Project Overview:

This documentation provides an in-depth overview of the architecture, components, and implementation plan for the IoT-Based Environmental Monitoring System. The system leverages free-tier technologies to ensure cost-effectiveness and includes features for real-time monitoring, basic analytics, alerts, and user management.

Hackathon: UST D3CODE Hackathon'24
Team Lead: Rishi Singh
Date: 08-09-2024

Formatted: Font: 16 pt

Formatted: Font: 16 pt

Formatted: Font: 16 pt

Formatted: Left

Project Documentation:

Formatted: Font: 16 pt

Formatted: Font: 11 pt

System Architecture Design

Formatted: Font: 14 pt

Architecture Overview This architecture leverages only free-tier technologies and services to ensure cost-effectiveness. It is designed to provide real-time monitoring, basic analytics, alerts, and user management for an IoT-based environmental monitoring system, with additional features for advanced analytics and user engagement.

1. Frontend:

- **Technology:** React.js
- **Components:**
  - **Dashboard:** Displays real-time data, alerts, and basic analytics.
  - **Monitoring Screens:** Show live sensor data visualizations and historical data trends.
  - **User Management:** Includes user login, profile management, and role-based access control.
  - **Mobile Application:** Dedicated mobile app for iOS and Android platforms to provide on-the-go access.
- **Implementation:** Use React.js libraries like Chart.js for data visualizations, and Axios or Fetch API for connecting to the backend. Implement responsive design to support mobile views.

2. Backend:

- **Technology:** Node.js with Express.js (REST API)
- **Components:**
  - **API Server:** Handles all API requests for real-time data, user authentication, and data processing.
  - **Authentication Service:** Manages user authentication (login/logout), session control, and access roles.
  - **Data Processing Service:** Processes incoming sensor data, performs simple analytics, and stores the data in the database.
  - **Automated Restoration Actions:** Integrates automation for certain restoration actions based on data analysis.
- **Implementation:** Use free-tier hosting solutions like Heroku or Vercel to deploy the backend.

3. Database:

- **Structured Data:**
  - **Technology:** PostgreSQL (Heroku Postgres Free Tier)
  - **Purpose:** Stores user accounts, system settings, and other structured information.
- **Unstructured Data:**
  - **Technology:** MongoDB Atlas (Free Tier)
  - **Purpose:** Stores unstructured sensor data logs and analytics.

4. Cloud Services:

- **Provider:** AWS (Free Tier) or Google Cloud (Free Tier)
- **Components:**
  - **Compute:** AWS Lambda or Google Cloud Functions (for data processing pipelines).
  - **Storage:** AWS S3 or Google Cloud Storage (for storing logs and data backups).
  - **Database:** AWS RDS (PostgreSQL) or Google Cloud SQL (PostgreSQL for structured data).

5. IoT Integration:

- **Protocols:** MQTT or HTTP (for communication between sensors and backend)
- **Components:**
  - **Sensors:** Use affordable, IoT-compatible sensors like DHT22 (for temperature and humidity).
  - **Gateway:** Aggregates sensor data and sends it to the backend via MQTT or HTTP.

Diagrams:

Formatted: Font: 14 pt

1. **System Architecture Diagram:**
   - Frontend: React components (Dashboard, Monitoring Screens, etc.) communicate with the backend via API.
   - Backend: Node.js handles API requests, processes sensor data, and stores it in the database.
   - Database: PostgreSQL and MongoDB are used to store structured and unstructured data.
   - Cloud Services: AWS/GCP components support data processing, storage, and hosting.
   - IoT Integration: Sensors send data through a gateway, which forwards the data to the backend.
2. **Data Flow Diagram:**
   - Sensor Data Flow: Sensors send real-time data to the backend via the gateway, which stores it in the database.
   - Data Processing: Backend processes the data and returns results/alerts to the frontend.
   - User Interaction: Users log in to view and manage data through the frontend interface, which pulls information from the backend.

Data Models

Formatted: Font: 14 pt

1. User Model:

- **Attributes:**
  - **UserID:** Unique identifier for each user.
  - **Name:** Full name of the user.
  - **Email:** User's email address.
  - **PasswordHash:** Hashed password for security.
  - **Role:** Role of the user (e.g., Admin, Standard User).
- **Purpose:** Stores user information, including access control permissions.

2. Sensor Data Model:

- **Attributes:**
  - **SensorID:** Unique identifier for each sensor.
  - **Timestamp:** The time at which the data was collected.
  - **SensorType:** Type of sensor (e.g., Temperature, Humidity).
  - **Location:** Physical location of the sensor.
  - **Data:** JSON format data containing sensor readings (e.g., { temperature: 22.5, humidity: 55 }).
- **Purpose:** Stores data from sensors in a flexible format, allowing for various types of sensors.

3. Restoration Activity Model:

- **Attributes:**
  - **ActivityID:** Unique identifier for each restoration activity.
  - **ActivityType:** Type of restoration action (e.g., Sensor Calibration, Maintenance).
  - **Description:** Detailed information about the activity.
  - **Timestamp:** Time when the activity was initiated.
  - **Status:** Status of the activity (e.g., Pending, Completed).
- **Purpose:** Tracks restoration activities and their progress/status.

4. Analytics Model:

- **Attributes:**
  - **AnalysisID:** Unique identifier for each analysis.
  - **Type:** Type of analysis performed (e.g., Trend Analysis, Anomaly Detection).
  - **Metrics:** JSON format data containing key metrics (e.g., averages, min/max values).
  - **Results:** Outcome of the analysis (e.g., anomalies detected).
- **Purpose:** Stores data from analytical processes and insights derived from sensor data.

Interfaces

1. Dashboard Wireframe:

- **Components:**
  - **Overview Panel:** Summary of system status, recent data, and alerts.
  - **Data Visualization:** Graphs and charts showing real-time sensor data (e.g., temperature trends).
  - **Notifications:** Alerts and messages related to system events or anomalies.
  - **Mobile App Interface:** Responsive design for mobile access.

2. Monitoring Screen Wireframe:

- **Components:**
  - **Live Data Feed:** Displays real-time sensor data in numerical and graphical format.
  - **Historical Data:** Graphs showing historical data for various time ranges (e.g., hourly, daily trends).
  - **Alerts:** Notification center for anomalies or abnormal sensor readings.

3. User Management Wireframe:

- **Components:**
  - **Login Page:** User authentication interface, with options for email and password entry.
  - **Profile Page:** Displays user details (e.g., name, email) and allows them to edit personal information.
  - **Permissions Management:** Interface for admins to assign or revoke roles and permissions from users.

Formatted: Font: 14 pt

MVP Development Plan

Formatted: Font: 16 pt

1. Core Features

Formatted: Font: 14 pt

1.1 Real-Time Monitoring:

- **Goal:** Display live data from sensors on the dashboard.
- **Implementation:**
  - Build a responsive React.js dashboard using WebSocket for real-time data fetching.
  - Implement a Node.js API that streams real-time sensor data from IoT devices.
  - Utilize Chart.js or D3.js for visualizing real-time data.

1.2 Basic Analytics:

- **Goal:** Provide simple analytics like trends and averages.
- **Implementation:**
  - Use Node.js to process incoming sensor data and calculate key metrics.
  - Visualize metrics such as average temperature or humidity trends using Chart.js.

1.3 Alerts System:

- **Goal:** Notify users of critical issues or anomalies.
- **Implementation:**
  - Build an alerting mechanism in the backend (e.g., anomaly detection using thresholds).
  - Use SendGrid's free tier for email notifications or in-app alert popups.

2. Pilot Setup

Formatted: Font: 14 pt

2.1 Prototype Sensors:

- **Goal:** Select a few sensors for testing the MVP.
- **Implementation:**
  - Integrate temperature and humidity sensors (e.g., DHT22), ensuring compatibility with MQTT/HTTP.

2.2 Pilot Environment:

- **Goal:** Deploy the system in a controlled test area.
- **Implementation:**
  - Use a small conservation area for testing, deploying a few sensors to gather real-time data.

2.3 Data Collection:

- **Goal:** Gather sensor data to test system functionality.
- **Implementation:**

- Use the real-time monitoring system to collect data, ensuring data accuracy and system reliability.

3. Build and Integrate

Formatted: Font: 14 pt

3.1 Frontend Development:

- **Build React Components:** Develop UI components for the dashboard, monitoring screens, and user management.
- **Connect to Backend:** Implement API calls using Axios to fetch real-time sensor data and user information.
- **Mobile App Development:** Develop and integrate mobile application features for enhanced accessibility.

3.2 Backend Development:

- **Build API Server:** Create RESTful endpoints for real-time data and user authentication.
- **Integrate IoT Sensors:** Implement endpoints for receiving sensor data from the MQTT/HTTP gateway.
- **Automated Actions:** Develop automation for certain restoration actions based on data analysis.

3.3 Data Processing

3.1 Data Storage:

- **Setup Databases:**
  - Use PostgreSQL (via Heroku Postgres) for structured data like user accounts.
  - Store unstructured sensor data in MongoDB Atlas Free Tier.
- **Data Ingestion:**
  - Implement pipelines to store sensor data, using AWS Lambda or Google Cloud Functions (free-tier) for lightweight processing.

3.2 Basic Analytics:

- **Data Processing:** Use in-memory processing to calculate metrics like averages, trends, and thresholds.
- **Visualization:** Display these insights using Chart.js on the React dashboard.

4. Additional Advanced Features

4.1 Advanced Analytics and Machine Learning:

- **Goal:** Implement predictive modeling and trend analysis.
- **Implementation:** Explore machine learning models for forecasting and anomaly detection.

4.2 Integration with Existing Systems:

- **Goal:** Enable data sharing with other environmental platforms.
- **Implementation:** Develop API endpoints for interoperability with external systems.

4.3 Community Reporting Tools:

- **Goal:** Allow users to report environmental issues.
- **Implementation:** Add forms and reporting tools to the frontend.

4.4 Multilingual Support:

- **Goal:** Make the platform accessible in multiple languages.
- **Implementation:** Implement i18n libraries in React and translation files.

4.5 Real-Time Alerts and Notifications:

- **Goal:** Provide timely updates through various channels.
- **Implementation:** Use free-tier services like SendGrid for email or Firebase Cloud Messaging for push notifications.

4.6 Feedback and Support System:

- **Goal:** Gather user feedback and provide support.
- **Implementation:** Add a feedback form and support ticket system to the frontend.

Contact Information:

Formatted: Font: 16 pt

- **Project Team Lead:** Rishi Singh
- **Email:** singhrishi1563@gmai.com
- **GitHub Repository:** https://github.com/NoviceNerd1
- **LinkedIn:** https://www.linkedin.com/in/rishisingh15/

Acknowledgments:

- Special thanks to the hackathon organizers for their support and guidance throughout the project.
- Appreciation to the open-source community for providing valuable tools and libraries used in this project.

Disclaimer:

This documentation is intended for use in the context of the **UST D3CODE Hackathon'24** and may include information about technology and services that are subject to change. The project is developed using free-tier services to maintain cost-effectiveness. The views expressed in this document are those of the project team and do not necessarily reflect the views of any affiliated organizations.

Formatted: Font: Bold