1. INTRODUCTION

1.1 PROJECT DESCRIPTION:

The project titled **"Ground Control Station for CanSats"** is a full-stack web application designed to simulate a real-world satellite ground control station for monitoring and managing CanSat telemetry data. CanSats, or miniature satellites packaged in the shape of a soft drink can, are widely used in academic and experimental aerospace missions due to their low cost and accessibility. This project aims to provide a centralized, web-based platform that enables users to monitor real-time telemetry data, manage user access securely, and visualize mission-critical parameters through an interactive and responsive interface.

The backend system is built using **Node.js with the Express framework**, adopting a RESTful API architecture that ensures scalability and modularity. It integrates **PostgreSQL with Sequelize ORM** for efficient database management of telemetry and user data. Advanced security measures such as **JWT-based authentication**, **bcrypt password hashing**, and **rate limiting** protect the system from unauthorized access and abuse. A key feature is the implementation of **TOTP-based two-factor authentication (2FA)** using Speakeasy and QR Code libraries, providing an additional layer of security via Google Authenticator during user login.

The frontend is developed using **React.js**, structured as a Single Page Application (SPA) with **React Router** handling client-side routing and protected routes. Users are assigned roles (admin, operator, viewer), and the dashboard dynamically adjusts access based on these roles. Once authenticated, users can access a telemetry dashboard that displays real-time data streamed via **Socket.IO**. This includes vital parameters such as temperature, pressure, altitude, latitude, longitude, and timestamps, all visualized using interactive charts and tables. Additional features such as dark mode, CSV export, and datetime-based filtering enhance usability and data analysis.

In summary, this project provides a comprehensive simulation of a CanSat ground control station, combining best practices in modern web development, cybersecurity, and real-time data processing. It is designed to be educational, scalable, and extensible for future CanSat missions or similar IoT-based telemetry systems.

1.2 PURPOSE:

The primary purpose of this project is to develop a secure, real-time, and user-friendly ground control system tailored for managing and monitoring CanSat missions. CanSats are increasingly used in aerospace education and experimental missions, requiring a robust platform to handle telemetry data, streamline mission operations, and provide clear situational awareness to users. This system addresses these needs by integrating modern web technologies, secure authentication protocols, and real-time communication tools. Onboard AI classification, reducing the need to transmit vast volumes of raw image data.

Secure user access and role management, ensuring that only authorized users such as admins, operators, or viewers can access sensitive telemetry data and perform mission-critical actions.

Real-time telemetry data streaming, enabling immediate visualization and analysis of temperature, pressure, altitude, and GPS coordinates from CanSats during flight.

Interactive dashboard and analytics tools, providing mission planners and engineers with clear data insights through charts, filtering tools, and CSV export features.

Responsive and accessible web interface, allowing users to access the platform from any device with consistent performance and an intuitive user experience.

By integrating advanced security features, real-time data handling, and interactive visualization, this project promotes a hands-on understanding of satellite telemetry management and serves as a valuable prototype for future aerospace applications. It aligns with modern aerospace mission goals focused on reliability, autonomy, and efficient data operations in compact satellite systems.

1.3 SCOPE:

This project covers a broad range of full-stack development components and real-time data processing functionalities tailored to simulate a real-world CanSat ground control station. It brings together aspects of user authentication, live data streaming, secure backend services, and dynamic frontend visualization:

User Authentication & Role Management: Implementation of secure user login and signup with JWT authentication, TOTP-based two-factor authentication, and role-based access control (admin, operator, viewer).

Real-Time Telemetry Streaming: Use of Socket.IO to transmit live CanSat telemetry data (temperature, pressure, altitude, latitude, longitude, timestamp) from the backend to the dashboard.

RESTful API Architecture: Modular Node.js backend with Express providing endpoints for telemetry data handling, user management, filtering operations, and system configuration.

Database Modeling: PostgreSQL database integrated via Sequelize ORM to manage structured data storage for users, telemetry entries, authentication tokens, and access logs.

Interactive Frontend Dashboard: React-based Single Page Application (SPA) with routing and protected components, offering charts, tables, CSV export, and date/time filtering for telemetry visualization.

Security and Rate Limiting: Incorporation of bcrypt for password hashing, express-rate-limit for brute force protection, CORS middleware, and email-based OTP for password recovery.

Responsive and Extendable UI: Clean and mobile-friendly UI with dark mode toggle, session management, and scope for future integration of map overlays or additional telemetry metrics.

1.4 OBJECTIVES:

The key objectives of this project are as follows:

- To develop a secure and modular full-stack application that simulates the functionalities of a real CanSat Ground Control Station, including telemetry monitoring, user access control, and administrative functions.
- To implement a robust user authentication system using JWT tokens and TOTP-based two-factor authentication for enhanced security and role-based access control (admin, operator, viewer).

- To enable real-time telemetry streaming and visualization by integrating Socket.IO for live updates of CanSat data such as temperature, pressure, altitude, latitude, longitude, and timestamp on a dynamic dashboard.
- To design an efficient, RESTful backend architecture using Node.js and Express, providing endpoints for telemetry retrieval, user registration/login, role verification, and password reset via email-based OTPs.
- To create a normalized relational database schema using PostgreSQL and Sequelize ORM for managing users, telemetry logs, TOTP secrets, password reset tokens, and role permissions.
- To offer a responsive and user-friendly frontend built with React.js and React Router, enabling seamless navigation, protected routes, and real-time data rendering in chart and table formats.
- To provide advanced telemetry filtering features such as datetime range selection, ascending/descending sorting, pagination, CSV downloads, and a dark mode toggle for improved user experience.
- To ensure data integrity and communication security using bcrypt for password hashing, express-rate-limit for brute-force protection, and CORS middleware for controlled frontend-backend interaction.
- To maintain code modularity and scalability, supporting future additions such as map-based visualizations, GPS tracking overlays, or AI-based telemetry anomaly detection.
- To follow best development practices including environment separation, reusable components, consistent API versioning, and structured error handling for long-term maintainability.

2. LITERATURE SURVEY

2.1 RESEARCH PAPERS AND INSIGHTS:

Recent advancements in microcontrollers, wireless telemetry, and software-defined radio (SDR) have significantly improved the capabilities of Ground Control Stations (GCS) for small-scale satellite systems such as CanSats and CubeSats. Key research works and technological insights include:

1. A. Sharma et al. (2021) developed a low-cost CanSat system that used GPS and sensor data transmitted over RF modules. Their GCS implemented real-time plotting and data logging using Python and GUI libraries like Tkinter.
2. X. Zhou et al. (2022) proposed an SDR-based telemetry framework that increased data transmission reliability for miniaturized satellites, demonstrating better signal-to-noise ratios and adaptability to environmental noise.
3. M. Kaya & Öztürk (2020) emphasized the importance of integrating real-time data visualization and alert mechanisms in CanSat ground stations, especially for competitions or environmental monitoring.
4. IEEE Conference (2023) proceedings showed the trend of using **WebSocket /Socket.IO** for real-time telemetry delivery and dashboards using **Node.js** and **React.js**, improving the interactivity and responsiveness of modern GCS implementations.
5. P. Lee, J. Zhang, and R. Thomas, Studies on IoT and embedded systems reveal that **MQTT and HTTP REST APIs** are gaining traction as communication protocols in telemetry streaming, offering lightweight and scalable solutions for real-time data handling.

RESEARCH GAP:

Despite progress, current GCS implementations for CanSats exhibit several limitations:

Limited Real-Time Interactivity

- Many systems use serial/USB-only interfaces with static plotting.
- Lacks web sockets for live telemetry streaming.

No User Authentication or Role Management

Existing GCS software lacks secure login, multi-user access, or admin controls for team-based operations.

No Historical Data Visualization

Most systems only show live data; historical telemetry records are not stored or accessible via queryable databases.

Poor Modularity and Extensibility

Lack of RESTful APIs or component-based architecture limits integration with external systems or mobile apps.

Absence of Alerting, Filtering, and Analytics

No smart filters or event detection to highlight anomalies like sudden altitude drops, signal loss, or sensor failure.

2.2 EXISTING SYSTEMS:

Several GCS implementations exist for educational, experimental, and competitive purposes. However, they vary in terms of interactivity, robustness, and scalability:

ArduStation GCS (by Arduino community): A simple serial-based interface used with GPS and barometer modules. It lacks internet-based access, multi-user support, or data persistence.

NASA CanSat Competition GCS: Typically implemented using ground software written in Processing or Python for plotting. Many are one-off, offline tools and lack database or user authentication support.

HabHub & UKHAS GCS: A high-altitude balloon telemetry system that uses LoRaWAN and maps for public tracking, but not designed for custom, secure CanSat missions.

SatNOGS (by Libre Space Foundation): A modular GCS for amateur satellites, offering automation and global ground station networking. It uses SDR and web UI but is complex for small projects like CanSats.

Most existing systems do not support **role-based access control**, **database-backed telemetry history**, or **modular APIs for telemetry and user management**

PROPOSED APPROACH:

To address these gaps, this project proposes a **web-based Ground Control Station (GCS)** platform with the following key innovations:

Real-Time Telemetry Streaming

Uses **Socket.IO** for pushing live telemetry (temperature, pressure, altitude, GPS, timestamp) to the frontend dashboard.

Role-Based User Authentication

- Implements **JWT-based login/signup**, with roles such as Admin, Engineer, and Observer.
- Uses **PostgreSQL** for user data and permission tracking.

Telemetry Database and Query System

- Stores telemetry in a normalized SQL schema.
- Enables time-range filtering, sorting, and export as CSV.

Responsive, Interactive Dashboard

- Built with **React.js** frontend and **Node.js/Express** backend.
- Displays charts (e.g., line graphs of temperature, altitude) and a data table.

Security and Session Control

- Includes logout, 2FA options (TOTP/Google Authenticator), and session management.
- Features a "Remember this device" toggle.

Modular Architecture

- REST APIs for telemetry, authentication, and user management.
- Frontend and backend can be independently deployed or containerized via Docker.

2.3 REAL WORLD USE CASES:

The proposed GCS can support multiple real-world CanSat mission scenarios:

Educational CanSat Competitions: Secure login for student teams, real-time scoring data, and instructor access.

Environmental Monitoring: Track altitude, temperature, and pressure changes in real time; log data for long-term research.

Disaster Simulation & Response: Simulate telemetry streams from CanSats used in forest fire or flood zone monitoring.

Telemetry for Agricultural Payloads: Monitor environmental conditions using CanSat payloads for crop or air quality analysis.

University Projects & Research: Enable reproducible data collection, peer access, and secure remote monitoring for academic CanSat projects.

TECHNOLOGICAL TRENDS SUPPORTING THIS SHIFT:

Modern web technologies and embedded platforms are now making scalable GCS software feasible even for small teams:

Socket.IO, WebSockets: Enable low-latency, bi-directional telemetry updates.

React.js & Charting Libraries: Provide real-time, interactive data visualizations with responsive design.

PostgreSQL + Sequelize ORM: Offers robust, queryable storage for telemetry and users.

OAuth2 + TOTP 2FA: Adds secure multi-user access suitable for educational or sensitive deployments.

Docker & Nginx: Allow containerization and deployment on low-cost servers or Raspberry Pi clusters.

3. SYSTEM SPECIFICATIONS

The project “GROUND Control Station for CanSats” is a full-stack web application designed to simulate and manage CanSat missions in real time using secure telemetry processing, interactive dashboards, and modular backend services. It includes multiple integrated modules such as user authentication, telemetry data visualization, socket-based real-time communication, and role-based control mechanisms.

This system is architected to provide a comprehensive ground control interface where users can securely log in, monitor CanSat telemetry, analyze live data through charts and tables, and manage user roles and permissions. The backend ensures robust API-driven data handling, while the frontend offers intuitive controls and visualization tools for mission planners and operators.

The specifications are broadly categorized into:

- Hardware Requirements
- Software Requirements
- Technology Stack
- External Tools & Libraries
- Data Sources

3.1 HARDWARE REQUIREMENTS:

Component		Specification
Processor	:	Intel Core i5, 8th Gen or higher
RAM	:	8 GB (minimum), 16 GB recommended
Storage	:	25 GB SSD minimum
Internet Bandwidth	:	Stable connection with > 20 Mbps
Operating System	:	Windows 10 or higher

The system was developed and tested on a standard development laptop using a real-time telemetry simulation module. While no GPU is strictly required, a stable network and a multi-core processor are essential to support Socket.IO-based real-time data transfer, PostgreSQL-based storage, and live data visualization through the React frontend.

3.2 SOFTWARE REQUIREMENTS:

Software		Version / Details
Node.js	:	Version 18.x (Backend server environment)
Express.js	:	Version 4.x (REST API backend framework)
PostgreSQL	:	Version 14.x (Relational database)
Sequelize ORM	:	For PostgreSQL database modeling and queries
Socket.IO	:	Real-time communication (WebSocket integration)
React.js	:	Version 18.x (Frontend framework)
Chart.js	:	For live telemetry data charting
Recharts	:	For responsive and interactive telemetry graphs
HTML/ CSS3/ Bootstrap	:	UI design and layout styling
Bcrypt / jsonwebtoken	:	For password hashing and secure user authentication
Nodemailer	:	For sending OTP-based reset password emails

TECHNOLOGY STACK

Backend

Language	:	JavaScript (Node.js)
Framework	:	Express.js
Database	:	PostgreSQL + Sequelize ORM
API Protocols	:	REST + Websocket(Socket.IO)
Security	:	JWT Auth, CORS-enabled, bcrypt password hashing

Real-Time Communication

• Library	:	Socket.IO
• Purpose	:	Streaming telemetry data to the frontend in real-time

Frontend

• Language / Framework	:	React.js + JSX
• UI Design	:	Tailwind CSS, HTML5, CSS3
• Visualization	:	Chart.js & Recharts (telemetry graphs)
• User Interaction	:	Responsive dashboard with role-based UI rendering

Utilities

• Authentication	:	JWT tokens with role-based access
• 2FA	:	Google Authenticator (TOTP using Speakeasy)
• Email Services	:	OTP-based password reset via Nodemailer

EXTERNAL TOOLS AND LIBRARIES

Library / Tool		**Purpose**
• Socket.IO	:	Real-time bidirectional communication between server and client for telemetry
• Sequelize	:	Object-Relational Mapping (ORM) for PostgreSQL database
• QRCode / Speakeasy	:	TOTP-based two-factor authentication (2FA) using Google Authenticator
• Nodemailer	:	Sending OTP emails for password reset and user verification
• Chart.js / Recharts	:	Interactive charting and graphing for telemetry data
• Tailwind CSS	:	Modern, utility-first CSS framework for responsive design

• React Icons / Lucide React	:	Icon libraries used for user interface enhancements
• Framer Motion	:	Animation library for smooth UI transitions
• VS Code	:	Development environment used for coding and testing
• pgAdmin	:	PostgreSQL database GUI for querying and managing data

DATA SOURCES:

The system utilizes both real-time telemetry and synthetic test data sources to simulate and manage CanSat operations:

Simulated Telemetry Streams: Synthetic real-time data for temperature, pressure, altitude, latitude, longitude, and timestamps generated using backend logic and Socket.IO.

PostgreSQL Database Records: Persistent storage of all telemetry logs, user credentials, roles, 2FA status, and session history.

User-Provided Data: Telemetry packets or image uploads shared by users for offline analysis or testing visualization/chart features.

Time Zone-Aware Timestamps: Stored with offset information (e.g., UTC+05:30) and normalized for backend querying and frontend filtering.

Historical Replay Dataset: Pre-recorded telemetry sessions used for dashboard simulation and regression testing of features like filters, charts, and CSV export.

4. SOFTWARE REQUIREMENTS SPECIFICATION

The purpose of this document is to define the Software Requirements Specification (SRS) for the project titled **“Ground Control Station for CanSats.”** This document outlines the functional and non-functional requirements of the system, including interactions, capabilities, technical architecture, and software standards. It serves as a blueprint for the development, testing, and deployment phases of the project, and acts as a reference for developers, testers, and project stakeholders.

4.1 PRODUCT PERSPECTIVE

The *Ground Control Station for CanSats* is a web-based telemetry monitoring system designed to provide real-time data visualization and secure user access for educational or research-based CanSat missions. It functions as a standalone platform that connects to live or simulated CanSat telemetry feeds via Socket.IO. The system replaces traditional serial-based desktop tools by offering a modern, browser-accessible interface built with React.js and Tailwind CSS. It enables users to interact with live sensor data (temperature, pressure, altitude, GPS) through interactive charts and tables, enhancing accessibility, usability, and data analysis.

The architecture follows a modular three-tier structure—frontend, backend, and database layers—ensuring scalability and maintainability. The backend, developed in Node.js with Express.js and Sequelize ORM, manages user authentication, session control, data filtering, and role-based access control. PostgreSQL serves as the persistent storage layer for telemetry data and user information. This flexible architecture allows for future integration with additional services like offline data sync, mission replay, and CanSat command modules, making it adaptable for larger satellite ground station implementations.

SYSTEM OVERVIEW:

The system is a web-based **Ground Control Station (GCS)** for real-time telemetry monitoring and control of CanSat missions. It provides an interactive dashboard that displays live sensor data received from CanSats and offers advanced user management, data filtering, and visualization functionalities. Key capabilities of the system include:

- Real-time telemetry data reception via **Socket.IO**.
- Interactive 2D dashboard with charts and tables showing:
  - Temperature
  - Pressure
  - Altitude
  - Latitude & Longitude
  - Timestamp
- Secure user registration, login, and **role-based access control**.
- Two-factor authentication (2FA) using **TOTP/Google Authenticator**.
- Database-backed **telemetry history** storage and retrieval using PostgreSQL.
- Filtering and sorting telemetry data by date/time range.
- Exporting telemetry data to CSV.
- User interface built using **React.js** and **Tailwind CSS**.
- RESTful APIs using **Node.js + Express.js**, with Sequelize ORM.
- Future extensibility for mission event alerts, CanSat control commands, and offline data upload.

The architecture is modular and scalable, supporting both real-time and historical data views for multiple user roles.

USERS OF THE SYSTEM:

Table No 4.1.1: Users of The System

User Type	Description
Administrator	Has full access to all system modules. Can create, update, or delete users. Can view all telemetry data, manage roles, and audit user actions.
Engineer	Can log in, view real-time telemetry, filter and download data, and upload mission data from offline sources.
Observer	Read-only user role. Can view the dashboard, charts, and telemetry tables but cannot modify or download any data.

4.2 FUNCTIONAL REQUIREMENTS

User Registration and Authentication

- Users can securely register, login, and logout.
- Passwords are hashed using **bcrypt**

Two-Factor Authentication

- Users can enable **TOTP-based 2FA** using apps like Google Authenticator.
- QR code is displayed during first login.

Real-time CanSat Telemetry Display

Receives telemetry data using **Socket.IO**, updates dashboard charts and tables in real-time.

Telemetry Data Storage

Stores all incoming telemetry (temperature, pressure, altitude, coordinates, timestamp) into a **PostgreSQL** database.

Historical Telemetry Filtering

Users can filter data by **date and time range** using datetime inputs, and sort in ascending/descending order.

Telemetry Data Export

Filtered data can be **downloaded as CSV** for further analysis

Dashboard Visualization

Real-time charts (line graphs) and interactive tables are rendered using **Chart.js** or **Recharts**.

User Role Management

Admins can add, delete, or modify users and assign roles (Admin, Engineer, Observer).

Secure Session Management

JWT-based tokens are used for session control; logout clears both local and session storage.

Responsive Frontend UI

Built with **React.js + Tailwind CSS**, works on desktops, tablets, and mobiles.

4.3 NON-FUNCTIONAL REQUIREMENTS

Performance: Real-time dashboard updates < 1 second latency.

Security: Uses bcrypt for password hashing, JWT for sessions, HTTPS, and TOTP for 2FA.

Scalability: Modular backend using Express.js and Sequelize allows easy API expansion.

Usability: Clean, responsive dashboard layout with real-time visual feedback.

Portability: Deployable on any platform supporting Node.js and PostgreSQL.

Reliability: Uses WebSocket error handling and reconnection logic to ensure persistent data flow.

SYSTEM ARCHITECTURE

The system is composed of the following major modules:

Frontend Client (React + Tailwind CSS + Chart.js)

- Manages UI components, chart rendering, filtering UI, and WebSocket events.
- Responsive design supports various screen sizes.
- Displays telemetry, login/signup, and admin pages.

Backend Server (Node.js + Express + Sequelize ORM)

- Exposes REST APIs for auth, user management, and telemetry filtering.
- Manages JWT tokens, TOTP verification, and email-based OTP (for password reset).

- Handles real-time data ingestion and socket communication.

Real-Time Telemetry Handler (Socket.IO)

- Receives telemetry data from CanSat or simulator and pushes updates to connected clients.
- Performs input validation and real-time broadcasting.

Database Layer (PostgreSQL with Sequelize)

- Stores users, roles, telemetry data, and login session history.
- Supports efficient querying and filtering with datetime fields.

5. SYSTEM DESIGN

System design is the blueprint for implementing the "Ground Control Station for CanSats" software system. This section provides a detailed overview of the high-level architecture, component interactions, data flow, and database entity relationships that guide the system’s development and integration.

The design adopts a **modular, layered architecture** to support **real-time telemetry**, **secure user access**, **role management**, and **visual data analysis**. Each component—frontend, backend, WebSocket server, and database—is clearly separated by function and communicates via **REST APIs or WebSocket protocols** to ensure scalability, maintainability, and reusability.

