Carbon Sense

Introduction

Welcome to the world of Carbon Sense, where we delve into the fascinating topic of carbon emissions and their impact on our planet. In this document, we will explore the causes and consequences of excessive carbon emissions, as well as discuss potential solutions to mitigate their effects. Whether you are a concerned individual or a policy maker, this document aims to provide you with valuable insights and knowledge to foster a more sustainable future for all. So, let's embark on this journey together and deepen our understanding of carbon sense!

Vision and Mission

1 User Management

The system must allow users to register, log in, and manage their profiles.

2 Carbon Emissions Tracking

The system must allow users to input data related to their activities (e.g., electricity usage, travel, etc.) and calculate the corresponding carbon emissions.

3 Monthly Emissions History

The system must store and display a history of the user's carbon emissions on a monthly basis.

4 Data Visualization & Reporting

The system should provide visual representations of the user's emissions data over time and allow users to generate reports of their carbon emissions.

Non-Functional Requirements

Performance

The system should calculate and display carbon emissions data with minimal latency

Scalability

The architecture should support increased load as more users begin tracking their emissions.

Security

Data must be encrypted in transit and at rest. Use JWT for authentication and authorization.

Availability & Maintainability

The system should have an uptime of 99.9%, with failover mechanisms in place. Code should follow SOLID principles, with high modularity and low coupling.

Technologies Used

Backend

Programming Language: Java 17 for backend services.

Frameworks: Spring Boot for building microservices, Spring Security for authentication and authorization.

Database: MongoDB for storing persistent data.

API Communication: RESTful APIs using Spring Web.

Frontend

Angular for building the user interface.

Other

Containerization: Docker for containerizing services.

High-Level Design: Architectural Overview

1

API Gateway

Acts as a single entry point for all client requests, routing them to the appropriate microservice.

2

User Service

Handles user-related operations such as registration, login, and profile management.

3

Carbon Emissions Service

Manages the tracking of user activities and the calculation of carbon emissions.

4

History Service

Stores and retrieves historical carbon emissions data for each user.

5

Reporting Service

Generates reports and visualizations based on user emissions data.

High-Level Design:

1

User Logs In

The user logs in and views their dashboard.

2

User Enters Activity Data

The user enters activity data (e.g., electricity usage, travel data).

3

System Calculates Emissions

The system calculates the corresponding carbon emissions.

4

System Updates Emissions Record

The system updates the carbon emissions record.

5

User Requests Report

The user requests an emission report.

6

System Generates Report

The system generates and displays the report.

Low-Level Design: Detailed Component Specification

Component	Description
Controller	Exposes RESTful endpoints and handles HTTP requests.
Service	Contains the business logic and interacts with repositories.
Repository	Manages data persistence and retrieval from MongoDB.

Low-Level Design: Security Considerations

Authentication

JWT is used for secure user authentication.

Authorization

Role-based access control ensures users only access permitted resources.

Data Encryption

Sensitive data is encrypted both in transit and at rest.

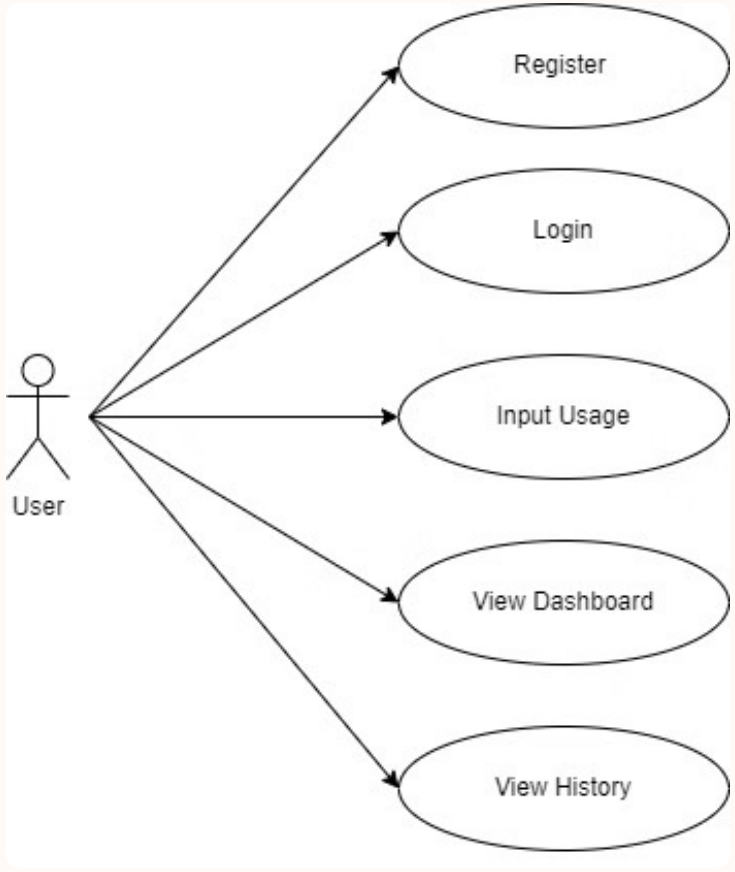
Input Validation

All user inputs are validated to prevent injection attacks.

