Energy Management System (EMS) UI

Introduction

The EMS UI is an interactive platform designed to streamline energy monitoring and management processes. This documentation provides an in-depth guide to understanding the structure, functionality, and implementation details of the EMS UI.

Technology Stack

- **Frontend:** React.js (CRA)
- **State Management:** React Context
- **UI Framework:** Material-UI
- **API Calls:** Axios
- **Testing:** Jest, React Testing Library
- **Containerization:** Docker

Installation Setup and Run

Follow these steps to run the project locally:

1. Clone the project:

git clone https://github.com/Alti-HW/EnergyManagementSystem-UI.git

2. Navigate to the project directory:

cd EnergyManagementSystem-UI

3. Install dependencies:

npm install

4. Start the server:

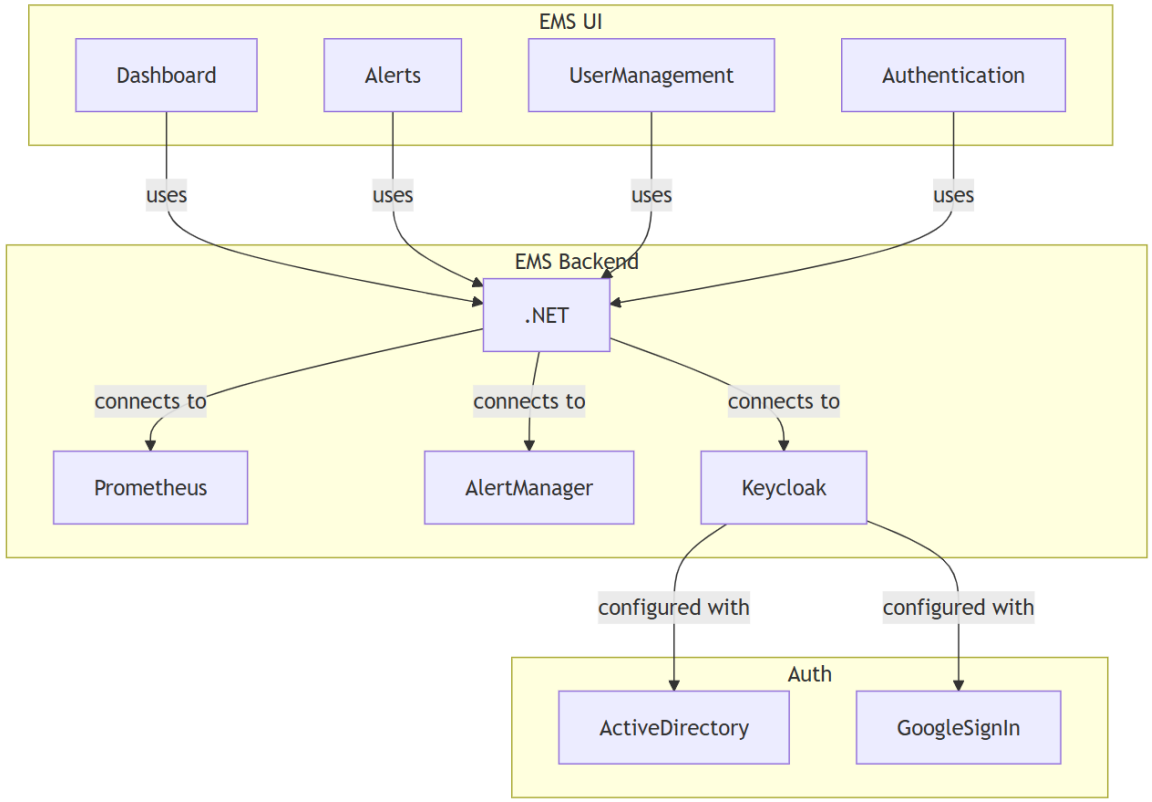
npm run start

Project Structure

```
EnergyManagementSystem-UI/
├── .gitignore
├── README.md
├── package-lock.json
├── package.json
├── public/
│   ├── favicon.ico
│   ├── index.html
│   └── manifest.json
├── src/
│   ├── App.css
│   ├── App.tsx
│   ├── authorization/
│   │   ├── FeatureAccessControl.tsx
│   │   └── user.access.constants.ts
│   ├── components/
│   │   ├── auth/
│   │   │   ├── Login.tsx
│   │   │   └── PrivateRoute.tsx
│   │   ├── dashboard/
│   │   │   ├── CustomDialog.tsx
│   │   │   ├── Dashboard.css
│   │   │   ├── Dashboard.tsx
│   │   │   ├── components/
│   │   │   │   ├── Alarms.tsx
│   │   │   │   ├── ChartHeading.tsx
│   │   │   │   └── ChartWrapper.tsx
│   │   └── user-management/
│   │       ├── AddUserModal.tsx
│   │       ├── DeleteUserModal.tsx
│   │       ├── EditUserModal.tsx
```

```
|   |       └── UserTable.tsx
|   ├── configs/
|   |   └── energyManagement.json
|   ├── constants/
|   |   └── routes.ts
|   ├── context/
|   |   └── AuthContext.tsx
|   ├── react-app-env.d.ts
|   ├── reportWebVitals.ts
|   ├── setupTests.ts
|   └── types/
|       └── dashboard.types.ts
├── tsconfig.json
└── yarn.lock
```

Project Architecture



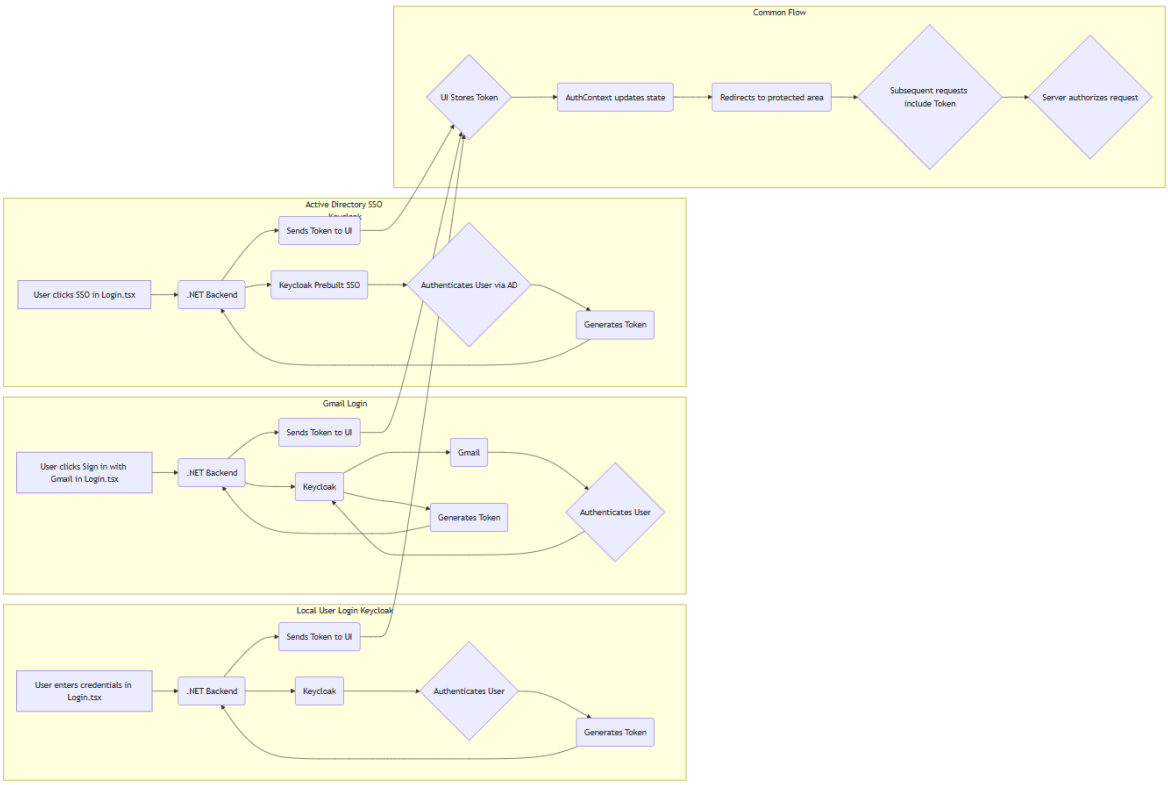
Pages

- **Dashboard:** Displays energy consumption metrics, including real-time data and historical trends.
- **Alerts:** Sends notifications of anomalies or critical energy usage events.

