Carbon Footprint Tracking Application
Functional Requirments Documentation

Vinay Kumar Satrasala,Mohammed Shahad

September 3, 2024

Contents

1 Scope

This document outlines the architecture and design of the Carbon Footprint Tracking Application. The project aims to help individuals monitor and reduce their carbon footprint by tracking their activities in areas such as electricity usage, travel, water consumption, dietary habits, waste management, and LPG/firewood usage. The application provides monthly emissions data and maintains a history of the user's carbon emissions over time.

2 Functional Requirements

The functional requirements define what the system should do. They include:

- **User Management**: The system must allow users to register, log in, and manage their profiles.
- **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.
- **Monthly Emissions History**: The system must store and display a history of the user's carbon emissions on a monthly basis.
- **Data Visualization**: The system should provide visual representations of the user's emissions data over time.
- **Reporting**: Users should be able to generate reports of their carbon emissions.

3 Non-Functional Requirements

Non-functional requirements define how the system performs its functions:

- **Performance**: The system should calculate and display carbon emissions data within 2 seconds of data entry.
- **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**: The system should have an uptime of 99.9%, with failover mechanisms in place.
- **Maintainability**: Code should follow SOLID principles, with high modularity and low coupling.

4 Technologies Used

The project employs the following technologies:

- **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.
- **Containerization**: Docker for containerizing services.

5 High-Level Design

5.1 Architectural Overview

The architecture is based on a microservices pattern, where each service is responsible for a specific business capability. Each microservice runs independently, communicating with others via HTTP REST APIs. The key components include:

- **API Gateway**: Acts as a single entry point for all client requests, routing them to the appropriate microservice.
- **User Service**: Handles user-related operations such as registration, login, and profile management.
- **Carbon Emissions Service**: Manages the tracking of user activities and the calculation of carbon emissions.
- **History Service**: Stores and retrieves historical carbon emissions data for each user.
- **Reporting Service**: Generates reports and visualizations based on user emissions data.

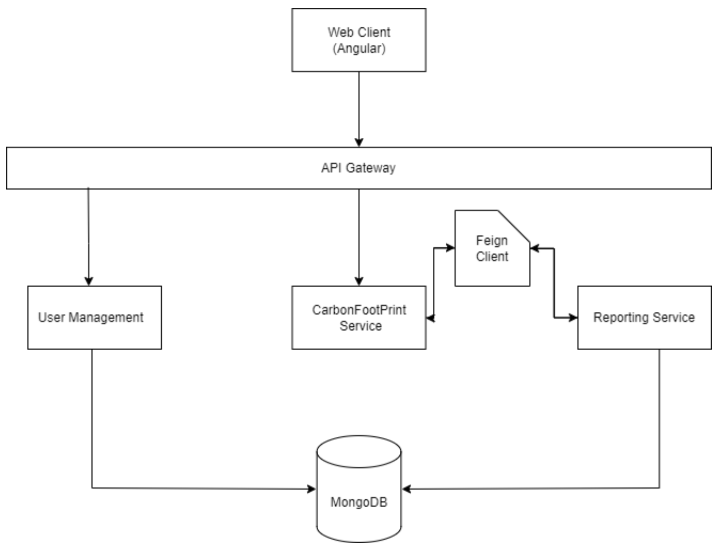

Figure 1: Activity Diagram: Carbon Emissions Workflow

5.2 Functional Specifications

This section details the functionality and behavior of each component. Below is an example sequence diagram that illustrates the process of logging user activities and calculating emissions:

- The user logs in and views their dashboard.

- The user enters activity data (e.g., electricity usage, travel data).
- The system calculates the corresponding carbon emissions.
- The system updates the carbon emissions record.
- The user requests an emission report.
- The system generates and displays the report.

