Depending on the user's role (CLIENT or ADMIN), the user is directed to either the Client or Admin page.
- The handleLogin function is responsible for setting the user state after a successful login and saving the user data to a cookie.

2. LoginPage Component:

- The LoginPage component handles user login, taking the username and password as inputs.
- It uses the login function from user_service to authenticate, and on success, calls onLogin (from the App component) to save the user state.
- It redirects to either the Client or Admin page based on the role.

3. Admin Component:

- The Admin component has a complex UI to manage users, devices, and mappings.
- It supports CRUD operations on users and devices, displaying data using components like User, Device, and modals (ModalAddUser, ModalEditUser, ModalAddDevice, ModalEditDevice).
- The component utilizes getUsers, getDevices, deleteUser, and deleteDevice from the services to fetch and modify backend data.
- It uses radio buttons for toggling views between users, devices, and mapping.
- Functions like addUserToList, updateUserInList, addDeviceToList, and updateDeviceInList are used to update the local state for instant UI refreshes after changes.

4. Client Component:

- The Client component shows devices associated with a logged-in client.
- It fetches devices using getDevices from client_service and displays them using the DeviceClient component.

5. Mapping Component:

- The Mapping component allows the admin to map or unmap devices to users.

- It uses createMapping and deleteMapping from mapping_service to handle these actions.
- The user ID is obtained using getUserId from user_service.

6. Service Layer:

- Services like device_service, user_service, client_service, login_service and mapping_service organize API requests and are shared across components.
- Example services include getDevices, deleteUser, and createMapping.
- Each service function performs API calls to the backend and returns the response data in JSON format.
- Login_service: Cookie handling functions, setCookie and getCookie, are used for storing and retrieving user data after login.

Data Flow and Communication Protocols

Communication between the frontend and backend is accomplished through HTTP requests, specifically using the Fetch API in JavaScript. The frontend sends asynchronous requests (like GET, POST, PUT, DELETE) to the backend's REST API endpoints, which are defined in the backend microservices, as follows:

User Microservice Endpoints

DELETE /user/delete: Deletes a user by their username.

GET /user/find: Retrieves a user by their ID, returning the user data if found.

GET /user/findAll: Fetches all users within the system.

GET /user/findId: Retrieves a user's ID based on their username.

PUT /user/update: Updates the details of an existing user based on provided data.

POST /auth/sign_in: Authenticates a user based on their username and password. If successful, it generates a JWT token and returns user details (including the token) in a JwtResponse. The token is used for subsequent requests, enabling stateless authentication.

POST /auth/sign_up: Registers a new user. Checks if the username or email is already in use, and if not, creates a new user account based on the SignupRequest details.

Device Microservice Endpoints

POST /device/create/mapping: Maps a device to a user based on information in the MappingDto object.

PATCH /device/delete/mapping: Removes an existing user-device mapping, as specified by the MappingDto.

GET /device/findAll/mapping: Retrieves all devices assigned to a specific user by their userId.

POST /device/create: Adds a new device to the system, returning an error if the device description is already in use.

PUT /device/update: Updates details of an existing device, using the information from DeviceDetailsDto.

DELETE /device/delete: Deletes a device based on the provided details in DeviceDetailsDto.

GET /device/find: Finds a device by its ID, returning the device details if found.

GET /device/findAll: Retrieves all devices in the system.

Each microservice communicates only with its respective database, maintaining data isolation. This approach prevents direct database interaction from the frontend, ensuring a separation of concerns.

Security Considerations

The UserManagement security configuration enforces several critical security measures, essential for a secure user management system:

1. **JWT-based Stateless Authentication:** The application employs JWT (JSON Web Token) for stateless authentication, eliminating the need for server-side session management. The token is signed with a secure key using HMAC SHA-512, ensuring that only valid, unaltered tokens are accepted. The JwtUtilsService is used for token generation and validation, protecting sensitive user information in transit.
2. **Role-Based Authorization:** Only authenticated users can access /user/** endpoints, while /auth/** endpoints remain open for registration and login. This ensures secure access control.
3. **Password Encryption:** User passwords are encrypted using BCryptPasswordEncoder, enhancing password security by storing them as hashed values rather than plain text.
4. **Cross-Origin Resource Sharing (CORS):** CORS is configured to allow only specific origins (like http://localhost:3000), which mitigates the risk of cross-origin attacks by controlling who can access the application resources.
5. **Error Logging for Token Validation:** The JwtUtilsService logs specific JWT validation errors, such as malformed or expired tokens, which helps with monitoring and auditing security events.

UML Deployment Diagram