- **User Management:** Enables administrators to manage users, assign roles, and control access.

Authentication

This section describes the authentication flow for the Energy Management System, covering login and logout processes, including standard username/password login, social media login (Google), and Active Directory Single Sign-On (SSO).



Functionality

- **Login:** Users can log in to the system using several methods:
    - **Standard Credentials:** Username/email and password. Successful login establishes a user session.
    - **Social Media (Google):** Users can authenticate using their Google accounts.
    - **Active Directory SSO:** Users can authenticate using their existing Active Directory credentials via Single Sign-On (SSO).
- **Logout:** Users can log out of the system, terminating their session and requiring them to re-authenticate to access protected resources.
- **Session Management:** The system manages user sessions, likely using cookies or local storage to maintain authentication state.
- **Authentication State Persistence:** The authentication state (e.g., whether a user is logged in) persisted across page reloads.

- **Protected Routes:** The system protects certain routes and resources, requiring users to be authenticated before accessing them.
- **Token Management:** After successful login, the server returns a token (JWT) that is stored in the client side and sent with every request to the server.

Components

- Login (src/components/auth/Login.tsx): The login form where users can choose their authentication method. This component likely contains buttons or links for Google and potentially handles the initiation of the Active Directory SSO flow.
- AuthContext (src/context/AuthContext.tsx): A context that provides authentication state and functions to components.
- PrivateRoute (src/components/auth/PrivateRoute.tsx): A component that protects routes, redirecting unauthenticated users to the login page.

Data Flow

- **Login Request:** The user initiates the login process:
- **Standard Credentials:** The user enters their credentials in the Login component (src/components/auth/Login.tsx) and submit the form.
- **Social media (Google):** The user clicks the "Sign in with Google" button in the Login component. This likely redirects the user to Google's authentication page.
- **Active Directory SSO:** The user is automatically redirected to the Active Directory authentication page (if they are not already authenticated) or seamlessly authenticated in the background.
- **Authentication API Call:**
- **Standard Credentials:** The Login component sends an authentication request to the server (likely /api/login or similar). The specific API endpoint is likely defined within the Login component.
- **Social media (Google):** After successful authentication with Google, Google redirects the user back to the application with an authorization code or token. The Login component then sends this code or token to the server (likely /api/login/google or similar).
- **Active Directory SSO:** The application receives a token or assertion from the Active Directory Identity Provider (IdP). This token is then sent to the server for validation (likely /api/login/ad or similar).
- **Server-Side Authentication:** The server verifies the user's identity:
- **Standard Credentials:** The server verifies the user's credentials against the database.
- **Social media (Google):** The server verifies the Google authorization code or token with Google's API. It then retrieves the user's profile information from Google and either creates a new user account in the system or links the Google account to an existing user account.

- **Active Directory SSO:** The server validates the token or assertion received from the Active Directory IdP. It then retrieves the user's information from the token and either creates a new user account in the system or links the Active Directory account to an existing user account.
- **Session Creation/Token Generation:** If the authentication is successful, the server creates a session or generates a JSON Web Token (JWT) and sends it back to the client.
- **Client-Side Storage:** The client stores the session identifier (e.g., in a cookie) or the JWT (e.g., in local storage or a cookie). The storage mechanism is likely implemented within the Login component or the AuthContext (src/context/AuthContext.tsx).
- **Authentication State Update:** The AuthContext (src/context/AuthContext.tsx) is updated to reflect the user's authenticated state. This likely involves setting the isAuthenticated state variable to true and storing the user's information.
- **Redirection:** The user is redirected to a protected area of the application (e.g., the dashboard). This redirection is likely handled by the Login component or a router.
- **Subsequent Requests:** For subsequent requests to protected resources, the client includes the session identifier or JWT in the request headers. This is likely handled by an interceptor or a wrapper around the fetch API.
- **Server-Side Authorization:** The server verifies the session identifier or JWT to authorize the request.
- **Logout Request:** When the user clicks the "Logout" button, a request is sent to the server (likely /api/logout or similar). The Logout component or function is not present in the project.
- **Session Termination/Token Invalidation:** The server terminates the user's session or invalidates the JWT.
- **Client-Side Cleanup:** The client removes the session identifier or JWT from storage and updates the AuthContext to reflect the user's unauthenticated state. This is likely handled by the AuthContext (src/context/AuthContext.tsx).
- **Redirection:** The user is redirected to the login page or a public area of the application. This redirection is likely handled by the AuthContext (src/context/AuthContext.tsx) or a router.

State Variables

- isAuthenticated (in AuthContext): A boolean state variable that indicates whether the user is currently authenticated.
- user (in AuthContext): An object that stores the user's information (e.g., username, email, roles).
- Loading state during the login process (likely within the Login component).
- Error state to display login errors (likely within the Login component).

Files

- src/components/auth/Login.tsx: Implements the login form and handles the login process, including Google and potentially Active Directory SSO initiation.
- src/context/AuthContext.tsx: Provides the authentication context and manages authentication state.
- src/components/auth/PrivateRoute.tsx: Protects routes, redirecting unauthenticated users to the login page.
- App.tsx: Uses the AuthContext and PrivateRoute to manage authentication and authorization throughout the application.

Notes

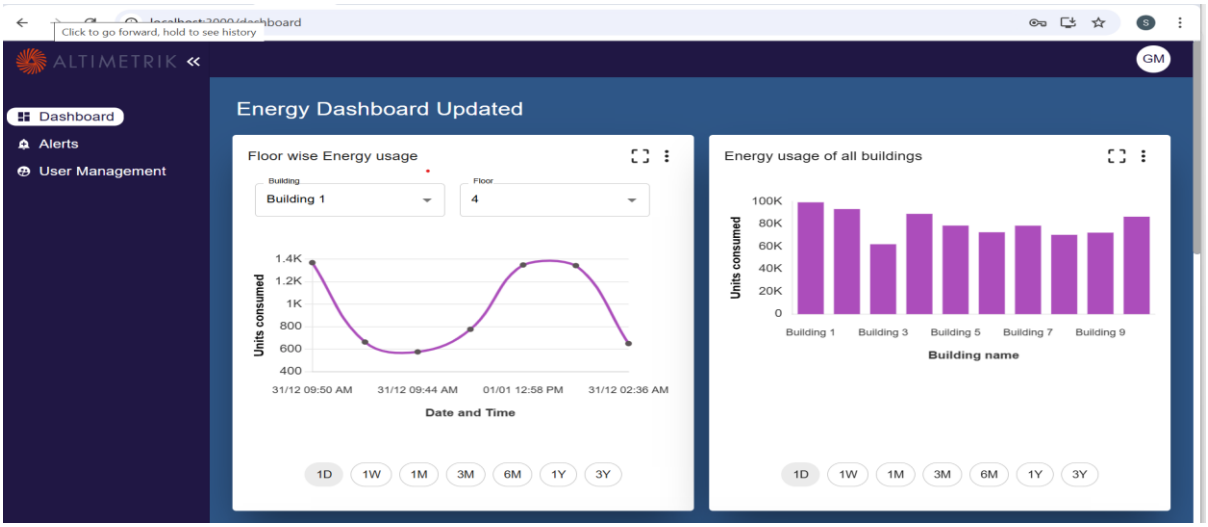
- A dedicated Logout component or function is not explicitly present in the provided file list. Logout functionality might be integrated within another component or handled directly within the AuthContext.
- The specific API endpoints for login and logout, as well as the mechanisms for session management and token storage, would require further investigation of the codebase.
- The presence and implementation of request interceptors for attaching authentication tokens would also require further investigation.
- The exact implementation details for Google and Active Directory SSO would depend on the specific libraries and configurations used in the project.