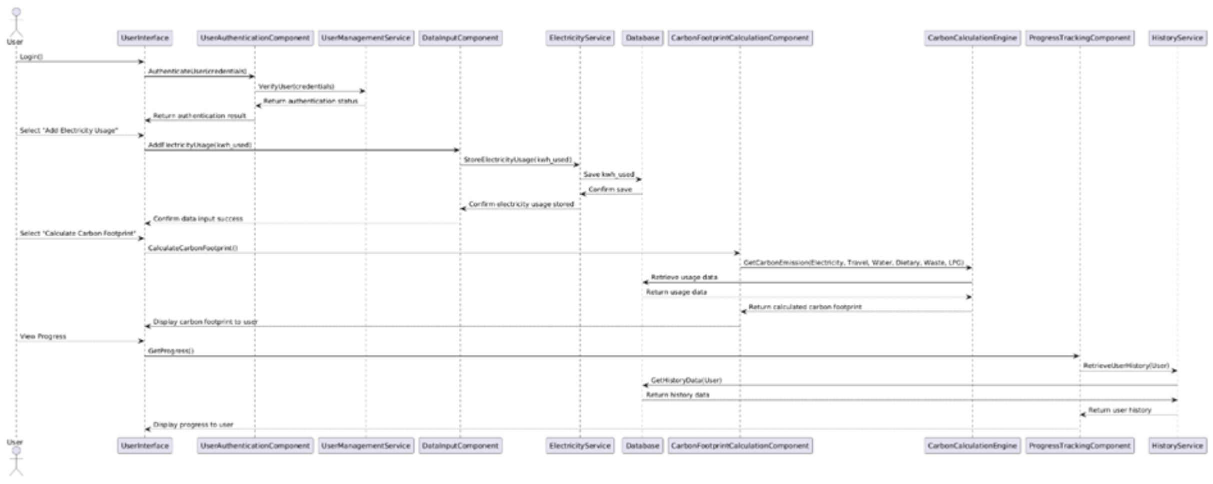
User
UserInterface
UserAuthenticationComponent
UserManagementService
DataInputComponent
ElectricityService
Database
CarbonFootprintCalculationComponent
CarbonCalculationEngine
ProgressTrackingComponent
HistoryService
Login()
AuthenticateUser(credentials)
VerifyUser(credentials)
Return authentication status
Return authentication result
Select "Add Electricity Usage"
AddElectricityUsage(kwh_used)
StoreElectricityUsage(kwh_used)
Save kwh_used
Confirm save
Confirm electricity usage stored
Confirm data input success
Select "Calculate Carbon Footprint"
CalculateCarbonFootprint()
GetCarbonEmission(Electricity, Travel, Water, Dietary, Waste, LPG)
Retrieve usage data
Return usage data
Return calculated carbon footprint
Display carbon footprint to user
View Progress
GetProgress()
RetrieveUserHistory(User)
GetHistoryData(User)
Return history data
Return user history
Display progress to user

Figure 2: Sequence Diagram: Logging Activities and Calculating Emissions

5.3 System Interactions

This section describes how different components interact with each other. The interactions are illustrated using a wireframe diagram that shows the flow between different pages and components of the system. For example:

- The homepage allows users to log in or view their emissions data.
- After logging in, users can access their profile, input activity data, and view historical emissions.
- The reporting feature allows users to generate reports based on their emissions history.

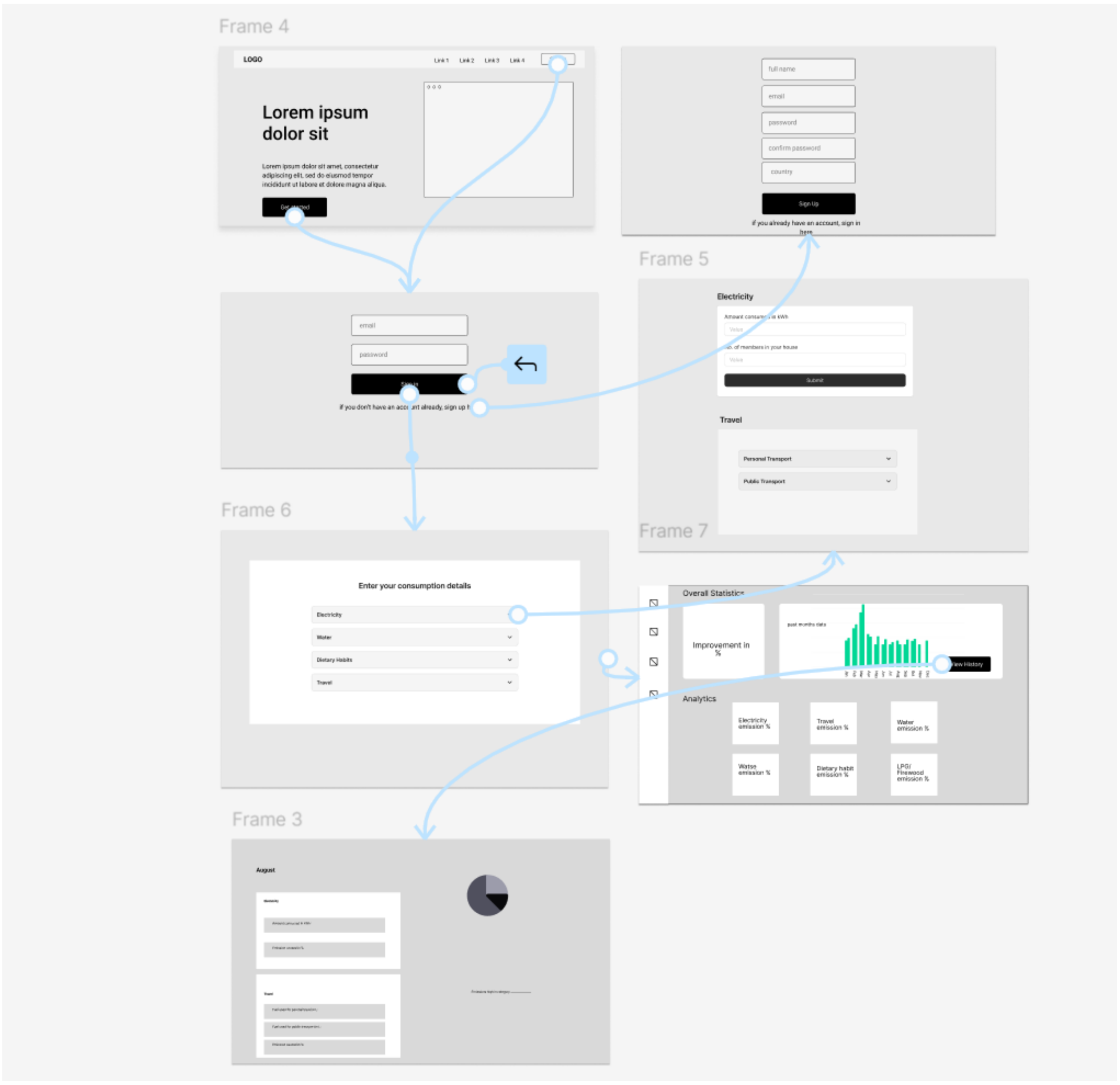

Figure 3: Wireframe Diagram: System Interactions

5.4 Performance Considerations

The system is designed to be highly performant. Key considerations include:

- **Caching**: Frequently accessed data, such as monthly emissions summaries, is cached to reduce load on the database.
- **Load Balancing**: Incoming requests are distributed across multiple instances of each microservice to ensure even load distribution.
- **Asynchronous Processing**: Non-critical tasks, such as report generation, are processed asynchronously using message queues to improve response times.
- **Database Optimization**: Indexing is used on frequently queried fields to speed up data retrieval.

5.5 Technology Stack

The technology stack includes:

- **Backend**: Java, Spring Boot, Spring Security.
- **Frontend**: Angular, Bootstrap, HTML, CSS.
- **Database**: MongoDB.
- **Containerization**: Docker.

6 Low-Level Design

6.1 Detailed Component Specification

Each microservice is composed of several components, including controllers, services, and repositories:

- **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.

For example, the Carbon Emissions Service includes:

- **CarbonEmissionsController**: Handles emissions-related API requests.
- **CarbonEmissionsService**: Contains the business logic for managing emissions data.
- **CarbonEmissionsRepository**: Interfaces with the database to manage emissions data.

6.2 Interface Definitions

Interfaces define the contracts between different services and components. Each service exposes a set of RESTful APIs that can be consumed by other services or the frontend. Example:

- **Carbon Emissions Service API**:
  - ‘POST /carbon-emissions‘: Logs a new activity and calculates the emissions.
  - ‘GET /carbon-emissions/month/month‘: Retrieves emissions data for a specific month.

6.3 Resource Allocation

Resource allocation involves distributing CPU, memory, and storage among the microservices. Each service is allocated resources based on its expected load:

- **Carbon Emissions Service**: Requires more memory for storing and processing emissions data.
- **History Service**: Needs higher storage allocation for maintaining historical records.
- **Reporting Service**: Allocated both CPU and memory for efficient report generation.

6.4 Error Handling and Recovery

Error handling is crucial to maintain system stability. The following strategies are implemented:

- **Try-Catch Blocks**: Used to handle exceptions in code.
- **Fallback Mechanisms**: In case of service failure, fallback methods are invoked to ensure continuity.
- **Logging**: Errors are logged with details for debugging and analysis.
- **Circuit Breaker Pattern**: Prevents a chain of failures when one service is down.

6.5 Security Considerations

Security is a top priority in the system design. Key considerations include:

- **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.

6.6 Code Structure

The codebase is structured according to the principles of modularity and separation of concerns. Each microservice follows a similar structure:

- **Controller Layer**: Manages HTTP requests.
- **Service Layer**: Contains business logic.
- **Repository Layer**: Manages database operations.
- **DTOs**: Data Transfer Objects are used to encapsulate data for transfer between layers.
- **Utils**: Utility classes and methods that are commonly used across the service.

7 Diagrams

7.1 ER Diagrams

The Entity-Relationship (ER) diagram provides a visual representation of the database schema. It shows the relationships between entities such as ‘Users‘, ‘CarbonEmissions‘, and ‘Emission-History‘.

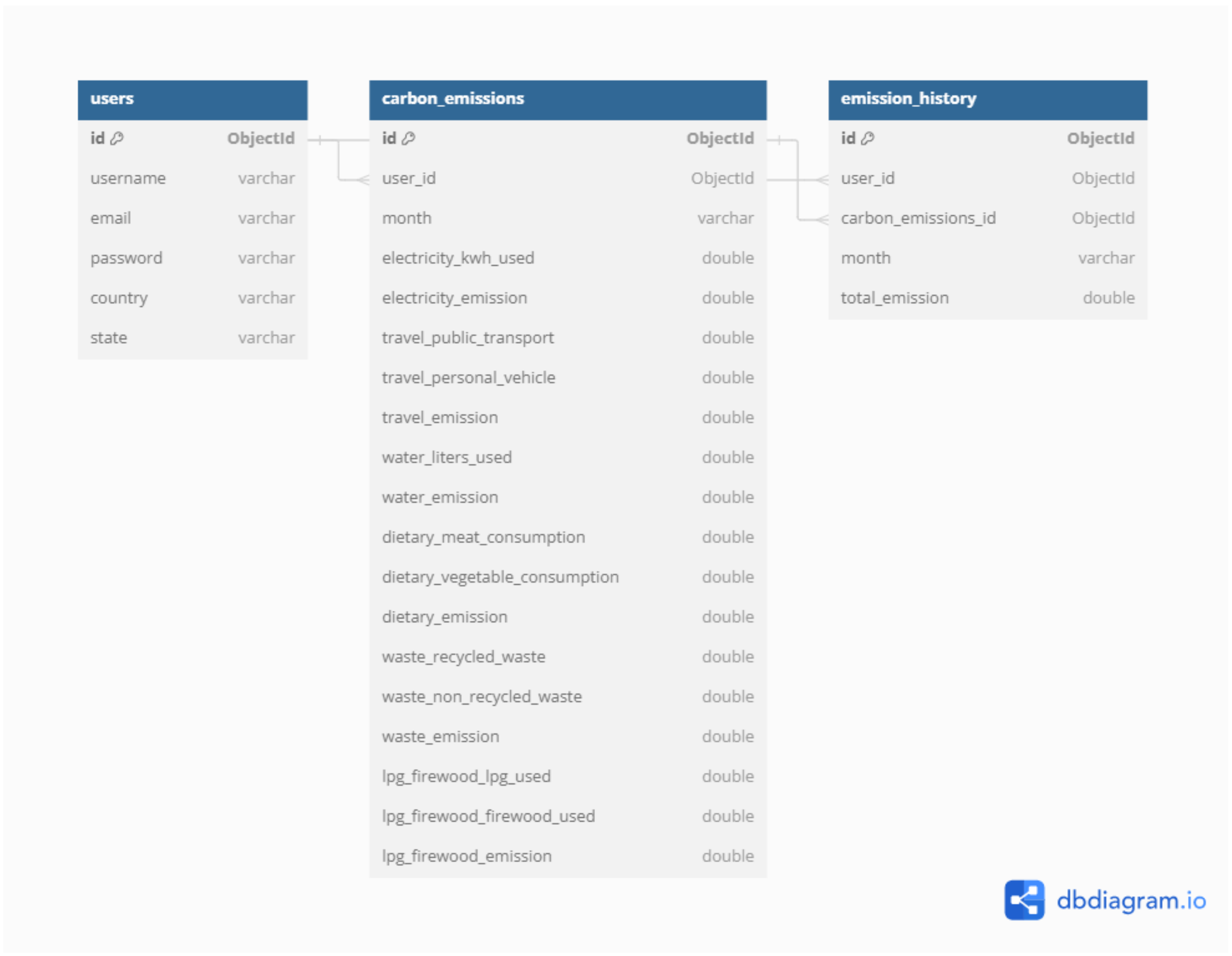
users
id ObjectId
username varchar
email varchar
password varchar
country varchar
state varchar
carbon_emissions
id ObjectId
user_id ObjectId
month varchar
electricity_kwh_used double
electricity_emission double
travel_public_transport double
travel_personal_vehicle double
travel_emission double
water_liters_used double
water_emission double
dietary_meat_consumption double
dietary_vegetable_consumption double
dietary_emission double
waste_recycled_waste double
waste_non_recycled_waste double
waste_emission double
lpg_firewood_lpg_used double
lpg_firewood_firewood_used double
lpg_firewood_emission double
emission_history
id ObjectId
user_id ObjectId
carbon_emissions_id ObjectId
month varchar
total_emission double
dbdiagram.io

Figure 4: ER Diagram

7.2 Class Diagrams

The class diagram illustrates the structure of the microservices, showing classes, attributes, methods, and the relationships between them.

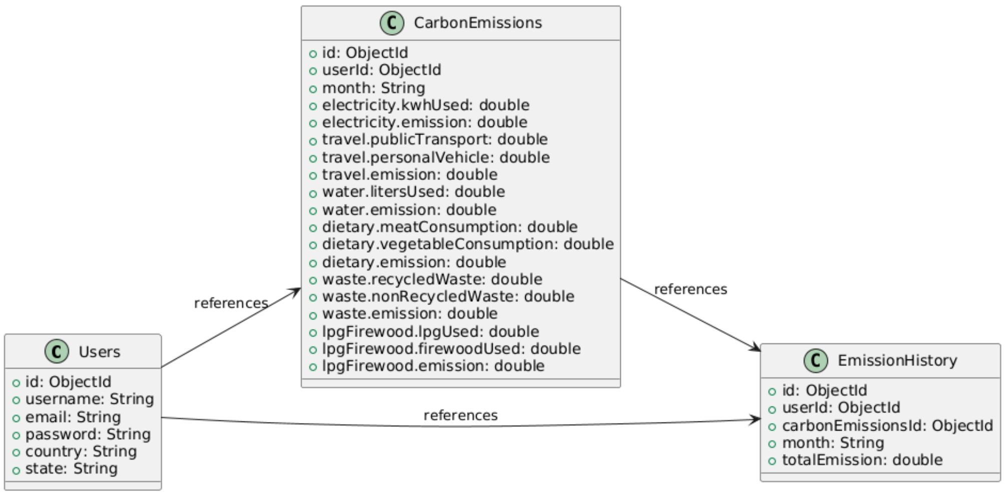

Figure 5: Class Diagram

7.3 Sequence Diagrams

Sequence diagrams depict the interaction between objects or components in a specific order, illustrating how processes or workflows are carried out.

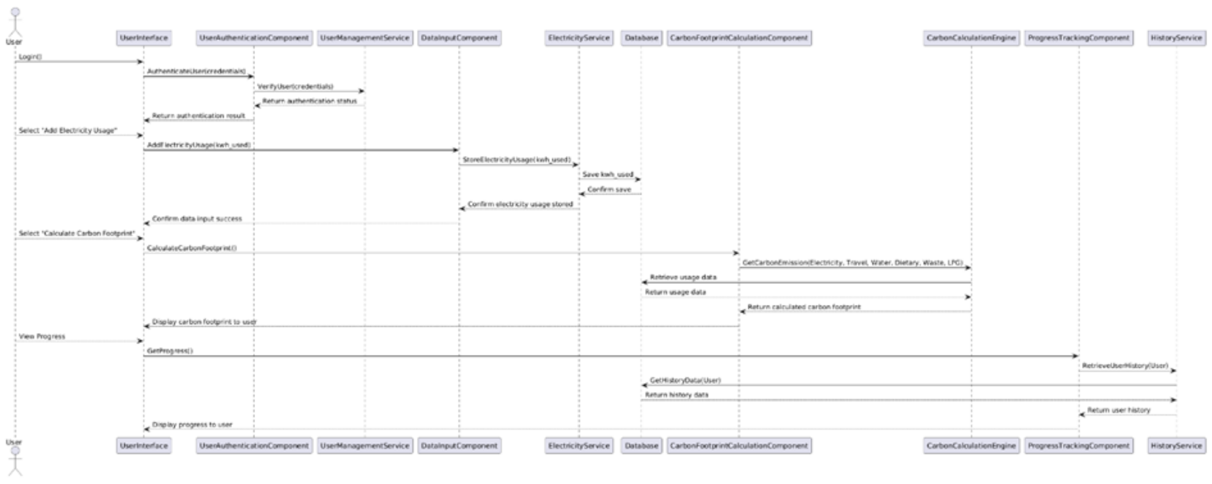
User
UserInterface
UserAuthenticationComponent
UserManagementService
DataInputComponent
ElectricityService
Database
CarbonFootprintCalculationComponent
CarbonCalculationEngine
ProgressTrackingComponent
HistoryService
Login()
AuthenticateUser(credentials)
VerifyUser(credentials)
Return authentication status
Return authentication result
Select "Add Electricity Usage"
AddElectricityUsage(kwh_used)
StoreElectricityUsage(kwh_used)
Save kwh_used
Confirm save
Confirm electricity usage stored
Confirm data input success
Select "Calculate Carbon Footprint"
CalculateCarbonFootprint()
GetCarbonEmission(Electricity, Travel, Water, Dietary, Waste, LPG)
Retrieve usage data
Return usage data
Return calculated carbon footprint
Display carbon footprint to user
View Progress
GetProgress()
RetrieveUserHistory(User)
GetHistoryData(User)
Return history data
Return user history
Display progress to user

Figure 6: Sequence Diagram: Logging Activities and Calculating Emissions

7.4 Activity Diagrams

Activity diagrams represent the flow of control or data through a sequence of actions or steps, showing the dynamic aspects of the system.

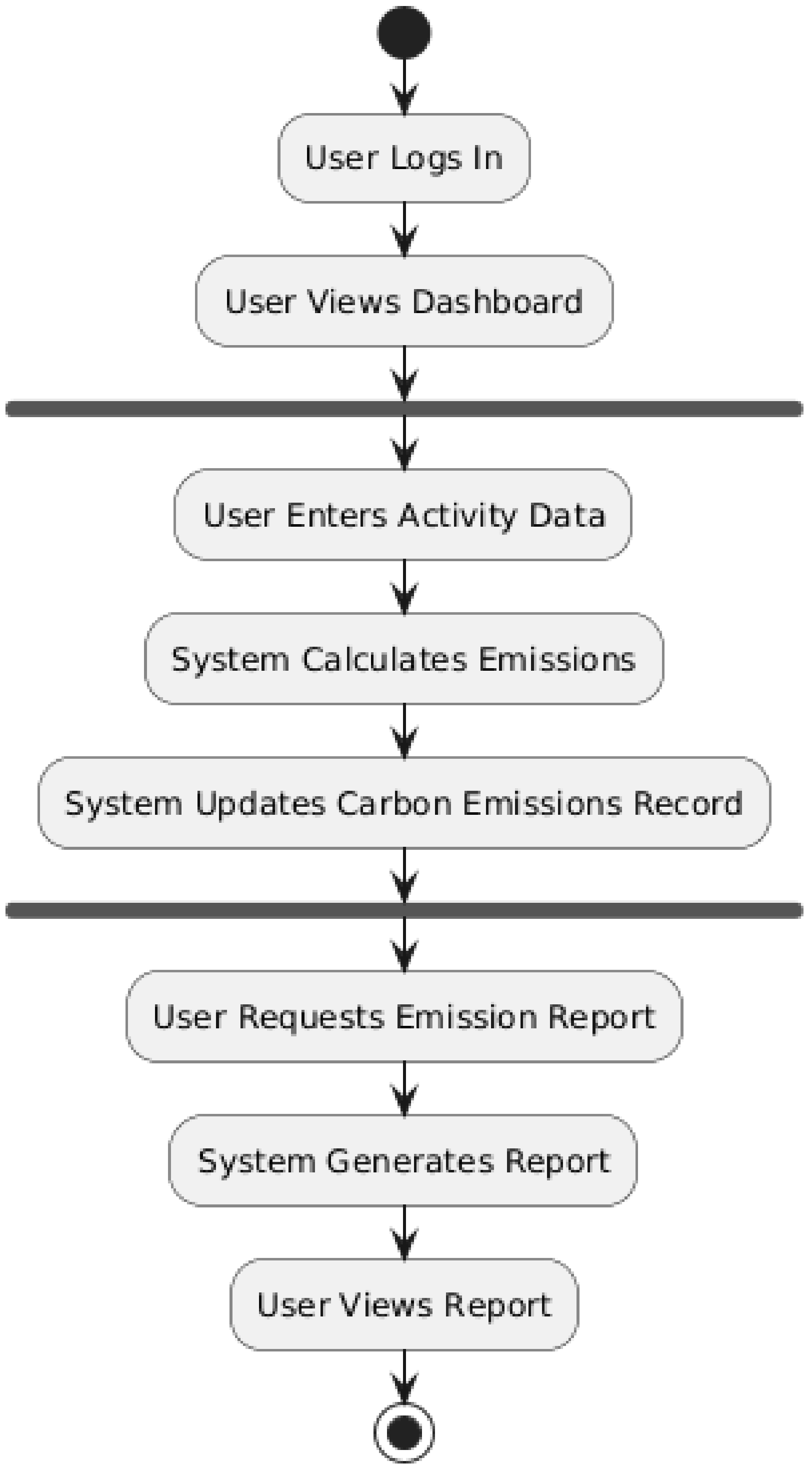
User Logs In
User Views Dashboard
User Enters Activity Data
System Calculates Emissions
System Updates Carbon Emissions Record
User Requests Emission Report
System Generates Report
User Views Report

Figure 7: Activity Diagram: Carbon Emissions Workflow

7.5 Use Case Diagrams

Use Case diagrams describe the interactions between users (actors) and the system, highlighting the functionalities provided by the system.

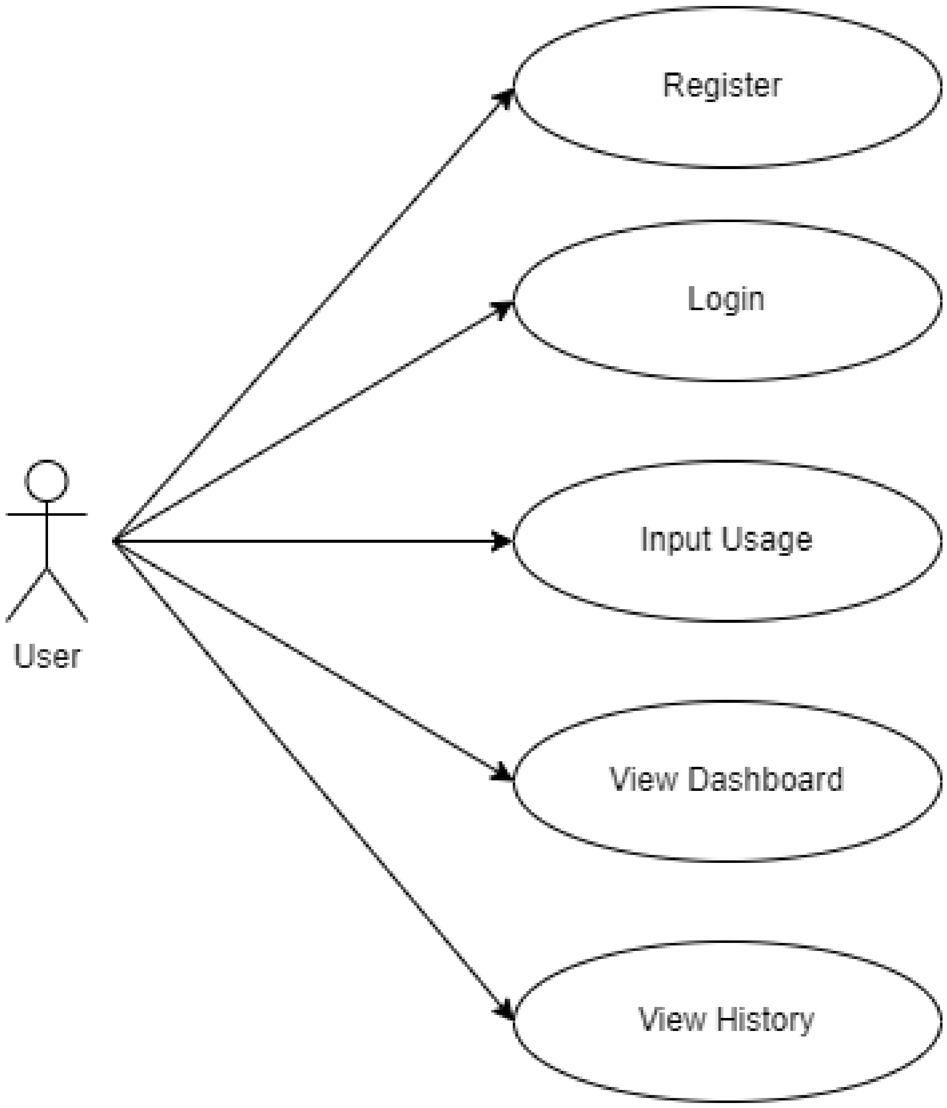

Figure 8: Use Case Diagram: System Interactions

8 Conclusion

In conclusion, the microservices architecture outlined in this document offers a scalable, maintainable, and secure solution for tracking carbon emissions. By breaking down the system into distinct services, each responsible for a specific function, the architecture promotes modularity and separation of concerns, allowing for easier development, testing, and deployment.

The use of Spring Boot, coupled with MongoDB, ensures a robust backend, while Angular provides a responsive and dynamic user interface. The incorporation of JWT for authentication and Docker for containerization further enhances the security and scalability of the system. Additionally, the architecture supports horizontal scaling, ensuring that the system can handle increased loads without compromising performance.

By adhering to the principles of microservices, this system is well-equipped to adapt to future changes and expansions, making it a sustainable solution for the long term. The diagrams and design decisions documented herein provide a clear roadmap for implementation, ensuring that the development process is both efficient and aligned with best practices. Overall, this architecture is poised to meet the project's goals and requirements, providing a strong foundation for the system's ongoing success.