Assignment Documentation 1

1. Detailed Conceptual Architecture of the Distributed System

The system in question is "Kaanergize," an energy management web application designed through a microservices architecture. This architecture is good for modularity and scalability, where the application to be seperated into different services that can be developed, deployed, and maintained independently. This design improves maintainability, allowing individual microservices to change and adapt without disrupting the entire system.

Backend Architecture

The backend architecture of Kaanergize is structured into several connected layers, with a specific role for correct operation. At the core of this architecture is the API layer, which serves as the interface between the application and users. This layer is responsible for handling incoming HTTP requests, exposing endpoints for various functionalities, including user authentication, device management, and data retrieval. Its essential in request validation, authorization, and response formatting, it also makes sure only valid requests are processed and appropriate responses are returned to users.

For security purposes, the API layer uses JSON Web Tokens (JWT) for authentication purposes. JWT is a compact means of representing claims to be transferred between two parties. It allows users to authenticate and gain access to protected resources by including a JWT in their request headers. Upon successful authentication, users receive a JWT, which they include in the headers of requests to access restricted endpoints.

Above the API layer is the service layer, which contains the application-specific logic for workflows across different services. This layer acts as a bridge between the API layer and the domain layer, managing interactions between services and coordinating tasks such as initiating transactions and enforcing business rules. For example, the service layer plays an important role in managing interactions between the User Management Service (UserMS) and the Device Management Service (DeviceMS). It ensures that requests for user creation and deletion flow smoothly from UserMS to DeviceMS, maintaining consistency throughout the application.

The domain layer contais the core business entities of the app and the logic governing their behavior. This layer houses the main data structures relevant to the application, such as user and device entities, and implements the rules that dictate how these entities interact. Additionally, the data access layer abstracts interactions with the database, facilitating the implementation of repositories that handle operations for specific entities. This layer provides a clean and organized method to perform CRUD (Create, Read, Update, Delete) operations on the data, ensuring efficient data management and retrieval.

Frontend Architecture

On the frontend, Kaanergize is developed using Angular, a framework that follows a component-based architecture for modularity and reusability. This approach allows to create encapsulated components that can be easily reused across different parts of the application. The main structure of the Angular application is within the app folder, which contains various subfolders for components, services, constants, and guards.

Within the components folder, each UI component is designed to handle specific functionality. The device management components, including Device Details, Device List, and User Device List, make user interactions with the devices in the system easier, allowing users to view detailed information, edit device settings, or delete devices as needed.

In the constants folder, shared constants across the application are defined to ensure that there is a single source of truth for static configurations. This is is important for maintaining consistency throughout the application. TypeScript enums are utilized for type safety, particularly for roles. Guards are implemented to manage route access based on the authentication status of users. For instance, AuthGuard and LoginGuard restrict access to certain routes, ensuring that only authenticated users can reach sensitive parts of the application, like user settings or device management features.

The frontend also uses Angular’s routing capabilities for navigation between different views and components. This routing configuration defines how users can navigate through the application, with guards protecting routes that require authentication. Services within the Angular application handle HTTP requests and encapsulate business logic, allowing components to focus on rendering and user interaction. This separation of concerns improves the overall structure of the application and makes it easier to maintain.

