UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

Assignment 3: Energy Management System

Distributed Systems 2024

Giurgiu Anca-Maria

Conceptual Architecture

The energy management system is designed with a microservices architecture, comprising three distinct microservices—User, Device, Monitoring and Chat—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.

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, and utilizes a 2-second interval for simulation instead of the usual 10 minutes to accommodate presentation requirements.

- **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.

Monitoring and Communication Microservice

Handles real-time energy consumption data by listening to device updates and saving hourly energy consumption records in a database. Messages are processed in batches of 6, representing theoretical intervals of 10 minutes per message, to calculate and store hourly consumption accurately. Alerts are triggered when the average hourly consumption exceeds a predefined limit, ensuring proactive notification and efficient energy monitoring.

- **Controller Layer:**
  Exposes API endpoints for device management (/monitoring).

- **Service Layer:**
  The MonitoringService is responsible for processing device data from a RabbitMQ message broker, managing energy consumption records, and sending alerts when certain thresholds are exceeded.

  The NotificationService handles sending notifications to clients via WebSocket using Spring's SimpMessagingTemplate.

- **Persistence Layer:**
  The MonitoringRepository communicates with the database for CRUD operations, ensuring messages are managed and stored correctly.

Chat Microservice

Enables real-time communication between users and administrators within the Energy Management System. It allows users to send messages asynchronously to administrators, who can respond in a real-time chat session. The service is designed to handle multiple concurrent chats, ensuring that users can communicate efficiently with administrators and receive instant notifications about message status and typing activity.

- **Controller Layer:**
  Exposes API endpoints to handle chat-related actions (/chat).

- **Service Layer:**
  The ChatService is responsible for managing the logic of the chat communication process. It listens for incoming messages, processes them, and handles communication between users and administrators.

  The NotificationService manages the sending of real-time notifications (e.g., message read, typing status) to both users and administrators using WebSocket communication.

- **Persistence Layer:**
  The ChatRepository communicates with the database for CRUD operations, ensuring messages 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**:
    - Displays devices fetched using the getDevices function from the client_service.
    - Devices are displayed using the Device component, which shows the device's description, address, and maximum hourly consumption.
    - Users can select multiple devices, tracked by the selectedDevices state.

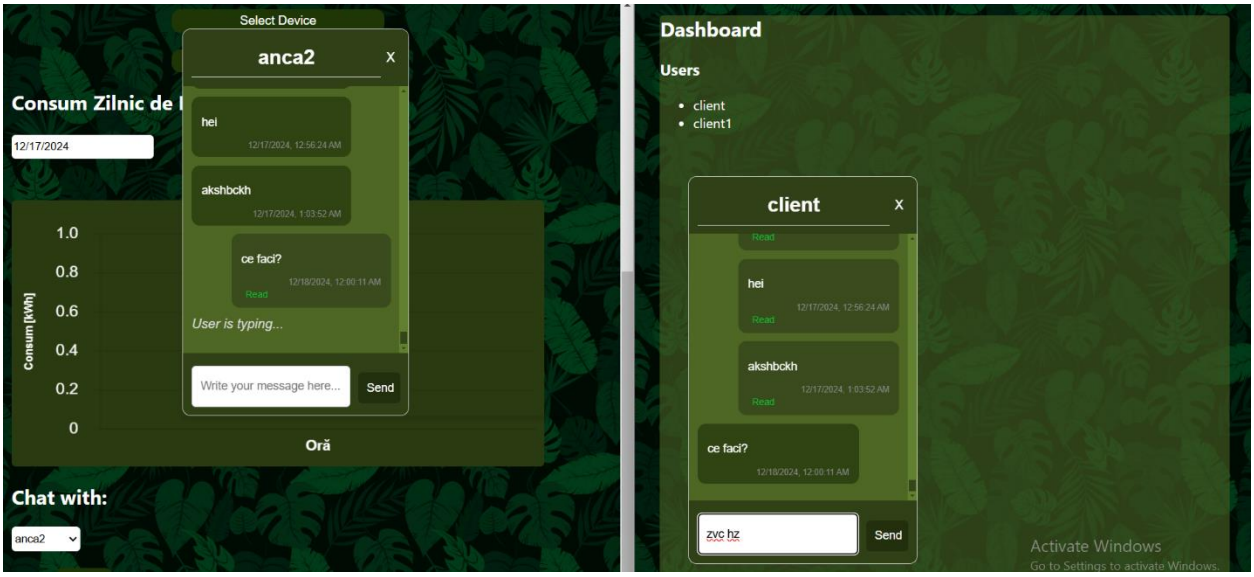
- Users can choose a simulation from a dropdown (Simulare 1 or Simulare 2) using handleDeviceChange.
- Once a device is selected, the findSelectDevice function assigns the selected simulation and sets up device-specific configurations (e.g., max consumption).
- A simulation can be started using the startSimulator function, which interacts with the monitoring_service to initiate the process.