5.1 SYSTEM ARCHITECTURE

The system architecture is **three-tiered**, consisting of:

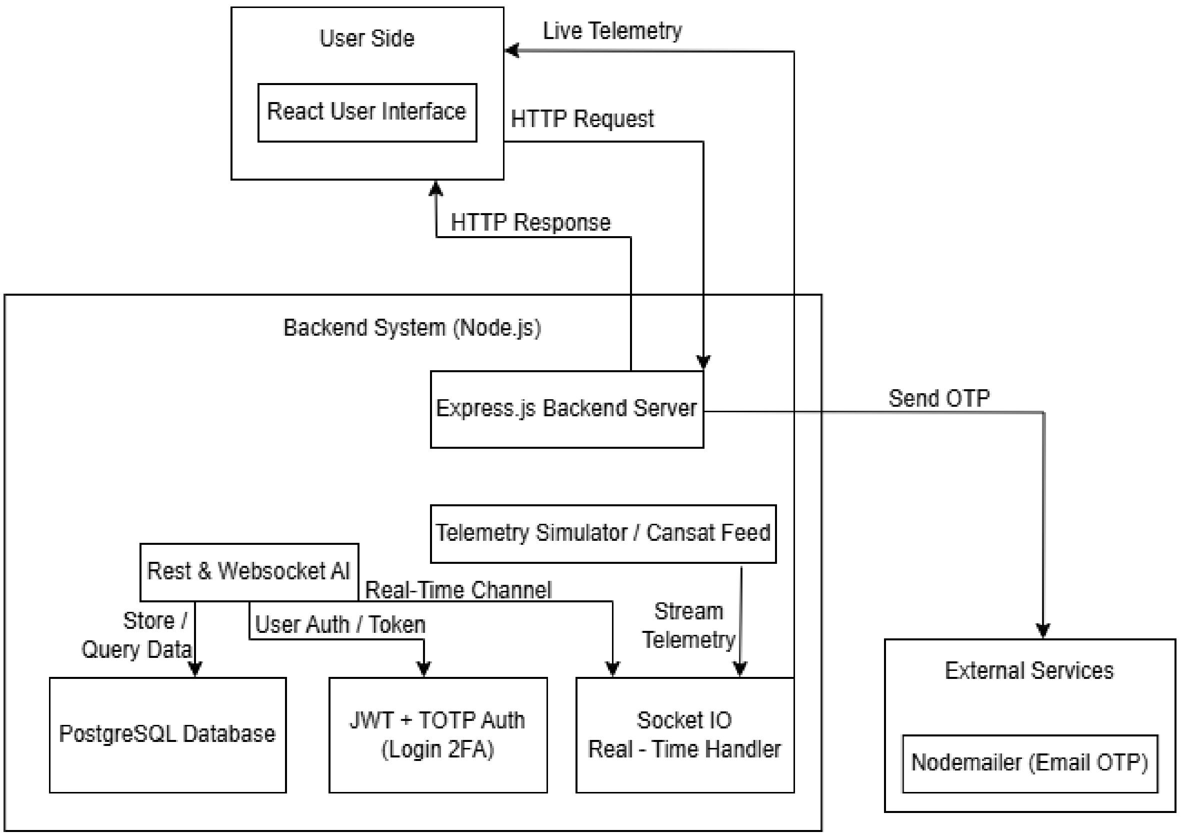


Figure No 5.1: System Architecture

Presentation Layer (Frontend)

- Built using React.js, Tailwind CSS, and Socket.IO for a dynamic single-page application.
- Users can register, log in, view real-time telemetry, and filter or download data.
- Provides responsive UI with chart visualizations, dark mode toggle, and secure TOTP setup.

Application Layer (Backend)

- Developed with Node.js and Express.js to handle all API, auth, and socket logic.
- Manages user registration, login, TOTP-based 2FA, and telemetry data processing.
- Supports REST and WebSocket APIs, email OTP for password reset, and secure token handling.

Data Layer (Database + Real-Time Input)

- Utilizes PostgreSQL with Sequelize ORM to manage structured data.
- Stores user info, TOTP secrets, telemetry records, and OTP history.
- Receives and logs real-time telemetry from a CanSat simulator or actual feed.

CONTEXT FLOW DIAGRAM (DFD Level 0)

The Context Flow Diagram represents the entire system as a single process with interactions from external entities.

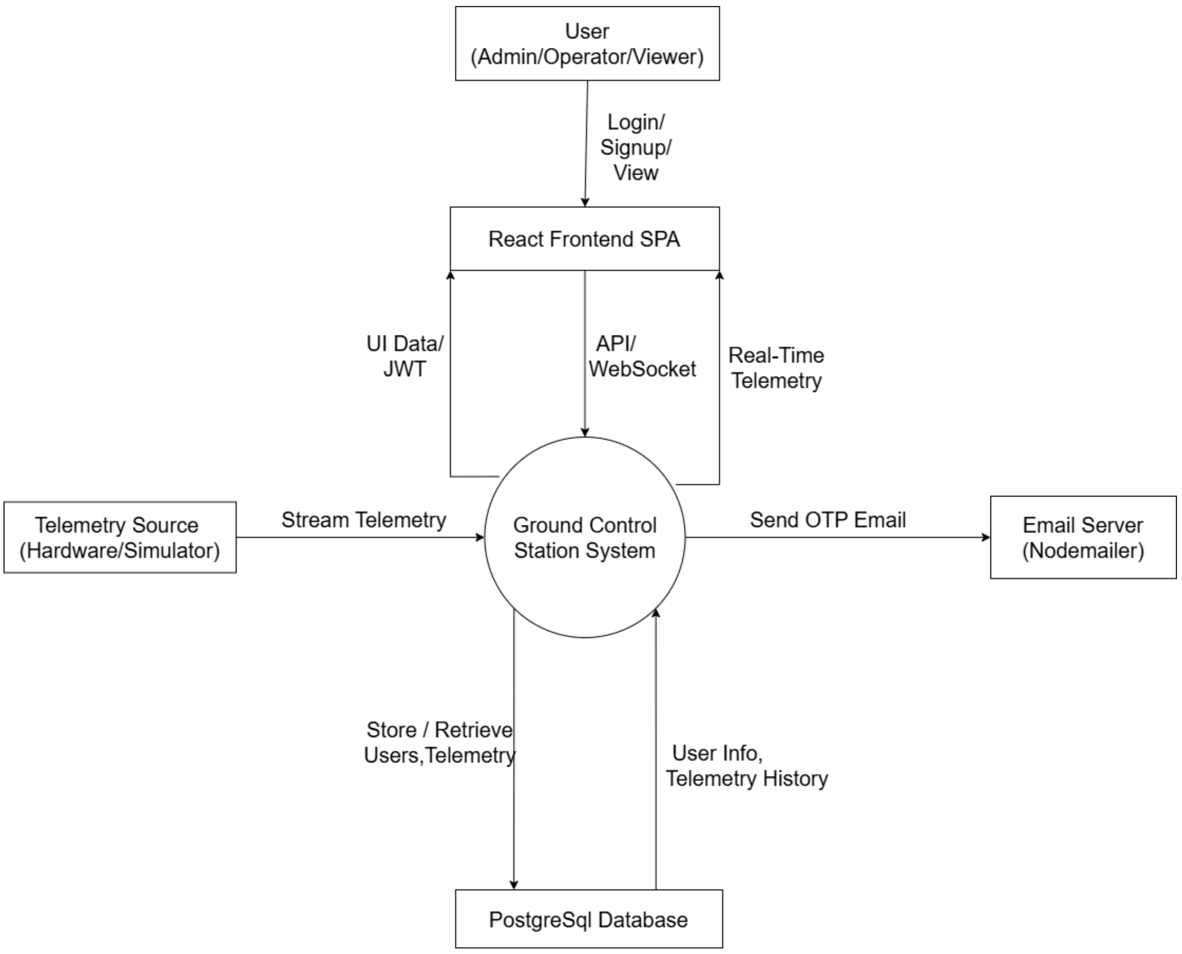


Figure No 5.2: Context Flow Diagram

5.2 DATA FLOW DIAGRAM

- A Data Flow Diagram (DFD) is a graphical representation of a system or subsystem that depicts how data moves through the system, including inputs, outputs, data stores, and processes.
- In the **Ground Control Station for CanSat**, the DFD models how telemetry data, user interactions, and system processes interact and transform data between the CanSat, users, and the database.
- DFDs are crucial for visualizing the system's internal and external components and

how data flows between them. This helps stakeholders understand the architecture and operations involved.

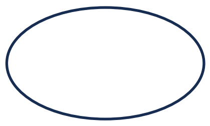
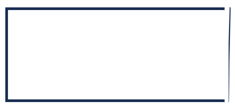
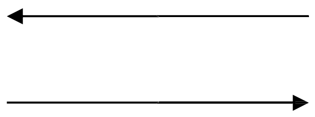
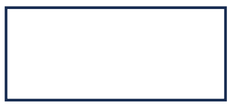
- The DFD captures telemetry flow from CanSat to Ground Station and through processes such as authentication, data storage, live visualization, and alerts.
- DFDs are also referred to as *Bubble Charts* or *Data Flow Graphs*. They can be represented at multiple levels of abstraction—from context-level (Level 0) to detailed levels (Level 1 and Level 2) that reveal finer functional breakdowns.

Rules for DFD construction

- A process cannot have only outputs.
- A process cannot have only inputs.
- Inputs to a process must be sufficient to produce the outputs.
- All data stores must be connected to at least one process.
- All data stores must be connected to a source or sink.
- A data flow can only have **one direction**; multiple flows must use separate arrows.
- If identical data flows to different destinations, it should be represented using a forked arrow.
- Data cannot flow directly back into the same process from which it came.
- All data flows must be named using **noun phrases**.

DFD Symbols

Table No 5.2.1: DFD Symbols

Name	**Notation**	**Description**
Process		Represents a function or activity that transforms input data into output data.
Data Store		Represents a repository for data that can be read from or written to.
Dataflows		Data flows are pipelines through which packets of information flow. Label the arrows with the name of the data that moves through it.
External Entity		External entities are objects outside the system with which the system communicates. External Entities are sources and destinations of the systems inputs andoutputs