Communication Between Microservices

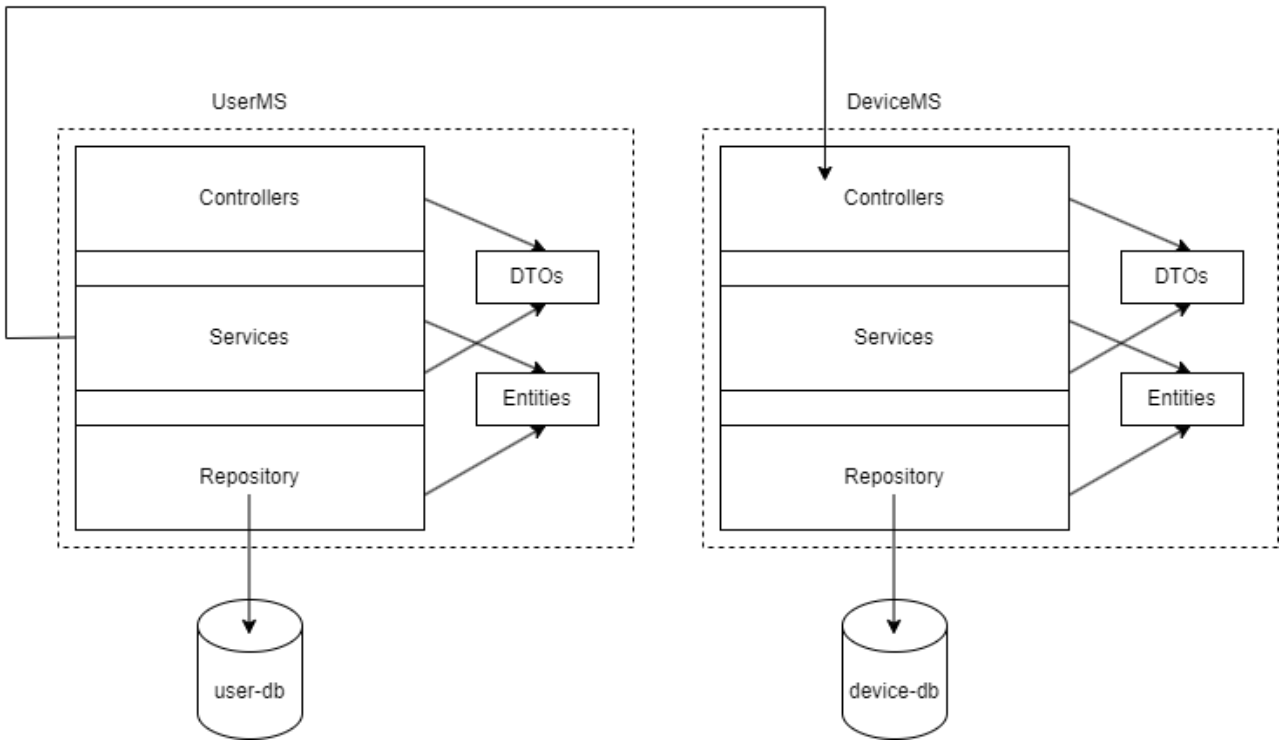
The UserMS is responsible for CRUD operations on user data, covering attributes such as user ID, name, email, and role (admin or client). It implements authentication mechanisms to control access to features based on user roles, ensuring that only authorized users can perform certain actions. For example, administrators have the ability to manage all users and devices, while regular clients have limited permissions primarily focused on viewing their devices.

In the same way, the DeviceMS manages CRUD operations on devices, characterized by attributes like device ID, description, address, and maximum hourly energy consumption. This service is responsible for tracking device details, managing updates, and ensuring that devices are correctly associated with users. When a new user is created, the UserMS handles this operation through an API call that initiates a transaction, ensuring that both user and device records are managed consistently. This transaction approach allows for cascading actions, such as associating new devices with the newly created user.

Role-based access control (RBAC) is a of the application, where administrators are granted full CRUD permissions for managing both users and devices, while clients have limited permissions, primarily focused on viewing devices.

Dockerization and Deployment

The application uses Docker to streamline the deployment and management of its various components. Each microservice is encapsulated in its own Docker container.
To manage the orchestration of these microservices, a Docker Compose file is utilized. This configuration defines how the various services, such as user and device databases, the UserMS, DeviceMS, and the frontend application, should be deployed and interact with each other. It establishes a bridge network for inter-service communication, allowing services to discover and communicate seamlessly.
Persistent volumes are also defined to retain data across container restarts, making sure that user and device information is preserved.



2. Deployment Diagram