TECHNICAL DOCUMENTATION

Developed for:

Comprehensive Database Management System

Developed by:

Students from the **Department of Information Technology**
PSG College of Technology, Coimbatore-641004

Table of Contents

1. **Introduction** 4
    a. Project Overview
    b. Purpose of the Document
    c. Scope and Audience
2. **Project Architecture**..... 6
    a. Overview of the Frontend Structure
    b. Component-Based Design
    c. State Management Approach
3. **Technology Stack**..... 10
    a. ReactJS
    b. Tailwind CSS
    c. Axios for API Calls
    d. Deployment on IIS
4. **File & Folder Structure**..... 13
    a. Explanation of `src/` Directory
    b. Index.js ,App.js and Constants
5. **Component Breakdown** 15
    a. AddConduct.js
    b. facultyentry.js
    c. facultyentrystaffdashboard.js
    d. admindashboard.js
    e. AssetReturn.js
    f. AssetIssue.js
    g. Assetlist.js
    h. AssetEntry.js
    i. AssetDetails.js
    j. Assetview.js
    k. Usermanagement.js
    l. Dashboard.js

m. NotificationPanel.js

6. **Routes & Controllers……** 23
   a. Authentication (`authApi.js`)
   b. File Uploading (`fileApi.js`)
   c. Notifications (`notificationApi.js`)
   d. User Management (`userApi.js`)
7. **State Management & Data Flow**..... 25
   a. Global State vs Local State
   b. Using React Context API
   c. API Call Flow
8. **User Interface & Styling** 28
   a. Responsiveness & Mobile Compatibility
9. **Error Handling & Edge Cases**..... 30
   a. Handling API Failures
   b. Form Validations & Error Messages
   c. Handling Unauthorized Access
   d. Network Connectivity Issues
10. **Deployment on IIS** 32
    a. Steps to Deploy on IIS
    b. Configuration & Environment Variables
    c. Troubleshooting Deployment Issues
11. **Testing & Debugging** 35
    a. Unit Testing for Components
    b. Debugging API Calls
    c. Common Bugs & Fixes
12. **Security Considerations** 37
    a. Securing API Requests
    b. Preventing Cross-Site Scripting (XSS)
    c. Authentication & Token Storage

13. **Conclusion** …… 39

a. Summary of the Documentation

b. Key Takeaways

14. **References & Appendix**…… 40

a. External Links & Docs

b. Additional Code Snippets

1. Introduction

1.1 Project Overview

Comprehensive Database Management System is a web-based application developed to efficiently manage and maintain asset and faculty records within an organization. This system enables users to **easily add, update, and track asset entries and faculty details**, ensuring data accuracy, centralized access, and streamlined operations. It enhances administrative productivity by reducing manual work and providing a **user-friendly interface for database management**.

Why This System?

In traditional asset and faculty management practices, records were maintained manually or through scattered spreadsheets, leading to data inconsistencies, difficulty in updates, and limited accessibility. To overcome these limitations, the **Comprehensive Database Management System** was developed to:

- Provide a centralized platform for managing assets and faculty information.
- Allow real-time updates and retrieval of records.
- Ensure data accuracy and eliminate redundancy.
- Offer secure, role-based access to prevent unauthorized modifications.
- Enable efficient reporting and auditing for administrative purposes.

Key Features of the System

- **Asset Management** – Allows users to add, update, and track institutional assets such as equipment, tools, and resources.
- **Faculty Management** – Enables administrators to maintain detailed records of faculty including designation, department, and assigned responsibilities.
- **Role-Based Access Control** – Ensures secure access for different user roles such as administrators, department heads, and data entry staff.
- **Real-Time Updates** – Any changes to asset or faculty records are reflected immediately across the system.
- **Search and Filter Functions** – Users can easily search for specific records or filter by department, category, or status.
- **Data Backup & Integrity** – Regular backups and validation mechanisms are in place to ensure data reliability.
- **Report Generation** – Administrators can generate customized reports for audits, reviews, or internal assessments.

Technologies Used in the Project

The system is built using modern web technologies:

Technology	Purpose
ReactJS	Frontend framework for UI development
Tailwind CSS	Styling and UI responsiveness
Axios	API communication and data fetching
React Router	Navigation and page transitions
Context API	Global state management
IIS	Deployment and hosting on Windows Server

1.2 Purpose of the Document

This **Frontend Technical Documentation** serves as a **detailed guide** for developers, maintainers, and administrators involved in the system's frontend implementation. The document aims to:

- **Provide a structured overview** of the frontend architecture, explaining key components and their roles.
- **Describe API integrations**, including authentication, complaint management, and notifications.
- **Explain the deployment process on IIS**, including configurations and troubleshooting steps.
- **Document state management strategies**, error handling, and best practices for UI development.
- **Assist developers and IT teams** in maintaining and upgrading the system efficiently.

This document will act as a **reference manual** for future improvements and debugging processes.

1.3 Scope and Audience

The document is intended for multiple stakeholders, including:

Developers & Software Engineers

- Understanding **frontend architecture, component structure, and state management**.
- Modifying or adding new features.
- Debugging common frontend issues.

System Administrators & IT Support

- Deploying and maintaining the **frontend on IIS**.
- Managing server configurations and performance monitoring.

- Troubleshooting deployment-related issues.

Quality Assurance (QA) Engineers

- Testing **UI components, API interactions, and form validations**.
- Identifying **UI inconsistencies, performance issues, and security vulnerabilities**.

End-Users & Stakeholders

- Understanding the **functionalities of the system**.
- Providing feedback for **usability improvements**.

What This Document Does Not Cover

- **Backend Implementation (Refer to Backend Documentation)**
- **Database Schema and Backend APIs**
- **Server-Side Authentication & Authorization**

This document is focused exclusively on **frontend implementation and deployment**. For backend configurations, refer to the backend documentation.

---

2. Project Architecture

The **Comprehensive Database Management System** frontend is developed using **ReactJS** with **Tailwind CSS**, structured in a **modular and scalable architecture**. This architecture follows the **separation of concerns principle**, ensuring better maintainability, reusability, and scalability. The application is designed to handle a **smooth user experience** while interacting with backend services for data retrieval and processing.

2.1 Overview of the Frontend Structure

The frontend of the system is responsible for **handling user interactions**, **rendering UI elements**, and **communicating with the backend APIs**. It is built with **ReactJS**, ensuring a fast and responsive UI, and follows **best practices in component-based development**.

2.1.1 Architectural Design

The frontend follows a **three-layered architecture**:

1. **Presentation Layer** (UI) – Built using **ReactJS** and **Tailwind CSS**, responsible for **rendering components, handling user inputs, and navigation**.
2. **Logic Layer** (State Management) – Utilizes **React Hooks (useState, useEffect, useContext)** for managing local and global states.
3. **Data Layer** (API Communication) – Uses **Axios** to fetch, send, and update data from the backend via REST APIs.

This separation ensures **loose coupling**, making the system easier to debug and extend.

2.1.2 High-Level Workflow

1. **User Interacts with the UI:** Users log in, submit complaints, view complaint status, and interact with the system.
2. **Component Renders Data:** React components update dynamically based on state changes and API responses.
3. **State Management Updates Data:** React Context API stores user session details, form inputs, and other global states.
4. **API Calls to Backend:** Axios is used to send and receive data, ensuring real-time updates.
5. **Backend Processes Requests:** The server authenticates users, retrieves records, and updates statuses in the database.
6. **Frontend Updates UI:** Data fetched from the backend updates UI components, reflecting the latest record's status.

2.1.3 Deployment and Hosting

The frontend is deployed on **IIS (Internet Information Services)**, a powerful web server that ensures **high availability, scalability, and security**. This deployment setup allows seamless hosting of the **React build files**, ensuring smooth performance and **load balancing for multiple users**.