Dashboard

The Dashboard page (Dashboard) provides a visual overview of energy management data, displaying charts. It allows users to monitor key metrics.

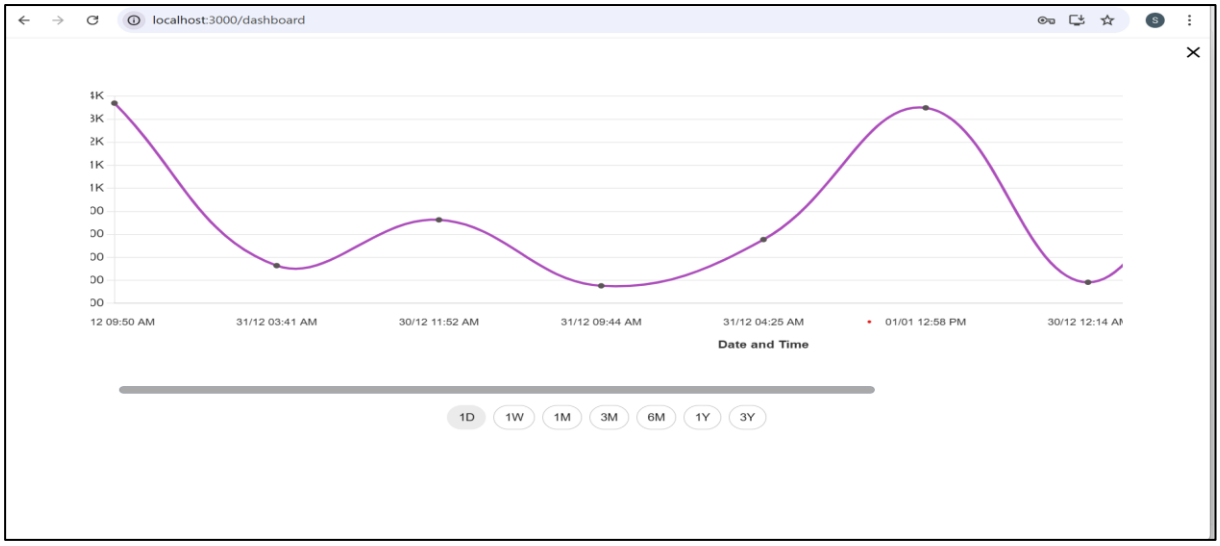
Functionality

- **Data Visualization:** The dashboard displays data using various chart types (e.g., bar, line, doughnut) via the ChartWrapper component.

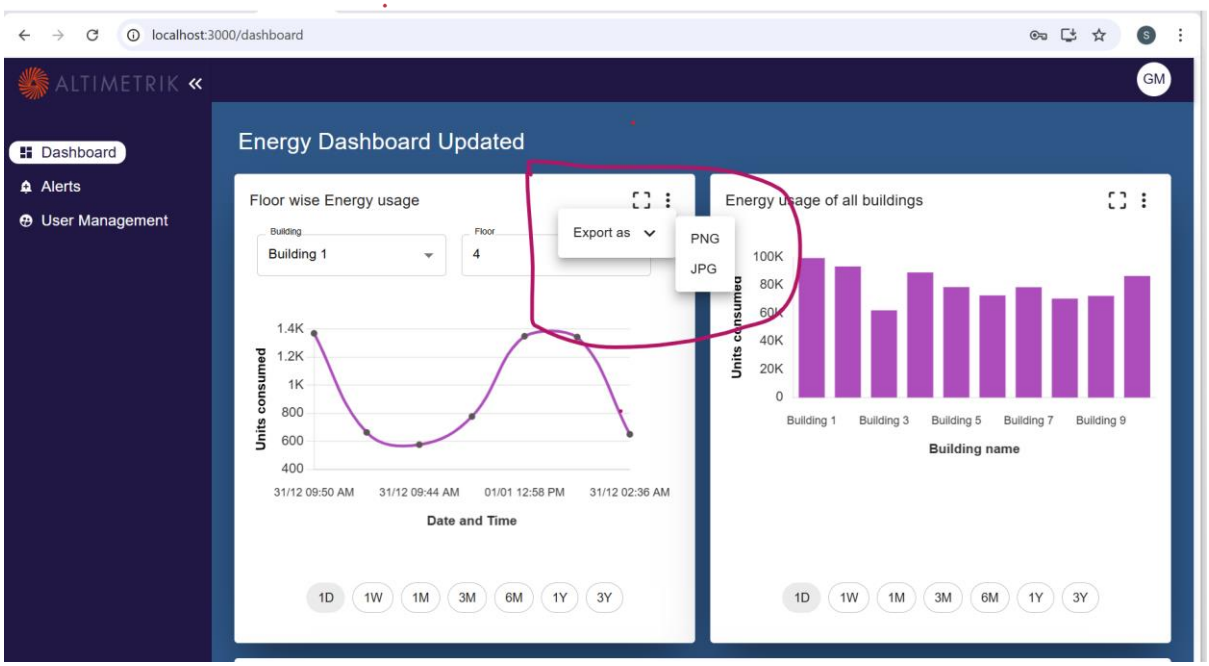


- **Customization:** Users can edit the dashboard layout, adding, removing, and rearranging components.
- **Real-time Monitoring:** The dashboard updates in real-time to reflect the latest data.
- **Full View:** The dashboard components can be viewed in full screen mode.

- **Scrollable charts**: Charts with x-axis as time have the provision to scroll horizontally which updates the chart with next/previous set of data



- **Export Chart data**: Chart configuration has provision to export chart data to PDF, PNG, JPG formats



Components

- Dashboard: The main dashboard component, responsible for rendering the layout and managing the components.
- ChartWrapper: A wrapper component for displaying charts.
- ChartHeading: A reusable heading component for charts, providing title and optional controls (e.g., edit, delete, expand).

Data Flow

1. The Dashboard component receives a defaultConfig prop, which defines the initial dashboard configuration. This configuration includes the dashboard title and an array of components to display.
2. The config state variable stores the current dashboard configuration. The setConfig function is used to update the configuration when the user edits the dashboard.
3. The componentMapper function iterates over the components array in the config and renders the appropriate component based on the type property of each component.
4. The ChartWrapper component displays the data.

Configuration

The dashboard's layout and components are configured via a JSON structure. The structure is described in the README.md file. The key parts of the configuration are:

- title: The title of the dashboard.
- components: An array of components to display on the dashboard. Each component has a type property that specifies the component to render (e.g., "chart").

Routes

The dashboard is accessible via the /dashboard route, as defined in src/constants/routes.ts.

State Variables

- enalbeEditDashboard: A boolean state variable that determines whether the dashboard is in edit mode.
- deleteDashboard: A boolean state variable that determines whether the dashboard is in delete mode.
- config: An object that stores the dashboard configuration.

Files

- src/components/dashboard/Dashboard.tsx: Main dashboard component.
- src/components/dashboard/components/ChartWrapper.tsx: Chart wrapper component.
- src/constants/routes.ts: Defines the routes for the application.
- README.md: Contains the configuration support documentation.