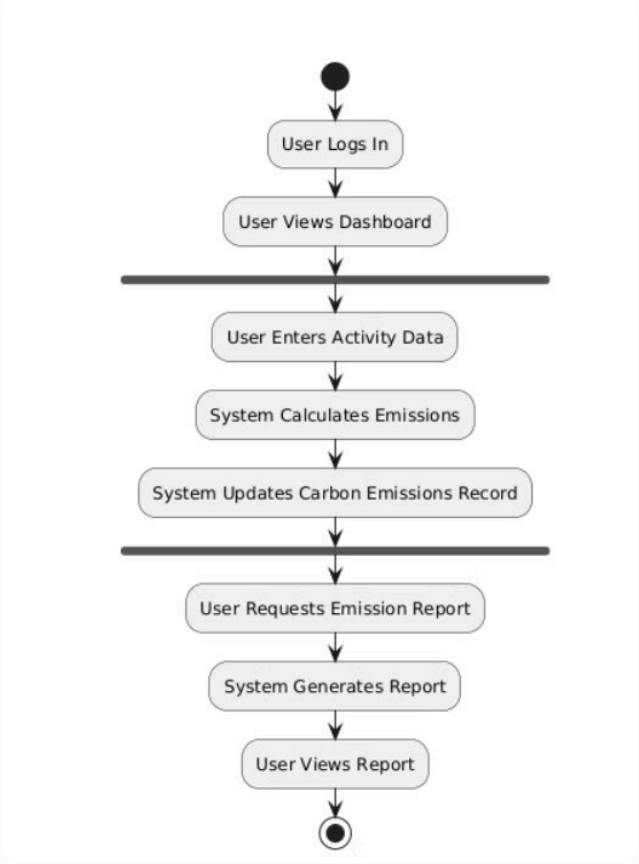
Exception Handling

1. **Global Exception Handling** Use a centralized error handler to manage exceptions across all microservice layers. Return custom error responses with detailed messages and appropriate HTTP status codes.
2. **Input Validation** Validate incoming requests to ensure they meet required formats and constraints. Validate DTOs using annotations like @NotNull, @Size, @Email.
3. **Client-Side Error Handling**
    - **Input Validation**: Before sending data to the server, the client-side (typically Angular) validates the input to prevent sending incorrect or malformed data.
    - **Error Interceptors**: HTTP interceptors are used to handle errors centrally in the Angular application, providing user-friendly error messages and handling specific error codes appropriately.
4. **Security Error Handling**
    - **Authentication and Authorization Failures**: Proper error handling mechanisms are implemented to handle cases where authentication or authorization fails. Unauthorized access attempts return a 401 Unauthorized or 403 Forbidden response.
    - **CSRF and XSS Protection**: Ensuring that proper security measures are in place to handle Cross-Site Request Forgery (CSRF) and Cross-Site Scripting (XSS) attacks by sanitizing input and implementing security headers.

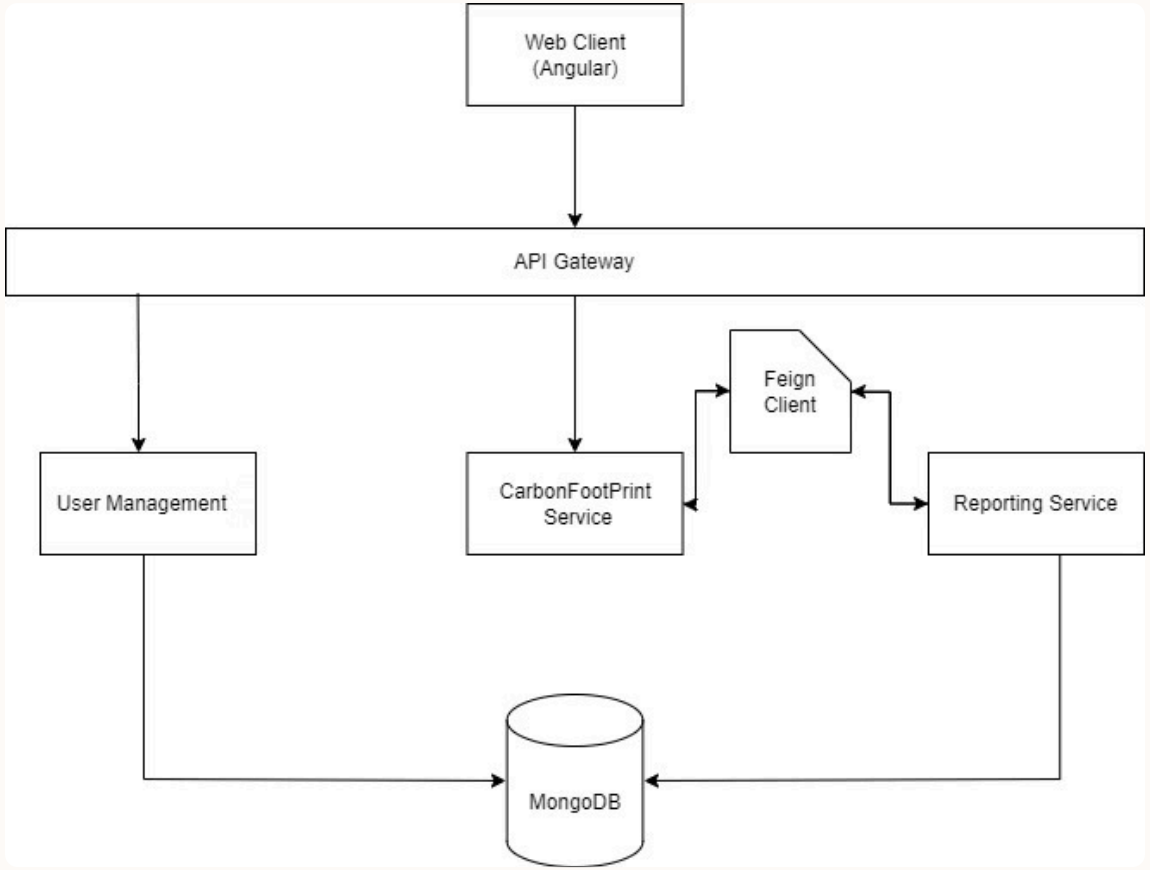
Usecase Diagram



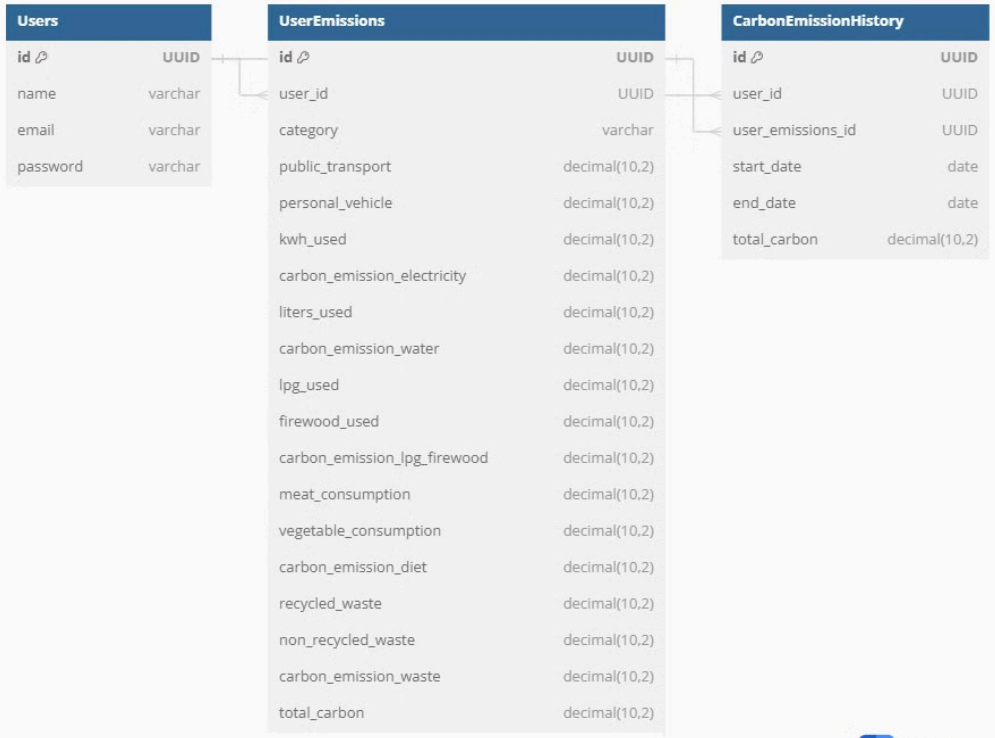
Activity Diagram



Architecture Diagram



ER Diagram



Wireframe Diagram