2.2 Component-Based Design

A **component-based approach** ensures **modularity, maintainability, and reusability**. Each component is designed to serve a **specific function** while being **independent and reusable**.

2.2.1 Folder Structure

The project is structured to **separate concerns**, keeping **UI, state, API calls, and utilities** in distinct directories.

```
/frontend
├── /public                 # Public assets (logos, icons, metadata)
├── /src                    # Main source code
│   ├── /assets             # Static assets (images, icons)
│   ├── /components         # Reusable UI components
│   ├── /fonts              # Page-specific components
│   ├── /styles             # React Context API for global state
│   ├── App.js              # API service handlers (Axios)
│   ├── main.js             # Helper functions (formatters, validators)
│
├── package.json            # Dependencies and scripts
├── .env                    # Environment variables
├── README.md               # Project documentation
```

This **structured organization** improves **code readability** and **scalability**, making it easier for new developers to contribute.

2.2.2 Key UI Components

Component	Description
addconduct.js	Principal can able to add the conduct of their internal and external faculty
adminassetapproval.js	Admin can be able to approve the assets.
assetissue.js	The can able to change the Asset Issue and manage the asset issue
assetmanagerdashboard.js	Assetmanager can be able to see the entire asset information.
assetreturn.js	After the asset usage ,the user returns into the store .
assetstore.js	Handles the entire Asset in a single place(new entry).
assetupdation.js	Handle the asset updation (changing the asset status).
facultyentry.js	Enter the new faculty entry(Internal and external faculty)
facultyentrystaffdashboard.js	Faculty entry staff dashboard shows the overall data of faculty entry
facultyverifierdashboard.js	It shows the faculty data that it overall data (Accept or reject)
headofofficedashboard.js	It shows the overall data of head of office
login.js	Handle the overall login and authentication of individual user
managerassetview.js	It shows the overall asset view using search filter
PrincipalAssetUpdation.js	The Principal can be able to update the asset
principalassetview.js	The Principal can be able to view the asset
principaldashboard.js	It shows the entire view of asset & faculty with notification
principalfacultyupdation.js	The principal can be to add conduct of the faculty(only conduct)
register.js	It allows to register the new user

Each component interacts with **backend APIs** for **data fetching and updates**.

2.3 State Management Approach

State management is **crucial** for handling UI interactions, API responses, and user authentication seamlessly. The system follows a **hybrid approach**, using both **local state (useState) and global state (Context API)**.

2.3.1 Global State Management (Context API)

React **Context API** is used for handling **authentication, user session, and global UI states** across components.

Authentication Context Example

```
import { createContext, useState } from "react";

export const AuthContext = createContext();

export const AuthProvider = ({ children }) => {
    const [user, setUser] = useState(null);

    return (
        <AuthContext.Provider value={{ user, setUser }}>
            {children}
        </AuthContext.Provider>
    );
};
```

This approach allows **centralized access** to the authentication state, avoiding **prop drilling**.

2.3.2 API Communication and Data Handling

API calls are managed using **Axios**, providing **asynchronous data fetching and error handling**.

Example: Fetching Data List

```
import axios from "axios";
import { useEffect, useState } from "react";

const DataList = () => {
    const [AssetData, setAssetData] = useState([]);

    useEffect(() => {
        axios.get(`${process.env.REACT_APP_API_URL}/complaints`)
            .then(response => setData(response.data))
            .catch(error => console.error("Error fetching data:", error));
    }, []);

    return (
        <ul>
            {complaints.map(complaint => (
                <li key={AssetId.id}>{Asset.description}</li>
            ))}
        </ul>
    );
};
```

This approach ensures **real-time updates**, reducing the need for manual refreshes.

---

3. Technology Stack

The **Comprehensive Database Management System** is built using modern **web technologies** to ensure **high performance, scalability, and maintainability**. The technology stack was carefully chosen to provide a **seamless user experience**, efficient data handling, and smooth communication with the backend.

3.1 ReactJS

ReactJS is the **primary framework** used for developing the frontend. It is a **declarative, component-based library** that allows for the efficient rendering of dynamic UI components.

3.1.1 Why ReactJS?

ReactJS was chosen due to the following benefits:

- **Component-Based Architecture** – Ensures reusability and modularity.
- **Virtual DOM** – Enhances performance by minimizing direct DOM manipulations.
- **One-Way Data Binding** – Ensures better control over application state.
- **React Hooks** – Enables efficient state and lifecycle management.

3.1.2 Key Features of ReactJS Used in the Project

Feature	Usage in the Project
React Components	Every UI element (navbar, complaint form, complaint list) is built as a reusable component.
React Router	Handles navigation between different pages without reloading the application.
React Hooks	Used for managing state (`useState`), side effects (`useEffect`), and global state (`useContext`).
Context API	Provides a global state for authentication and user session management.

3.1.3 Example: Creating a Simple React Component

```
import React from 'react';

const Greeting = ({ name }) => {
    return <h1>Hello, {name}! Welcome to the Grievance Redressal System.</h1>;
};

export default Greeting;
```

3.2 Tailwind CSS

To design a **responsive, modern, and scalable UI**, we have used **Tailwind CSS**, a utility-first CSS framework that provides:

- **Faster UI development** with pre-defined utility classes.
- **Highly customizable** design system using the `tailwind.config.js` file.
- **Improved performance** due to automatic CSS purging in production.

3.2.1 Why Tailwind CSS?

Advantage	Impact on Project
Utility-Based Styling	Eliminates the need for writing custom CSS classes, making styling simpler.
Flexbox & Grid Support	Used to create responsive layouts with minimal effort.
Dark Mode Support	Easily configurable to enhance user accessibility.
Customization	The design is fully customized to fit the branding and usability needs.

3.2.2 Example: Styling a Button with Tailwind CSS

```
<button className="bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded">
    Submit Complaint
</button>
```

This ensures that the button follows a **consistent design** while responding to **hover actions**.

3.3 Axios for API Calls

To handle **asynchronous communication with the backend**, the project uses **Axios**, a promise-based HTTP client for making API requests.

3.3.1 Why Axios?

- **Simplifies API Requests** – Provides an easy-to-use API for making HTTP requests.
- **Automatic JSON Handling** – Parses responses automatically.
- **Request & Response Interceptors** – Helps in modifying API requests globally.
- **Error Handling** – Provides better error messages and retry mechanisms.

3.3.2 Example: Fetching Complaints Using Axios

```
import axios from "axios";
import { useEffect, useState } from "react";

const ComplaintList = () => {
    const [complaints, setComplaints] = useState([]);

    useEffect(() => {
        axios.get(`${process.env.REACT_APP_API_URL}/complaints`)
            .then(response => setComplaints(response.data))
            .catch(error => console.error("Error fetching complaints:", error));
    }, []);

    return (
        <ul>
            {complaints.map(complaint => (
                <li key={complaint.id}>{complaint.description}</li>
            ))}
        </ul>
    );
```

```
};
```

3.4 Deployment on IIS (Internet Information Services)

To **host and serve the frontend application**, the system is deployed using **IIS (Internet Information Services)**. IIS is a **robust and scalable web server** developed by Microsoft, widely used for hosting **React and other web applications**.

3.4.1 Why IIS for Deployment?

- **Optimized for Windows Server** – Seamless integration with Windows infrastructure.
- **High Performance & Scalability** – Handles multiple requests efficiently.
- **Built-in Security Features** – Provides SSL support and authentication mechanisms.
- **Reverse Proxy Support** – Works well with backend APIs and microservices.