- src/configs/energyManagement.json: Contains the default configuration for the dashboard.

Permissions

Alerts feature is protected by permissions defined in user.access.constants.ts. Key permissions include:

- ` VIEW_DASHBOARD `: Allows view the dashboard
- ` EXPORT_REPORTS `: Allows exporting chart data in configured formats

Dashboard config example


```
{
 "containers": [
  {
   "type": "dashboard",
   "name": "Dashboard",
   "title": "Energy Dashboard Updated",
   "components": [
    {
     "type": "chart",
     "chartType": "line",
     "title": "Floor wise Energy usage",
     "dataApi": [
      {
       "url": "http://localhost:5050/api/Energy/energy-consumption",
       "userFor": "chartData",
       "method": "POST",
       "dataPathKey": [
        "floorConsumptions",
        "floorDetails"
       ],
       "payload": {
        "startDate": "string",
        "endDate": "string",
```

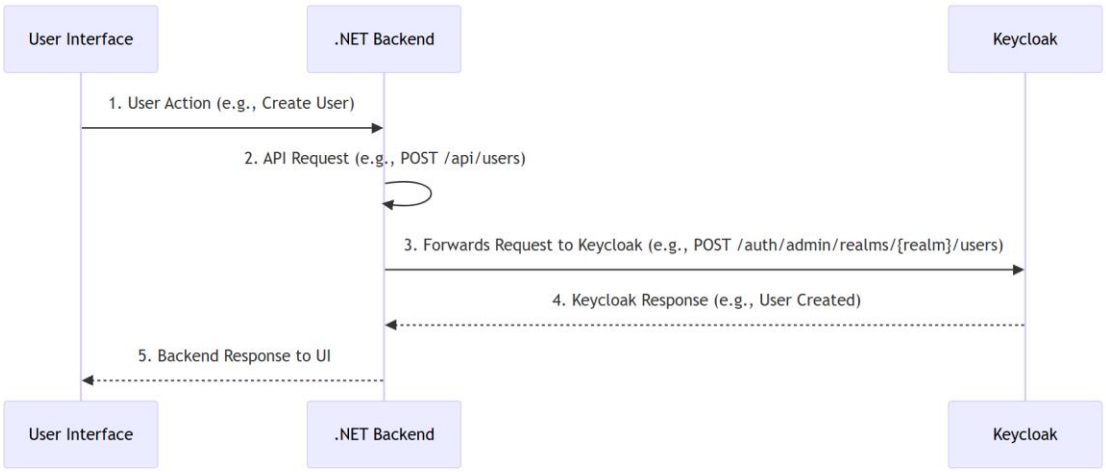
```
      "buildingId": 1,
      "floorId": 1
    }
  },
  {
    "url": "http://localhost:5050/api/Building/GetAllBuildingsWithFloors",
    "userFor": "filters",
    "method": "GET"
  }
],
"xAxisKey": "timestamp",
"xAxisLabel": "Date and Time",
"yAxisKey": "energyConsumedKwh",
"yAxisLabel": "Units consumed",
"yAxisDataFormatter": "number",
"xAxisDataFormatter": "date",
"dateFilter": {
  "filterType": "date",
  "filterValues": [
    "1D",
    "1W",
    "1M",
    "3M",
    "6M",
    "1Y",
    "3Y"
  ],
  "showFilter": "true"
},
"filters": [
  {
```

```
          "filterType": "dropdown",
          "showFilter": "true",
          "filterLabel": "Building",
          "filterDataValueKey": "buildingId",
          "filterDataLabelKey": "name",
          "filterId": "buildingId",
          "dataPath": "",
          "isParentFilter": "true"
        },
        {
          "filterType": "dropdown",
          "showFilter": "true",
          "filterLabel": "Floor",
          "filterDataValueKey": "floorId",
          "filterDataLabelKey": "floorNumber",
          "isItDependentFilter": "true",
          "dependentFilterId": "building",
          "filterId": "floorId",
          "dataPath": "floors"
        }
      ],
      "enableScrollInFullview": "true",
      "color": "#a26cb6",
      "exportOptions": [
        "PNG",
        "JPG"
      ]
    }
  ]
 }
]
```

}

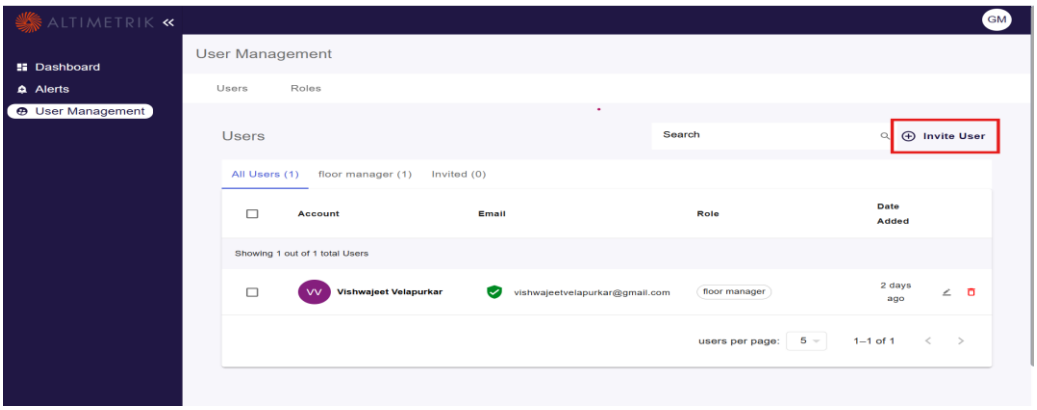
User Management

The User Management section allows administrators to manage user accounts, roles, and permissions within the Energy Management System.



Functionality

- **User Creation**: Administrators can create new user accounts, specifying usernames, passwords, and other relevant details.



- **User Editing**: Administrators can modify existing user accounts, including updating usernames, passwords, roles, and permissions.
- **User Deletion**: Administrators can delete user accounts.
- **Role Management**: Administrators can create, edit, and delete roles, which define sets of permissions.

- **Permission Management**: Administrators can assign permissions to roles, controlling access to various features and data within the system.
- **User Listing and Search**: The system provides a way to list all users and search for specific users based on criteria like username or email.

Role-Based Access Control (RBAC)

RBAC ensures secure access to EMS features based on user roles.

Permissions Constants

tsx
```
const userAccess = {
  ADD_USER: ["Add User"],
  DELETE_USER: ["Delete User"],
  VIEW_DASHBOARD: ["View Dashboard"],
};
export default userAccess;
```

FeatureAccessControl HOC This Higher-Order Component conditionally renders features based on roles:

tsx
```
import { useUser } from "../context/user.context";

const FeatureAccessControl = ({ requiredRoles, children }: any) => {
  const { user } = useUser();

  const hasRequiredRole = requiredRoles?.some((role) =>
    user?.resource_access?.EMS?.roles?.includes(role)
  );

  if (!hasRequiredRole) {
    return null;
  }

  return children;
};

export default FeatureAccessControl;
```