DFD Level 1

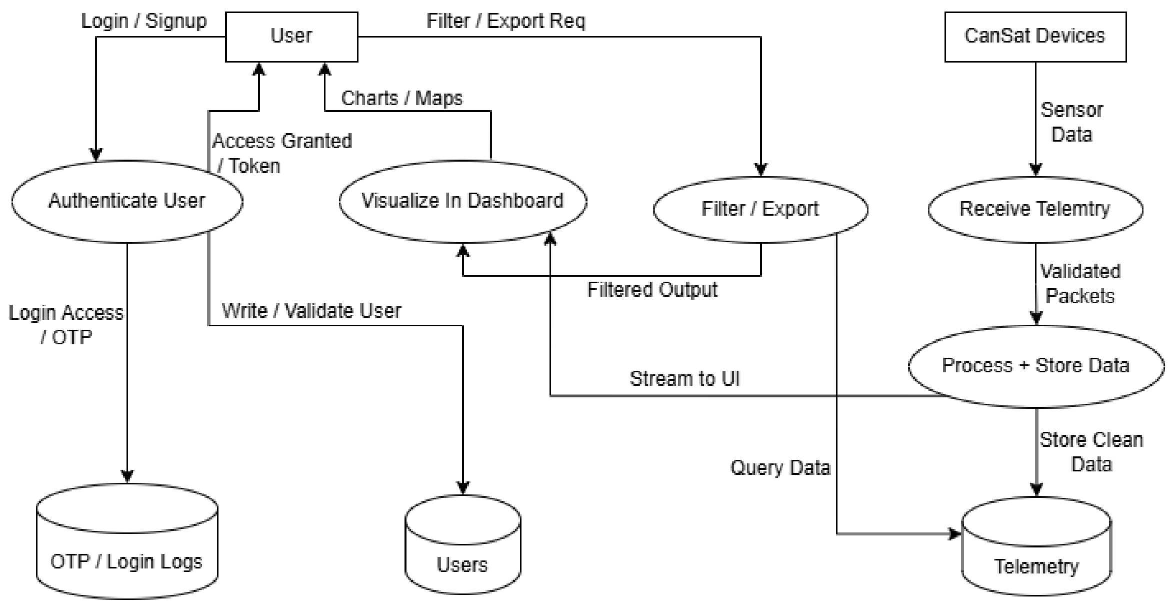


Figure No 5.2.2: General

DFD Level 2 – User Authentication & Role Management

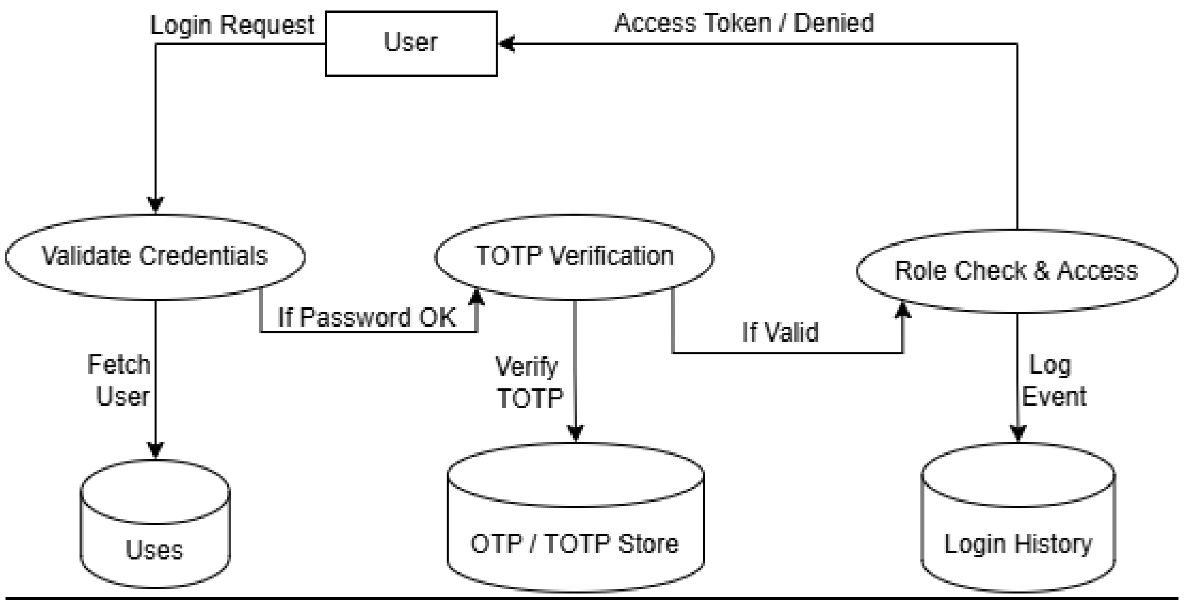


Figure No 5.2.3: User Authentication & Role Management

DFD Level 2 – CanSat Telemetry Acquisition

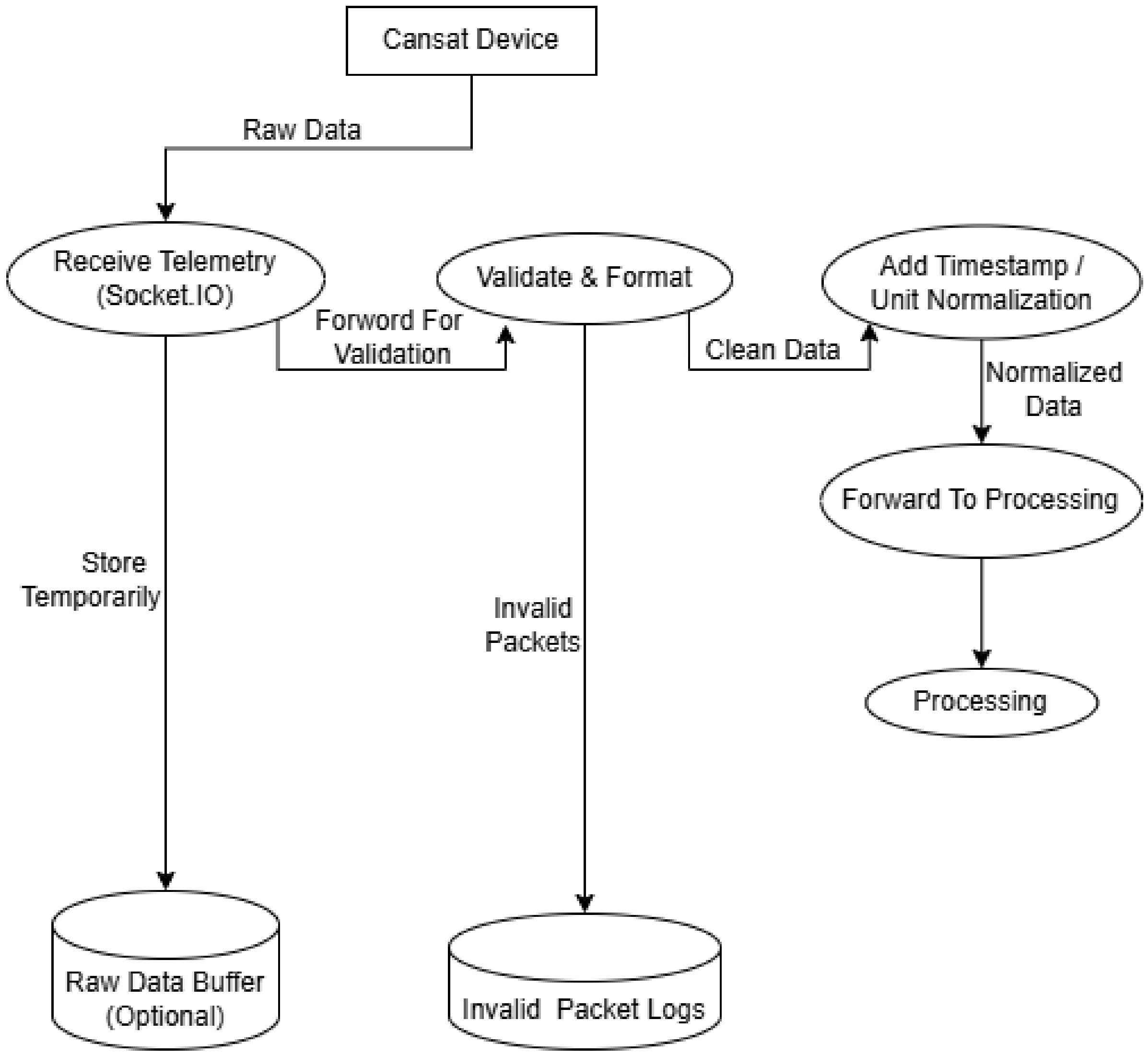


Figure No 5.2.4: CanSat Telemetry Acquisition

DFD Level 2 – Data Processing & Storage

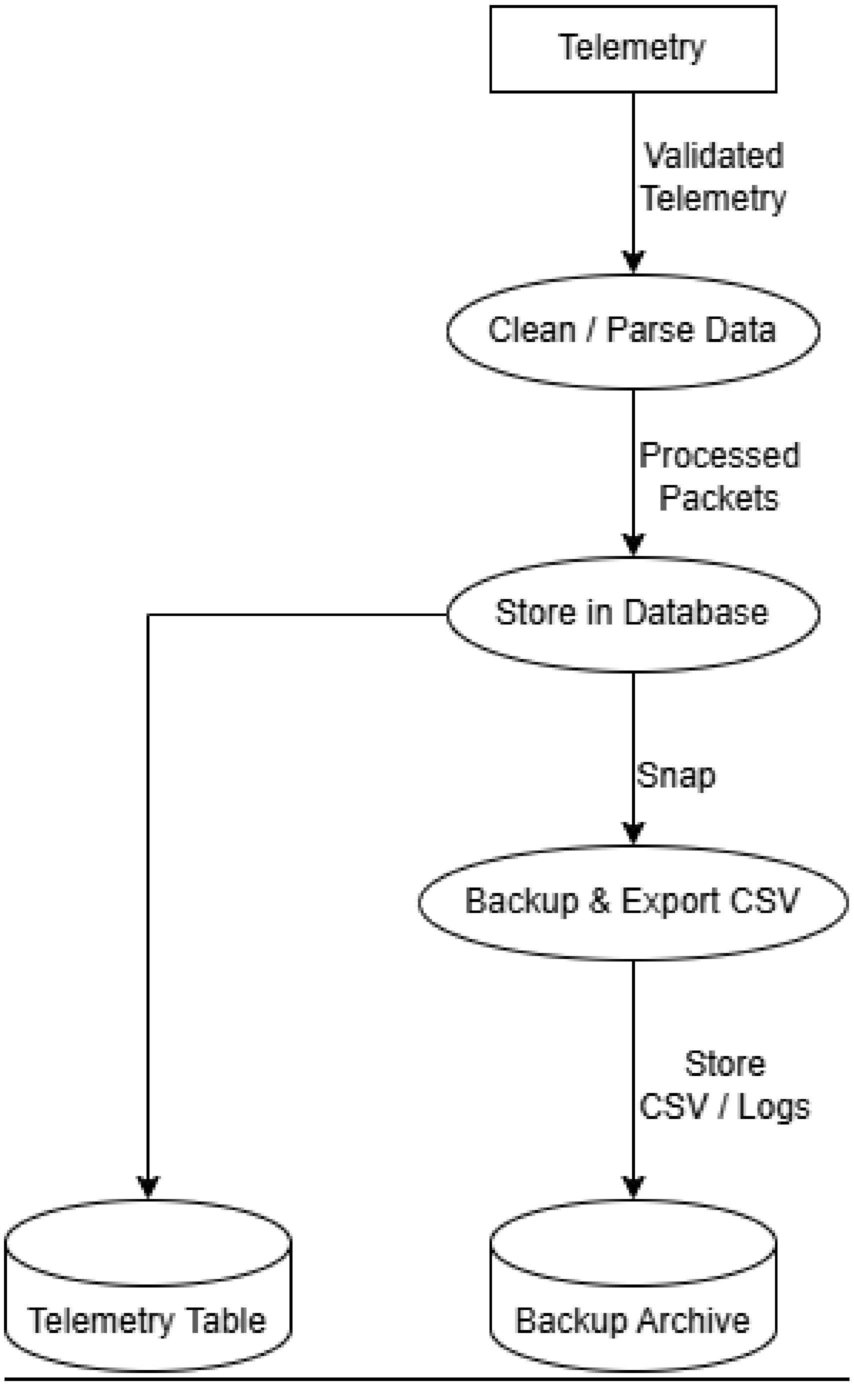


Figure No 5.2.5: Data Processing & Storage

DFD Level 2 – Real-Time Visualization & UI/UX

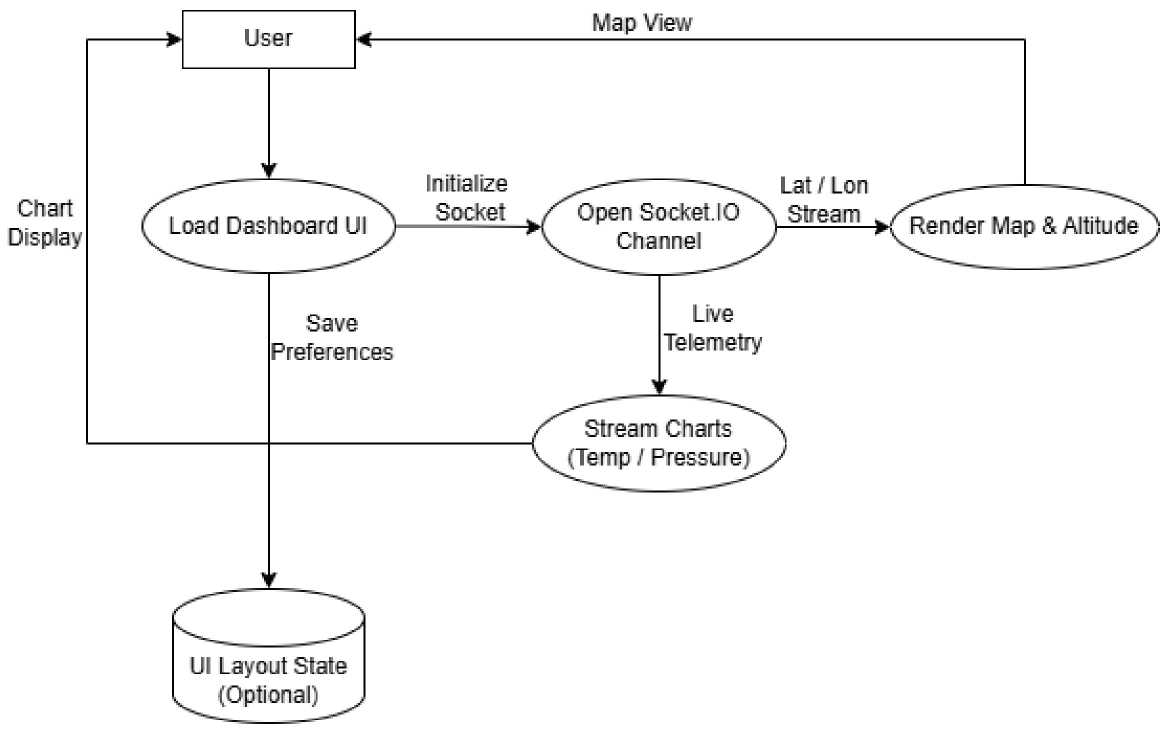


Figure No 5.2.6: Real-Time Visualization & UI/UX

DFD Level 2 – Data Filtering & Analysis

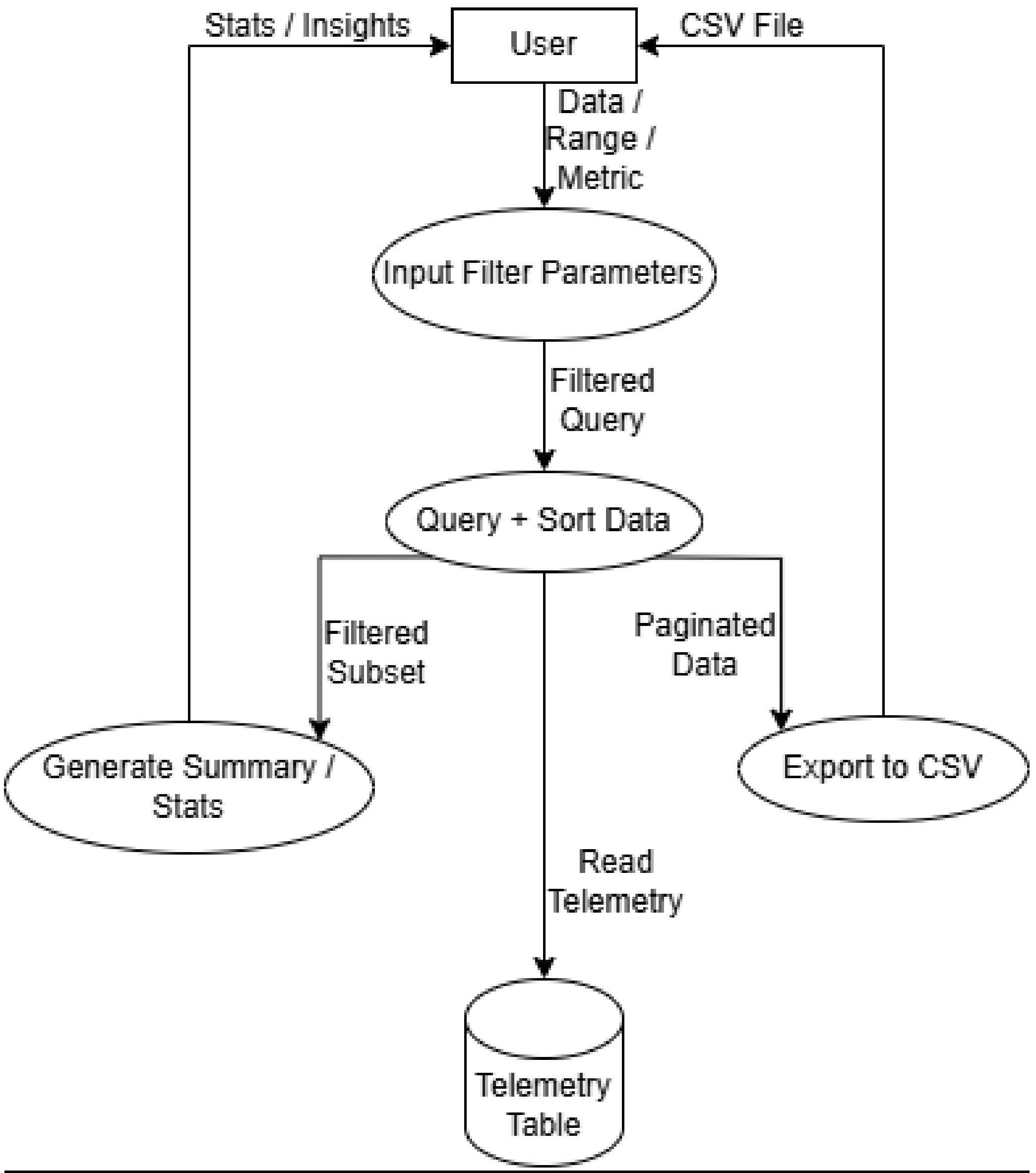


Figure No 5.2.7: Data Filtering & Analysis

6. DATABASE DESIGN

The database is a core component of the Ground Control Station for CanSats project. It acts as the central repository for user information, telemetry data, authentication tokens, and activity logs.

PostgreSQL is used with Sequelize ORM to ensure efficient querying, relational integrity, and support for real-time updates and historical telemetry analysis.

6.1 OBJECTIVES OF DATABASE DESIGN

- To store telemetry data (temperature, pressure, altitude, etc.) from CanSats in real time.
- To securely manage user data, roles, and multi-factor authentication credentials.
- To enable efficient data retrieval for charts, filters, and download features.
- To track OTPs for password reset and QR code secrets for TOTP 2FA setup.
- To maintain a log of login history, device tracking, and user activity for auditability.

DATABASE SCHEMA OVERVIEW

The system uses the following key tables:

1. **users** – Stores user credentials, roles, and TOTP secret keys.
2. **telemetry** – Records CanSat telemetry data: temperature, pressure, GPS, and timestamp.
3. **otps** – Temporarily stores OTP codes sent via email for password reset.
4. **devices** – (Optional) Tracks trusted devices if "Remember This Device" is enabled.
5. **login_history** – Logs user login timestamps, IP addresses, and authentication mode.

TABLE DEFINITIONS

USERS Table

Table No 6.1.1: Users

Column Name	Data Type	Constraints	Description
user_id	Integer	Primary Key, Auto-increment	Unique ID for each registered user
Username	String	Unique, Not Null	Display name of the user
Email	String	Unique, Not Null	Registered email address
password_hash	String	Not Null	Hashed password using bcrypt
Role	String	Default: 'user'	Role of the user (admin/user)
totp_secret	String	Nullable	TOTP secret for 2FA
isApproved	Boolean	Default: false	Indicates if the user is approved
isDisabled	Boolean	Default: false	Indicates if the user is disabled / banned
createdAt	DateTime	Default: Current timestamp	Account creation timestamp
updatedAt	DateTime	Auto-updated on modification	Last updated timestamp of the user record

TELEMETRY Table

Table No 6.1.2: Telemetry

Column Name	Data Type	Constraints	Description
telemetry_id	Integer	Primary Key, Auto-increment	Unique telemetry entry ID
Temperature	Float	Not Null	Temperature reading from CanSat

Pressure	Float	Not Null	Pressure reading
Altitude	Float	Not Null	Altitude data from sensor
Latitude	Float	Not Null	Latitude coordinate
Longitude	Float	Not Null	Longitude coordinate
Timestamp	DateTime	Not Null	Time of data recording

OTPS Table

Table No 6.1.3: Otps

Column Name	Data Type	Constraints	Description
otp_id	Integer	Primary Key, Auto-increment	Unique OTP record ID
email	String	Not Null	Associated user email
otp_code	String	Not Null	One-time password sent to email
expires_at	DateTime	Not Null	Expiration timestamp of OTP

DEVICES Table (Optional)

Table No 6.1.4: Devices

Column Name	Data Type	Constraints	Description
device_id	Integer	Primary Key, Auto-increment	Unique ID for each trusted device
user_id	Integer	Foreign Key (users.id)	Associated user ID
device_info	String	Not Null	Device metadata
trusted	Boolean	Default: false	Whether the device is trusted

LOGIN_HISTORY Table

Table No 6.1.5: Login_history

Column Name	Data Type	Constraints	Description
login_id	Integer	Primary Key, Auto-increment	Unique login entry
user_id	Integer	Foreign Key (users.id)	Associated user
timestamp	DateTime	Not Null	Login timestamp
ip_address	String	Nullable	IP address of the login source
method	String	Not Null	Login method (password/2FA)

Relationships Summary

Table No 6.1.6: Relationships

Entity A	Relation Type	Entity B	Linking Field
User	1 → Many	Telemetry	user_id (optional FK)
User	1 → Many	OTPS	email (implicit)
User	1 → Many	Devices	user_id
User	1 → Many	LoginHistory	user_id

6.2 E-R DIAGRAM

The Entity-Relationship (E-R) diagram models the logical structure of the Ground Control Station database, identifying entities, attributes, and the relationships among them. Though real-time telemetry data is handled via Socket.IO, the data is persistently stored in PostgreSQL, and the conceptual E-R model remains key to designing a scalable and structured system.

Data Objects (Entities)

The following key data objects (entities) are identified in the CanSat Ground Control Station:

User: Represents authenticated users (engineers, admins, or observers).

TelemetryData: Stores real-time telemetry values sent by the CanSat.

Role: Represents role-based access control for users (e.g., Admin, Viewer).

Session: Represents a user session or a mission session for data logging.

Device: Represents a CanSat device or module transmitting telemetry.

Alert: Generated when a telemetry reading crosses critical thresholds.

EventLog: Tracks user/system actions for audit purposes.

ATTRIBUTES

Table No 6.2.1: Attributes

Entity	Attributes
User	id, username, email, passwordHash, roleId, is2FAEnabled, createdAt
Role	id, name, description
TelemetryData	id, deviceId, temperature, pressure, altitude, latitude, longitude, timestamp, sessionId
Device	id, name, serialNumber, status, lastActive, createdAt
Session	id, name, startedAt, endedAt, createdBy
Alert	id, deviceId, type, value, threshold, timestamp, acknowledged
EventLog	id, userId, action, details, timestamp

RELATIONSHIPS

User → Session: A user can initiate multiple sessions.

User → EventLog: All user/system actions are recorded.

User → Role: Each user is assigned one role.

Role → Users: A role can be assigned to many users.

Device → TelemetryData: Each CanSat device sends multiple telemetry data entries.

Device → Alert: One device may generate multiple alerts.

TelemetryData → Session: Each telemetry entry is linked to a session.

Alert → Device: Alerts originate from specific devices.

CARDINALITY

Cardinality defines the numerical relationship between two entities:

One-to-Many (1:N):

- One user → many sessions
- One user → many event logs
- One role → many users
- One device → many telemetry data records
- One device → many alerts
- One session → many telemetry data records

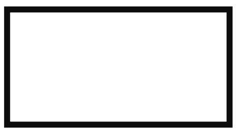
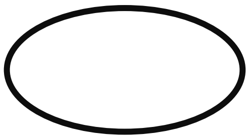
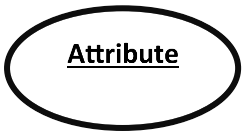
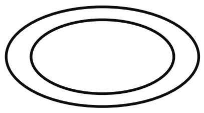
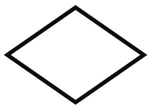
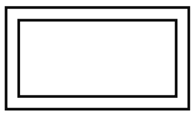
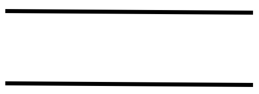
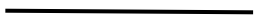
One-to-One (1:1):

- One user → one 2FA config (if stored separately)
- One device → one last active status record (latest)

Many-to-One (N:1):

- Many users → one role
- Many telemetry records → one session
- Many alerts → one device
- Many event logs → one user

Table No 6.2.2: Cardinality

Name	Notation	Description
Entity		Represents a real-world object or concept, such as a person, place, or event.
Attribute		Denotes a property or characteristic of an entity or relationship.
Key Attribute		Uniquely identifies an entity in an entity set.
Multivalued Attribute		Can have multiple values for a single entity (e.g., phone numbers).
Relationship		Shows an association between two or more entities.
Weak Entity		An entity that cannot exist without a related strong entity.
Participation (Total)		Indicates all instances must participate in the relationship (total participation).
Participation (Partial)		Indicates some instances may participate in the relationship (partial participation).
Cardinality(1:1,1:N, M:N)	Marked on connecting lines (1, N, M)	Specifies number of instances involved in the relationship.

ER Diagram

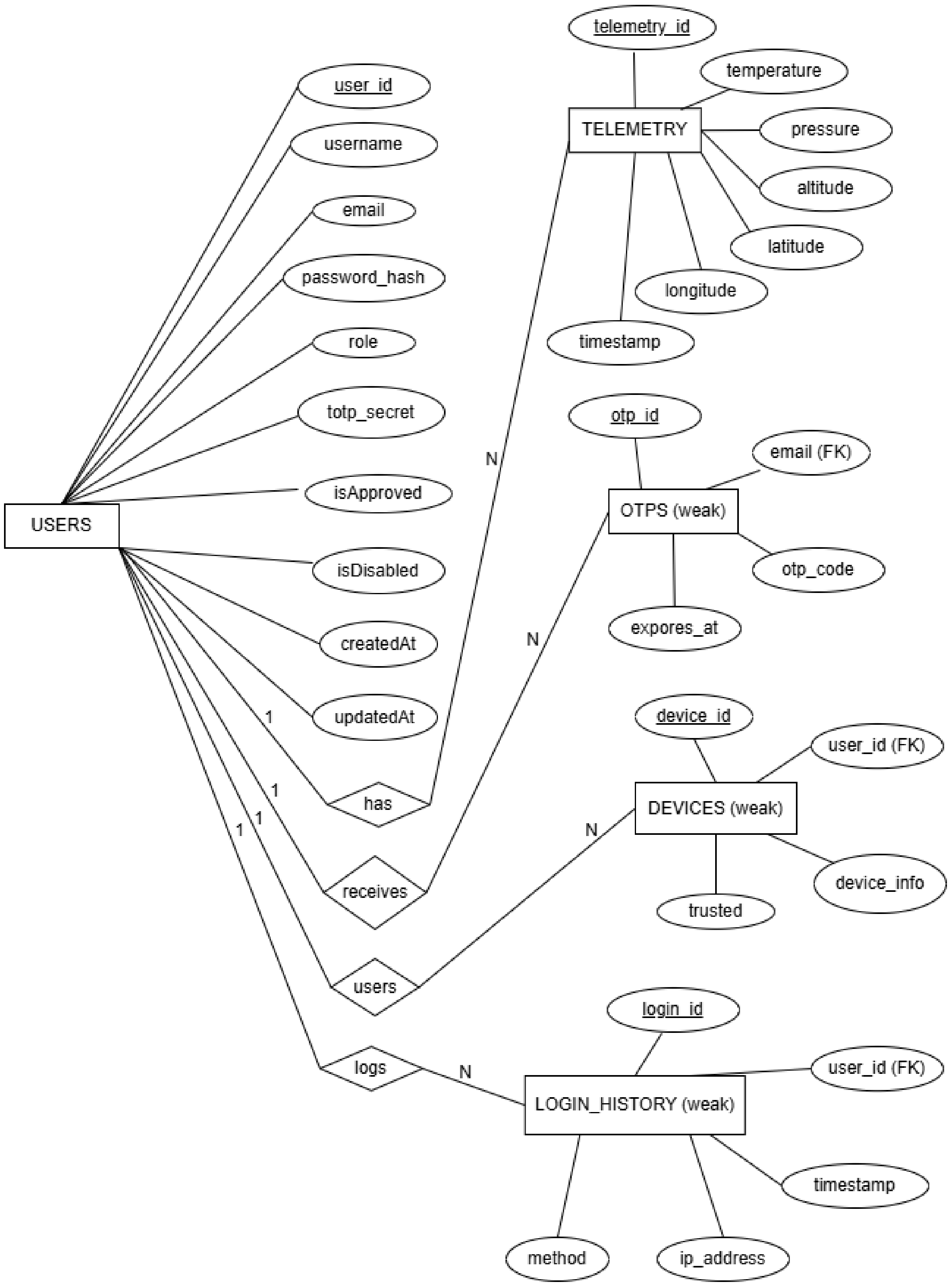


Figure No 6.2.1: ER Diagram

7. DETAILED DESIGN

The detailed design phase translates the high-level architecture into actionable components that can be directly implemented. It specifies the internal logic, data flow, and interconnections of modules such as authentication, telemetry streaming, and dashboard visualization. This ensures clarity in development and aligns all modules for secure, scalable, and efficient operation.

In the context of the **Ground Control Station for CanSats**, this phase defines the blueprint for real-time telemetry processing, user role management, 2FA, data visualization, and storage. It establishes how user sessions are handled, how telemetry data flows from socket streams to the database, and how filtered data is visualized. The modular Node.js + React.js architecture supports independent module testing, reusability, and maintainability.

This phase includes:

Structure Diagrams: Represent system modules such as Auth Service, Telemetry Service, and UI Layer.

Flowcharts: Depict logic for processes like user login, 2FA validation, telemetry parsing, and CSV export.

Module Breakdown: Defines how frontend, backend, and database layers are structured and communicate.

Procedural Specifications: Outline steps for key operations like login, telemetry filtering, or chart rendering.

The system emphasizes:

Real-time data processing and event-driven design using Socket.IO and Express.

Frontend-backend communication through secure REST APIs and WebSocket streams.

Security enforcement, including JWT, 2FA, and role-based access control at all critical points.