3.4.2 Deployment Steps

1. **Build the React App:**
2. `npm run build` This generates the build folder containing **`static files.`**
3. **Copy the Build Files to IIS Server:**
    - Place the `build` folder in a directory accessible by IIS.
    - Example location: `C:\inetpub\wwwroot\grievance-system`.
4. **Configure IIS to Serve the React App:**
    - Open IIS Manager.
    - Add a new **website** and set the **physical path** to the build folder.
    - Assign a domain or **localhost** for testing.
5. **Enable React Router Support in IIS:**
    - Since React is a **Single Page Application (SPA)**, add a `web.config` file to handle **URL rewrites**:

```
<configuration>
    <system.webServer>
        <rewrite>
            <rules>
                <rule name="React SPA" stopProcessing="true">
                    <match url=".*" />
                    <conditions logicalGrouping="MatchAll">
                        <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true" />
                        <add input="{REQUEST_FILENAME}" matchType="IsDirectory" negate="true" />
                    </conditions>
                    <action type="Rewrite" url="/index.html" />
                </rule>
            </rules>
        </rewrite>
    </system.webServer>
</configuration>
```

6. **Restart IIS and Access the Application:**
7. `iisreset`

- Open a browser and visit http://localhost/grievance-system.

4. File & Folder Structure

To ensure scalability, maintainability, and a clear separation of concerns, the **Comprehensive Database Management System** follows a structured and modular file organization. This section breaks down the key directories and components of the `src/` folder that collectively make up the frontend architecture.

a. Explanation of `src/` Directory

The `src/` directory is the heart of the frontend React application. It contains all the functional and visual components, utilities, and configuration files necessary for the seamless operation of the user interface.

Primary structure inside `src/`:

```
src/
│
├── assets/                → Static files like images, logos, icons
├── components/            → Reusable UI components like Header, Sidebar, Cards
├── fonts/                 → Route-based components like Dashboard, GrievanceList
├── styles/                → API call abstractions using Axios
├── App.js                 → Utility functions (e.g., token management, role-based access)
├── main.js                → Global constants (roles, status codes, API routes)
```

Each of these folders has a specific responsibility, contributing to clean code and separation of concerns.

b. index.js, App.js, and Constants

1. **Index (`index.js`)**
   - Contains functions for HTTP requests using Axios.
   - Centralized API handling allows easier configuration of headers, tokens, and base URLs.
   - Example: `index.js`

```
//index.js
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import './index.css';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

2. **App(`App.js`)**
   - **Purpose**: Root component that holds the overall structure of your app.
   - **Usage**: Contains routes, navigation, and layout logic.
   - Example: App.js

```
function App() {
  return (
    <div>
```

```
      <Header />
      <MainRoutes />
    </div>
  );
}
export default App;
```

3. **Assets (`Assets/`)**
    - **Purpose**: Stores static files like images, fonts, SVGs, or videos
    - **Example**: logo.png, background.jpg
    - **Usage**: Access using imports like:

      ```
      import logo from './assets/logo.png';
      ```

This structured organization ensures a maintainable and collaborative development environment, allowing multiple team members to work independently on isolated modules without conflicts.

5. Component Breakdown

In this section, we provide an in-depth breakdown of the major components in the frontend application of the **Comprehensive Database Management System**. Each component is designed with modularity and reusability in mind, conforming to modern ReactJS principles. The components fall into functional categories such as navigation, user interaction, complaint submission, and administrative management.

---

a.AddConduct.js

Purpose:
Allows the Principal to view and update a faculty member's conduct and remarks. The form is pre-filled with data fetched from the backend based on faculty ID.

Key Features:

- Uses React Router's `useParams` to get the `facultyId` from the URL.
- Fetches faculty details (`name`, `conduct`, and `remarks`) via an API.
- Enables the Principal to update conduct and remarks through a form.
- Displays saving status and uses SweetAlert2 for success/error messages.

- Responsive and styled using inline styles and external CSS.

Typical Elements:

```
<input type="text" value={facultyName} disabled />
<select value={conduct} onChange={...}>
  <option value="best">Best</option>
  ...
</select>
<input type="text" value={remarks} onChange={...} />
<button type="submit">Submit</button>
```

---

b. facultyentry.jsx

Purpose:
Allows staff to enter, edit, and auto-save faculty data including personal details, educational qualifications, publications, modules, and other roles.

Key Features:

- Uses `useLocation` to extract faculty data passed via navigation state.
- Auto-saves faculty form inputs on change.
- Provides full faculty data structure with fields like name, date of birth, courses handled, modules, awards, etc.
- Includes major and minor domain selection with interlinked options.
- Validates all inputs before submission.
- Uses `Swal.fire` (SweetAlert2) for user alerts and notifications.
- Automatically loads drafts if available via API.
- Upload preview for photograph with size restriction.

Typical Elements:

```
<input type="text" name="name" value={facultyData.name} onChange={...} />
<select value={facultyType} onChange={handleFacultyTypeChange}>
  <option value="internal">Internal</option>
</select>
<input type="date" name="dateOfBirth" value={facultyData.dateOfBirth} onChange={...} />
<input type="file" onChange={handleFileChange} />
<button onClick={handleSubmit}>Submit</button>
```

---

c. facultyentrystaffdashboard.jsx

Purpose:
Provides a dashboard for staff to manage faculty entry operations and review notifications related to faculty approvals and verifications.

Key Features:

- Displays sidebar with navigation to entry/view modules.
- Pulls and displays real-time notification data (rejections, alerts).
- Allows “re-enter” on rejected entries via navigation with state.
- Allows clearing individual or all notifications.
- Uses `Helmet` for metadata, responsive layout, and icons.
- Handles notification expansion for more detail view.

Typical Element:

```
<button onClick={toggleNotificationPanel}><i className="fas fa-bell" /></button>
{notifications.map(notification => (
  <button onClick={() => handleReenter(notification)}>Re-enter</button>
))}
<a href={`/facultyentrydashboard?username=${username}`}>Home</a>
```

---

d.admindashboard.jsx

Purpose:
Manages all asset approval-related operations including purchase, issue, return, exchange, service, disposal, and building upgrade.

Key Features:

- Uses `useEffect` to load asset data based on active tab (return, issue, disposal, etc.).
- Handles approvals and rejections with remarks via modals (SweetAlert2).
- Detailed views for each asset type with formatted tables.
- Filtering/searching for return assets with debounced input.
- Displays modals/popups for asset details.
- Full CRUD handling for asset workflows.

Typical Elements:

<button onClick={() => setActiveTab("return")}>Return</button>

<input type="text" value={searchTerm} onChange={...} />

<button onClick={() => approveReturn(asset._id, asset.newCondition)}>Approve</button>

<select onChange={(e) => handleConditionChange(asset._id, e.target.value)} />

---

e.AssetReturn.jsx

Purpose:
Displays the list of returned assets pending approval and allows managers to review, change condition, approve, or reject them.

Key Features:

- Fetches Permanent and Consumable return assets via API.
- Allows filtering by item name with search debounce.
- Each asset has a "Change Condition" dropdown and action buttons.
- Handles different condition mappings for Permanent vs Consumable.
- SweetAlert2 used for confirmation and success/error messages.
- Uses consistent card-based layout for each asset entry.

Typical Element:

<select value={asset.newCondition} onChange={(e) => handleConditionChange(asset._id, e.target.value)}>

<option value="Good">Good</option>

</select>

<button onClick={() => approveReturn(asset._id, asset.newCondition)}>Approve</button>

<button onClick={() => rejectReturn(asset._id)}>Reject</button>

---

f. AssetIssue.jsx

Purpose:
Handles acknowledgement and display of issued assets along with options to approve or reject with remarks.

Key Features:

- Fetches issued assets that have been acknowledged by staff.
- Displays full asset details including recipient and quantity.
- Allows admin to approve or reject asset issuance.
- SweetAlert2 modal prompts for rejection reason.
- Uses table layout for clear action controls.

Typical Element:

```
<td>{asset.itemName}</td>
<button onClick={() => approveIssue(asset._id)}>Approve</button>
<button onClick={() => rejectIssue(asset._id)}>Reject</button>
```

g. AssetList.jsx

Purpose:
Displays a list of all assets in the system with options to view details, edit, or delete assets.

Key Features:

- Fetches and displays all assets from the database.
- Supports pagination and filtering by asset category or status.
- Provides buttons for editing or deleting assets.
- Links to detailed asset view for each entry.
- Responsive table design for better usability on all devices.

Typical Element:

```
<tr>
  <td>{asset.itemName}</td>
  <td>{asset.category}</td>
  <button onClick={() => editAsset(asset._id)}>Edit</button>
  <button onClick={() => deleteAsset(asset._id)}>Delete</button>
</tr>
```

h. AssetEntry.jsx

Purpose:
Provides a form for creating or updating asset details in the system.

Key Features:

- Supports both create and edit modes based on provided asset ID.
- Validates input fields (e.g., item name, quantity, category).
- Displays error messages for invalid inputs.
- Submits data to the backend API and shows success/error notifications.
- Includes a cancel button to discard changes.

Typical Element:

```
<div>
  <label>Item Name</label>
  <input
    type="text"
    value={formData.itemName}
    onChange={(e) => setFormData({ ...formData, itemName: e.target.value })}
  />
  <button onClick={handleSubmit}>Save</button>
```

```
</div>
```

i. AssetDetails.jsx

Purpose:
Renders detailed information about a specific asset, including its history and current status.

Key Features:

- Fetches asset data by ID from the backend.
- Displays asset attributes like name, category, quantity, and issuance history.
- Shows a timeline of asset transactions (e.g., issued, returned).
- Allows navigation back to the asset list.
- Handles loading and error states gracefully.

Typical Element:

```
<div>
  <h2>{asset.itemName}</h2>
  <p>Category: {asset.category}</p>
  <p>Quantity: {asset.quantity}</p>
  <button onClick={() => navigate('/assets')}>Back to List</button>
</div>
```

j. Assetview.jsx

Purpose:
Displays the transaction history of an asset, including issuance and return records.

Key Features:

- Retrieves and lists all transactions related to a specific asset.
- Shows details like date, recipient, quantity issued, and status.
- Supports sorting by date or transaction type.
- Uses a collapsible table for detailed transaction notes.
- Exports history as a CSV file for reporting.

Typical Element:

```
<tr>
  <td>{transaction.date}</td>
  <td>{transaction.recipient}</td>
  <td>{transaction.quantity}</td>
  <button onClick={() => toggleDetails(transaction._id)}>
    {showDetails ? 'Hide' : 'Show'} Details
  </button>
</tr>
```

k. UserManagement.jsx

Purpose:
Manages user accounts with options to create, update, or deactivate users.

Key Features:

- Fetches and displays a list of users with their roles (e.g., admin, staff).
- Allows admins to create new users or edit existing ones.
- Supports deactivating/reactivating user accounts.
- Uses modals for user creation and confirmation dialogs for deactivation.
- Filters users by role or status.

Typical Element:

```
<tr>
  <td>{user.name}</td>
  <td>{user.role}</td>
  <button onClick={() => editUser(user._id)}>Edit</button>
  <button onClick={() => deactivateUser(user._id)}>Deactivate</button>
</tr>
```

l. Dashboard.jsx

Purpose:
Provides an overview of the asset management system with key metrics and quick actions.

Key Features:

- Displays summary statistics (e.g., total assets, issued assets, pending approvals).
- Includes charts for visualizing asset distribution by category.
- Offers quick links to common tasks (e.g., add asset, view issues).
- Updates metrics in real-time using API polling.
- Responsive layout for desktop and mobile views.

Typical Element:

```
<div>
  <h3>Total Assets: {stats.totalAssets}</h3>
  <p>Issued Assets: {stats.issuedAssets}</p>
  <button onClick={() => navigate('/assets/new')}>Add New Asset</button>
</div>
```

m. NotificationPanel.jsx

Purpose:
Manages and displays system notifications for users, such as pending actions or alerts.

Key Features:

- Fetches unread and read notifications for the logged-in user.
- Marks notifications as read individually or in bulk.
- Supports filtering by notification type (e.g., alerts, reminders).
- Displays notifications in a dropdown or sidebar.
- Integrates with WebSocket for real-time updates.

Typical Element:

```
<div>
  <p>{notification.message}</p>
  <span>{notification.date}</span>
  <button onClick={() => markAsRead(notification._id)}>Mark as Read</button>
</div>
```

Additional Features:

- Color-coded status indicators.
- Option to upload response files.
- Dynamic tab view based on status.

---

6. Model View Controller(MVC)

The Model-View-Controller (MVC) design pattern is a widely used architectural framework that separates an application into three interconnected components. Each component has distinct responsibilities, promoting modularity, scalability, and maintainability. Below is a detailed explanation of each component in the MVC pattern, presented separately with general information and structured similarly to your example.

---

a. Model

The Model represents the data, business logic, and rules of the application. It is responsible for managing the underlying structure and storage of data, performing operations like fetching, updating, or deleting data, and notifying the View of any changes. The Model is independent of the user interface and communicates with the Controller to process user input or with the View to reflect updates.

Responsibilities:

- Manage the application's data and state.
- Handle data operations (e.g., CRUD: Create, Read, Update, Delete).
- Communicate with databases, APIs, or other backend services.
- Notify the View of data changes (e.g., through events or observables).
- Enforce business rules and data validation.

Example:

```
import axios from 'axios';


export const fetchUser = async (userId) => {
  const response = await axios.get(`/api/users/${userId}`);
  return response.data;
};

export const updateUser = async (userId, userData) => {
  const response = await axios.put(`/api/users/${userId}`, userData);
  return response.data;
};

export const deleteUser = async (userId) => {
  await axios.delete(`/api/users/${userId}`);
};
```

---

b. View

The View is the user interface of the application, responsible for displaying the data from the Model to the user. It is a visual representation of the Model’s data and listens for updates from the Model or Controller to refresh the display. The View is passive and does not directly manipulate data; it forwards user interactions to the Controller.

Responsibilities:

- Render the user interface based on the Model's data.
- Display updates when the Model changes.
- Capture user input (e.g., clicks, form submissions) and pass it to the Controller.
- Provide a responsive and interactive experience for the user.

Example:

```
// view/UserProfile.js (React example)
import React, { useState, useEffect } from 'react';

const UserProfile = ({ user, onUpdate }) => {
  const [name, setName] = useState(user.name);
  const [email, setEmail] = useState(user.email);

  const handleSubmit = (e) => {
    e.preventDefault();
    onUpdate({ name, email });
  };

  return (
    <div>
      <h2>User Profile</h2>
      <form onSubmit={handleSubmit}>
        <label>
          Name:
          <input
            type="text"
            value={name}
            onChange={(e) => setName(e.target.value)}
          />
        </label>
        <label>
          Email:
          <input
            type="email"
            value={email}
            onChange={(e) => setEmail(e.target.value)}
          />
        </label>
        <button type="submit">Update</button>
      </form>
    </div>
  );
};

export default UserProfile;
```

---

c. Controller

The Controller acts as an intermediary between the Model and the View. It handles user input from the View, processes it (often updating the Model), and ensures the View reflects the updated state of the Model. The

Controller contains the application's logic for responding to user actions and coordinating between the Model and View.

Responsibilities:

- Receive and process user input from the View.
- Update the Model based on user actions.
- Trigger updates in the View when the Model changes.
- Coordinate the flow of data and actions between the Model and View.
- Handle errors and edge cases.

Example:

```
export const fetchNotifications = () => axios.get("/notifications");
export const markAsRead = (id) => axios.put(`/notifications/read/${id}`);
```

Integration in Notifications.js:

```
useEffect(() => {
  fetchNotifications()
    .then(res => setNotifications(res.data))
    .catch(() => setError("Failed to load notifications"));
}, []);
```

---

d. User Management

Manages user-related data: role assignment, profile update, listing.

Responsibilities:

- Fetch all users (Storekeeper,Assetmanager,Headofoffice,Principal,Facultyentrystaff,facultyverifier,Viewer)
- Create an new user
- Filter users by role (admin, staff, student)

Example:

```
export const getAllUsers = () => axios.get("/users");
export const updateUser = (id, data) => axios.put(`/users/${id}`, data);
export const getProfile = () => axios.get("/users/me");
```

Best Practices Followed in Service Layer:

Principle	Applied Methodology
Centralization	All APIs grouped into separate files per module
Error Handling	Axios interceptors and local `try/catch` blocks
Token Security	Tokens auto-attached in headers using interceptors
Reusability	Services decoupled from components
Consistency in API Calls	Uniform naming and structure for all service functions

---

7. State Management & Data flow

Efficient state management is pivotal in ensuring that data flows smoothly across the components of the **Comprehensive Database Management System** frontend UI. The application leverages both **local state** for UI-level responsiveness and the **React Context API** for managing shared/global state. This section outlines the approaches adopted, how data flows across modules, and strategies for scalability.

---

a. Global State vs Local State

Understanding the difference between **local** and **global** state is key to managing the complexity of modern SPAs.

Type	Usage	Example
Local State	Maintains state within a component	Form inputs, toggles, modals
Global State	Shared across multiple components or routes	Logged-in user data, notification count on top

Examples:

- **Local State (useState):**

```
const [notification, setnotification] = useState("");
const [error, setError] = useState(null);
```

- **Global State (React Context):**

```
const { user } = useContext(AuthContext);
```

---

b. Using React Context API

To avoid **prop drilling**, the application uses **React Context API** for managing global state such as:

- User authentication info
- Current user role
- Global notifications
- Theme settings (optional for future)

AuthContext.js:

```
export const AuthContext = createContext();

export const AuthProvider = ({ children }) => {
  const [user, setUser] = useState(null);

  const login = (userData) => setUser(userData);
  const logout = () => setUser(null);

  return (
    <AuthContext.Provider value={{ user, login, logout }}>
      {children}
    </AuthContext.Provider>
  );
};
```

Usage in App.js:

```
<AuthProvider>
  <BrowserRouter>
    <Routes> ... </Routes>
  </BrowserRouter>
</AuthProvider>
```

---

c. API Call Flow

Each API-triggered interaction follows a structured flow ensuring **asynchronous safety**, **loading indicators**, and **error fallback**.

Typical Flow:

1. **Action Initiated** → Button Click / useEffect
2. **Loading State Set** → `setLoading(true)`
3. **API Call (Axios)** → With try/catch
4. **State Updated on Success** → `setData(response.data)`
5. **Handle Error Gracefully** → Toasts or error component
6. **Reset Loading State** → `setLoading(false)`

Example in Notifications.js:

```
useEffect(() => {
  const fetchData = async () => {
    try {
      setLoading(true);
      const res = await fetchNotifications();
```

```
      setNotifications(res.data);
    } catch (err) {
      toast.error("Failed to load notifications");
    } finally {
      setLoading(false);
    }
  };
  fetchData();
}, []);
```

---

d. State Synchronization Between Components

- **Example 1: Notification count changes in one component (e.g., AdminDashboard)** is reflected globally due to useContext.
- **Example 2: User login in `Login.js` updates global `AuthContext`**, allowing role-based routing.

---

e. Benefits of This Approach

Feature	Benefit
Centralized Context	Easy to manage auth and global variables
Local State Isolation	Prevents unnecessary re-renders
Predictable Data Flow	Each interaction follows a consistent pattern
Scalable	Easy to extend for new modules or UI changes
Testable	Components remain decoupled and isolated

---

8. User Interface & Styling

The User Interface (UI) of the **CASFOS Comprehensive Database Management System** is built using **ReactJS** and styled with **Tailwind CSS**, ensuring a modern, responsive, and accessible experience across devices. This section outlines the design methodology, responsiveness considerations, and the visual aesthetics integrated into the platform.

---

a. Responsiveness & Mobile Compatibility

Responsiveness ensures the system is accessible on desktops, tablets, and smartphones. Tailwind CSS enables responsive design with **breakpoints** such as:

- `sm`: (≥ 640px)
- `md`: (≥ 768px)
- `lg`: (≥ 1024px)

- `xl`: (≥ 1280px)

Implementation Examples:

1. **Grid Layouts:**

```
<div className="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3 gap-6">
  {/* Complaint cards or dashboard metrics */}
</div>
```

2. **Conditional Rendering Based on Screen Size:**

```
<div className="hidden md:block">
  {/* Show sidebar only on medium and up */}
</div>
```

3. **Mobile Navigation Handling:**

- Toggle-based mobile menu for sidebar navigation
- Dropdowns optimized for touch input

Color Scheme and Branding

The application reflects a professional and institutional look using:

- **Primary Colors**: Green (`bg-green-600`)
- **Alert Colors**: Red for errors, Yellow for warnings, Blue for info
- **Typography**: System fonts with `font-semibold`, `text-lg`, etc.

Visual Consistency:

- Same button styles across modules
- Alerts and toast notifications follow a color-coded convention
- Dark/light shades used for contrast and accessibility

---

9. Error Handling Strategy

Robust error handling is critical to delivering a stable and reliable user experience. In the **Grievance Redressal System**, errors are anticipated, intercepted, and resolved gracefully at every layer of the frontend, from API failures to form validation issues. This section outlines the structured approach to error management, user feedback mechanisms, and fallback strategies employed in the application.

---

a. Types of Errors Considered

The application handles several categories of errors, including:

1. **Network/API Errors**: Server unavailability, timeouts, or incorrect endpoints.
2. **Authentication/Authorization Errors**: Invalid tokens, expired sessions, unauthorized route access.
3. **Validation Errors**: User input errors during form submission.
4. **Application Logic Errors**: State mismatches, improper data rendering, or component failures.
5. **Client-Side JavaScript Errors**: Unexpected exceptions in rendering or event handling.

---

b. API-Level Error Handling (Axios)

Axios interceptors are used to globally capture errors from API calls. This helps standardize error responses across the app.

```
axiosInstance.interceptors.response.use(
  response => response,
  error => {
    if (!error.response) {
      // Network error
      alert("Network error. Please check your internet connection.");
    } else if (error.response.status === 401) {
      // Unauthorized
      localStorage.clear();
      window.location.href = "/login";
    } else if (error.response.status === 403) {
      alert("Access Denied. You are not authorized.");
    } else if (error.response.status >= 500) {
      alert("Server Error. Please try again later.");
    }
    return Promise.reject(error);
  }
);
```

---

c. Component-Level Error Boundaries

React error boundaries are used to catch **runtime rendering errors** in components, preventing the entire application from crashing.

Example :

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
```

```
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  render() {
    if (this.state.hasError) {
      return <h2>Something went wrong. Try refreshing the page.</h2>;
    }

    return this.props.children;
  }
}
```

Wrap major routes/components inside:

```
<ErrorBoundary>
  <YourComponent />
</ErrorBoundary>
```

---

d. Form Validation and Input Feedback

Forms are validated both on client and server side. Client-side validation uses:

- HTML5 validation attributes (e.g., `required`, `pattern`)
- Manual checks with helpful error messages

Example:

```
if (!email.includes('@')) {
  setError("Please enter a valid email.");
  return;
}
```

Server-side validation errors are returned and mapped:

```
catch(error => {
  if (error.response?.data?.errors) {
    setFieldErrors(error.response.data.errors);
  } else {
    setGeneralError("Submission failed.");
  }
});
```

Navigation & Session Handling Errors

Unauthorized access attempts (invalid JWTs or wrong role) are intercepted via route guards and redirected to:

- **Login Page**
- **Unauthorized Component**
- **Error Message Toasts**

Unauthorized Component Example:

```
return (
  <div>
```

```
    <h3>403 - Unauthorized</h3>
    <p>You do not have permission to access this page.</p>
  </div>
);
```

Global UI Feedback (Toasts & Alerts)

Errors and notifications are shown using:

- **Toasts (e.g., react-toastify)**
- **Modal Dialogs for critical errors**
- **Snackbar-style alerts for actions**

```
toast.error("Error submitting form. Try again later.");
toast.success("Complaint registered successfully!");
```

Logging and Debugging Aids

In development mode, all API errors are logged with complete context:

```
if (process.env.NODE_ENV === 'development') {
  console.error("API Error:", error.response);
}
```

Benefits of Centralized Error Handling

Feature	Benefit
Axios Interceptors	Uniform handling of all API errors
Error Boundaries	Prevent full app crashes
Toasts & Alerts	Real-time user feedback
Form Validation	Minimizes incorrect submissions
Role-Based Guards	Prevents unauthorized access
Debug Logs	Accelerates issue diagnosis during development

---

10. Deployment on IIS

The frontend of the CASFOS Grievance Redressal System is deployed on **Internet Information Services (IIS)**, a secure and scalable web server from Microsoft. Hosting the ReactJS application on IIS ensures high availability, efficient request handling, and support for enterprise-grade deployments in a Windows environment.

10.1 Steps to Deploy on IIS

The deployment process involves building the React app, configuring IIS, and ensuring the appropriate permissions and MIME types are set.

10.1.1 Build the React Application

1. Ensure all environment variables in `.env` are set appropriately for production.
2. Run the production build command:

```
npm run build
```

3. This creates a `build/` directory with static HTML, CSS, JS, and asset files.

10.1.2 Setup IIS for Hosting

1. Open **IIS Manager** on the Windows server.
2. Create a new **website** or application under *Default Web Site*.
3. Set the **physical path** to the React `build/` directory.
4. Configure the site **binding**:

- Type: `http` or `https` (recommended for security)
- Port: Typically `80` or `443`
- Hostname: Set based on your domain

10.1.3 MIME Type Configuration

Ensure IIS supports serving modern JS and CSS files:

- Add or verify MIME types like:
    - `.js` → `application/javascript`
    - `.json` → `application/json`
    - `.css` → `text/css`
    - `.svg` → `image/svg+xml`

10.1.4 Configure URL Rewrite (for SPA Routing)

React apps using React Router require client-side routing fallback to `index.html`:

1. Install **IIS URL Rewrite Module** (if not already).
2. Add a `web.config` file inside the `build/` directory:

```
<configuration>
  <system.webServer>
    <rewrite>
      <rules>
        <rule name="React Routes" stopProcessing="true">
          <match url=".*" />
          <conditions logicalGrouping="MatchAll">
            <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true" />
```

```
            <add input="{REQUEST_FILENAME}" matchType="IsDirectory" negate="true" />
          </conditions>
          <action type="Rewrite" url="/index.html" />
        </rule>
      </rules>
    </rewrite>
  </system.webServer>
</configuration>
```

10.2 Configuration & Environment Variables

Proper configuration ensures separation between development and production environments.

10.2.1 Environment Variables in `.env`

Place this in the root directory:

```
REACT_APP_API_URL=https://api.casfos.example.com
REACT_APP_ENV=production
```

- These values are compiled at build time and referenced using `process.env.REACT_APP_*`.

10.2.2 IIS Configuration Parameters

- **Application Pool**: Use `No Managed Code` for React.
- **Permissions**: Grant `IIS_IUSRS` read access to the build directory.
- **Static Content**: Ensure the "Static Content" feature is enabled in Windows Features.

10.2.3 Handling HTTPS (Optional but Recommended)

- Bind SSL certificates in IIS for HTTPS.
- Ensure proper redirection from HTTP to HTTPS using rewrite rules.

10.3 Troubleshooting Deployment Issues

Issue	Cause	Resolution	Code / Configuration Example
1. Blank Page on Route Refresh (React Router SPA)	IIS tries to find a physical file or folder for the route	Use **URL Rewrite** in `web.config` to redirect all unmatched routes to `index.html`	`xml <configuration> <system.webServer> <rewrite> <rules> <rule name="React Routes" stopProcessing="true"> <match url=".*" /> <conditions logicalGrouping="MatchAll"> <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true" /> <add input="{REQUEST_FILENAME}" matchType="IsDirectory" negate="true" /> </conditions> <action type="Rewrite" url="/index.html" /> </rule> </rules> </rewrite> </system.webServer> </configuration>`
2. 404 Errors for JS/CSS Files	Missing or incorrect MIME types in IIS	Add the required MIME types in IIS Manager → MIME Types	Example types to add: `.js → application/javascript.css → text/css.json → application/json`
3. Environment	Variables not prefixed	Ensure the `.env` file is created	Example `.env:env REACT_APP_API_URL=https://api.example.com<br`

Issue	Cause	Resolution	Code / Configuration Example
Variables Not Loaded in Build	with `REACT_APP_` or `.env` not loaded during build	**before running the build** and variables are properly prefixed	`>REACT_APP_ENV=production `Access in code:`js const baseUrl = process.env.REACT_APP_API_URL; `
4. 500 Internal Server Error	Missing read permissions on IIS directory or corrupted build	- Give read permissions to `IIS_IUSRS` group- Rebuild project using `npm run build`	Command to reset permissions (run in PowerShell as admin):`powershell icacls "C:\path\to\build" /grant "IIS_IUSRS:(OI)(CI)RX" /T `
5. CORS Errors When Calling Backend	Backend does not allow frontend origin	Enable CORS on backend by allowing the React app domain in the response headers	**Node.js Express example**:`js app.use(cors({  origin: "http://yourfrontend.com",  credentials: true })); `
6. IIS Serves Directory Listing or Index Not Rendered	`Default Document` feature not enabled in IIS	Enable “Default Document” in **IIS Features** and move `index.html` to the top of the list	IIS Manager → Site → “Default Document” → Ensure `index.html` is present and ordered at the top

---

11. Testing & Debugging

Thorough testing and efficient debugging are essential for maintaining the reliability and quality of the CASFOS Grievance Redressal System. This section outlines approaches for component testing, debugging techniques for API integration, and common issues encountered during development or deployment.

11.1 Unit Testing for Components

Unit testing ensures that individual React components function correctly in isolation.

11.1.1 Testing Framework

The application uses:

- **Jest**: JavaScript testing framework.
- **React Testing Library**: For testing React components by simulating user behavior.

11.1.2 Running Tests

```
npm test
```

- Automatically detects and runs `.test.js` files.
- Displays pass/fail status with coverage information.

11.2 Debugging API Calls

Debugging API calls is critical to ensure proper communication with the backend.

11.2.1 Using Browser DevTools

- Open **Chrome DevTools → Network Tab**.
- Monitor outgoing requests (URLs, methods, payloads).
- Check:
    - HTTP status codes (e.g., 200, 401, 500).
    - Response body.
    - Request headers (e.g., Authorization tokens).