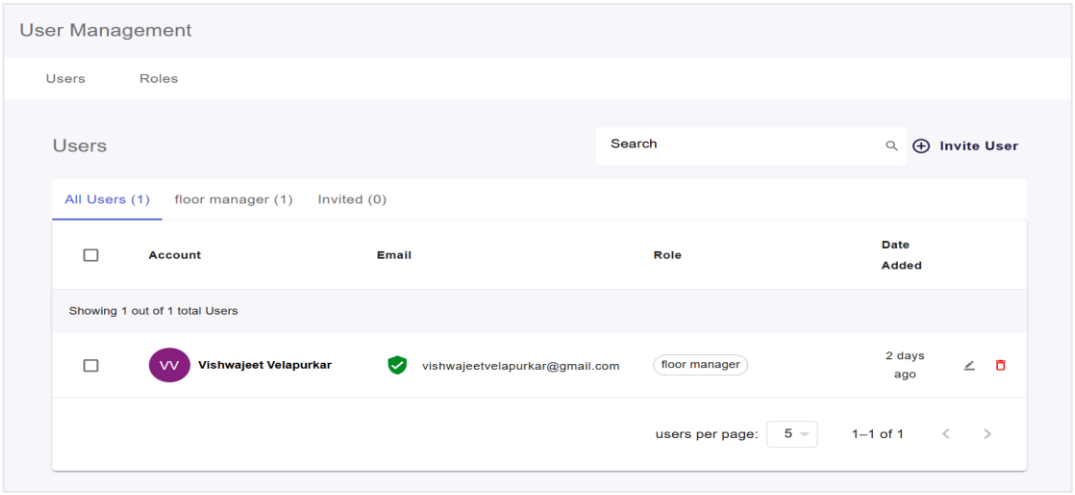
Components

- `UserTable` (src/components/user-management/UserTable.tsx): Displays a table of users with options to edit or delete them.
- `EditUserModal` (src/components/user-management/EditUserModal.tsx): A modal for editing user details.
- `DeleteUserModal` (src/components/user-management/DeleteUserModal.tsx): A modal for confirming user deletion.
- `AddUserModal` (src/components/user-management/AddUserModal.tsx): A modal for adding new users.
- `FeatureAccessControl` (src/authorization/FeatureAccessControl.tsx): Controls access to user management features based on user roles and permissions.

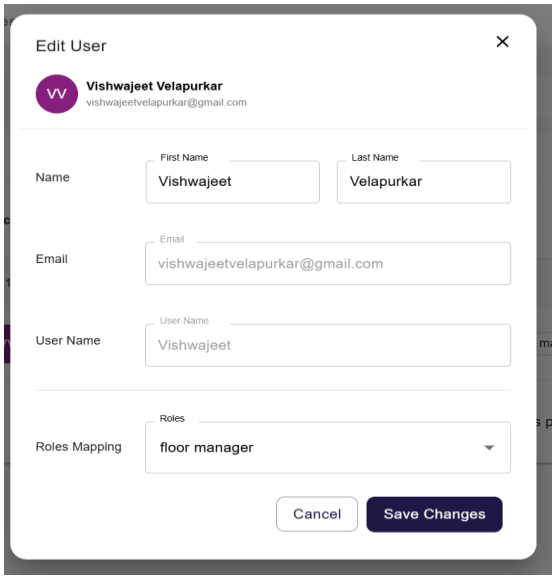
Data Flow

1. Fetching User Data: The `users` component fetches user data from an API endpoint (likely `/api/users` or similar).

2. Displaying User Data: The `UserTable` component renders the user data in a tabular format, including columns for username, email, role, and actions (edit, delete).

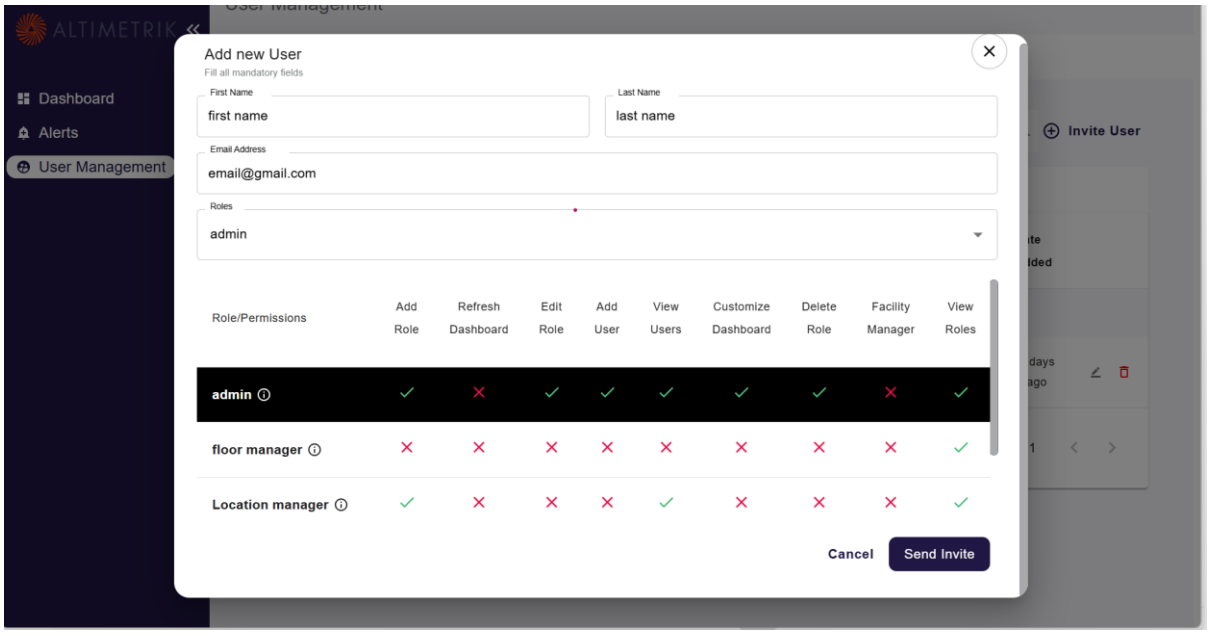


3. Editing User Data: When an administrator clicks the "Edit" button for a user, the `EditUserModal` is displayed, pre-populated with the user's data. The administrator can modify the data and submit the changes. The changes are sent to an API endpoint (likely `/api/users/{userId}`) to update the user in the database.



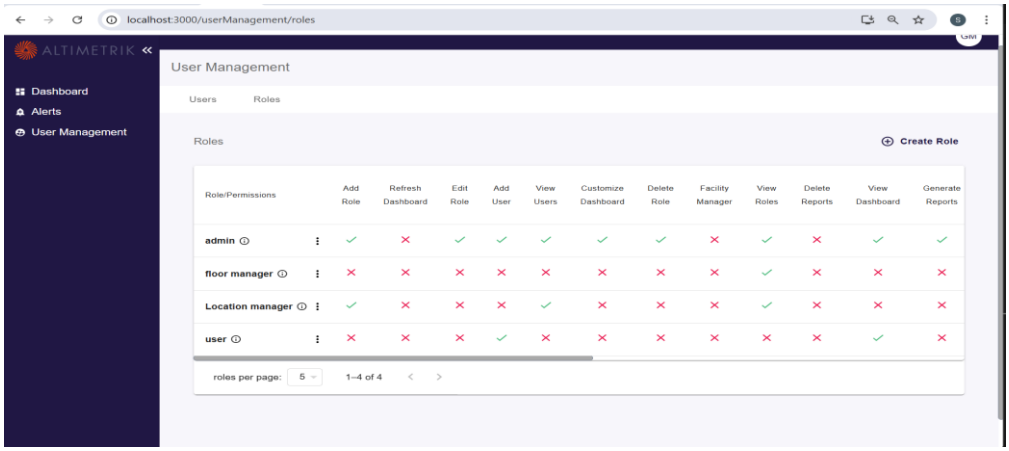
4. Deleting User Data: When an administrator clicks the "Delete" button for a user, the `DeleteUserModal` is displayed, prompting the administrator to confirm the deletion. If confirmed, a request is sent to an API endpoint (likely `/api/users/{userId}`) to delete the user from the database.

5. Adding User Data: When an administrator clicks the "Add User" button, the `AddUserModal` is displayed, allowing the administrator to enter the new user's data. The data is sent to an API endpoint (likely `/api/users`) to create the new user in the database.