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.
7. **EnergyConsumptionChart Component:**
   - Users can pick a date using a DatePicker component to update the chart data dynamically.
   - Fetches and processes energy consumption data for the selected date and device.
   - Displays hourly energy consumption in a line chart using Chart.js.
8. **WebSocketNotifications Component:**
   - Establishes a WebSocket connection using SockJS and Stomp.
   - Connects to the /topic/notifications/{deviceId} endpoint based on the selected device.
   - Receives messages in real time for the selected device.
   - Extracts and validates messages containing the device ID.
   - Tracks the frequency of each unique message and displays a badge for repeated messages.
   - Updates the message list dynamically as new messages are received.

9. **Dashbord Component:**
   - Fetches the list of users for the admin using the getUsersForAdmin service.
   - Displays a list of users and allows the admin to open a chat with any user by clicking on their name.

- Manages multiple chat sessions at once by dynamically adding and removing chat boxes.

10. **ClientChat Component:**
    - Allows the client to select an admin from a dropdown list and start a chat by clicking the "Start to chat" button.
    - Ensures that the client selects an admin before starting the chat. If no admin is selected, an alert is shown.
    - Displays the chat box once a valid admin is selected, and the chat session can proceed.
11. **ChatBox Component:**
    - Establishes a WebSocket connection using SockJS and Stomp to handle real-time message delivery and notifications.
    - Listens for messages from the WebSocket server and updates the messages state accordingly, ensuring that new messages are displayed instantly.
    - Supports typing indicators for both users. When one user types, a "User is typing..." message is displayed.
    - Automatically scrolls the chat to the latest message when a new message is received.
    - Tracks read/unread messages and updates the UI to show when a message has been read by the recipient.
    - Handles marking messages as read and sending typing notifications through the WebSocket connection.



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.

Monitoring and Communication Microservice Endpoints

GET /monitoring/chart: Retrieves the hourly energy consumption for a specified device on a given day.

POST /monitoring/maxConsumption: Updates the maximum allowable hourly consumption threshold.

Chat Microservice Endpoints

POST /chat/create: Creates a new chat message and sends it to the recipient.

POST /chat/read: Marks all messages as read between two users, sending a reading notification.

GET /chat/messagesBetweenUsers: Retrieves the chat messages between two specified users.

GET /chat/usersForAdmin: Retrieves a list of users that an admin can chat with.

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.

Read and Typing Functionality

Typing:

The onKeyPress event in the input field triggers the handleTyping function when the user starts typing. The handleTyping function checks if the user is already typing (isTyping state). Sends a "typing" notification to the server via the WebSocket (stompClient.send). Sets a timeout of 2 seconds to reset the isTyping state. The WebSocket subscribes to /topic/notifications /{currentUser.id}. If a message of type TYPING is received, the setUserTyping state is set to true. After 3 seconds, the typing indicator is hidden by resetting setUserTyping to false. The /typing/{recipientId} endpoint listens for typing notifications. Calls sendTypingNotification to notify the recipient. Sends a WebSocket message to /topic/notifications/{recipientId}.

Read:

A useEffect hook checks if the chat has any unread messages and sends a "read" notification to the backend when the user clicks on the component. The WebSocket (stompClient) sends messages to /app/read/{senderId}. The WebSocket subscribes to /topic/notifications/read/{currentUser.id} /{user.id}. If the received message is "read", the messages state is updated to mark the relevant messages as read. A POST request to the /read endpoint processes the "read" notifications. Identifies the senderId and recipientId from the request. Sends a WebSocket message to /topic/notifications/read/{recipientId}/{senderId}. Notifies all subscribers that the messages have been marked as read.

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