11.2.2 Axios Debug Example

```
axios.get(`${process.env.REACT_APP_API_URL}/complaints`)
  .then((res) => console.log("Data:", res.data))
  .catch((err) => console.error("Error:", err.response || err));
```

11.2.3 Enabling Axios Interceptors for Logging

```
axios.interceptors.request.use((config) => {
  console.log(`[Request] ${config.method?.toUpperCase()} - ${config.url}`);
  return config;
});

axios.interceptors.response.use(
  response => response,
  error => {
    console.error(`[Response Error] ${error.response?.status}:`, error.response?.data);
    return Promise.reject(error);
  }
);
```

11.3 Common Bugs & Fixes

Bug	Cause	Fix
Component not rendering data	`useEffect` not triggered or dependency array is incorrect	Ensure correct dependency array: `js useEffect(() => fetchData(), []); `

Bug	Cause	Fix
Token not sent in API requests	Axios requests missing headers	Add token to default headers: `js axios.defaults.headers.common['Authorization'] = `Bearer ${token}`; `
Form doesn't reset after submission	State not cleared post-submit	Manually reset state: `js setComplaint(""); `
Page refresh clears session	User state only stored in React state	Store token in localStorage/sessionStorage and rehydrate: `js localStorage.setItem("token", token); const savedToken = localStorage.getItem("token"); `
Unhandled API errors crash UI	Missing try/catch or `.catch()` blocks	Always handle errors: `js try { await axios.get(...); } catch (e) { ... } `

12. Security Considerations

Security is a critical aspect of any web application, especially one handling sensitive grievance data. This section highlights key strategies and implementation practices used to safeguard API communications, prevent vulnerabilities, and ensure user authentication security.

12.1 Securing API Requests

12.1.1 Using HTTPS

- All API requests must be made over **HTTPS** to ensure encrypted communication.
- Enforce `https://` endpoints in `.env` files:

```
REACT_APP_API_URL=https://api.casfos.in
```

12.1.2 Authorization Headers

- All authenticated requests include a bearer token in the `Authorization` header.
- Automatically inject token using Axios interceptors:

```
axios.interceptors.request.use((config) => {
  const token = localStorage.getItem('token');
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }
  return config;
});
```

12.1.3 Rate Limiting and Throttling (Backend Support)

- While primarily a backend concern, the frontend can handle:
    - Retry delays

- Showing appropriate UI messages
- Temporarily disabling buttons after failed login attempts

12.2 Preventing Cross-Site Scripting (XSS)

12.2.1 Avoid Direct HTML Injection

- Avoid using `dangerouslySetInnerHTML` unless necessary. Always sanitize if used:

```
import DOMPurify from 'dompurify';

const safeHTML = DOMPurify.sanitize(dirtyHTML);
```

12.2.2 Escape User Input

- Sanitize any user-generated content before rendering.

Example:

```
const escapeHTML = (str) =>
  str.replace(/[&<>"']/g, (m) => ({
    '&': '&amp;',
    '<': '&lt;',
    '>': '&gt;',
    '"': '&quot;',
    "'": '&#039;',
  }[m]));
```

12.2.3 Content Security Policy (CSP)

- Configure IIS to use CSP headers:

```
<httpProtocol>
  <customHeaders>
    <add name="Content-Security-Policy" value="default-src 'self'; script-src 'self';
style-src 'self' 'unsafe-inline';"/>
  </customHeaders>
</httpProtocol>
```

12.3 Authentication & Token Storage

12.3.1 Token Storage Strategy

- Store tokens in **`localStorage`** for persistence, but be cautious:
    - Avoid storing sensitive data in plaintext
    - Protect against XSS to prevent token theft

12.3.2 Token Handling on Login

```
authApi.login(credentials).then(res => {
  localStorage.setItem("token", res.data.token);
  setUser(res.data.user);
});
```

12.3.3 Logout and Cleanup

```
const handleLogout = () => {
  localStorage.removeItem("token");
  setUser(null);
  navigate("/login");
};
```

12.3.4 Session Validation on Page Load

```
useEffect(() => {
  const token = localStorage.getItem("token");
  if (token) {
    // Optionally verify with backend
    setUser(decodeToken(token));
  } else {
    navigate("/login");
  }
}, []);
```

12.3.5 Protecting Routes (Private Routes)

```
const PrivateRoute = ({ children }) => {
  const { user } = useContext(AuthContext);
  return user ? children : <Navigate to="/login" />;
};
```

12.4 Additional Frontend Security Practices

Practice	Description
Disable DevTools in Production	Use tools like `react-devtools-detector` or obfuscate source maps.
Set `X-Content-Type-Options`	Prevents MIME sniffing: `<add name="X-Content-Type-Options" value="nosniff" />`
Disable Autocomplete in Sensitive Forms	Prevents password suggestions in complaint forms: `<input type="password" autoComplete="off" />`
Form Validation	Prevent malformed input from reaching backend via regex checks and length limits.
CORS Configuration on Backend	Ensure only the frontend origin is whitelisted to make API calls.

---

13. Conclusion

13.1 Summary of the Documentation

The **CASFOS Comprehensive Database Management System Frontend Documentation** provides a comprehensive walkthrough of the architecture, components, state management, API integration, deployment setup, and security mechanisms. Key highlights include:

- A modular **ReactJS** frontend structure powered by **Tailwind CSS** for fast, responsive UI.
- Robust **state management** through a combination of React Hooks and Context API to ensure smooth data flow across components.

- **Service-oriented API interaction** using Axios, with structured service files for authentication, complaint handling, notifications, and file uploads.
- Strategic **error handling** and edge-case coverage to ensure graceful failure and better user experience.
- A streamlined **IIS deployment process**, enabling production-level hosting with environmental configuration and troubleshooting steps.
- Integration of **testing and debugging practices** to ensure reliability and maintainability of the frontend system.
- Emphasis on **security**, including token management, XSS protection, and route guarding for authenticated access.

13.2 Key Takeaways

- **Scalability**: The component-based design and well-structured folder hierarchy enable the system to scale efficiently as new features are added.
- **Maintainability**: Clear separation of concerns across presentation, logic, and data layers makes the codebase easier to understand and maintain.
- **User-Centric Design**: Responsive layouts and smooth state transitions offer an optimal experience across devices.
- **Deployment-Ready**: The frontend is production-ready and integrates seamlessly with backend APIs when deployed on IIS.
- **Secure and Resilient**: Proper validation, error handling, and authentication practices help in creating a secure and reliable user interface.

This documentation aims to serve as a valuable resource for current and future developers contributing to the CASFOS Comprehensive Database Management System, ensuring consistency, clarity, and quality in frontend development.

---

14. References & Appendix

14.1 External Links & Docs

1. **ReactJS Documentation**
   Official React documentation providing guides and API references.
   React Documentation
2. **Tailwind CSS Docs**
   Tailwind CSS is a utility-first CSS framework used to style the application.
   Tailwind CSS Documentation
3. **Axios GitHub Repo**
   Axios is the HTTP client used for making API requests.
   Axios GitHub Repository

4. **IIS Deployment Guide**
   A comprehensive guide for deploying applications on Internet Information Services (IIS).
   IIS Documentation

5. **React Router Docs**
   React Router allows for dynamic routing within a React application.
   React Router Documentation

6. **React Testing Library Docs**
   React Testing Library helps with testing React components.
   React Testing Library Documentation

7. **JWT Authentication**
   Understanding how JWT tokens work for securing APIs.
   JWT.io

8. **Socket.IO Documentation**
   Real-time web socket communication library (for future enhancements).
   Socket.IO Documentation

14.2 Additional Code Snippets

14.2.1 Protected Route Example

This code shows how to restrict access to certain routes based on the authentication state.

```
import { useContext } from "react";
import { Navigate } from "react-router-dom";
import { AuthContext } from "../context/AuthContext";

const PrivateRoute = ({ children }) => {
  const { user } = useContext(AuthContext);
  return user ? children : <Navigate to="/login" />;
};

export default PrivateRoute;
```

14.2.2 Creating a Custom Axios Instance

A custom Axios instance helps in centralizing HTTP request configurations such as headers or base URL.

```
import axios from "axios";

const axiosInstance = axios.create({
  baseURL: process.env.REACT_APP_API_URL,
});

axiosInstance.interceptors.request.use((config) => {
  const token = localStorage.getItem("token");
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }
  return config;
});
```

```
export default axiosInstance;
```