7.1 STRUCTURE DIAGRAM

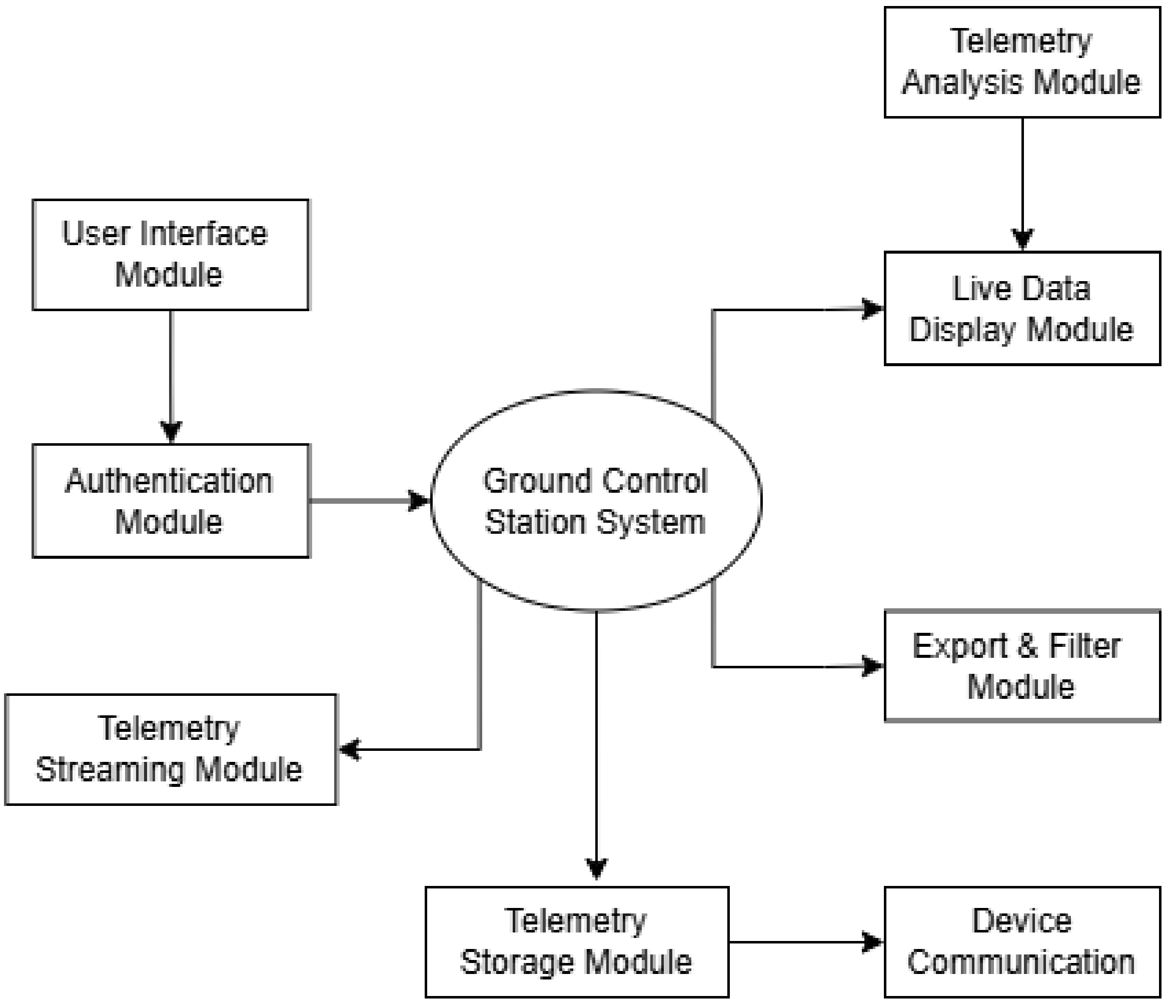


Figure No 7.1: Structure Diagram

7.2 FLOW CHARTS

User Login Flow Chart

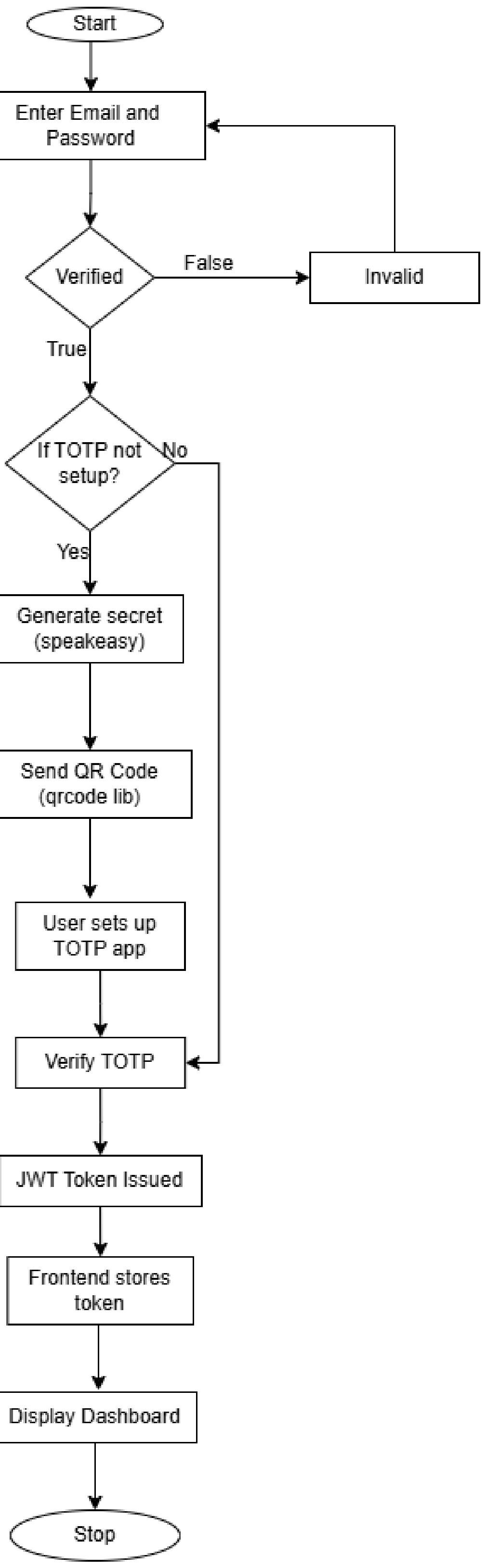


Figure No 7.2.1: User Login

Registration

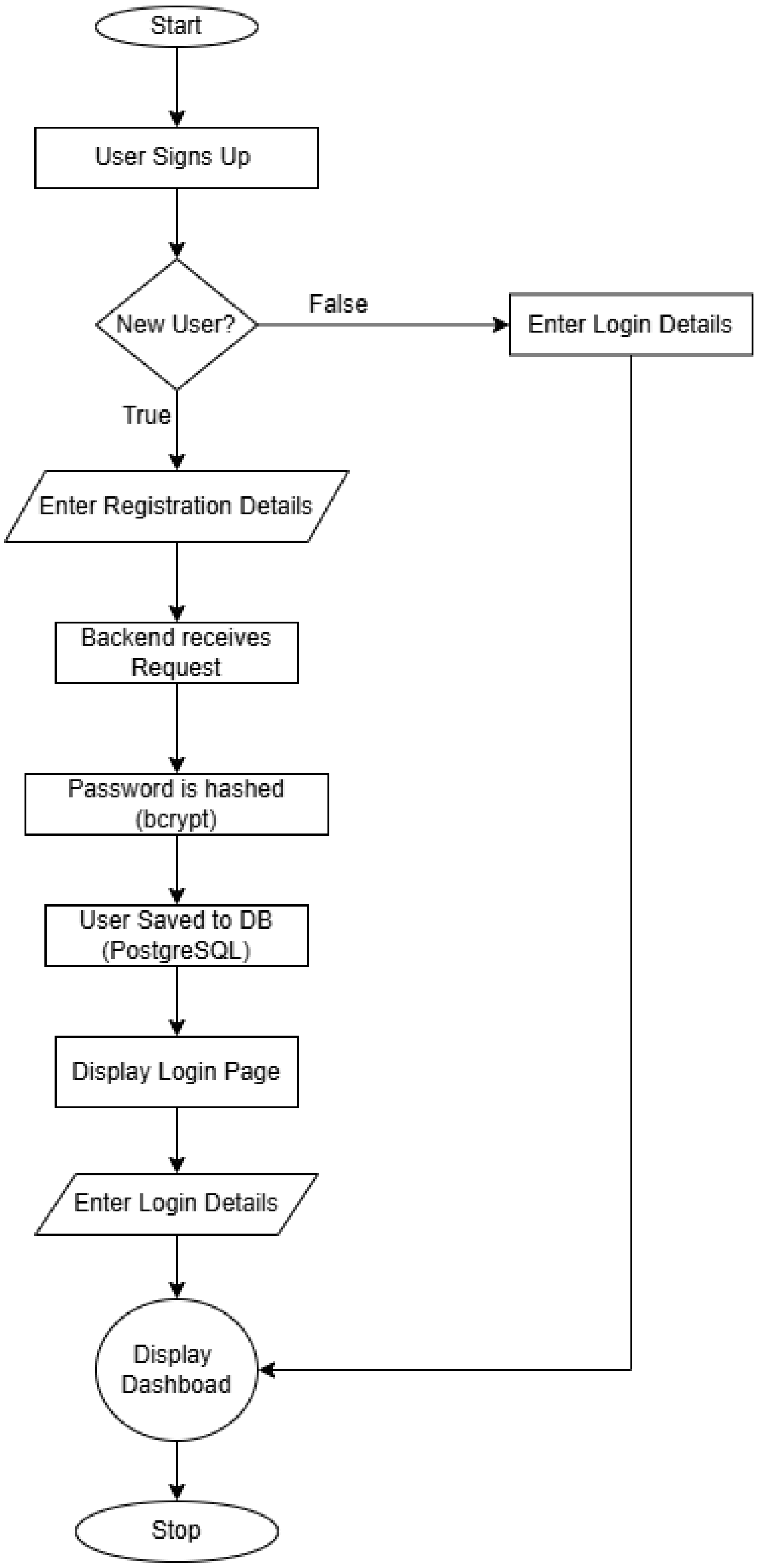


Figure No 7.2.2: Registration

CanSat Data Acquisition

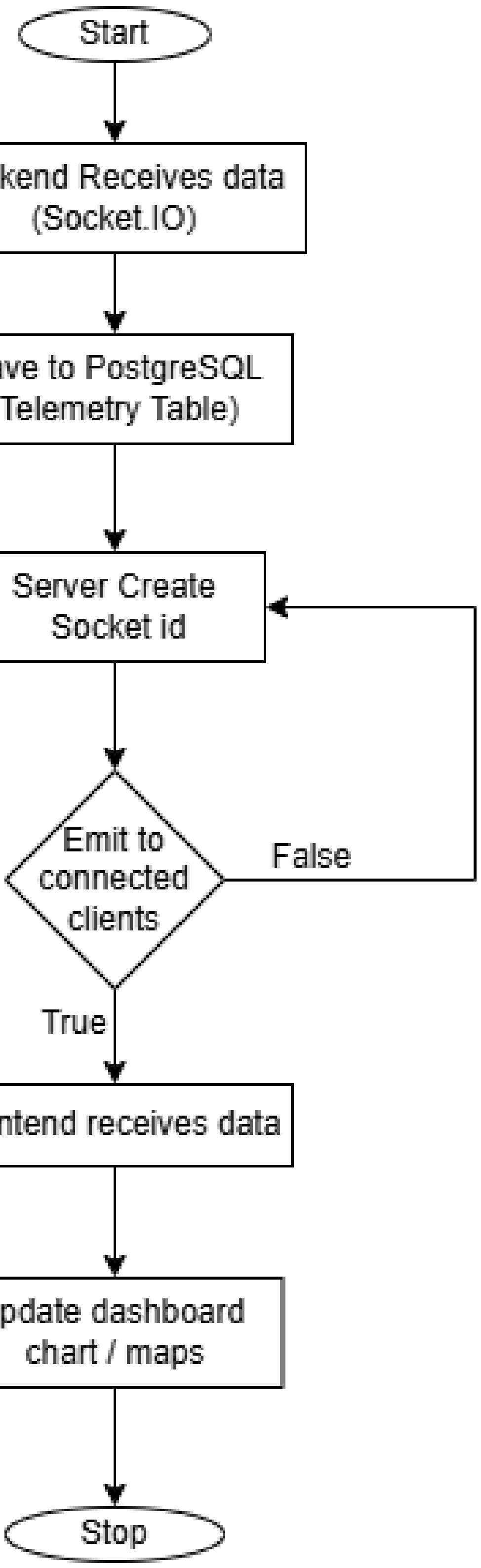

Start
Backend Receives data (Socket.IO)
Save to PostgreSQL (Telemetry Table)
Server Create Socket id
Emit to connected clients
False
True
Frontend receives data
Update dashboard chart / maps
Stop

Figure No 7.2.3: CanSat Data Acquisition

Data Processing & Storage

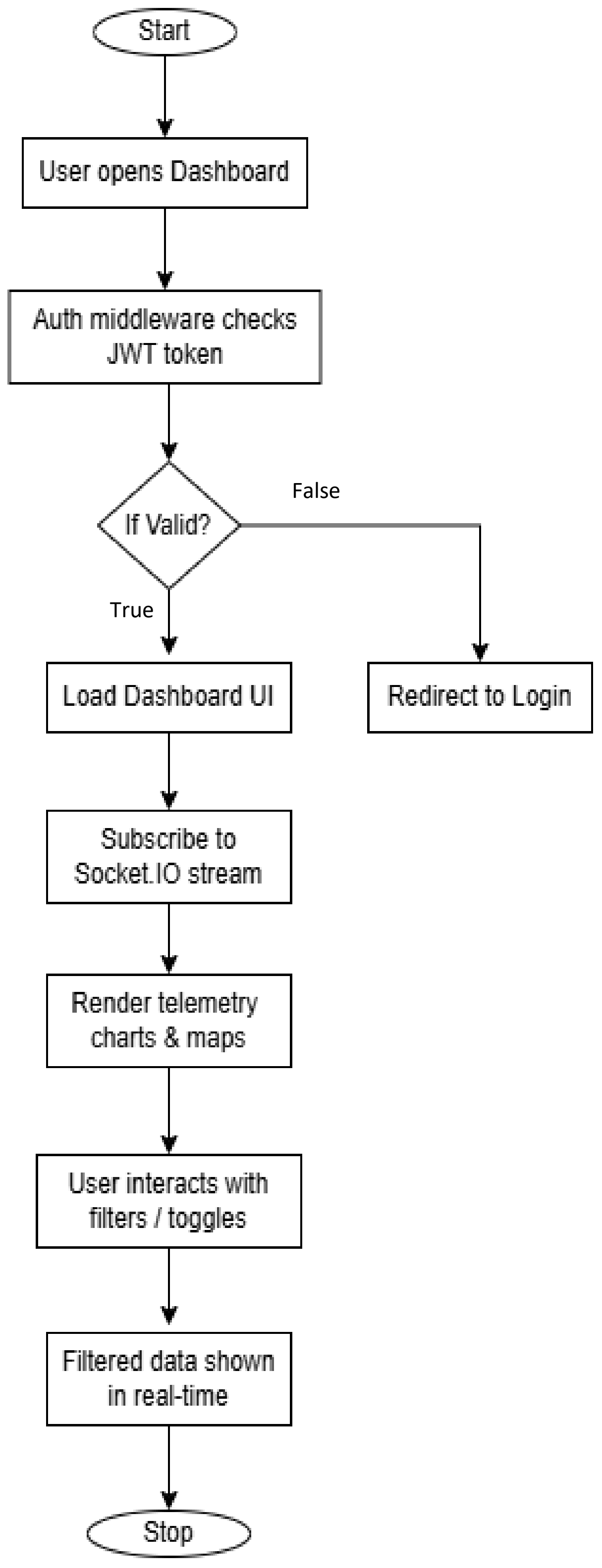

Start
User opens Dashboard
Auth middleware checks JWT token
If Valid?
False
True
Load Dashboard UI
Redirect to Login
Subscribe to Socket.IO stream
Render telemetry charts & maps
User interacts with filters / toggles
Filtered data shown in real-time
Stop

Figure No 7.2.4: Data Processing & Storage

Data Filtering & Analysis

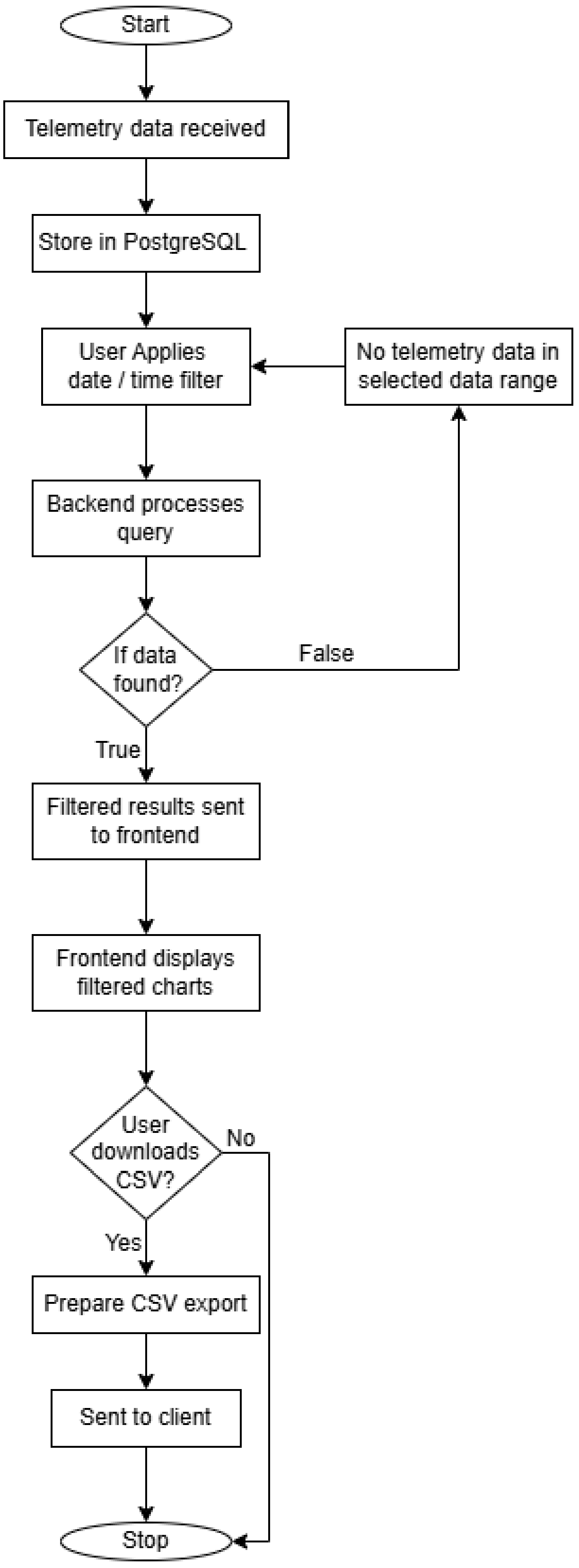


Figure No 7.2.5: Data Filtering & Analysis

Admin Flow Chart

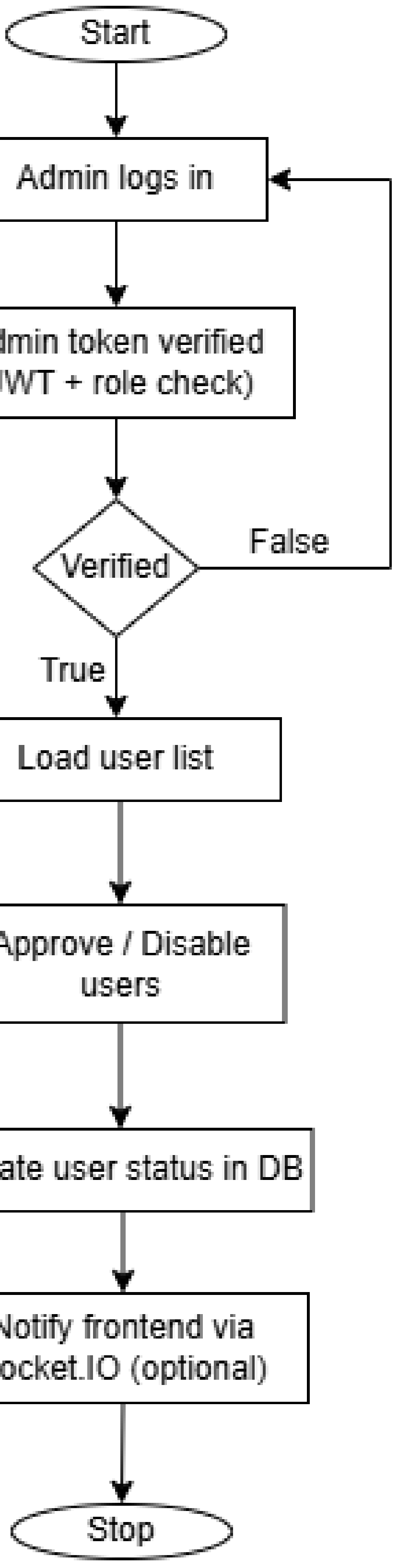


Figure No 7.2.6: Admin

7.3 MODULAR DECOMPOSITION OF COMPONENTS

User Authentication & Role Management

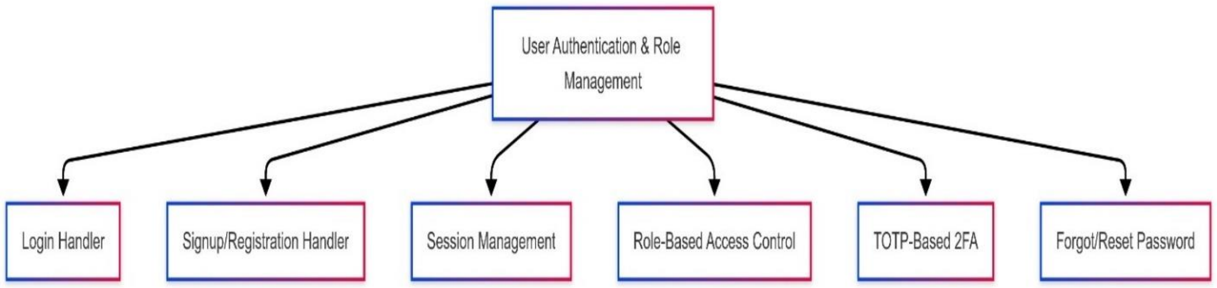

User Authentication & Role Management
Login Handler
Signup/Registration Handler
Session Management
Role-Based Access Control
TOTP-Based 2FA
Forgot/Reset Password

Figure No 7.3.1: User Authentication & Role Management

CanSat Data Acquisition Module

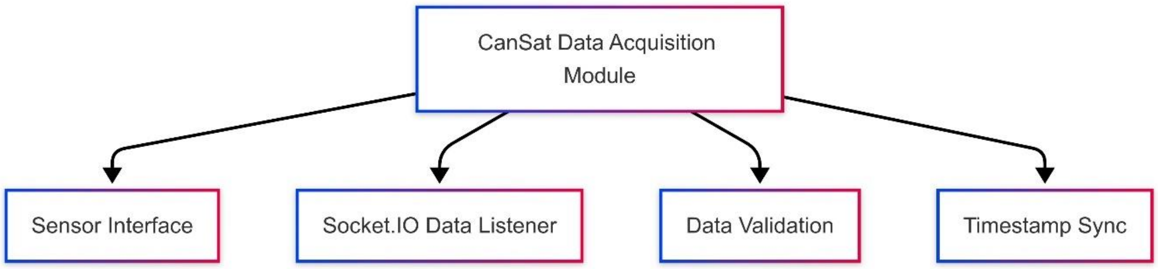


Figure No 7.3.2: CanSat Data Acquisition Module

Data Processing & Storage Module

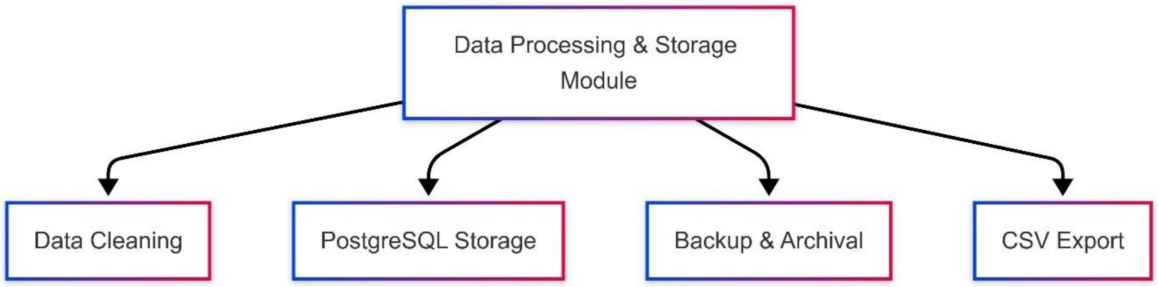


Figure No 7.3.3: Data Processing & Storage Module

Real-Time Data Visualization & UI/UX Module

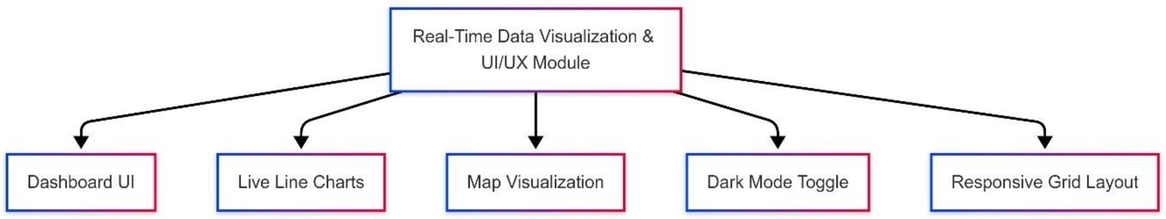


Figure No 7.3.4: Real-Time Data Visualization & UI/UX Module

Data Filtering & Analysis Module

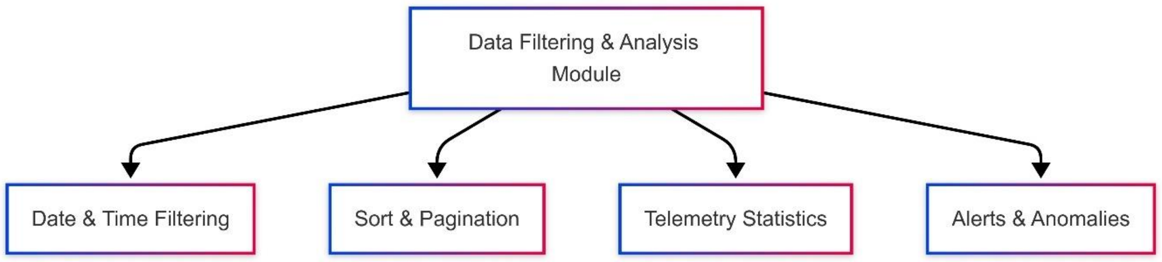


Figure No 7.3.5: Data Filtering & Analysis Module

7.4 MODULAR ARCHITECTURE OVERVIEW

The **GROUND Control Station for CanSats** is structured as a modular full-stack system designed for scalability, real-time telemetry visualization, and user management. The system is composed of the following major components:

Table No 7.4: Modular Architecture Overview

Module	Description
Node.js Backend	RESTful and WebSocket API layer handling user auth, telemetry processing, and DB
Socket.IO Engine	Enables real-time communication between CanSat data feed and frontend clients
React Frontend	SPA dashboard with data charts, telemetry tables, filters, and dark mode toggle
PostgreSQL DB	Stores user credentials, telemetry logs, session data, and configuration states

Backend Design (Node.js + Express.js)

The backend is built with **Express.js** using modular route definitions and middleware for clean separation of logic and extensibility.

Route Structure

Figure No 7.4.1: Route Structure

Route File	Path Prefix	Responsibilities
`auth.routes.js`	`/api/auth`	User registration, login, logout, 2FA setup
`telemetry.routes.js`	`/api/telemetry`	Stores, filters, and retrieves telemetry data
`users.routes.js`	`/api/users`	Role management, user profile operations

`export.routes.js`	`/api/export`	Exports telemetry data to CSV or JSON
`alerts.routes.js`	`/api/alerts`	Threshold-based alert setup and notifications (planned)

REST API Endpoints

Table No 7.4.2: REST API Endpoints

Endpoint	Method	Purpose
`/api/auth/login`	`POST`	User login with optional 2FA
`/api/auth/register`	`POST`	Registers new users with email and password
`/api/telemetry/latest`	`GET`	Fetches the latest real-time telemetry values
`/api/telemetry/filter`	`POST`	Filters telemetry based on datetime and parameters
`/api/export/csv`	`GET`	Exports filtered data as a CSV file
`/api/alerts/setup`	`POST`	Creates custom threshold alerts (future enhancement)

Real-Time Engine (Socket.IO)

The **Socket.IO** server enables real-time telemetry flow between the CanSat simulator/device and connected clients.

Real-Time Workflow

- The CanSat simulator or hardware emits telemetry data through Socket.IO.
- Server broadcasts to all authenticated clients.
- Frontend updates charts, maps, and tables instantly without refresh.

Frontend Design (React + Chart.js + Tailwind CSS)

The frontend is a **single-page React application** styled with Tailwind CSS and designed for responsiveness, real-time updates, and user interactivity.

Major Pages:

Table No 7.4.3: Major Pages

File	Description
`LoginPage.js`	Handles user login, 2FA QR rendering, and error feedback
`Dashboard.js`	Displays telemetry charts, tables, filters, and user info
`SignupPage.js`	User registration with validation and backend integration
`TelemetryChart.js`	Line chart for live plotting of telemetry data (temp, alt, etc.)
`Filters.js`	Allows users to sort/filter telemetry based on date/time range

Component Responsibilities:

Table No 7.4.4: Component Responsibilities

Component	Purpose
`SocketClient.js`	Manages live connection with Socket.IO server
`AuthContext.js`	Provides global auth state across components
`DataTable.js`	Paginated telemetry table with export and sort functionality
`DarkModeToggle.js`	Toggles dark/light UI themes
`ExportButton.js`	Triggers CSV download of telemetry data

Data and Control Flow

Telemetry Data Flow

- CanSat device emits real-time telemetry via Socket.IO.
- Backend receives and stores the data in PostgreSQL.
- Frontend receives the broadcast and updates UI (charts + tables).
- Data is optionally filterable and exportable to CSV.

User Authentication Flow

- User registers and logs in via `/api/auth`.
- JWT is issued and stored in localStorage/sessionStorage.
- For users with 2FA, a QR code is rendered during first login.
- Authenticated users access the dashboard with role-based views.

PROCEDURAL DETAILS OF KEY MODULES

Each of the following modules represents a core functionality of the CanSat Ground Control Station. The logic for each is outlined in a step-by-step format to assist developers in implementing or testing these modules with clarity and precision.

User Login Module

Objective: Validate user credentials and establish a secure session.

Procedure:

1. User enters email and password on the login screen.
2. Frontend sends a POST request to `/api/auth/login`.
3. Backend queries the `users` table for the provided email.
4. If user exists:
   • Compares entered password with `password_hash` using bcrypt.
5. If password is valid:
   • Generates JWT token and sends it to the frontend.
   • If TOTP is enabled, prompts for 2FA; otherwise, redirects to `/dashboard`.
6. Else:
   • Returns error: “Invalid credentials”.

User Registration Module

Objective: Register new users with secure credentials and role setup.

Procedure:

1. User fills the signup form with username, email, and password.
2. Frontend sends POST request to `/api/auth/signup`.
3. Backend:
   • Validates uniqueness of email and username.
   • Hashes password using bcrypt.
   • Stores user data with `isApproved=false` by default.
4. Sends a success response and redirects user to the login page.
5. If duplicate entry, returns error: "User already exists".

Telemetry Reception Module

Objective: Receive and store real-time telemetry data from the CanSat device.

Procedure:

1. CanSat simulator/device emits telemetry via WebSocket connection.
2. Socket.IO server receives the data packet.
3. Backend parses data fields:
   temperature, pressure, altitude, latitude, longitude, timestamp.
4. Saves telemetry data into the **`telemetry`** table.
5. Emits the same data to all connected frontend clients.

Telemetry Visualization Module

Objective: Visualize real-time and historical telemetry data on the dashboard.

Procedure:

1. Dashboard loads and connects to WebSocket server.
2. Listens for incoming telemetry updates.

3. Frontend updates line charts and tables live.
4. User selects filters (date, sort order, range).
5. Sends request to `/api/telemetry/filter`.
6. Backend queries `telemetry` table and returns filtered results.
7. Frontend updates charts and tables accordingly.

TOTP-Based 2FA Module

Objective: Provide Two-Factor Authentication for user login security.

Procedure:

1. On first login, backend generates TOTP secret using speakeasy.
2. Sends QR code image to frontend using `qrcode` module.
3. User scans QR with Google Authenticator app.
4. On future logins, user is prompted for 6-digit OTP.
5. Frontend sends OTP to `/api/auth/verify-2fa`.
6. Backend verifies code and returns JWT on success.

Password Reset Module (OTP via Email)

Objective: Allow users to reset forgotten passwords securely via OTP.

Procedure:

1. User enters email on the "Forgot Password" page.
2. Frontend sends POST to `/api/auth/request-reset`.
3. Backend generates 6-digit OTP and expiry timestamp.
4. Sends OTP to user's email using Nodemailer.
5. User enters OTP and new password in the reset form.
6. Frontend sends POST to `/api/auth/reset-password`.
7. Backend verifies OTP and updates password hash in DB.

User Role & Access Module

Objective: Grant different access rights to Admin, Engineer, and Observer roles.

Procedure:

1. Admin assigns roles via `/api/users/manage`.
2. JWT includes role claim during login.
3. Protected routes check role before granting access.
4. Admin-only features like "Manage Users" are visible only to admins.
5. Observers cannot access data download, deletion, or filters.

CSV Export Module

Objective: Allow users to download telemetry data in CSV format.

Procedure:

1. User clicks "Download CSV" on the dashboard.
2. Frontend sends GET request to `/api/telemetry/export`.
3. Backend filters data based on user selection.
4. Formats it into a CSV string using `json2csv`.
5. Sends file as downloadable blob to frontend.

8. IMPLEMENTATION

The implementation phase translates the detailed system design into fully functional modules. The GROUND Control Station for CanSats is implemented as a secure, scalable full-stack web application using:

Backend: Node.js with Express.js, Sequelize ORM, Socket.IO for real-time telemetry, and PostgreSQL.

Frontend: React.js for dynamic UI rendering and real-time dashboard interactions.

Security: JWT for authentication, TOTP for two-factor verification, and bcrypt for password hashing.

Visualization: Real-time telemetry visualization via responsive charts and tables.

BACKEND IMPLEMENTATION (Node.js + Express.js)

Table No 8.1: BACKEND IMPLEMENTATION

File / Folder	Purpose
`server.js`	Main entry point of the server; initializes Express, middlewares, and Socket.IO.
`config/database.js`	Sequelize configuration for PostgreSQL database connection.
`models/User.js`	Defines the `User` model (includes createdAt, updatedAt, isApproved, isDisabled).
`models/Telemetry.js`	Sequelize model for telemetry data (temperature, pressure, altitude, etc.).
`routes/authRoutes.js`	Routes for user login, signup, logout, and 2FA verification.
`routes/telemetryRoutes.js`	REST API to fetch, filter, and stream telemetry data to frontend.
`routes/passwordRoutes.js`	Routes for forgot-password and reset-password via email OTP.

`middleware/authMiddleware.js`	Middleware for JWT token authentication and role checking.
`utils/totpUtils.js`	Utility functions to generate and verify TOTP using `speakeasy`.
`utils/emailService.js`	Sends OTP via email for password reset.
`sockets/telemetrySocket.js`	Emits real-time telemetry to connected clients using Socket.IO.

FRONTEND IMPLEMENTATION (React.js)

Table No 8.2: FRONTEND IMPLEMENTATION

File / Folder	Purpose
App.js	Main React app entry, defines route structure and layout.
pages/LoginPage.jsx	User login form with TOTP verification if required.
pages/SignupPage.jsx	User registration form with validation.
pages/Dashboard.jsx	CanSat telemetry dashboard showing charts, data grid, and user info.
components/Navbar.jsx	Responsive navigation bar with logout and username display.
components/TelemetryChart.jsx	Line chart components for displaying real-time telemetry (e.g., altitude).
components/DataTable.jsx	Table displaying telemetry records with sort/filter/pagination options.
components/DarkModeToggle.jsx	Toggle button to enable dark/light theme.
components/DateFilter.jsx	Inputs to filter telemetry data by exact date and time using datetime-local.
services/api.js	Axios service layer to communicate with backend REST APIs.
utils/tokenUtils.js	Helpers for handling JWT tokens in localStorage/sessionStorage.

8.1 CODE SNIPPET

```
# ===========================
# App.js
# ===========================
import React from "react";
import { BrowserRouter as Router, Routes, Route, Navigate } from "react-router-dom";
import AboutPage from "./pages/AboutPage";
import SignupPage from "./pages/SignupPage";
import LoginPage from "./pages/LoginPage";
import DashboardPage from "./pages/DashboardPage";
import ForgotPasswordPage from "./pages/ForgotPasswordPage";
import ResetPasswordPage from "./pages/ResetPasswordPage";
import ManageUsersPage from "./pages/ManageUsersPage";

const ProtectedRoute = ({ children }) => {
  const token = localStorage.getItem("token") || sessionStorage.getItem("token");
  if (!token) {   return <Navigate to="/login" replace />;       }
  return children;
};

function App() {          return (
    <Router>     <Routes>
        <Route path="/about" element={<AboutPage />} />
        <Route    path="/manage-users"     element={
            <ProtectedRoute>
              <ManageUsersPage />
            </ProtectedRoute>
          }
        />
        <Route path="/signup" element={<SignupPage />} />
        <Route path="/login" element={<LoginPage />} />
        <Route path="/dashboard" element={<DashboardPage />} />
        <Route path="/" element={<Navigate to="/about" />} />
        <Route path="/forgot-password" element={<ForgotPasswordPage />} />
        <Route path="/reset-password/:token" element={<ResetPasswordPage />} />
      </Routes>    </Router>  );}

export default App;
#==========================================================
# AboutPage.js
#==========================================================
import React from "react";
import { useNavigate } from "react-router-dom";

const AboutPage = () => {
  const navigate = useNavigate();

  const handleDashboardClick = async () => {
    const token = localStorage.getItem("token") || sessionStorage.getItem("token");

    if (!token) {
      alert("Please login to access the Dashboard.");
      return navigate("/login");    }

    try {      const res = await fetch("http://localhost:5000/api/auth/verify-token", {
        method: "GET",   headers: { Authorization: `Bearer ${token}` },    });

      if (!res.ok) {    throw new Error("Invalid or expired token");            }

      navigate("/dashboard");
    } catch (err) {      alert("Session expired. Please login again.");
      localStorage.clear();      sessionStorage.clear();   navigate("/login");    }  };


  return (    <div style={styles.container}>      <div style={styles.navbar}>
        <button onClick={() => navigate("/login")} style={styles.link}>Login</button>
        <button onClick={() => navigate("/signup")} style={styles.link}>Register</button>
        <button onClick={handleDashboardClick} style={styles.link}>Dashboard</button>
      </div>
      <div style={styles.content}>
        <h1>📡 CanSat Ground Control Station</h1>
        <p style={{textIndent:'40px'}}>
          The <b>CanSat Ground Control Station</b> is an interactive, full-stack web
applicationdesigned to  simulate real-world satellite telemetry monitoring in a secure and
visually engaging environment. It enables users to track vital CanSat parameters like
<b>temperature, pressure, altitude, latitude,  longitude, </b> and <b>timestamp</b> — all
updated in real-time and displayed through intuitive charts, gauges,   and map visualizations.
  </p>
        <p style={{textIndent:'40px'}}>
          Built using <b> React.js, Node.js, PostgreSQL,</b> and <b>Socket.IO,</b> this
platform offers robust  features like<b> JWT-based login, TOTP two-factor authentication
(2FA),</b> and<b> role-based access  control</b> for Admins and Viewers.  </p>
  <p> Click the buttons on the top-right to Login, Register, or access the Dashboard.        </p>
      </div>    </div>  );};
```

```
const styles = {
  container: {    minHeight: "100vh", padding: "30px", backgroundColor: "#f4f4f4",
    backgroundImage: "url('/images/cansat.jpg')", backgroundSize: "cover",
    backgroundPosition: "center", backgroundRepeat: "no-repeat",
    fontFamily: "Arial, sans-serif" },
  navbar: {    position: "absolute", top: 20,  right: 30,  display: "flex", gap: "10px", },
  link: {    padding: "8px 16px",    backgroundColor: "#007BFF",    color: "#fff",
    border: "none",    borderRadius: "4px",    cursor: "pointer",    fontWeight: "bold",
  },
  content: {    maxWidth: "700px",    margin: "100px auto",    textAlign: "justify",
    backgroundColor: "#fff",    padding: "40px",    borderRadius: "10px",
    boxShadow: "0 0 10px rgba(0,0,0,0.1)",  },};
export default AboutPage;
# ============================================================
# DashboardPage.js
# ============================================================
// ✅ Fully Working CanSat Dashboard with Accurate Filtering, Sorting, Pagination, and
Real-Time Charts (Backend Filter)
import React, { useEffect, useState } from "react";
import { useNavigate } from "react-router-dom";
import NavBar from "../components/NavBar";
import io from "socket.io-client";
import axios from "axios";
import { LineChart, Line, XAxis, YAxis, Tooltip, Legend, ResponsiveContainer,
  CartesianGrid,} from "recharts";
import { MapContainer, TileLayer, Marker, Popup } from "react-leaflet";
import "leaflet/dist/leaflet.css";

const socket = io("http://localhost:5000", {
  transports: ["websocket"],  withCredentials: true,});

function DashboardPage() {
  const [telemetryData, setTelemetryData] = useState([]);
  const [isConnected, setIsConnected] = useState(false);
  const [socketId, setSocketId] = useState(null);
  const [user, setUser] = useState(null);
  const [darkMode, setDarkMode] = useState(localStorage.getItem("theme") === "dark");
  const [sortKey, setSortKey] = useState("timestamp");
  const [ascending, setAscending] = useState(false);
  const [startDate, setStartDate] = useState("");
  const [endDate, setEndDate] = useState("");
  const [cansatId, setCansatId] = useState("");
  const [page, setPage] = useState(1);
```

```
 const perPage = 10;
 const navigate = useNavigate();
 useEffect(() => {
   const token = localStorage.getItem("token") || sessionStorage.getItem("token");
          const      username      =      localStorage.getItem("username")      ||
sessionStorage.getItem("username");
   const email = localStorage.getItem("email") || sessionStorage.getItem("email");
   const role = localStorage.getItem("role") || sessionStorage.getItem("role");


   if (!token) {    alert("You must be logged in to access the dashboard!");
     navigate("/login");    return;   }

   setUser({ username, email, role });
   document.body.className = darkMode ? "dark-mode" : "light-mode";
   localStorage.setItem("theme", darkMode ? "dark" : "light");

   socket.on("connect", () => {    setIsConnected(true);    setSocketId(socket.id);  });
   socket.on("disconnect", () => setIsConnected(false));
   socket.on("telemetry", (data) => {
     if (cansatId && data.cansatId !== cansatId) {
       return; // Ignore data not matching selected cansatId
     }
     const formatted = { ...data, timestamp: new Date(data.timestamp).toISOString() };
     setTelemetryData((prev) => [...prev.slice(-49), formatted]);   });

   return () => {    socket.off("connect");   socket.off("disconnect");
     socket.off("telemetry");   };
 }, [navigate, darkMode, cansatId]);

 useEffect(() => {
   const fetchFilteredData = async () => {
     try {     const query = {};
       if (startDate) query.start = new Date(startDate).toISOString();
       if (endDate) query.end = new Date(endDate).toISOString();

       const url = `http://localhost:5000/api/telemetry/filter${
        Object.keys(query).length ? "?" + new URLSearchParams(query).toString() : ""   }`;

       const response = await axios.get(url);
       const sorted = [...response.data].sort((a, b) => {
        const aVal = sortKey === "timestamp" ? new Date(a[sortKey]).getTime() : a[sortKey];
           const  bVal  =  sortKey  ===  "timestamp"  ?  new  Date(b[sortKey]).getTime()  :
b[sortKey];
```

```
        return ascending ? aVal - bVal : bVal - aVal; });

      setTelemetryData(sorted);   setPage(1);
    } catch (err) {   console.error("❌ Error fetching filtered data:", err);      }    };

    fetchFilteredData();
  }, [startDate, endDate, sortKey, ascending]);

  const downloadCSV = () => {
      const  headers  =  ["Temperature",  "Pressure",  "Altitude",  "Latitude",  "Longitude",
"Timestamp (Local Time)"];
    const rows = telemetryData.map((item) => [  item.temperature,  item.pressure,
      item.altitude,     item.latitude,     item.longitude,
      new Date(item.timestamp).toLocaleString(),     ]);
      const csvContent = [headers, ...rows].map((e) => e.join(",")).join("\n");
    const blob = new Blob([csvContent], { type: "text/csv" });
    const url = URL.createObjectURL(blob);    const a = document.createElement("a");
    a.href = url;    a.download = "telemetry_data.csv";    a.click();
    URL.revokeObjectURL(url);  };

  const pageData = telemetryData.slice((page - 1) * perPage, page * perPage);
  const chartColors = {
    temperature: "#ff4c4c",    pressure: "#00b8d9",    altitude: "#7a63ff",
    latitude: "#34c759",    longitude: "#ff9500",  };
  return (
    <div style={styles(darkMode).container}>
      <NavBar />
      <h2 style={styles(darkMode).header}>CanSat Dashboard</h2>

            {user    &&    <p    style={styles(darkMode).welcome}>👋    Welcome    back,
{user.username?.trim() || user.email}!</p>}

      {user?.role === "admin" && (
       <button    onClick={() => navigate("/manage-users")}
         style={{
          ...styles(darkMode).downloadButton,     backgroundColor: "#6a0dad",
          color: "white",    }}>  👥 Manage Users   </button>    )}
      <div style={styles(darkMode).controls}>
                              <button       onClick={()       =>       setDarkMode(!darkMode)}
style={styles(darkMode).darkToggle}>
        {darkMode ? "🌞 Light Mode" : "🌙 Dark Mode"}
       </button>
      <button onClick={downloadCSV} style={styles(darkMode).downloadButton}>📥
Download CSV</button>
    <input  type="text"  placeholder="Filter by CanSat ID"  value={cansatId}
     onChange={(e) => setCansatId(e.target.value)}
     style={{ ...styles(darkMode).input, width: "200px" }}   />
                  <select    onChange={(e)    =>    setSortKey(e.target.value)}
style={styles(darkMode).select}>
     <option value="timestamp">Sort by Time</option>
     <option value="temperature">Temperature</option>
     <option value="pressure">Pressure</option>
     <option value="altitude">Altitude</option>
    </select>
                   <button    onClick={()    =>    setAscending(!ascending)}
style={styles(darkMode).downloadButton}>
     {ascending ? "⬆ Ascending" : "⬇ Descending"}
    </button>
             <input  type="datetime-local"  value={startDate}  onChange={(e)  =>
setStartDate(e.target.value)} style={styles(darkMode).input} />
             <input  type="datetime-local"  value={endDate}  onChange={(e)  =>
setEndDate(e.target.value)} style={styles(darkMode).input} />
    <button  onClick={() => {  setStartDate("");  setEndDate(""); }}
     style={styles(darkMode).downloadButton}  > ❌ Clear Filters </button>
   </div>

   <p style={{ color: isConnected ? "#4caf50" : "#ff3b30", fontWeight: "bold" }}>
    Status: {isConnected ? "Connected" : "Disconnected"}
   </p>
   {socketId && (
    <p style={{ fontWeight: "bold" }}> Socket ID: {socketId} </p> )}

   {telemetryData.length === 0 ? (
     <p style={{ margin: "40px 0", fontStyle: "italic" }}>⚠ No telemetry data in the
selected date range.</p>
   ) : (
    <>
     {/* First Row: Temperature Chart, Altitude Chart, Temperature Speedometer */}
     <div style={{ display: "grid", gridTemplateColumns: "repeat(auto-fit, minmax(250px,
1fr))", gap: "20px", marginBottom: "40px" }}>
      <div style={styles(darkMode).chartBox}>
       <h4 style={styles(darkMode).chartTitle}>Temperature Over Time</h4>
       <ResponsiveContainer width="100%" height={250}>
        <LineChart data={telemetryData}>
         <CartesianGrid strokeDasharray="3 3" stroke={darkMode ? "#555" : "#ccc"} />
```

```
            <XAxis dataKey="timestamp" stroke={darkMode ? "#fff" : "#000"} />
            <YAxis stroke={darkMode ? "#fff" : "#000"} />
            <Tooltip contentStyle={{ backgroundColor: darkMode ? "#333" : "#fff" }} />
            <Legend />
                                             <Line type="monotone" dataKey="temperature"
stroke={chartColors.temperature} dot={false} />
          </LineChart>   </ResponsiveContainer>
        </div>
        <div style={styles(darkMode).chartBox}>
          <h4 style={styles(darkMode).chartTitle}>Altitude Over Time</h4>
          <ResponsiveContainer width="100%" height={250}>
            <LineChart data={telemetryData}>
            <CartesianGrid strokeDasharray="3 3" stroke={darkMode ? "#555" : "#ccc"} />
            <XAxis dataKey="timestamp" stroke={darkMode ? "#fff" : "#000"} />
            <YAxis stroke={darkMode ? "#fff" : "#000"} />
            <Tooltip contentStyle={{ backgroundColor: darkMode ? "#333" : "#fff" }} />
            <Legend />
                <Line type="monotone" dataKey="altitude" stroke={chartColors.altitude}
dot={false} />   </LineChart>   </ResponsiveContainer>
        </div>

        <div style={styles(darkMode).mapBox}>
          {telemetryData.length ? (
              <MapContainer center={[telemetryData[telemetryData.length - 1].latitude,
telemetryData[telemetryData.length - 1].longitude]}   zoom={13}
            style={{ height: "300px", borderRadius: "8px" }}>
            <TileLayer   url="https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png"
              attribution='&copy; <a href="http://osm.org/copyright">OpenStreetMap</a>
contributors'/>
                  <Marker position={[telemetryData[telemetryData.length - 1].latitude,
telemetryData[telemetryData.length - 1].longitude]}>
              <Popup>
                Latitude: {telemetryData[telemetryData.length - 1].latitude}<br />
                Longitude: {telemetryData[telemetryData.length - 1].longitude}
              </Popup>
            </Marker>
          </MapContainer>
          ) : (
            <p style={{ color: darkMode ? "#fff" : "#000" }}>No location data available</p>
          )}    </div>
      </div>

      {/* Second Row: Pressure Speedometer, Longitude and Latitude Map */}
      <div style={{ display: "grid", gridTemplateColumns: "repeat(auto-fit, minmax(250px,
1fr))", gap: "20px" }}>
       <div style={styles(darkMode).speedometerBox}>
         <h4 style={styles(darkMode).chartTitle}>Temperature °C/°F</h4>
            <CustomSpeedometer  maxValue={100}  value={telemetryData.length ?
telemetryData[telemetryData.length - 1].temperature : 0}
          needleColor="red"   startColor="green"  endColor="red" height={350}
          width={300}/>
        </div>

        <div style={styles(darkMode).speedometerBox}>
        <h4 style={styles(darkMode).chartTitle}>Pressure K/PSI</h4>
            <CustomSpeedometer  maxValue={2000}  value={telemetryData.length ?
telemetryData[telemetryData.length - 1].pressure : 0}
         needleColor="#6a0dad"  startColor="#b57edc"  endColor="#6a0dad"
         height={350}  width={300} />  </div>
        </div>

       {/* Live Telemetry Data Table */}
       <div style={styles(darkMode).tableContainer}>
        <h3 style={styles(darkMode).chartTitle}>Live Telemetry Data</h3>
        <div style={{ overflowX: "auto" }}>
         <table style={styles(darkMode).table}>
          <thead> <tr>  <th style={styles(darkMode).th}>Temperature</th>
            <th style={styles(darkMode).th}>Pressure</th>
            <th style={styles(darkMode).th}>Altitude</th>
            <th style={styles(darkMode).th}>Latitude</th>
            <th style={styles(darkMode).th}>Longitude</th>
            <th style={styles(darkMode).th}>Timestamp</th>
           </tr>  </thead>
          <tbody>
           {pageData.map((item, idx) => (
            <tr key={idx}>
             <td style={styles(darkMode).td}>{item.temperature}</td>
             <td style={styles(darkMode).td}>{item.pressure}</td>
             <td style={styles(darkMode).td}>{item.altitude}</td>
             <td style={styles(darkMode).td}>{item.latitude}</td>
             <td style={styles(darkMode).td}>{item.longitude}</td>
                                          <td  style={styles(darkMode).td}>{new
Date(item.timestamp).toLocaleString()}</td>  </tr>  ))}
          </tbody>  </table>  </div>
        <div style={{ marginTop: "10px" }}>
          <button onClick={() => setPage(p => Math.max(p - 1, 1))} disabled={page ===
1}>◀ Prev</button>
```

```
            <span style={{ margin: "0 10px" }}>Page {page}</span>
            <button onClick={() => setPage(p => (p * perPage < telemetryData.length ? p + 1
: p))}>Next ▶</button>
          </div>    </div>    </>   )}       </div> );}

// CustomSpeedometer component using SVG
const CustomSpeedometer = ({ maxValue, value, needleColor, startColor, endColor, height,
width }) => {
  const radius = Math.min(width, height) / 2;  const centerX = width / 2;
  const centerY = height / 2; const angleRange = 180; // semi-circle
  const valueRatio = Math.min(Math.max(value / maxValue, 0), 1);
  const needleAngle = 180 - valueRatio * angleRange;

  // Calculate needle end point
  const needleLength = radius * 0.9;  const radian = (needleAngle * Math.PI) / 180;
  const needleX = centerX + needleLength * Math.cos(radian);
  const needleY = centerY - needleLength * Math.sin(radian);

  // Gradient id for arc
  const gradientId = `gradient-${needleColor.replace("#", "")}`;

  return (
    <svg width={width} height={height} viewBox={`0 0 ${width} ${height}`}>
      <defs>  <linearGradient id={gradientId} x1="0%" y1="0%" x2="100%" y2="0%">
          <stop offset="0%" stopColor={startColor} />
          <stop offset="100%" stopColor={endColor} />
        </linearGradient>
      </defs>
      {/* Arc background */}
      <path
        d={`
          M ${centerX - radius} ${centerY}
          A ${radius} ${radius} 0 0 1 ${centerX + radius} ${centerY}
        `}
        fill="none"
        stroke={`url(#${gradientId})`}
        strokeWidth={20}
        strokeLinecap="round"
      />
      {/* Needle */}
      <line        x1={centerX}       y1={centerY}       x2={needleX}       y2={needleY}
        stroke={needleColor}       strokeWidth={3}      />
      {/* Center circle */}
      <circle cx={centerX} cy={centerY} r={5} fill={needleColor} />
    {/* Text value */}
    <text      x={centerX}      y={height - 50}      textAnchor="middle"
      fontSize={30}      fill={needleColor}      fontWeight="bold">
      {Math.round(value)}
    </text>   </svg>  );};

const styles = (darkMode) => ({
  container: {   textAlign: "center",   padding: "20px",   minHeight: "100vh",
    backgroundColor: darkMode ? "#121212" : "#f8f9fa",   color: darkMode ? "#fff" : "#000",
    overflowX: "hidden",  },
  header: { fontSize: "28px", fontWeight: "bold", marginBottom: "20px" },
  welcome: { marginBottom: 10 },
  controls: {   display: "flex",   flexWrap: "wrap",   justifyContent: "center",   gap: "10px",
    marginBottom: "20px",  },
  darkToggle: {   padding: "10px 16px",
    backgroundColor: darkMode ? "#555" : "#007bff",   color: "#fff",   border: "none",
    borderRadius: "5px",   cursor: "pointer",  },
  downloadButton: {   padding: "10px 16px",   backgroundColor: "#28a745",
    color: "#fff",   border: "none",   borderRadius: "5px",   cursor: "pointer",  },
  select: {   padding: "8px",   fontSize: "16px",   borderRadius: "5px",
    border: "1px solid #ccc",
  },
  input: {   padding: "8px",   borderRadius: "5px",   border: "1px solid #ccc",  },
  frameset: {   display: "grid",   gridTemplateColumns: "repeat(3, 1fr)",   gap: "20px",
    padding: "20px",   width: "100%",   boxSizing: "border-box",
    "@media (max-width: 900px)": {
      gridTemplateColumns: "repeat(auto-fit, minmax(250px, 1fr))",   },  },
  chartBox: {   backgroundColor: darkMode ? "#1e1e1e" : "#fff",   padding: "10px",
    borderRadius: "8px",
    boxShadow: darkMode ? "0 0 8px rgba(255,255,255,0.05)" : "0 0 10px rgba(0,0,0,0.1)",
    height: "300px",   display: "flex",   flexDirection: "column",  },
  speedometerBox: {   backgroundColor: darkMode ? "#1e1e1e" : "#fff",
    padding: "10px",   borderRadius: "8px",
    boxShadow: darkMode ? "0 0 8px rgba(255,255,255,0.05)" : "0 0 10px rgba(0,0,0,0.1)",
    height: "300px",   width: "95%",   display: "flex",   flexDirection: "column",
    justifyContent: "center",   alignItems: "center",  },
  mapBox: {   backgroundColor: darkMode ? "#1e1e1e" : "#fff",   padding: "10px",
    borderRadius: "8px",
    boxShadow: darkMode ? "0 0 8px rgba(255,255,255,0.05)" : "0 0 10px rgba(0,0,0,0.1)",
    height: "300px",  },
  placeholderBox: {   backgroundColor: darkMode ? "#2a2a2a" : "#f0f0f0",
    borderRadius: "8px",   height: "300px",  },
  sectionTitle: {   fontWeight: "bold",   fontSize: "22px",   marginTop: "30px",
    marginBottom: "10px",   color: darkMode ? "#fff" : "#000",  },
  chartTitle: {    fontWeight: 600,    color: darkMode ? "#fff" : "#000",    marginBottom: 10,
  },
  tableContainer: {    width: "90%",    margin: "30px auto",  },
  table: {    width: "100%",    borderCollapse: "collapse",
    backgroundColor: darkMode ? "#1e1e1e" : "#fff",    color: darkMode ? "#fff" : "#000", },
  th: {    padding: "10px",    border: "1px solid #888",
  backgroundColor: darkMode ? "#333" : "#f1f1f1",    color: darkMode ? "#fff" : "#000",
    fontWeight: "bold",
  },
  td: {    padding: "10px",    border: "1px solid #888",    textAlign: "center",  },
});

export default DashboardPage;
#================================================================
# NavBar.js
#================================================================
import React from "react";
import { Link, useNavigate } from "react-router-dom";

function NavBar() {
  const navigate = useNavigate();
  const user = JSON.parse(localStorage.getItem("user")); // 👈 Fetch user from localStorage

  const handleLogout = () => {
    localStorage.clear();
    alert("Logged out successfully!");
    navigate("/login");
  };

  return (
    <nav style={{ backgroundColor: "#282c34", padding: "10px", color: "white", display:
"flex", justifyContent: "space-between", alignItems: "center" }}>
      <div>
            <Link to="/dashboard" style={{ marginRight: "20px", color: "white"
}}>Dashboard</Link>

      {/* 👇 Conditionally show Manage Users link if role is Admin */}
      {user && user.role === "Admin" && (
        <Link to="/manage-users" style={{ color: "white" }}>Manage Users</Link>
      )}
      </div>
      <button onClick={handleLogout} style={styles.logoutButton}>Logout</button>
    </nav>
  );
}

const styles = {
  logoutButton: {    background: "red",    color: "white",    padding: "8px 12px",
    border: "none",    borderRadius: "50px",    cursor: "pointer",  },
};

export default NavBar;
#===============================================================
# LoginPage.js
#===============================================================
import React, { useState } from "react";
import { useNavigate } from "react-router-dom";

const LoginPage = () => {
  const [formData, setFormData] = useState({ email: "", password: "" });
  const [totp, setTotp] = useState("");
  const [qrData, setQrData] = useState(null);
  const [step, setStep] = useState(1); // 1 = login, 2 = QR, 3 = TOTP
  const [error, setError] = useState("");
  const [loading, setLoading] = useState(false);
  const [rememberMe, setRememberMe] = useState(false);
  const [showPassword, setShowPassword] = useState(false); // 👁 toggle
  const navigate = useNavigate();

  const handleChange = (e) => {
    setFormData({ ...formData, [e.target.name]: e.target.value });
  };

  const handleLogin = async (e) => {
    e.preventDefault();
    setError("");
    setLoading(true);

    try {
      const res = await fetch("http://localhost:5000/api/auth/login", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify(formData),
      });

      const data = await res.json();
      if (!res.ok) throw new Error(data.error || "Login failed");

      if (data.qrCode) {
        setQrData({ qrCode: data.qrCode, manualCode: data.manualCode });
        setStep(2);
      } else if (data.requiresTOTP) {
        setStep(3);
      } else {
        throw new Error("Unexpected server response");
      }
    } catch (err) {
      setError(err.message || "Login failed");
    } finally {
      setLoading(false);
    }
  };

  const handleVerifyTotp = async () => {
    try {
      const res = await fetch("http://localhost:5000/api/auth/verify-totp", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ email: formData.email, token: totp }),
      });

      const data = await res.json();
      if (!res.ok) throw new Error(data.error || "Invalid code");

      if (rememberMe) {
        localStorage.setItem("token", data.token);
        localStorage.setItem("role", data.user?.role || "viewer");
        localStorage.setItem("email", data.user?.email || "");
        localStorage.setItem("username", data.user?.username || "");
      } else {
        sessionStorage.setItem("token", data.token);
        sessionStorage.setItem("role", data.user?.role || "viewer");
        sessionStorage.setItem("email", data.user?.email || "");
        sessionStorage.setItem("username", data.user?.username || "");
      }


      alert("Login successful!");
      navigate("/dashboard");
    } catch (err) {
```

```
    setError(err.message || "TOTP verification failed");
  }
};

return (
  <div style={styles.container1}>
  <div style={styles.container}>
    <h2 style={{ color: "whitesmoke"}}>Login</h2>
    {error && <p style={styles.error}>{error}</p>}

    {step === 1 && (
     <form onSubmit={handleLogin} style={styles.form}>
       <input
         type="email"
         name="email"
         placeholder="Email"
         value={formData.email}
         onChange={handleChange}
         required
         style={styles.input}
       />

       <div style={{ position: "relative" }}>
         <input
           type={showPassword ? "text" : "password"}
           name="password"
           placeholder="Password"
           value={formData.password}
           onChange={handleChange}
           required
           style={{ ...styles.input, paddingRight: "10px" }}
         />
         <span
           onClick={() => setShowPassword(!showPassword)}
           style={{
             position: "absolute",
             right: "0px",
             top: "50%",
             transform: "translateY(-50%)",
             cursor: "pointer",
             userSelect: "none",
           }}
         >
```

```
          {showPassword ? "🙈" : "👁"}
        </span>
      </div>

      <label style={{ fontSize: "14px", marginBottom: "5px",color:'Window' }}>
        <input
          type="checkbox"
          checked={rememberMe}
          onChange={() => setRememberMe(!rememberMe)}
          style={{ marginRight: "5px" }}
        />
        Remember this device
      </label>

      <button type="submit" disabled={loading} style={styles.button}>
        {loading ? "Logging in..." : "Login"}
      </button>

      <b><p style={{ color: "white" }}>
        New User?{" "}
        <a href="/signup" style={{ color: "white" }}>
          Register Here
        </a>
        <br />
        <a href="/forgot-password" style={{ color: "yellow" }}>
          Forgot Password?
        </a>
      </p></b>
    </form>
  )}

  {step === 2 && qrData && (
    <div style={styles.form}>
          <p style={{ color:"whitesmoke" }}>📱 Scan this QR code using Google
Authenticator:</p>
        <img src={qrData.qrCode} alt="QR Code" style={{ width: "200px", margin: "10px
0" }} />
      <p style={{ color:"whitesmoke" }}>
              🔐 Or enter this code manually: <b style={{ color:"whitesmoke"
}}>{qrData.manualCode}</b>
      </p>
      <button onClick={() => setStep(3)} style={styles.button}>
        I've set it up
        </button>
      </div>
    )}

    {step === 3 && (
      <div style={styles.form}>
        <p style={{color:'yellow'}}>Enter the 6-digit code from Google Authenticator</p>
        <input
          type="text"
          value={totp}
          onChange={(e) => setTotp(e.target.value)}
          placeholder="Enter TOTP"
          style={styles.input}
        />
        <button onClick={handleVerifyTotp} style={styles.button}>
          Verify Code
        </button>
      </div>
    )}
  </div></div>
 );
};

const styles = {
 container1: {    minHeight: "100vh",    padding: "30px",    backgroundColor: "#f4f4f4",
   backgroundImage: "url('/images/Login.jpg')",    backgroundSize: "cover",
   backgroundPosition: "center",    backgroundRepeat: "no-repeat",
   fontFamily: "Arial, sans-serif",
 },
 container: { width: "300px", margin: "150px auto", textAlign: "center" },
 form: { display: "flex", flexDirection: "column", gap: "10px" },
 input: {
   padding: "10px",    fontSize: "16px",    borderRadius: "10px",    border: "1px solid #ccc",
   width: "100%",
 },
 button: {    padding: "10px",    fontSize: "16px",    backgroundColor: "darkblue",
   color: "white",    border: "none",    borderRadius: "5px",    cursor: "pointer",
   boxShadow: "0 0 8px 2px rgba(0, 123, 255, 0.7)",
   transition: "box-shadow 0.3s ease-in-out",
 },
 buttonActive: {    boxShadow: "0 0 15px 5px rgba(0, 123, 255, 1)",
 },
 error: { color: "red", fontWeight: "bold" },
};
```

```
export default LoginPage;
#====================================================================
# ForgotPasswordPage.js
#====================================================================
import React, { useState } from "react";
import axios from "axios";
import { useNavigate } from "react-router-dom";

function ForgotPasswordPage() {
  const [email, setEmail] = useState("");
  const [step, setStep] = useState(1);
  const [otp, setOtp] = useState("");
  const [newPassword, setNewPassword] = useState("");
  const [showPassword, setShowPassword] = useState(false);
  const navigate = useNavigate();

  const handleEmailSubmit = async (e) => {
    e.preventDefault();
    try {
      const res = await axios.post("http://localhost:5000/api/auth/request-reset", { email });
      alert(res.data.message);
      setStep(2);
    } catch (err) {
      alert(err.response?.data?.error || "Something went wrong!");
    }
  };

  const handleResetPassword = async (e) => {
    e.preventDefault();
    try {
      const res = await axios.post("http://localhost:5000/api/auth/reset-password", {
        email,
        otp,
        newPassword,
      });
      alert(res.data.message);
      navigate("/login");
    } catch (err) {
      alert(err.response?.data?.error || "Something went wrong!");
    }
  };

  return (
  <div style={styles.container1}>
  <div style={styles.container}>
   {step === 1 ? (
    <>
     <h2>Forgot Your Password?</h2>
     <form onSubmit={handleEmailSubmit} style={styles.form}>
      <input
       type="email"
       placeholder="Enter your registered email"
       value={email}
       onChange={(e) => setEmail(e.target.value)}
       required
       style={styles.input}
      />
      <button type="submit" style={styles.button}>Send OTP</button>
      <p><a href="/login" style={{ color: "yellow" }}>Back to Login</a></p>
     </form>
    </>
   ) : (
    <>
     <h2>Enter OTP & New Password</h2>
     <form onSubmit={handleResetPassword} style={styles.form}>
      <input
       type="text"
       placeholder="Enter 6-digit OTP"
       value={otp}
       onChange={(e) => setOtp(e.target.value)}
       required
       style={styles.input}
      />
      <div style={{ position: "relative", width: "300px" }}>
       <input
        type={showPassword ? "text" : "password"}
        placeholder="Enter new password"
        value={newPassword}
        onChange={(e) => setNewPassword(e.target.value)}
        required
        style={{ ...styles.input, width: "93%" }}
       />
       <span
        onClick={() => setShowPassword(!showPassword)}
        style={{
         position: "absolute",
         top: "50%",
            right: "5px",
            transform: "translateY(-50%)",
            cursor: "pointer",
            fontSize: "18px",
            userSelect: "none",
          }}
        >
          {showPassword ? "🙈" : "👁"}
        </span>
      </div>
      <button type="submit" style={styles.button}>Reset Password</button>
      <p><a href="/login" style={{ color: "yellow" }}>Back to Login</a></p>
    </form>
  </>
 )}
 </div></div>
 );
}

const styles = {
 container1: {    minHeight: "100vh",    padding: "30px",    backgroundColor: "#f4f4f4",
   backgroundImage: "url('/images/Signup.jpg')",    backgroundSize: "cover",
   backgroundPosition: "center",    backgroundRepeat: "no-repeat",
   fontFamily: "Arial, sans-serif",
 },
 container: { textAlign: "center", padding: "40px" , margin:"90px auto"},
 form: { display: "flex", flexDirection: "column", alignItems: "center", gap: "10px" },
 input: { padding: "10px", width: "280px" },
 button: { padding: "10px 20px", backgroundColor: "blue", color: "white", border: "none"
},
};

export default ForgotPasswordPage;
#=============================================================
# ResetPasswordPage.js
#=============================================================
import React, { useState } from "react";
import axios from "axios";
import { useParams, useNavigate } from "react-router-dom";

function ResetPasswordPage() {
 const { token } = useParams();
 const navigate = useNavigate();
 const [newPassword, setNewPassword] = useState("");

 const handleReset = async (e) => {
   e.preventDefault();
   try {
     const res = await axios.post(`http://localhost:5000/api/auth/reset-password/${token}`, {
       newPassword,
     });
     alert(res.data.message);
     navigate("/login"); // ✅ redirect to login
   } catch (err) {
     alert(err.response?.data?.error || "Something went wrong!");
   }
 };

 return (
   <div style={styles.container}>
     <h2>Reset Your Password</h2>
     <form onSubmit={handleReset} style={styles.form}>
       <input
         type="password"
         placeholder="Enter new password"
         value={newPassword}
         onChange={(e) => setNewPassword(e.target.value)}
         required
         style={styles.input}
       />
       <button type="submit" style={styles.button}>Reset Password</button>
       <p><a href="/login">Back to Login</a></p>
     </form>
   </div>
 );
}

const styles = {
 container: { textAlign: "center", padding: "40px" },
 form: { display: "flex", flexDirection: "column", alignItems: "center", gap: "10px" },
 input: { padding: "10px", width: "300px" },
 button: { padding: "10px 20px", backgroundColor: "blue", color: "white", border: "none"
},
};

export default ResetPasswordPage;
```

```
#==================================================================
# ManageUserPage.js
#==================================================================
import React, { useEffect, useState } from 'react';
import axios from 'axios';

// Helper function to get token from localStorage or sessionStorage
const getToken = () => {
  return localStorage.getItem("token") || sessionStorage.getItem("token");
};

const ManageUsersPage = () => {
  const [users, setUsers] = useState([]);
  const [error, setError] = useState(null);
  const [darkMode, setDarkMode] = useState(localStorage.getItem("theme") === "dark");
  const [page, setPage] = useState(1);
  const perPage = 10;

  const fetchUsers = async () => {
    try {
      const token = getToken();
      if (!token) {
        setError("No authentication token found. Please log in.");
        return;
      }
      const res = await axios.get('http://localhost:5000/api/admin/users', {
        headers: { Authorization: `Bearer ${token}` },
        withCredentials: true
      });
      setUsers(res.data);
      setError(null);
    } catch (error) {
        console.error('Error fetching users:', error.response ? error.response.data :
error.message);
       setError(error.response ? error.response.data.message || error.response.data.error :
error.message);
    }
  };

  const handleAction = async (id, action, data = {}) => {
    try {
      const token = getToken();
      if (!token) {
        setError("No authentication token found. Please log in.");
        return;
      }
      const url = `http://localhost:5000/api/admin/users/${id}${action}`;
      await axios[action === '' ? 'put' : 'patch'](url, data, {
        headers: { Authorization: `Bearer ${token}` },
        withCredentials: true
      });
      fetchUsers();
    } catch (error) {
          console.error('Error performing action:', error.response ? error.response.data :
error.message);
        setError(error.response ? error.response.data.message || error.response.data.error :
error.message);
    }
  };

  const handleDelete = async (id) => {
    try {
      const token = getToken();
      if (!token) {
        setError("No authentication token found. Please log in.");
        return;
      }
      await axios.delete(`http://localhost:5000/api/admin/users/${id}`, {
        headers: { Authorization: `Bearer ${token}` },
        withCredentials: true
      });
      fetchUsers();
    } catch (error) {
      console.error('Error deleting user:', error.response ? error.response.data : error.message);
        setError(error.response ? error.response.data.message || error.response.data.error :
error.message);
    }
  };

  useEffect(() => {
    fetchUsers();
  }, []);

  const pageData = users.slice((page - 1) * perPage, page * perPage);

  useEffect(() => {
    document.body.className = darkMode ? "dark-mode" : "light-mode";
    localStorage.setItem("theme", darkMode ? "dark" : "light");
```

```
 }, [darkMode]);

 return (
   <div style={styles(darkMode).container}>
     <h2 style={styles(darkMode).header}>Manage Users</h2>
     <div style={styles(darkMode).controls}>
                                <button onClick={() => setDarkMode(!darkMode)} 
style={styles(darkMode).darkToggle}>
         {darkMode ? "🌞 Light Mode" : "🌙 Dark Mode"}
       </button>
     </div>
     {error && <div style={styles(darkMode).error}>{error}</div>}
     <div style={{ overflowX: "auto" }}>
       <table style={styles(darkMode).table}>
         <thead>
           <tr>
             <th style={styles(darkMode).th}>No.</th>
             <th style={styles(darkMode).th}>Email</th>
             <th style={styles(darkMode).th}>Username</th>
             <th style={styles(darkMode).th}>Role</th>
             <th style={styles(darkMode).th}>Approved</th>
             <th style={styles(darkMode).th}>Disabled</th>
             <th style={styles(darkMode).th}>TOTPSecret</th>
             <th style={styles(darkMode).th}>Created At</th>
             <th style={styles(darkMode).th}>Updated At</th>
             <th style={styles(darkMode).th}>Actions</th>
           </tr>
         </thead>
         <tbody>
           {pageData.map((user, index) => (
                 <tr key={user.id} style={index % 2 === 0 ? styles(darkMode).evenRow : 
styles(darkMode).oddRow}>
               <td style={styles(darkMode).td}>{(page - 1) * perPage + index + 1}</td>
               <td style={styles(darkMode).td}>{user.email}</td>
               <td style={styles(darkMode).td}>{user.username}</td>
               <td style={styles(darkMode).td}>
                 <select
                   value={user.role}
                   onChange={(e) => handleAction(user.id, '', { role: e.target.value })}
                   style={styles(darkMode).select}
                 >
                   <option value="admin">admin</option>
                   <option value="operator">operator</option>
```

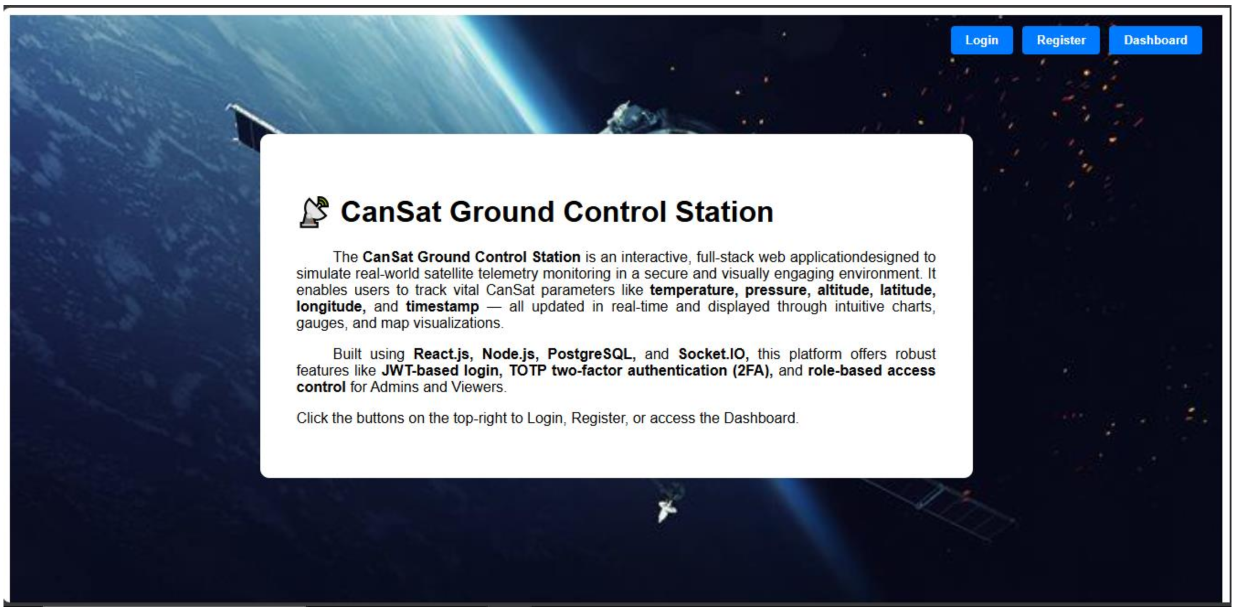
```
          <option value="viewer">viewer</option>
         </select>
        </td>
       <td style={{ ...styles(darkMode).td, textAlign: 'center' }}>{user.isApproved ? '✅'
: '❌'}</td>
        <td style={{ ...styles(darkMode).td, textAlign: 'center' }}>{user.isDisabled ? '⛔'
: '✔️'}</td>
           <td style={{ ...styles(darkMode).td, width: "80px", whiteSpace: "nowrap",
overflow: "hidden", textOverflow: "ellipsis" }}>{user.totpSecret || 'N/A'}</td>
                <td style={styles(darkMode).td}>{user.createdAt ? new
Date(user.createdAt).toLocaleString() : 'N/A'}</td>
                <td style={styles(darkMode).td}>{user.updatedAt ? new
Date(user.updatedAt).toLocaleString() : 'N/A'}</td>
        <td style={styles(darkMode).td}>
         {!user.isApproved && <button onClick={() => handleAction(user.id, '/approve')}
style={styles(darkMode).actionButton}>Approve</button>}
          {!user.isDisabled && <button onClick={() => handleAction(user.id, '/disable')}
style={styles(darkMode).actionButton}>Disable</button>}
                <button onClick={() => handleDelete(user.id)} style={{
...styles(darkMode).actionButton, ...styles(darkMode).deleteButton }}>Delete</button>
        </td>
       </tr>
      ))}
     </tbody>
    </table>
   </div>
   <div style={styles(darkMode).pagination}>
    <button onClick={() => setPage(p => Math.max(p - 1, 1))} disabled={page === 1}
style={styles(darkMode).paginationButton}>◀ Prev</button>
    <span style={styles(darkMode).paginationInfo}>Page {page}</span>
     <button onClick={() => setPage(p => (p * perPage < users.length ? p + 1 : p))}
disabled={page * perPage >= users.length}
style={styles(darkMode).paginationButton}>Next ▶</button>
   </div>
  </div>
 );
};

const styles = (darkMode) => ({
 container: {   textAlign: "center",   padding: "20px",   minHeight: "100vh",
  backgroundColor: darkMode ? "#121212" : "#f8f9fa",
  color: darkMode ? "#fff" : "#000",   overflowX: "hidden",  },
 header: { fontSize: "28px", fontWeight: "bold", marginBottom: "20px" },
  controls: {    display: "flex",    justifyContent: "center",    marginBottom: "20px",  },
  darkToggle: {    padding: "10px 16px",
    backgroundColor: darkMode ? "#555" : "#007bff",    color: "#fff",    border: "none",
    borderRadius: "5px",    cursor: "pointer",  },
  error: {    marginBottom: "20px",    padding: "10px",    backgroundColor: "#f8d7da",
    color: "#721c24",    borderRadius: "5px",    maxWidth: "600px",
    margin: "0 auto 20px auto",  },
  table: {    width: "100%",    borderCollapse: "collapse",
    backgroundColor: darkMode ? "#1e1e1e" : "#fff",   color: darkMode ? "#fff" : "#000",  },
  th: {    padding: "10px",    border: "1px solid #888",
    backgroundColor: darkMode ? "#333" : "#f1f1f1",
    color: darkMode ? "#fff" : "#000",    fontWeight: "bold",  },
  td: {    padding: "10px",    border: "1px solid #888",    textAlign: "center",  },
  evenRow: {    backgroundColor: darkMode ? "#2a2a2a" : "#f9f9f9",  },
  oddRow: {    backgroundColor: "transparent",  },
  select: {    padding: "6px",    borderRadius: "4px",    border: "1px solid #ccc",
    fontSize: "14px",  },
  actionButton: {    margin: "2px",    padding: "4px 10px",    borderRadius: "4px",
    border: "none",    cursor: "pointer",    backgroundColor: "#28a745",    color: "#fff",  },
  deleteButton: {    backgroundColor: "#dc3545",  },
  pagination: {    marginTop: "15px",    display: "flex",    justifyContent: "center",
    alignItems: "center",    gap: "10px",  },
  paginationButton: {    padding: "6px 12px",    borderRadius: "4px",
    border: "1px solid #888",    backgroundColor: darkMode ? "#333" : "#f1f1f1",
    color: darkMode ? "#fff" : "#000",    cursor: "pointer",  },
  paginationInfo: {    fontWeight: "bold",  },
});

export default ManageUsersPage;
```

8.2 USER INTERFACE (SCREENSHOTS)

ABOUTPAGE



CanSat Ground Control Station

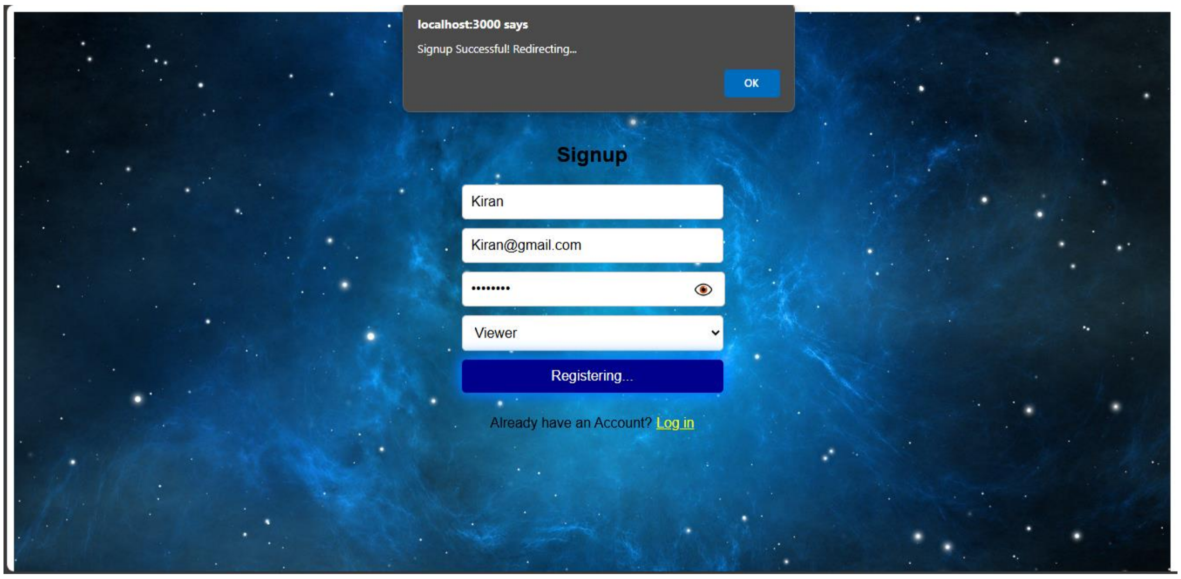
The **CanSat Ground Control Station** is an interactive, full-stack web applicationdesigned to simulate real-world satellite telemetry monitoring in a secure and visually engaging environment. It enables users to track vital CanSat parameters like **temperature, pressure, altitude, latitude, longitude,** and **timestamp** — all updated in real-time and displayed through intuitive charts, gauges, and map visualizations.

Built using **React.js, Node.js, PostgreSQL,** and **Socket.IO,** this platform offers robust features like **JWT-based login, TOTP two-factor authentication (2FA),** and **role-based access control** for Admins and Viewers.

Click the buttons on the top-right to Login, Register, or access the Dashboard.

Figure No 8.2.1: ABOUTPAGE

SIGNUPPAGE



localhost:3000 says

Signup Successful! Redirecting...

Kiran

Kiran@gmail.com

••••••••

Viewer

Already have an Account? Log in

Figure No 8.2.2: SIGNUPPAGE

LOGINPAGE

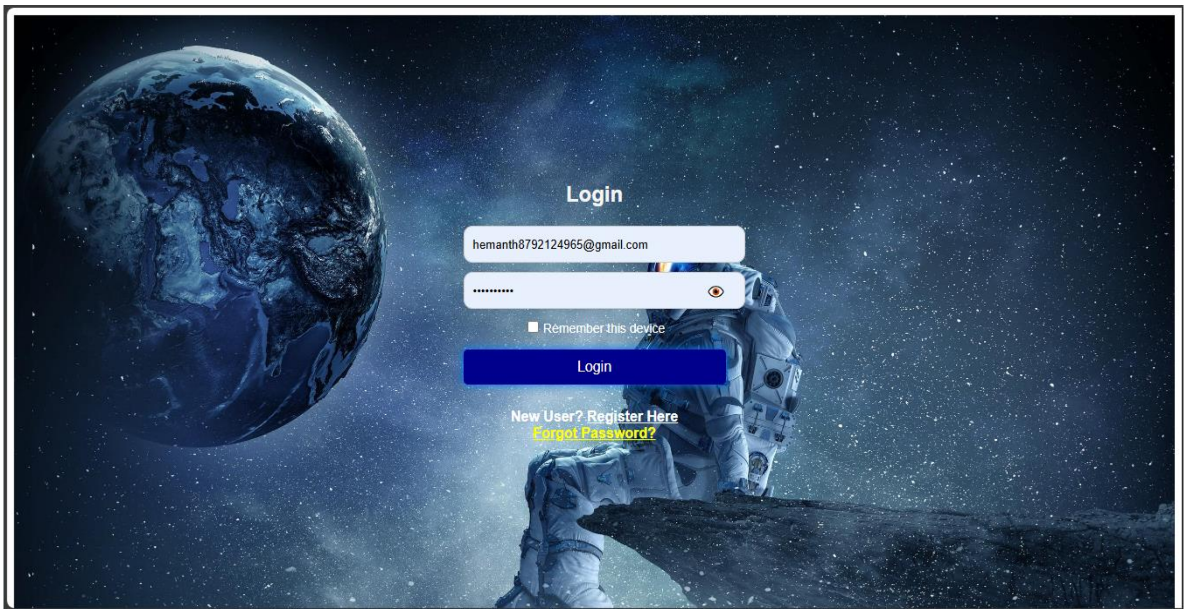
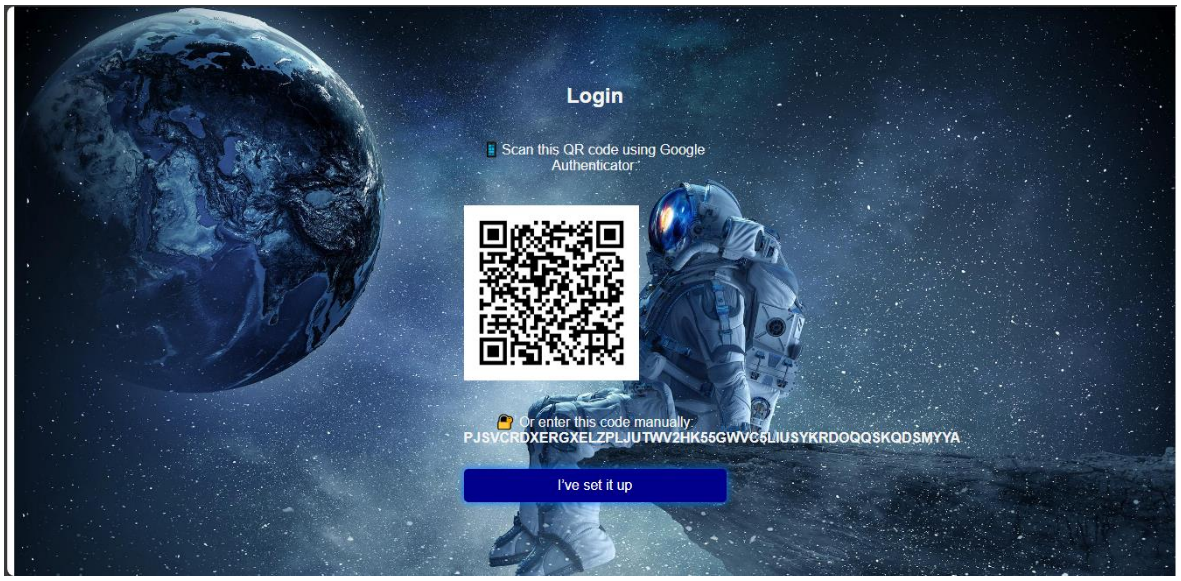


Figure No 8.2.3: LOGINPAGE

TOTP_SETUP

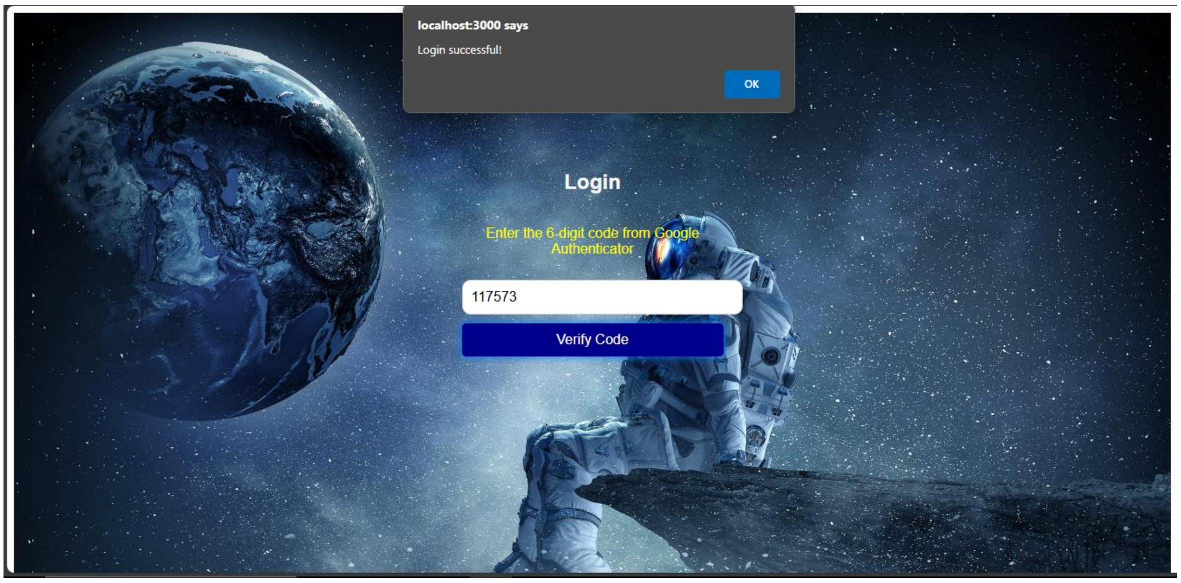


Scan this QR code using Google Authenticator:

Or enter this code manually:
PJSVCRDXERGXELZPLJUTWV2HK55GWVC5LIUSYKRDOQQSKQDSMYYA

Figure No 8.2.4: TOTP_SETUP

TOTPPAGE

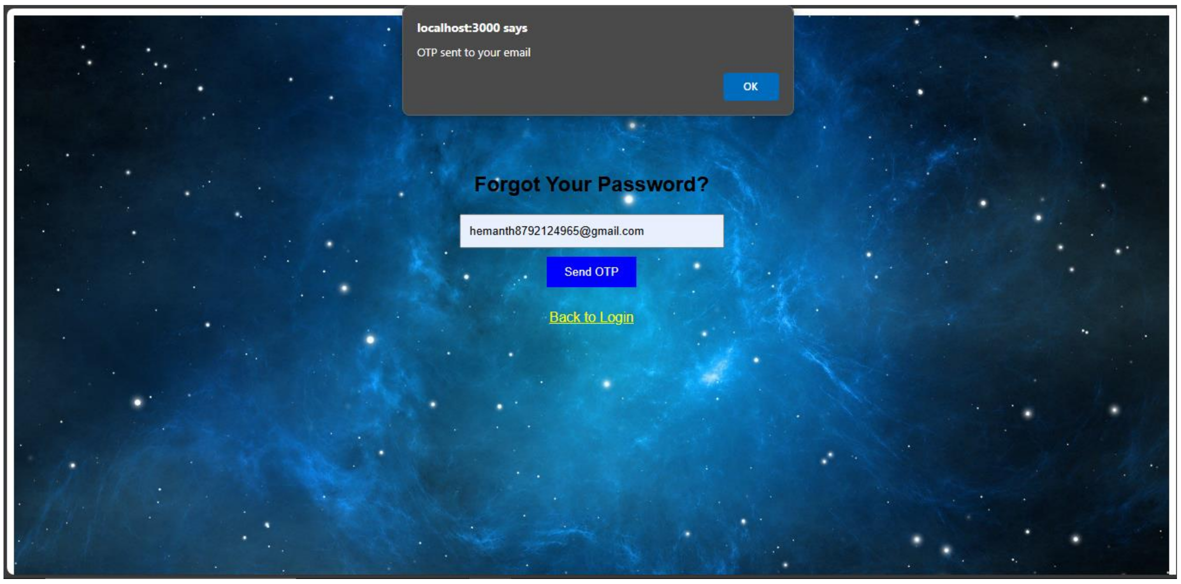

localhost:3000 says

Login successful!

Figure No 8.2.5: TOTPPAGE

FORGOTPASSWORDPAGE

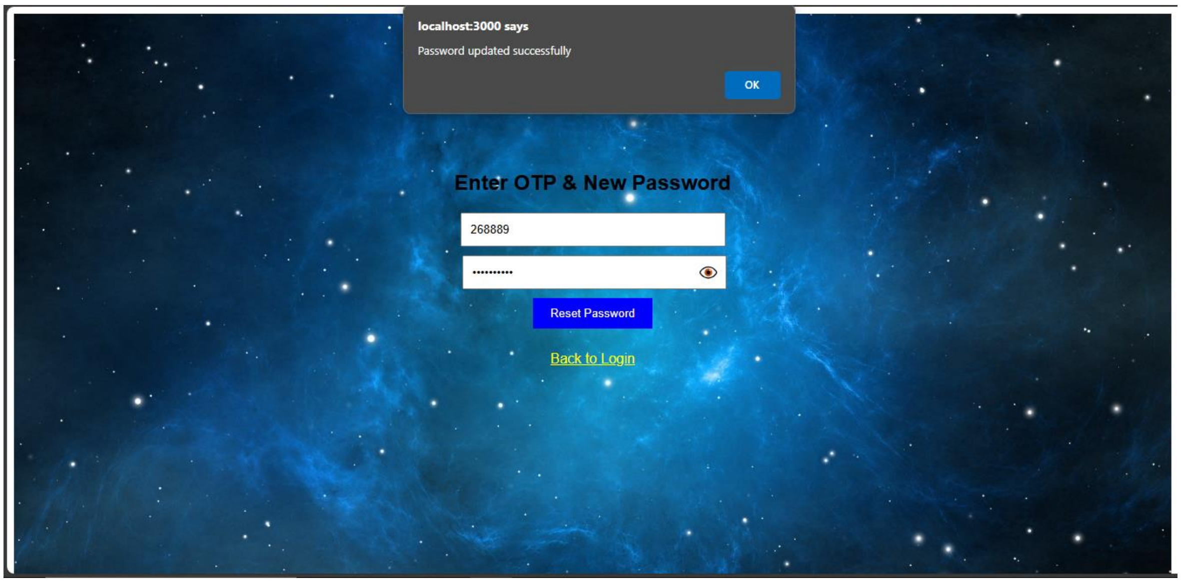
EMAIL


localhost:3000 says

OTP sent to your email

Figure No 8.2.6.1: EMAIL

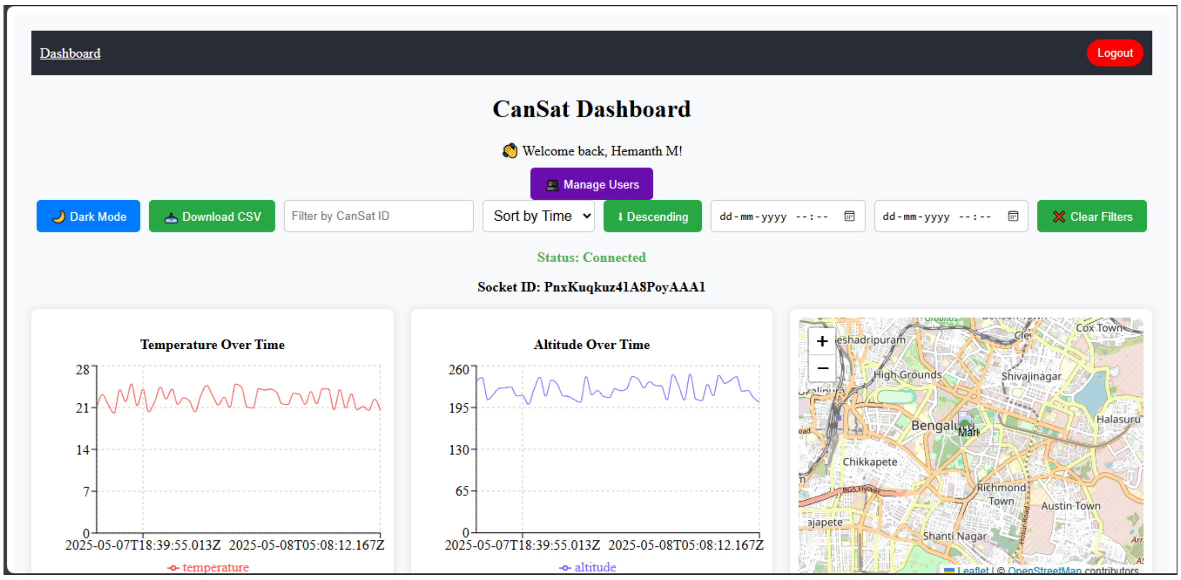
OTP & NEWPASSWORD


localhost:3000 says

Password updated successfully

Figure No 8.2.6.2: OTP & NEWPASSWORD

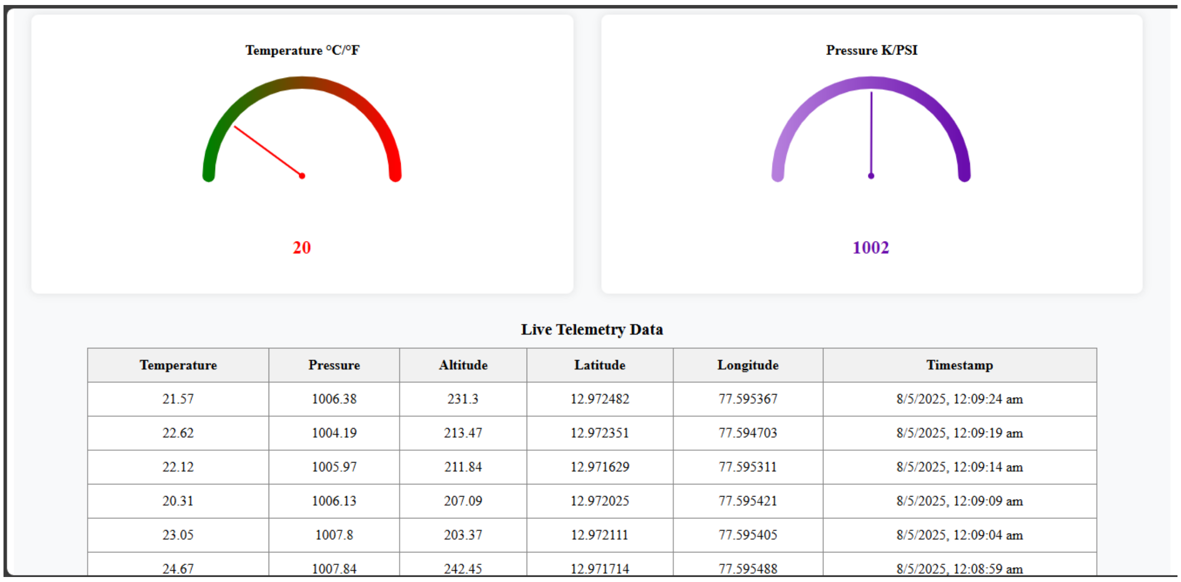
DASHBOARDPAGE



Status: Connected

Socket ID: PnxKuqkuz41A8PoyAAA1

Figure No 8.2.7: DASHBOARDPAGE

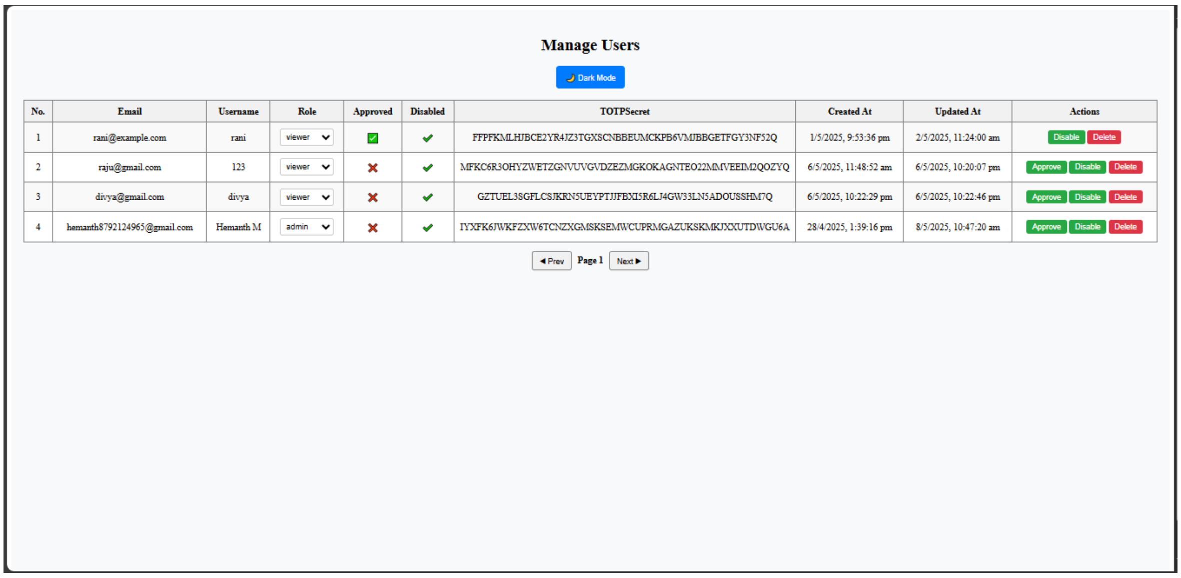


Live Telemetry Data

Temperature	Pressure	Altitude	Latitude	Longitude	Timestamp
21.57	1006.38	231.3	12.972482	77.595367	8/5/2025, 12:09:24 am
22.62	1004.19	213.47	12.972351	77.594703	8/5/2025, 12:09:19 am
22.12	1005.97	211.84	12.971629	77.595311	8/5/2025, 12:09:14 am
20.31	1006.13	207.09	12.972025	77.595421	8/5/2025, 12:09:09 am
23.05	1007.8	203.37	12.972111	77.595405	8/5/2025, 12:09:04 am
24.67	1007.84	242.45	12.971714	77.595488	8/5/2025, 12:08:59 am

Figure No 8.2.8: DASHBOARDPAGE

MANAGEUSERPAGE


Manage Users

No.	Email	Username	Role	Approved	Disabled	TOTPSecret	Created At	Updated At	Actions
1	rani@example.com	rani	viewer	✓	✓	FFPFKMLHJBCE2YR4JZ3TGXSCNBBEUMCKPB6VMJBBGETFGY3NF52Q	1/5/2025, 9:53:36 pm	2/5/2025, 11:24:00 am	Disable Delete
2	raju@gmail.com	123	viewer	✗	✓	MFKC6R3OHYZWETZGNVUVGVDZEZMGKOKAGNTEO22MMVEEIM2QOZYQ	6/5/2025, 11:48:52 am	6/5/2025, 10:20:07 pm	Approve Disable Delete
3	divya@gmail.com	divya	viewer	✗	✓	GZTUEL3SGFLCSJKRN5UEYPTJJFBXI5R6LJ4GW33LN5ADOUSSHM7Q	6/5/2025, 10:22:29 pm	6/5/2025, 10:22:46 pm	Approve Disable Delete
4	hemanth8792124965@gmail.com	Hemanth M	admin	✗	✓	IYXFK6JWKFZXW6TCNZXGMSKSEMWCUPRMGAZUKSKMKJXXUTDWGU6A	28/4/2025, 1:39:16 pm	8/5/2025, 10:47:20 am	Approve Disable Delete

Figure No 8.2.9: MANAGEUSERPAGE

9. SOFTWARE TESTING

Software testing ensures that the system performs as expected and meets both functional and non-functional requirements. For the **Ground Control Station for CanSats**, testing was done at various levels to verify:

Given the modular nature of the project, testing was carried out for:

- Real-time telemetry handling
- User authentication and TOTP-based 2FA
- Dashboard UI components (charts, tables, filtering)
- Telemetry data filtering and sorting
- Security and session management

The testing encompassed backend APIs, WebSocket connections, PostgreSQL database integrity, frontend UI behaviour, and overall application flow.

9.1 TYPES OF TESTING PERFORMED

Table No 9.1: Types of Testing Performed

Type of Testing	Description
Unit Testing	Validated backend services and utility functions (e.g., JWT, OTP handling)
Integration Testing	Verified API endpoints with DB, TOTP setup, and socket connection flow
Functional Testing	Tested telemetry reception, filtering, sorting, 2FA, login/logout actions
UI Testing	Manually tested charts, buttons, inputs, and dark mode toggle
Smoke Testing	Confirmed app boots, login works, and dashboard renders properly
Regression Testing	Re-tested modules after bug fixes (e.g., TOTP, datetime filtering)

9.2 TESTING TOOLS USED

- **Postman**: API testing (`/api/auth/login`, `/api/telemetry/filter`, `/api/auth/signup`)
- **Browser DevTools**: UI event inspection and WebSocket traffic
- **Jest (Optional)**: For unit testing backend utilities (e.g., token validation)
- **Manual Console Logs**: To verify backend and frontend processing during TOTP and socket events
- **Socket.IO Debug Mode**: For real-time socket event tracking

9.3 DETAILED TEST CASES

Test Case 1: User Signup and Email Storage

Table No 9.3.1: User Signup and Email Storage

Field	Details
Test Case ID	TC_SIGNUP_01
Test Scenario	New user registers
Test Steps	1. Open signup page 2. Fill form 3. Submit
Expected Result	User created and saved in DB
Actual Result	Signup successful, user stored
Status	Pass

Test Case 2: Login + TOTP First-Time Setup

Table No 9.3.2: Login + TOTP First-Time Setup

Field	Details
Test Case ID	TC_TOTP_01
Test Scenario	First login after signup shows QR code for Google Authenticator
Test Steps	1. Login with email/password 2. Receive QR code 3. Scan it
Expected Result	QR displayed, TOTP verified
Actual Result	QR rendered and verified successfully
Status	Pass

Test Case 3: TOTP Verification After First-Time Setup

Table No 9.3.3: TOTP Verification After First-Time Setup

Field	Details
Test Case ID	TC_TOTP_02
Test Scenario	Login using correct TOTP after initial setup
Test Steps	1. Login 2. Enter TOTP code 3. Redirect to dashboard
Expected Result	Access granted and session created

Actual Result	Login successful
Status	Pass

Test Case 4: Fetching Live Telemetry via Socket.IO

Table No 9.3.4: Fetching Live Telemetry via Socket.IO

Field	Details
Test Case ID	TC_SOCKET_01
Test Scenario	Receiving real-time telemetry stream
Test Steps	1. Open dashboard 2. Establish socket 3. View updates
Expected Result	Live telemetry shown in table/charts
Actual Result	Data received every 2 sec and displayed
Status	Pass

Test Case 5: Dark Mode Toggle

Table No 9.3.5: Dark Mode Toggle

Field	Details
Test Case ID	TC_UI_01
Test Scenario	Toggling between light and dark themes

Test Steps	1. Click toggle 2. Observe theme switch
Expected Result	UI switches theme and persists preference
Actual Result	Dark mode works as expected
Status	Pass

Test Case 6: Date Range Filtering (Backend)

Table No 9.3.6: Date Range Filtering

Field	Details
Test Case ID	TC_FILTER_01
Test Scenario	Backend filtering of telemetry by datetime range
Test Steps	1. Input `from` and `to` datetime 2. Submit
Expected Result	Only telemetry within range shown
Actual Result	Correct subset displayed in table and chart
Status	Pass

Test Case 7: Download Telemetry Data as CSV

Table No 9.3.7: Download Telemetry Data as CSV

Field	**Details**
Test Case ID	TC_EXPORT_01
Test Scenario	Exporting telemetry table as CSV
Test Steps	1. Click export button
Expected Result	CSV file downloaded
Actual Result	File downloaded with telemetry records
Status	Pass

Test Case 8: Unauthorized Dashboard Access

Table No 9.3.8: Unauthorized Dashboard Access

Field	**Details**
Test Case ID	TC_AUTH_01
Test Scenario	Access dashboard without login
Test Steps	1. Clear storage 2. Visit `/dashboard` URL
Expected Result	Redirect to login page
Actual Result	Access blocked and redirected
Status	Pass

Test Case 9: Sort Toggle (Ascending/Descending)

Table No 9.3.9: Sort Toggle

Field	Details
Test Case ID	TC_SORT_01
Test Scenario	Toggling telemetry sort order
Test Steps	1. Click sort button
Expected Result	Table and charts update accordingly
Actual Result	Sort order reflected correctly
Status	Pass

Test Case 10: Upload Invalid TOTP Code

Table No 9.3.10: Upload Invalid TOTP Code

Field	Details
Test Case ID	TC_TOTP_FAIL_01
Test Scenario	Entering incorrect TOTP after login
Test Steps	1. Login 2. Enter wrong TOTP code
Expected Result	“Invalid TOTP code” shown
Actual Result	Error displayed
Status	Pass

10. CONCLUSION

The project titled **“GROUND Control Station for CanSats”** delivers a secure, responsive, and real-time telemetry monitoring platform designed for academic and research-focused CanSat missions. It incorporates modern web technologies like React, Node.js, and PostgreSQL to simulate real-world satellite ground station operations. Key features include a real-time telemetry dashboard using Socket.IO, which streams live CanSat data such as temperature, pressure, altitude, latitude, longitude, and timestamp. The frontend offers intuitive visualizations, filtering by datetime range, sort toggles, and CSV export for offline analysis, all wrapped in a clean, responsive user interface with dark mode support.

In addition to functionality, the platform emphasizes **security and scalability** through robust user authentication, including JWT-based login, role management, and Two-Factor Authentication (TOTP) using Google Authenticator. The backend architecture is modular and secure, protecting access via route guards and enabling secure password recovery through OTP-based email verification. This project demonstrates how low-cost satellite payloads like CanSats can be supported by professional-grade software solutions, offering a scalable, extensible framework suitable for real-time mission control, educational competitions, and research deployments.

11. FUTURE ENHANCEMENTS

While the current implementation of the **GROUND Control Station for CanSats** meets core functional and usability requirements, several enhancements are planned to increase scalability, user experience, and operational depth:

GPS Map Integration Integrate a live map using tools like Leaflet.js or Mapbox to display the CanSat's real-time location based on telemetry latitude and longitude. This adds geospatial context and is essential for mission tracking and recovery. Telemetry Alerts and Notifications Implement real-time threshold-based alerts (e.g., high temperature or low altitude). These can notify users via on-screen popups or optional email/SMS, enabling timely responses to critical conditions.

Mission Replay Mode Enable playback of previously stored telemetry logs to simulate past missions. This feature is valuable for training, analysis, troubleshooting, and academic presentations. Role-Based Access Control (RBAC) Introduce differentiated access for Admins, Engineers, and Viewers to ensure that sensitive functions (like user management or system settings) are restricted based on user roles, enhancing security and organization.

BIBLIOGRAPHY

1. A. Sharma, R. Gupta, and N. Kumar, “Development of a Low-Cost Ground Control Station for CanSat Telemetry Using Python,” *International Journal of Electronics and Communication Engineering*, vol. 14, no. 3, pp. 120–125, 2021.
2. X. Zhou and H. Li, “A Software-Defined Radio Framework for Satellite Telemetry Systems,” *IEEE Transactions on Aerospace and Electronic Systems*, vol. 58, no. 2, pp. 78–85, 2022.
3. M. Kaya and S. Öztürk, “Design and Implementation of Real-Time Data Visualization in CanSat Ground Stations,” *Proc. of the National Space Research Symposium*, 2020, pp. 98–104.
4. IEEE Aerospace Conference, “Next-Gen Ground Control with Node.js and React.js for CubeSat Telemetry,” *IEEE Aerospace Conf. Proc.*, 2023.
5. P. Lee, J. Zhang, and R. Thomas, “Adopting WebSocket and RESTful APIs for Lightweight IoT-Based Telemetry,” *Journal of Embedded Systems and Applications*, vol. 19, no. 1, pp. 33–40, 2023.

Referred Website links

1. Arduino Community, “ArduStation – A Serial-Based Ground Control Interface,” [Online]. Available: https://playground.arduino.cc/
2. NASA, “CanSat Competition: Ground Control Software Guidelines,” [Online]. Available: https://www.nasa.gov/cansat
3. UKHAS, “Tracking High-Altitude Balloons with LoRaWAN – HabHub Project,” [Online]. Available: https://ukhas.org.uk/
4. Libre Space Foundation, “SatNOGS: An Open Network of Ground Stations,” [Online]. Available: https://satnogs.org/
5. Mozilla Developer Network, “Introduction to Socket.IO and Real-Time Web Applications,” [Online]. Available: https://developer.mozilla.org/

Plagiarism Checker X Originality Report

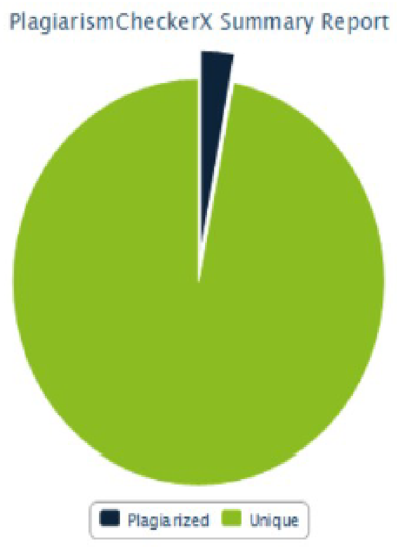
Similarity Found: 3%

Title: **Ground Control Station for CanSats**

Date: Monday, May 19, 2025

Statistics: 490 words Plagiarized / 15486 Total words

Remarks: Low Plagiarism Detected - Your Document needs Optional Improvement.



Date	Monday, May 19, 2025
Words	490 Plagiarized Words / Total 15486 Words
Sources	More than 76 Sources Identified.
Remarks	Low Plagiarism Detected - Your Document needs Optional Improvement.