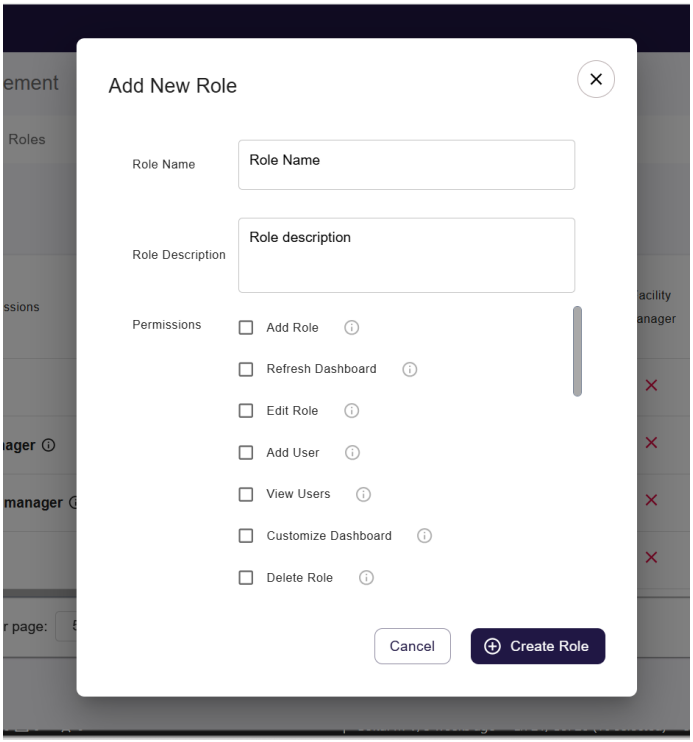


6. Fetching Role & Permissions Data: ‘roles’ component fetches the permissions and roles data and does the mapping among them

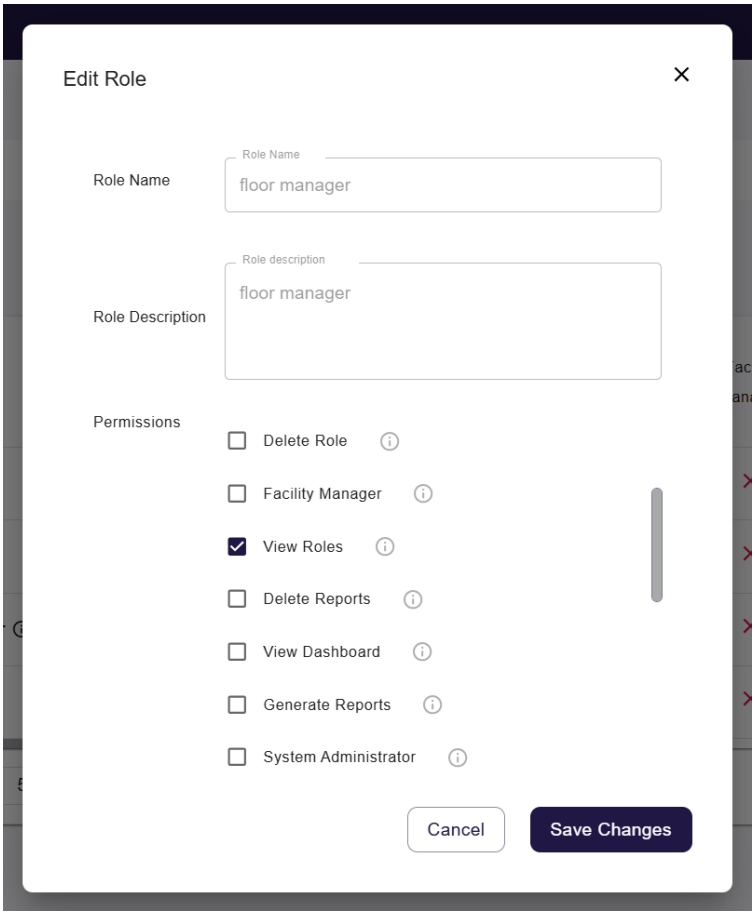
7. Displaying roles: ‘rolesTable’ displays the mapped permissions and roles data in tabular format.



8. Create new role: add_role provides UI for creating new role. And posts the consolidated form data to API end point (` api/roles/create`)



9. Editing existing role: ‘edit_role’ components allow to edit the existing rule and posts the updated data to API end point(‘api/roles/update-rolepermissions`)



10. Access Control: The `FeatureAccessControl` component is used to control access to user management features based on the user's roles and permissions. For example, only users with the `EDIT_USER` permission can edit user accounts.

Permissions

User management features are protected by permissions defined in user.access.constants.ts. Key permissions include:

- `VIEW_USERS`: Allows viewing the list of users.
- `ADD_USER`: Allows creating new users.
- `EDIT_USER`: Allows editing existing users.
- `DELETE_USER`: Allows deleting users.
- `VIEW_ROLES`: Allows viewing the list of roles.
- `ADD_ROLE`: Allows creating new roles.
- `EDIT_ROLE`: Allows editing existing roles.
- `DELETE_ROLE`: Allows deleting roles.

State Variables

State variables within the `UserTable` component manage the list of users, loading state, and error state.

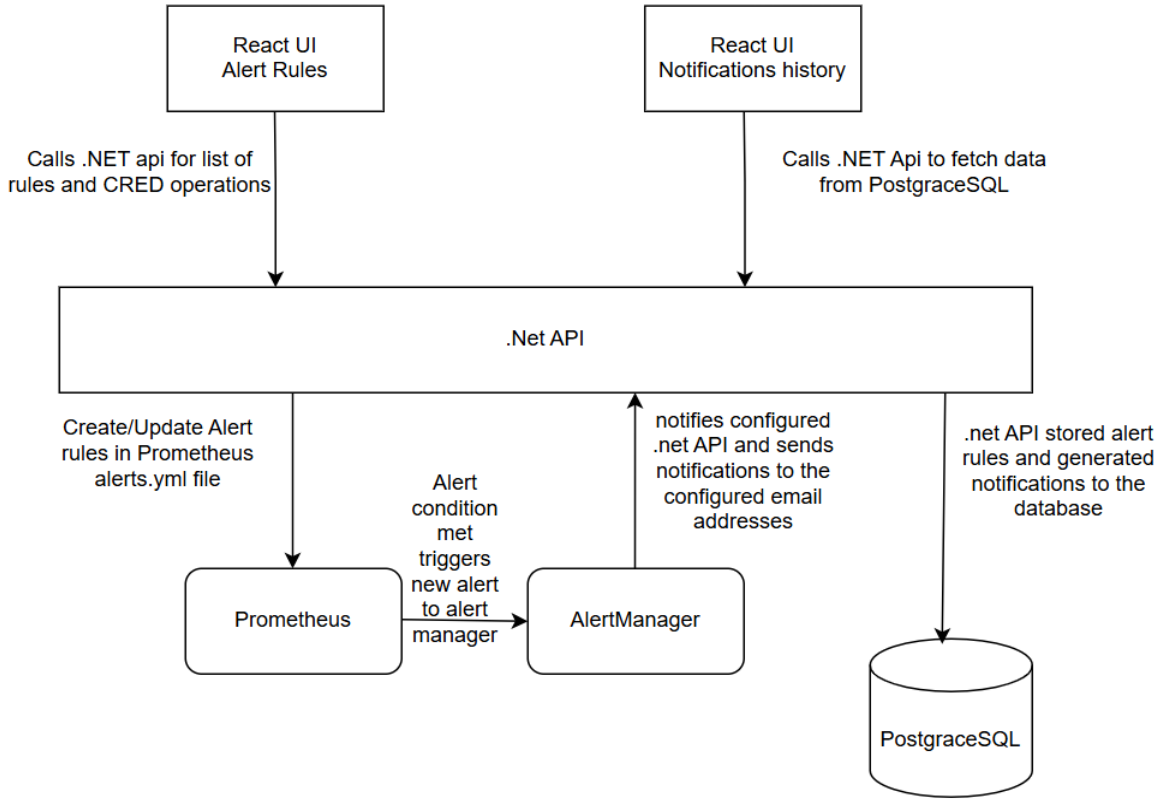
* State variables within the `EditUserModal`, `DeleteUserModal`, and `AddUserModal` components manage the form data, validation errors, and submission state.

Files

- `src/components/user-management/UserTable.tsx`: Displays the user table and handles user listing, editing, and deletion.
- `src/components/user-management/EditUserModal.tsx`: Provides a modal for editing user details.
- `src/components/user-management/DeleteUserModal.tsx`: Provides a modal for confirming user deletion.
- `src/components/user-management/AddUserModal.tsx`: Provides a modal for adding new users.
- `src/authorization/FeatureAccessControl.tsx`: Controls access to user management features based on user roles and permissions.
- user.access.constants.ts: Defines user access permissions.

Alerts

Alerts section allows admins to configure alert rules to trigger alerts and can be able to see the notifications history as well



Functionality

- **Alert Rule Creation**: Administrators can create new rules by configuring required metrics.
- **Alert Rule Modify**: Administrators can Edit/Delete existing rule
- **View alert Rule**: Administrators can view the alerts in Prometheus UI
- **Alerts history:** Can view the history of alerts with status
- **Refresh history:** By refreshing will get updated/new alerts from database

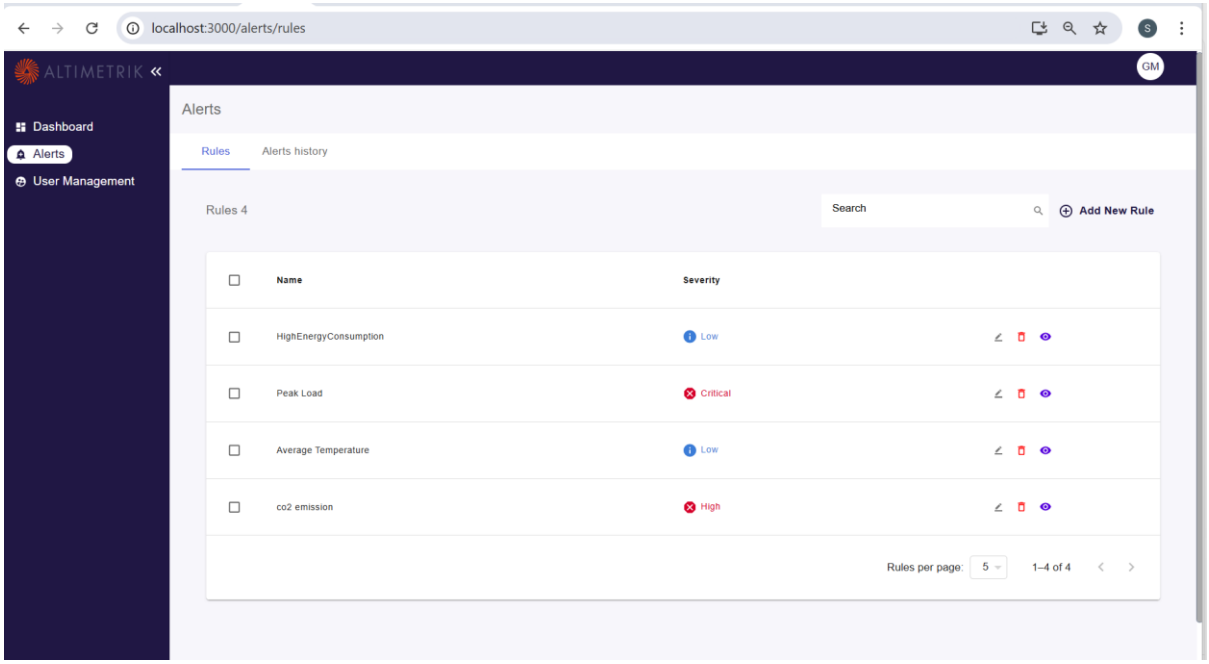
Components

- `AlertsList` (src\containers\alerts\list\AlertsList.tsx): Wrapper component for alert rules tab
- `Rules table` (src\containers\alerts\list\RulesTable.tsx): Displays a table of Rules with CRED options
- `CreateAlert` (src\containers\alerts\list\CreateAlert.tsx): Opens a drawer for creating/editing the alert rules
- `Notifications` (src\containers\alerts\notifications\Notifications.tsx): Wrapper component for Notifications/alerts history components
- `NotificationsTable`(src\containers\alerts\notifications\NotificationsTable.tsx): Displays a notifications/alerts list in tabular format

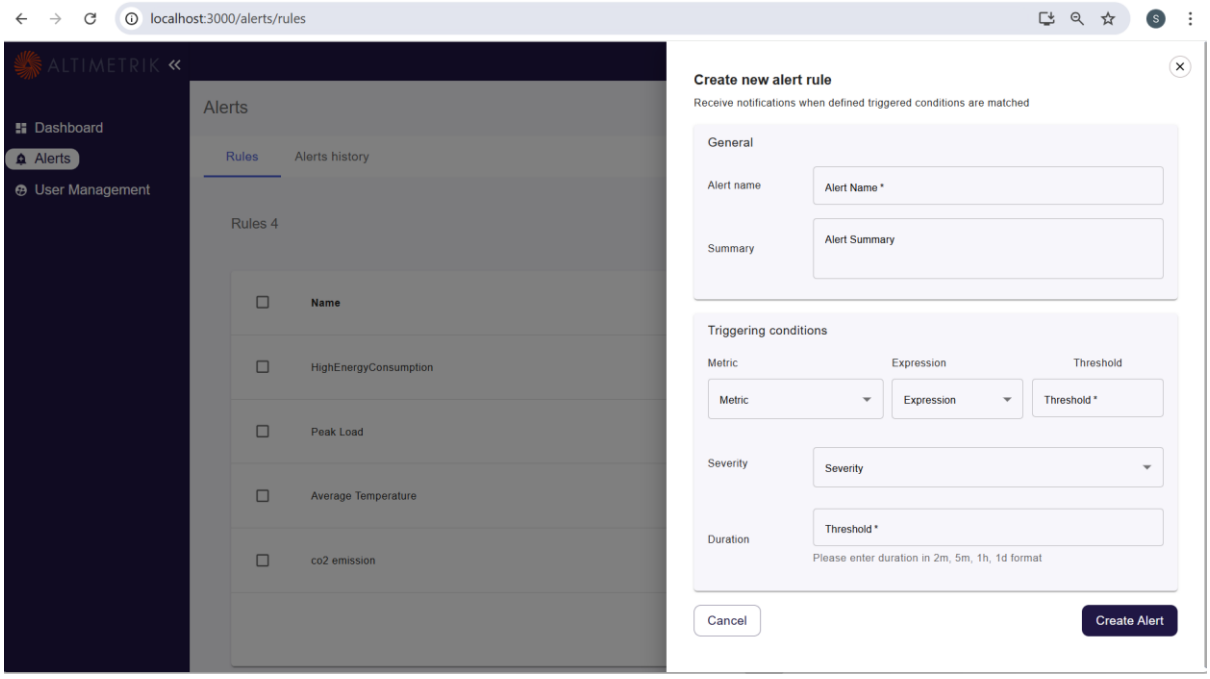
Data Flow

1. Fetching Alert Rules Data: The ` AlertsList` component fetches list of alert rules from API endpoint (‘api/alerts/rules’).

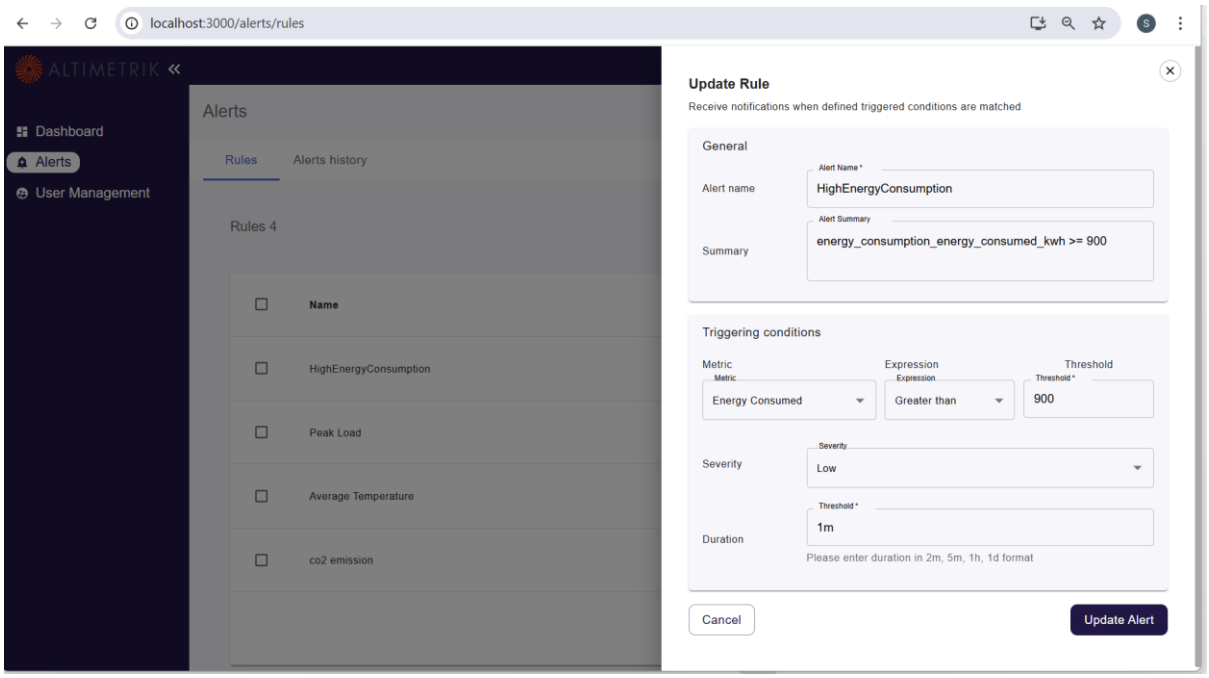
2. Displaying Alert rules data: ‘RulesTable’ component receives the rules list as props from ‘AlertsList’ and displays it in tabular format



3. Creating new Alert rule: ‘CreateAlert’ component provides UI for configuring Rule Name, description, metric for which that rule must monitor, threshold value along with condition. This component posts the consolidated data to API endpoint (‘api/alerts/rules’)



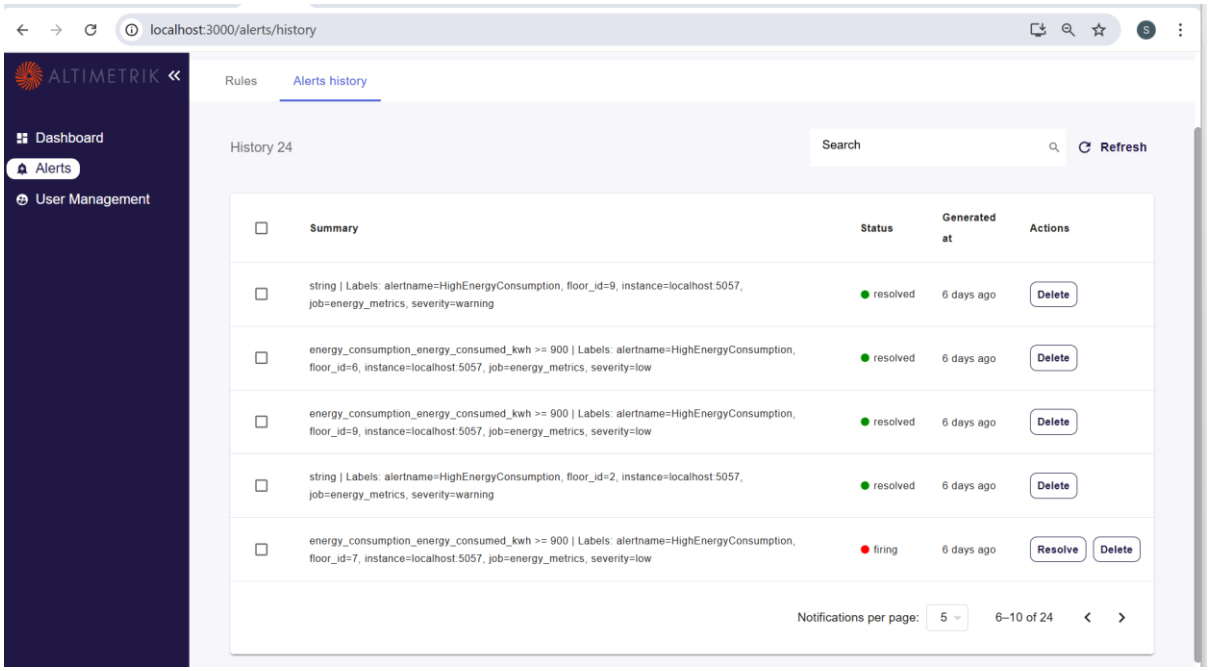
4. Editing existing Alert rule: ‘CreateAlert’ component provides UI for editing the Rule Name, description, metric for which that rule must monitor, threshold value along with condition. This component posts the consolidated data to API endpoint (‘api/alerts/rules/${rule_name}’)



5. Deleting Alert rule: When an administrator clicks the "Delete" button, the `DeleteUserModal` is displayed, prompting the administrator to confirm the deletion. If confirmed, a request is sent to an API endpoint (likely `/api/alerts/rules/${rule_name} `) to delete the user from the database.

6. Fetching Alerts Data: ‘Notifications’ component fetches list of alerts from API endpoint (‘api/alerts/get_alerts’)

7. Displaying Alerts: ‘NotificationsTable’ component receives the list of alerts from Notifications and displays in tabular format



8. Delete alert: ‘NotificationsTable’ provides a button to delete the alert from the history, an administrator clicks the "Delete" button, the `DeleteUserModal` is displayed, prompting the administrator to confirm the deletion. If confirmed, a request is sent to an API endpoint (likely `/api/alerts/delete_alert/${alert_id} `) to delete the user from the database.

9. Resolve alert: ‘NotificationsTable’ provides a button to resolve the alert from the history, an administrator clicks the "Resolve" button, a request is sent to an API endpoint (likely `/api/alerts/resolve_alert/${alert_id} `) to delete the user from the database.

Permissions

Alerts feature is protected by permissions defined in user.access.constants.ts. Key permissions include:

- ` ADD_ALERT_RULE `: Allows creating new alert rules
- ` EDIT_ALERT_RULE `: Allows editing existing alert rules
- ` DELETE_ALERT_RULE `: Allows deleting alert rules
- ` DELETE_ALERT `: Allows deleting alerts.
- ` RESOLVE_ALERT `: Allows resolving alerts.

10. Plugins and Modules Used

- **React Router:** For routing.
- **Axios:** For API calls.

- **React Chart.js 2:** For chart rendering.
- **React Context API:** For state management.
- **Material UI (MUI)**: For UI elements and ICONs

11. Summary

This documentation serves as a comprehensive guide for understanding and working with the EMS UI.