14.2.4 Error Handling in API Calls

This example shows how to handle errors globally for API calls using Axios.

```
import axiosInstance from "./axiosInstance";

export const AssetDataApi = {
  getComplaints: async () => {
    try {
      const response = await axiosInstance.get("/ Asset Entry");
      return response.data;
    } catch (error) {
      console.error("Error fetching complaints", error);
      throw error; // Ensure the error is thrown so it can be handled by the caller
    }
  },
  submitComplaint: async (AssetData) => {
    try {
      const response = await axiosInstance.post("/assets", AssetData);
      return response.data;
    } catch (error) {
      console.error("Error submitting data", error);
      throw error;
    }
  }
};
```

14.2.5 Using React Context API for Global State Management

The React Context API is used to manage global states like user authentication or session information.

```
// context/AuthContext.js
import { createContext, useState } from "react";

export const AuthContext = createContext();

export const AuthProvider = ({ children }) => {
  const [user, setUser] = useState(null);

  return (
    <AuthContext.Provider value={{ user, setUser }}>
      {children}
    </AuthContext.Provider>
  );
};
```

14.3 Tools & Libraries Used

- **ReactJS**: A declarative, efficient, and flexible JavaScript library for building user interfaces.
- **Tailwind CSS**: A utility-first CSS framework to create custom designs without leaving HTML.
- **Axios**: Promise-based HTTP client for the browser and Node.js, used for making API requests.
- **React Router**: A library for routing in React applications, enabling navigation between views.

- **React Context API**: A way to manage global state without the need for prop drilling.
- **React Testing Library**: A simple and complete testing library for React components.
- **JWT Authentication**: JSON Web Tokens used for secure API requests and user authentication.
- **Socket.IO**: A JavaScript library for real-time web applications (planned feature).
- **Jest**: JavaScript testing framework for unit and integration testing.
- Certainly! Below is the **Common Troubleshooting** section with code examples, presented in a different style for clarity and impact:

14.5 Common Troubleshooting Tips (with Code)

1. Issue: Axios Requests Failing with CORS Error

Problem: When making API calls using Axios, you may encounter a `CORS` (Cross-Origin Resource Sharing) error, indicating that the backend server is not allowing requests from the frontend.

Solution: Ensure that the server supports CORS. If the server does not allow cross-origin requests, configure CORS on the server side. Alternatively, during development, you can set up a proxy to avoid CORS issues.

Code Example:

```
// In your React App (Development Setup: Proxy in package.json)
"proxy": "http://localhost:5000",  // Point to your backend API

// Example of Axios request (frontend)
import axios from "axios";

const fetchComplaints = async () => {
  try {
    const response = await axios.get("/complaints"); // This will be proxied
    console.log(response.data);
  } catch (error) {
    console.error("Error fetching complaints", error);
  }
};
```

Server-Side Solution: In case the backend is using Node.js with Express, you can use the `cors` package to enable CORS:

```
// On the server-side (Node.js with Express)
const express = require('express');
const cors = require('cors');
const app = express();

// Allow all domains or specify origins
app.use(cors());

app.get('/complaints', (req, res) => {
  res.json([{ id: 1, description: "Complaint 1" }]);
});

app.listen(5000, () => {
  console.log('Server is running on http://localhost:5000');
```

```
});
```

2. Issue: State Not Updating on Component Re-Render

Problem: You may encounter a situation where the state is not updating as expected, causing the component to render stale or incorrect data.

Solution: Make sure you're using the state update function (`setState`) correctly. Avoid mutating the state directly. Always return a new copy of the state if you need to modify it.

Code Example:

```
// Incorrect: Direct mutation of the state
const [list, setList] = useState([1, 2, 3]);

const addItem = () => {
  list.push(4);  // Mutating state directly (bad practice)
  setList(list);
};

// Correct: Creating a new state object instead of mutating
const addItemCorrectly = () => {
  setList((prevList) => [...prevList, 4]);  // Correct approach
};
```

3. Issue: Tailwind CSS Classes Not Applying Correctly

Problem: Tailwind CSS classes may not be applied properly or might not be showing the desired results.

Solution: Ensure that you have correctly set up Tailwind CSS in your project. If you're using PostCSS, make sure that it processes the Tailwind classes. Additionally, check for typos or conflicting styles in your class names.

Code Example:

```
<!-- Example of incorrect Tailwind class usage -->
<div class="bg-color-red p-x-10">  <!-- Error: 'bg-color-red' is invalid -->
  This will not apply styles properly.
</div>

<!-- Corrected Tailwind class usage -->
<div class="bg-red-500 px-10">   <!-- Correct class names -->
  This will apply styles properly.
</div>
```

Configuration Check: Verify that your `tailwind.config.js` is set up properly and Tailwind is integrated correctly with PostCSS:

```
// tailwind.config.js
module.exports = {
  content: [
    "./src/**/*.{js,jsx,ts,tsx}", // Ensure Tailwind scans these files
  ],
  theme: {
    extend: {},
  },
  plugins: [],
};
```

4. Issue: React Components Not Rendering or Rendering with Old Data

Problem: React components might fail to re-render, or they may render outdated data due to issues with state or props.

Solution: Ensure that data passed as props to the component is updated correctly. Check if you're using the correct method to trigger re-renders, and avoid mutating state or props directly.

Code Example:

```
// Incorrect: Direct mutation of props or state
const ComplaintDetails = ({ complaint }) => {
  // Mutating props directly (bad practice)
  complaint.status = 'Resolved';
  return <div>{complaint.status}</div>;
};

// Correct: Use a state update to trigger re-render
const ComplaintDetails = ({ complaint }) => {
  const [status, setStatus] = useState(complaint.status);

  const handleResolve = () => {
    setStatus('Resolved'); // Correct way to update the state
  };

  return (
    <div>
      <p>{status}</p>
      <button onClick={handleResolve}>Resolve</button>
    </div>
  );
};
```

5. Issue: React Hook Form Validation Not Working

Problem: Form validation might not work correctly if the form data or validation rules are not set up correctly.

Solution: Ensure that the validation rules are correctly defined using `React Hook Form`. If necessary, make use of `resolver` to integrate third-party validation libraries like `Yup`.

Code Example:

```
// Using React Hook Form with validation
import { useForm } from "react-hook-form";

const ComplaintForm = () => {
  const { register, handleSubmit, formState: { errors } } = useForm();

  const onSubmit = (data) => {
    console.log("Complaint submitted:", data);
  };

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <textarea
        {...register("complaint", { required: "Complaint description is required" })}
      />
      {errors.complaint && <span>{errors.complaint.message}</span>}

      <button type="submit">Submit</button>
    </form>
```

```
  );
};
```

6. Issue: App Not Updating After Login (State Persistence)

Problem: After a user logs in, the app may not update to reflect the logged-in user or persist the user session.

Solution: Use React Context or local storage to persist user authentication state across page reloads. Ensure that session data is correctly stored and retrieved.

Code Example:

```
// Using React Context for session management
import { createContext, useState, useContext } from "react";

// Create a context for user authentication
const AuthContext = createContext();

export const AuthProvider = ({ children }) => {
  const [user, setUser] = useState(null);

  const loginUser = (userData) => {
    setUser(userData);
    localStorage.setItem("user", JSON.stringify(userData)); // Persist data
  };

  const logoutUser = () => {
    setUser(null);
    localStorage.removeItem("user");
  };

  return (
    <AuthContext.Provider value={{ user, loginUser, logoutUser }}>
      {children}
    </AuthContext.Provider>
  );
};

export const useAuth = () => useContext(AuthContext);

// In your components, use the `useAuth` hook to access user data
```

This approach uses different formatting to ensure clarity, with code examples clearly separated by issues and solutions. Each troubleshooting step includes a solution with code snippets to guide the reader through resolving common problems in a structured and easy-to-follow way.

NOTES: