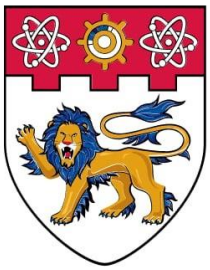
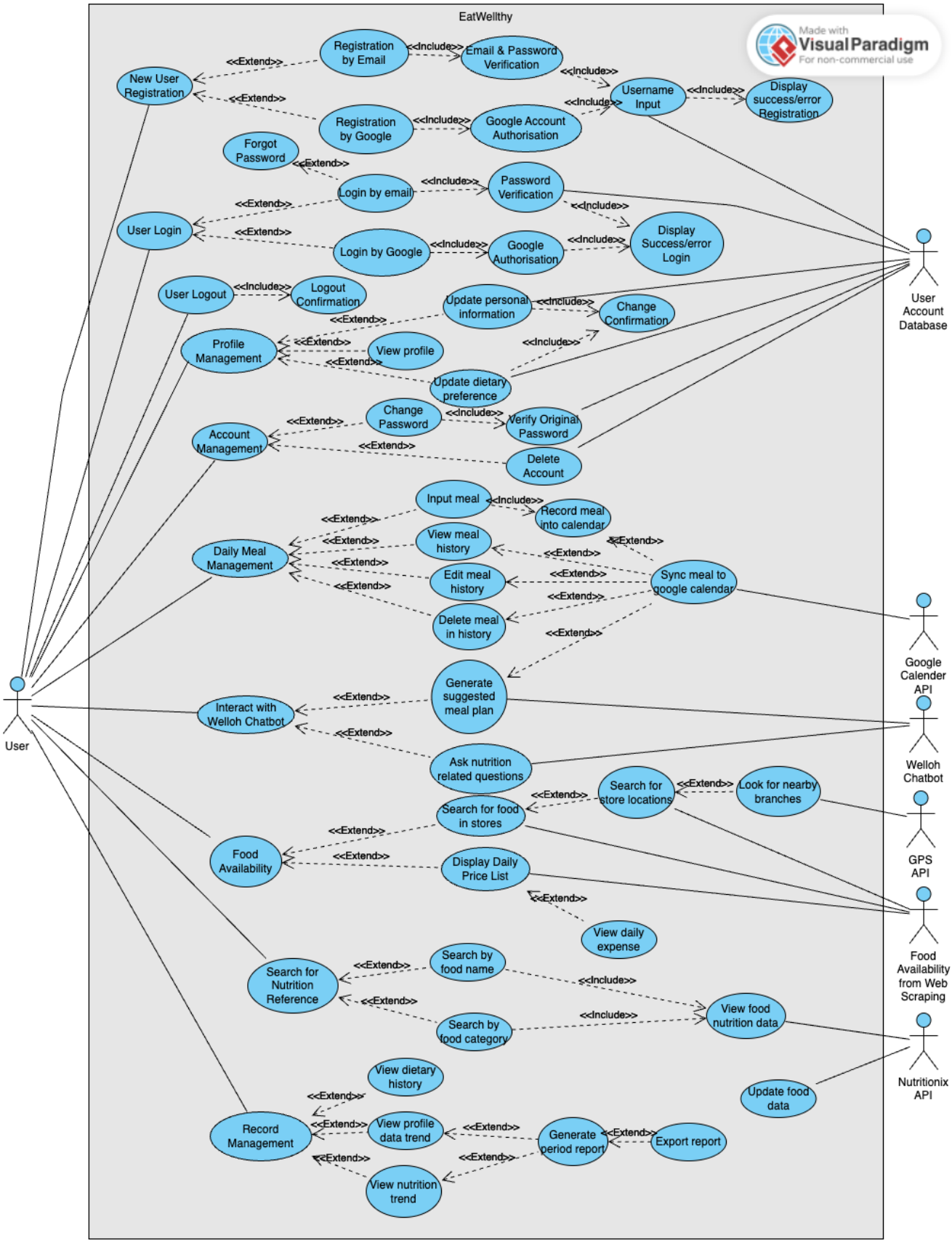


NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

SC2006 Lab3 Deliverables

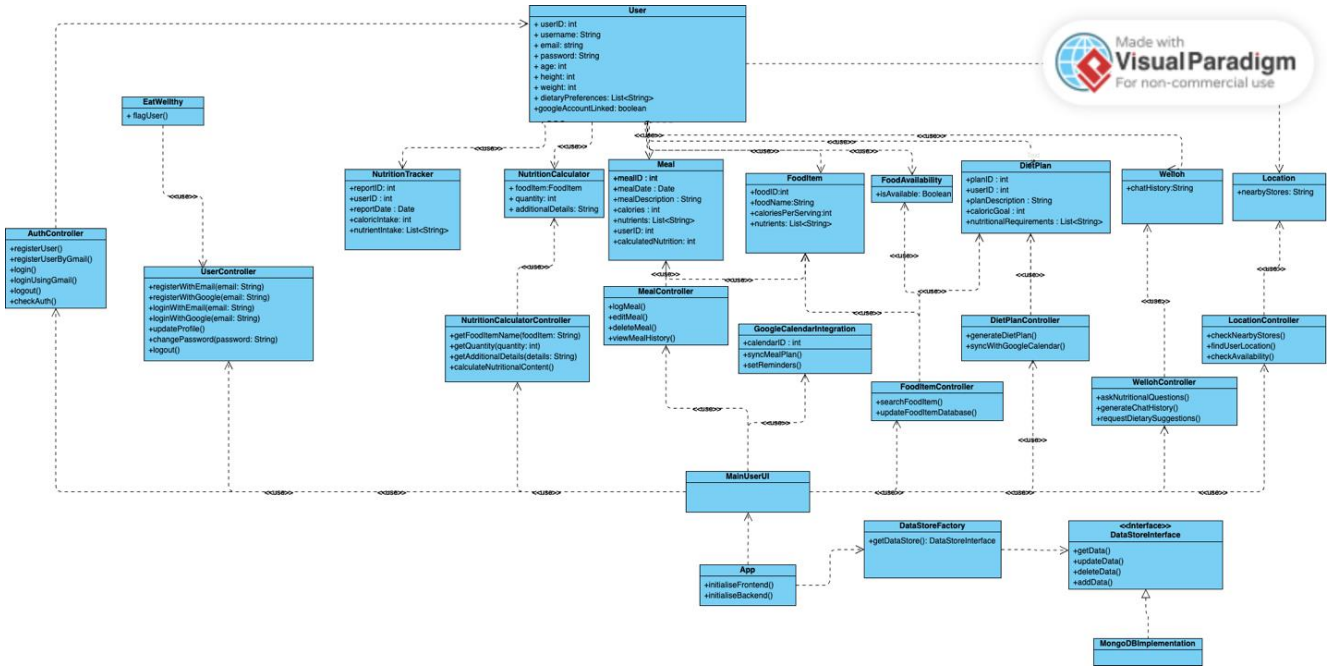
Lab index	SDAA	
Group number	31	
Group members	Liu Xiaotao	U2320836F
	Low Jo Yi, Nicole	U2321370D
	Mahi Pandey	U2321382F
	Mehta Rishika	U2323133H
	Zhang Yichi	U2320736J
	Zhao Qixian	U2321752L

1. Complete Use Case Model



2. Design Models

a. Class Diagram



Main Functional Components

1. App
    - Acts as the main entry point for initializing both the frontend and backend components of the application.
    - Operations include:
        - initialiseFrontend(): Sets up the user interfaces.
        - initialiseBackend(): Starts the backend processes and connects to the data store.
2. MainBoard
    - Represents the main user interface that integrates other UIs like meal management, diet plans, and nutrition tracking.
3. User
    - Manages core user data, including login details, user preferences, and nutritional goals.
    - Attributes include userID, username, email, dietaryPreferences, etc.

Controllers (Facade Pattern)

The controllers handle communication between the user interface and the data management components:

1. AuthController
    - Manages user authentication, login, signup, and session handling.
    - Key methods: registerUserByEmail(), login(), logout(), updatePassword(), etc.
2. UserController
    - Handles user registration, profile updates, and account management.
    - Coordinates with the User class and interacts with the data store for saving user information.
3. NutritionCalculationsController
    - Provides functionality to calculate nutritional intake, suggest daily diet goals, and analyze food data.
    - Interacts with classes like NutritionTracker, Meal, and DietPlan to offer recommendations.
4. MealController
    - Manages user meal data, including adding, updating, and deleting meals.
    - Interacts with the Meal class to ensure meals are logged correctly and align with user preferences.
5. DietPlanController
    - Allows users to create, edit, and manage diet plans.
    - Integrates with the DietPlan class to suggest meal plans based on nutritional needs.
6. FoodItemController
    - Manages the addition, deletion, and update of food items in the app's database.
    - Interacts with the FoodItem class to maintain accurate nutritional information for each item.
7. LocationController

- Manages location-based services, like finding nutrition-related locations or events.
- Integrates with classes like Location and uses mapping services to provide location data.

8. GoogleCalendarIntegration
    - Synchronizes meal plans with the user's Google Calendar, allowing for better scheduling.
    - Methods include: addEventToCalendar() and removeEventFromCalendar().

Data Handling Components

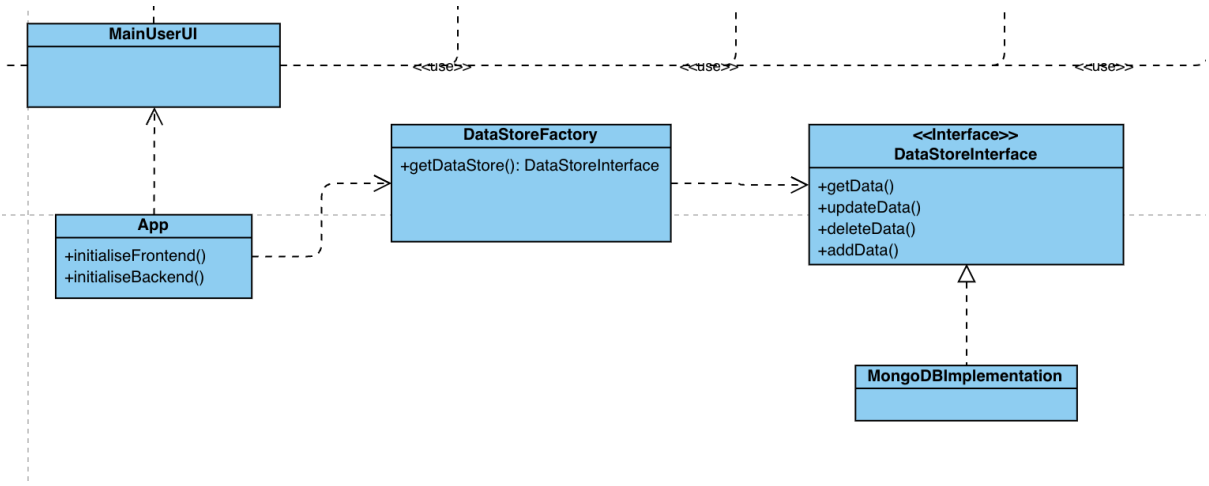
1. DataStoreInterface (Interface)
    - Defines methods for data operations:
        - getData()
        - updateData()
        - deleteData()
        - addData()
2. DataStoreFactory
    - Creates instances of data store implementations based on the configuration.
3. MongoDBImplementation
    - Implements the DataStoreInterface specifically for MongoDB operations, allowing CRUD actions tailored to MongoDB's architecture.

Purpose

The class diagram follows a modular design, promoting flexibility, scalability, and easier integration of new features. The Facade Pattern simplifies interactions between the user interfaces and backend services, while the DataStoreInterface and factory classes provide a clear structure for data management, enabling the use of different database systems like MongoDB.

Data Access Layer — DataStore (Factory Pattern + Strategy Pattern)

The **DataStore** layer ensures that data access, modification, and storage are consistently handled across the application. It facilitates efficient management of large-scale data operations, offering flexibility to interact with different data storage systems without altering the core functionality.



Design Patterns Implemented

1. Factory Pattern

- The **Factory Pattern** is used to instantiate specific implementations of the DataStoreInterface without embedding the instantiation logic in the core application code.
- The **DataStoreFactory** class creates instances of different data store classes, making it easier to switch between different storage systems (e.g., MongoDB, MySQL) based on configuration.
- In this design, the getDataStore() method in DataStoreFactory is responsible for creating and returning the appropriate data store instance, promoting loose coupling between the application and its data storage components.

2. Strategy Pattern

- The **Strategy Pattern** provides a flexible way to switch between different data storage implementations (e.g., MongoDB, MySQL) without modifying the core logic of the application.
- This pattern is applied through the following components:
    - **DataStoreInterface**: An interface defining the common set of data operations, such as:
        - getData(): Retrieves data from the database.

        - updateData(): Updates existing records in the database.
        - deleteData(): Removes data from the database.
        - addData(): Adds new data to the database.
    - **MongoDBImplementation**: A concrete implementation of the DataStoreInterface tailored for MongoDB. It encapsulates MongoDB-specific data operations and integrates with MongoDB databases.

Key Components

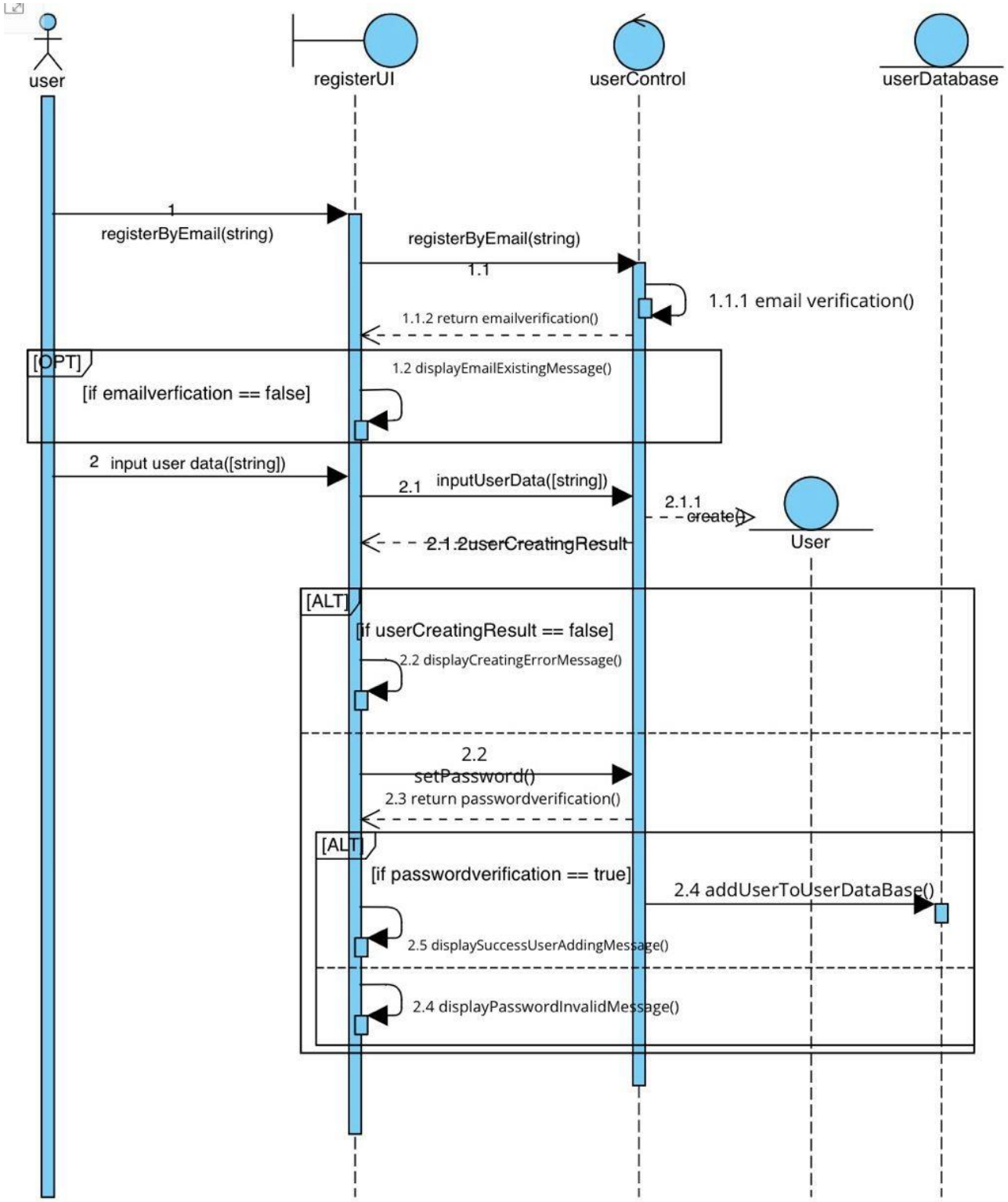
1. **DataStoreFactory**
    - This factory class determines the appropriate data store type based on application configuration and creates instances accordingly.
    - It implements the getDataStore() method, which returns an instance of the DataStoreInterface (e.g., MongoDBImplementation).
    - This class helps in maintaining a clean separation of concerns by centralizing the creation logic for data store instances.
2. **DataStoreInterface**
    - This interface defines the standard set of operations for data manipulation, providing a unified approach to data handling irrespective of the underlying database.
    - By using the **Strategy Pattern**, it allows for seamless interchange between different storage mechanisms, such as MongoDB and MySQL, without altering client-side code.
3. **MongoDBImplementation**
    - Implements the DataStoreInterface specifically for MongoDB, supporting CRUD operations.
    - This implementation handles MongoDB-specific logic, ensuring that data access is managed efficiently while adhering to MongoDB's structure and commands.

Purpose and Benefits

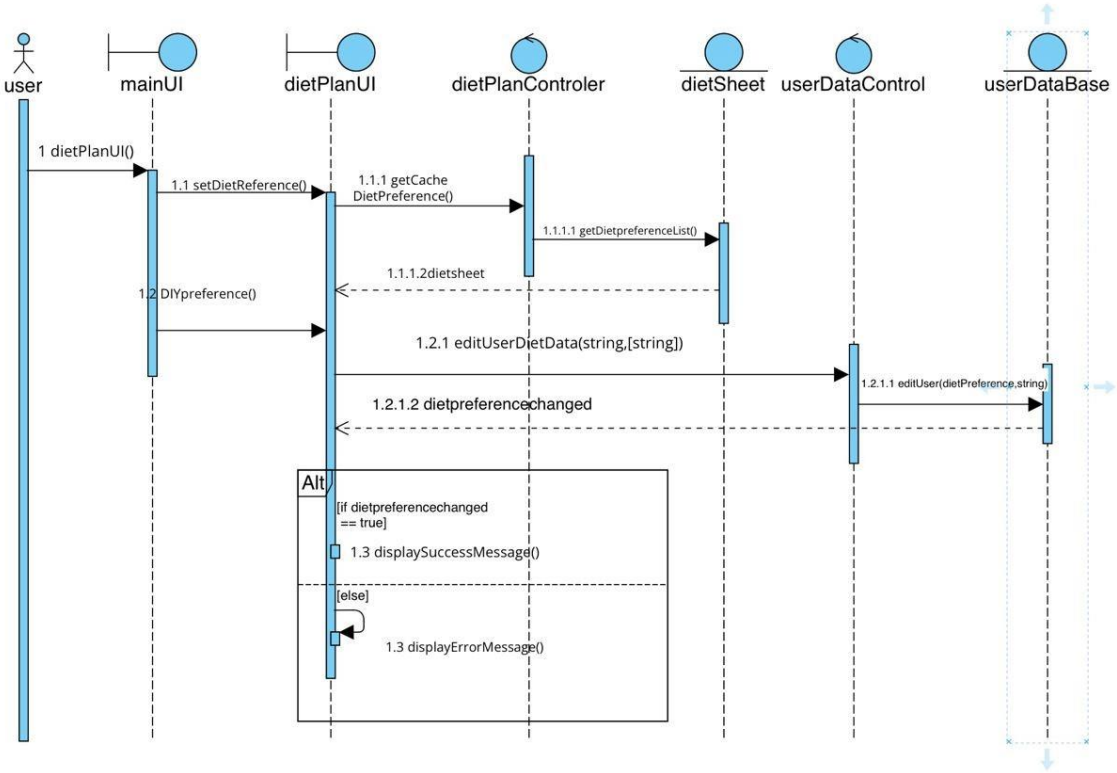
The **DataStore Layer** ensures flexible, scalable, and maintainable data management through abstraction and encapsulation. By integrating the **Factory** and **Strategy Patterns**, it enhances adaptability, enabling seamless integration or switching of data storage solutions. This centralized approach minimises errors, as modifications to data access logic occur in one place without impacting other components.

b. Sequence Diagrams

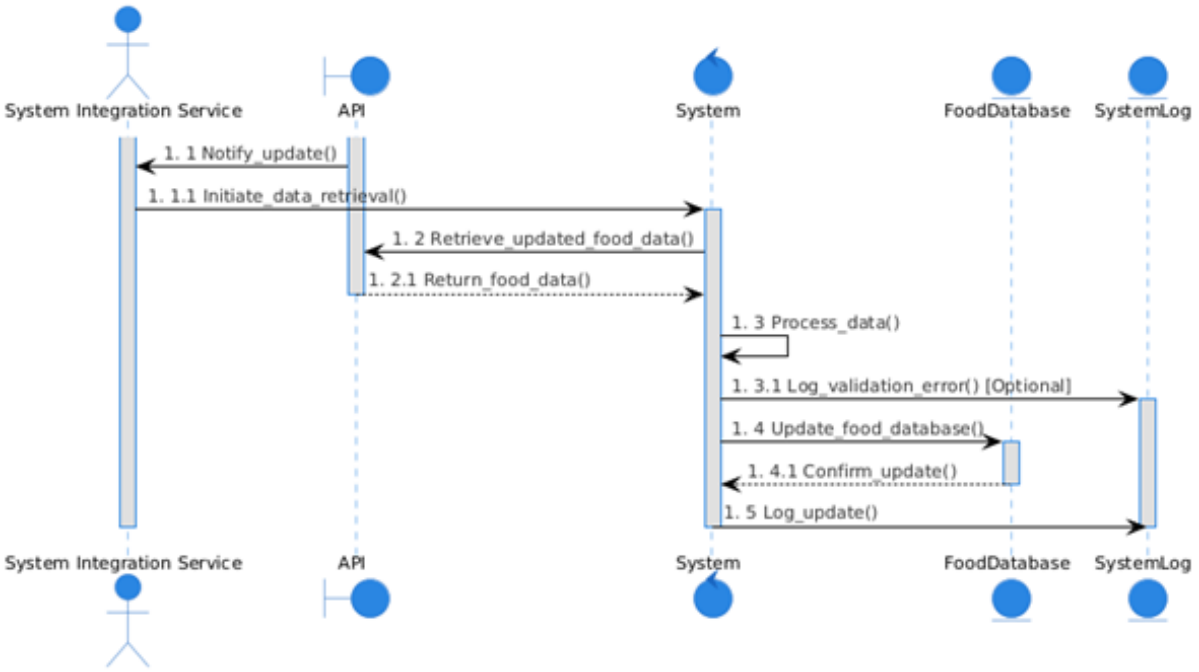
i. U0101 – New User Registration using Email



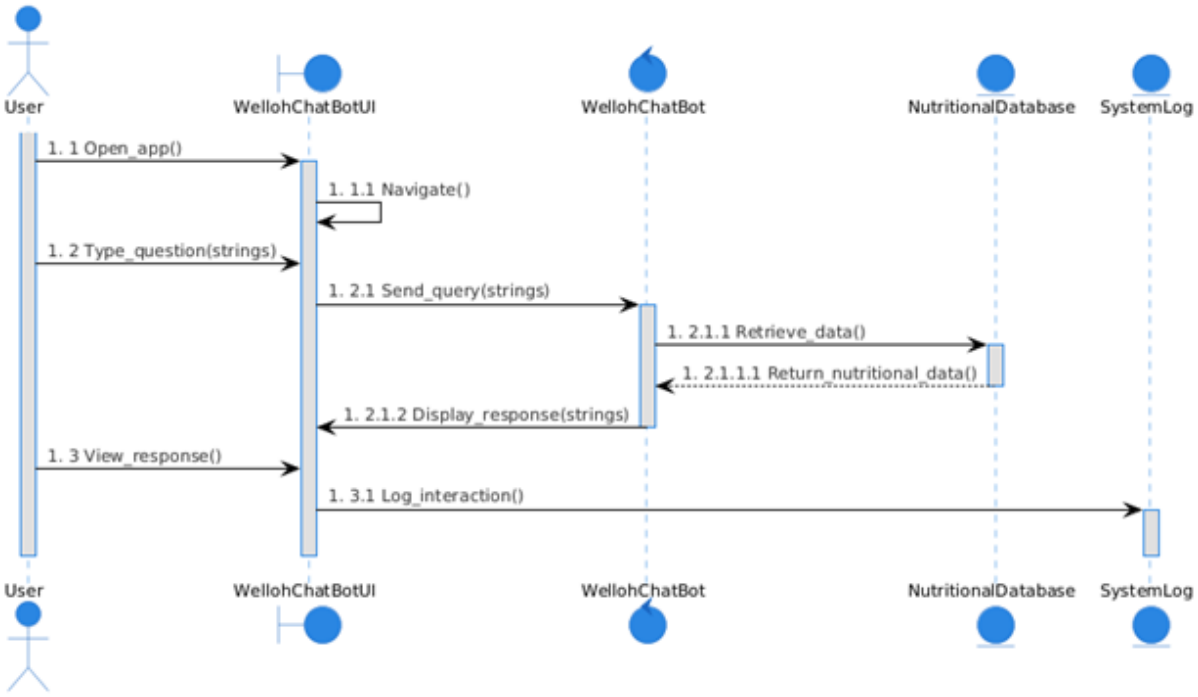
ii. U0203 – Manage Dietary Preference



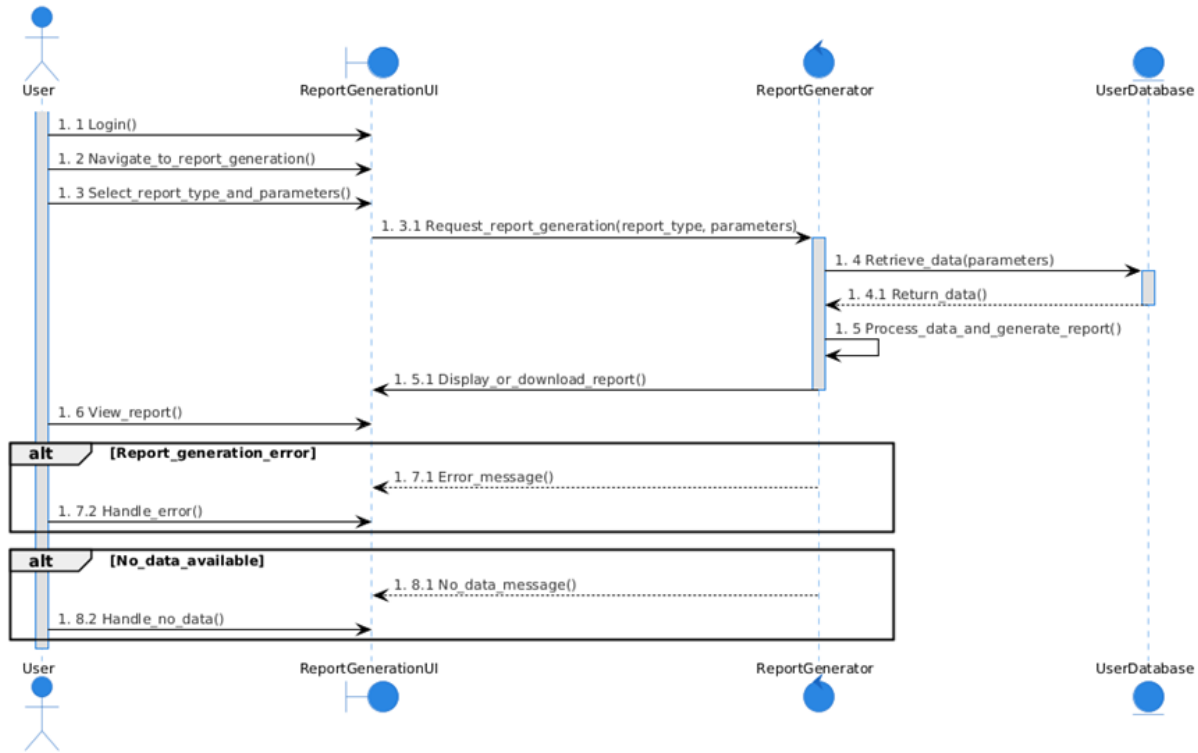
iii. U0301 – Log Daily Meal



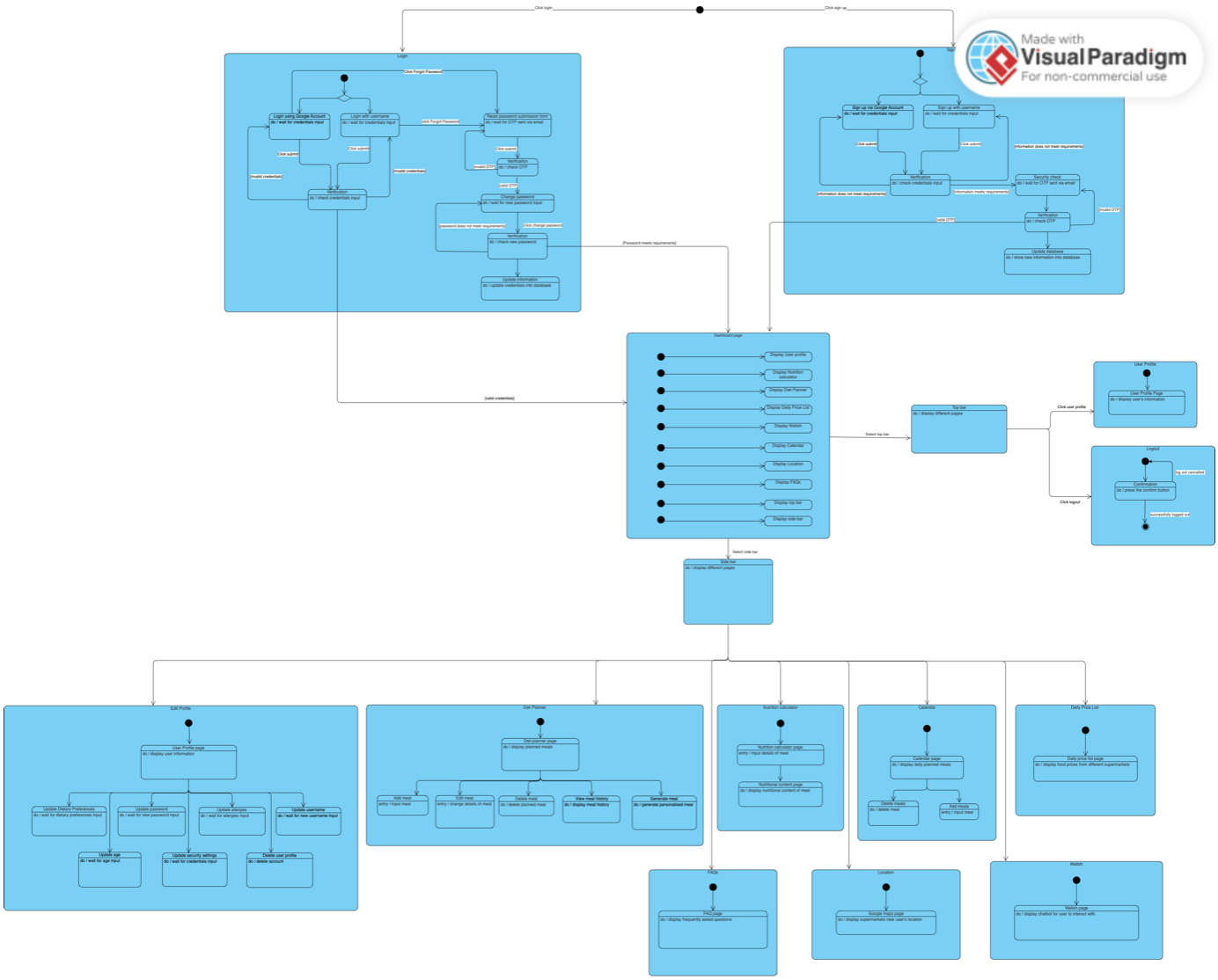
iv. U0401 – Ask Nutritional Questions



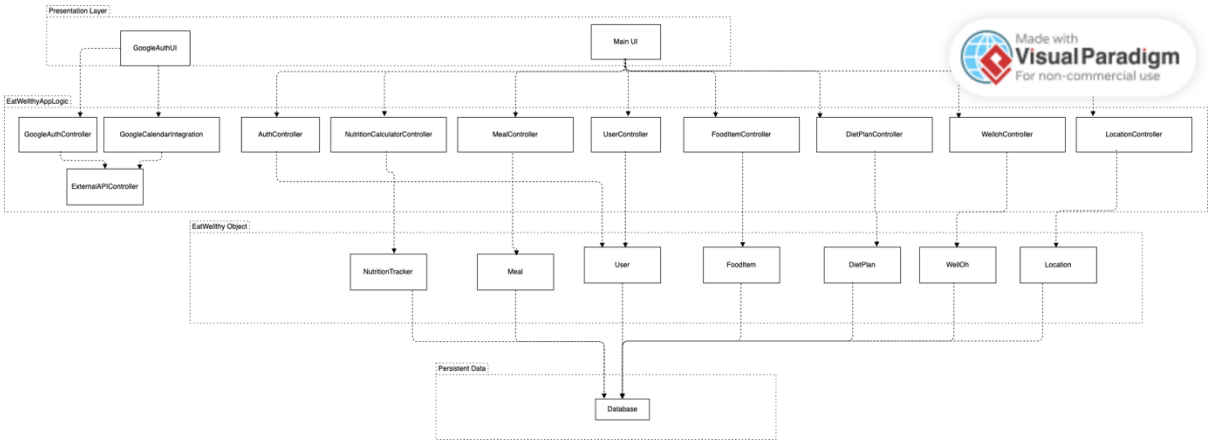
v. U0603 – Generate User Report



c. Dialog Map



3. System Architecture



1. Presentation Layer

- This layer contains the user interfaces that interact directly with the users of the application:
    - **GoogleAuthUI**:
        - Handles user login via Google Authentication.
        - Communicates with the **GoogleAuthController** to manage authentication processes.
    - **MainUI**:
        - Acts as the primary user interface for the app.
        - Coordinates with multiple controllers such as **UserController**, **NutritionCalculatorController**, **MealController**, **FoodItemController**, **DietPlanController**, and others for general user interactions.

2. EatWellthy App Logic

- This layer contains controllers that implement the core business logic of the application, processing requests from the presentation layer and interacting with the object layer:
    - **GoogleAuthController**:
        - Manages Google authentication processes, linking with the **GoogleAuthUI**.
    - **GoogleCalendarIntegration**:
        - Handles synchronization of user meal plans with Google Calendar.
        - Works closely with **ExternalAPIController** for API interactions.
    - **AuthController**:
        - Manages user authentication (login, logout, and session handling).
        - Interacts with the **UserController** for user account management.
    - **ExternalAPIController**:
        - Handles all external API interactions (e.g., Google Calendar and mapping APIs).
    - **UserController**:

        - Manages user-related functionalities such as profile updates and user data retrieval.
    - **NutritionCalculatorController**:
        - Calculates nutritional metrics for meals and diets.
        - Works with **NutritionTracker** to update user nutrition stats.
    - **MealController**:
        - Manages meal creation, modification, and deletion processes.
        - Directly interacts with the **Meal** object.
    - **FoodItemController**:
        - Handles CRUD operations for food items, interacting with the **FoodItem** object.
    - **DietPlanController**:
        - Manages user diet plans, creating and editing plans, and linking them to meals.
    - **LocationController**:
        - Manages location-based services, such as finding relevant nutrition events.

3. EatWellthy Object Layer

- This layer contains the entities that represent the core objects of the application. These entities store, manage, and update data as per the app logic:
    - **User**:
        - Represents user information, including account details, preferences, and profile data.
    - **NutritionTracker**:
        - Tracks user nutrition metrics and stores daily nutrition logs.
    - **Meal**:
        - Represents meal data, including meal names, descriptions, and nutritional information.
    - **FoodItem**:

        - Represents individual food items, including nutritional details and attributes.
    - **DietPlan**:
        - Represents user diet plans, including linked meals and dietary goals.
    - **WellOh**:
        - Handles wellness-related data, potentially integrating health metrics.
    - **Location**:
        - Manages location data related to user nutrition events or locations of interest.

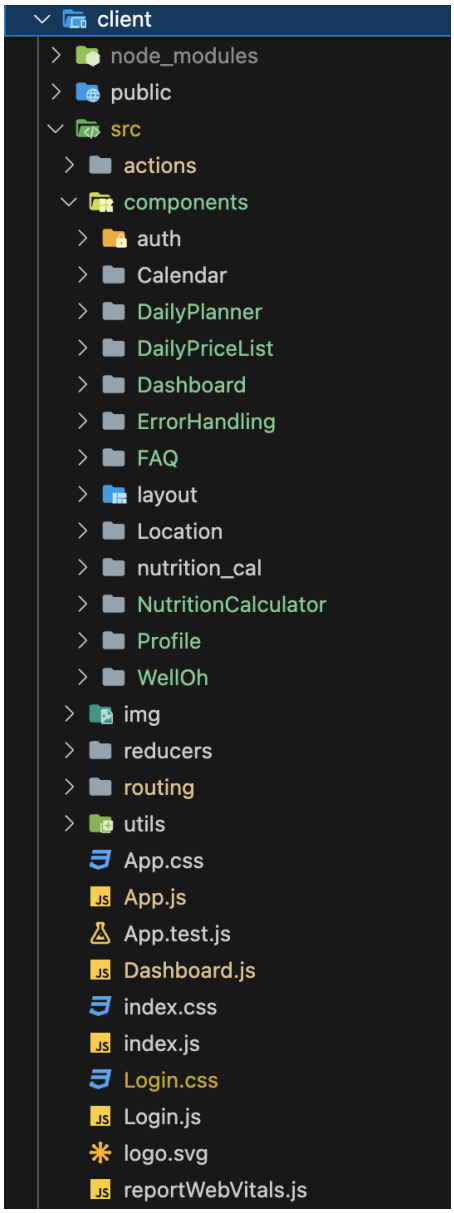
4. Persistent Data Layer

- **Database**:
    - Serves as the central storage for all persistent entities.
    - Stores data for users, meals, food items, diet plans, nutrition logs, and location information.

4. Application Skeleton

Please refer to the source code uploaded in the github repository for the application skeleton

a. Frontend



The frontend/ client (React.js) mainly consists of different directories to render the user interface in the webpage. The client application is structured to ensure clarity and ease of collaboration, featuring a well-organized architecture that enhances maintainability and scalability. At the core, App.js serves as the entry point, orchestrating the rendering of various components and screens.

Within the frontend directory, we have defined several key folders (as recommended by the framework used):

- **actions**: This folder houses all the action creators, defining the various interactions and data manipulation methods that can be dispatched throughout the application. These actions facilitate communication between components and the Redux store.

- **components**: This folder contains reusable UI components, ensuring a modular approach to development. Each component is designed to encapsulate specific functionality and styling, promoting code reusability and easier maintenance.

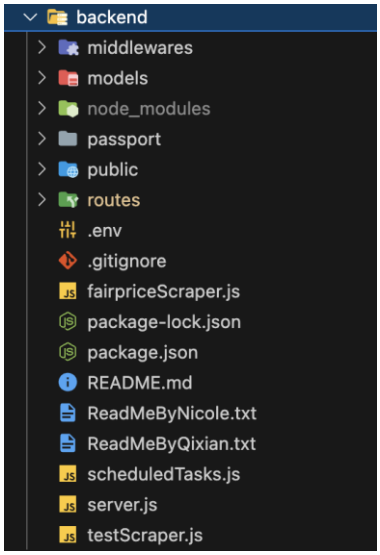
- **reducers**: Here, you'll find the reducers that manage the state of the application. These functions specify how the application's state changes in response to dispatched actions, providing a clear structure for state management.

- **routers**: This folder is responsible for defining the application's routing logic, determining how different URLs map to specific components and enabling smooth navigation throughout the app.

- **utils**: This folder contains utility functions and helpers that streamline various operations across the application, making the codebase cleaner and more efficient.

Together, these divisions create a cohesive and organized frontend structure, promoting efficient and effective collaboration.

b. Backend



The Node.js backend is designed with a modular and organized architecture, ensuring maintainability, scalability, and ease of collaboration. At the heart of the application is server.js, the main entry point that initializes the server and handles configuration.

Our key directories include:

- **middlewares**: This folder contains middleware functions that intercept and process HTTP requests before they reach the routes. These functions handle tasks such as authentication, logging, error handling, and validation, providing a clean and consistent workflow for request processing.

- **models**: The models folder defines the business objects (mongoose database schemas) that represent the core data structures of the application. These models map directly to database entities, enabling seamless interaction with the database.

- **routes**: This folder contains the route handlers, which define the REST API endpoints that manage communication between the frontend and backend. Each route is linked to a controller or service that handles the core logic behind the data processing.

- **server.js**: As the entry point of the backend, this file is responsible for setting up the server, loading the middleware, defining the routes, and connecting to the database. It ensures the entire backend infrastructure is properly initialized and ready to handle requests.

Together, this structure provides a clean, efficient, and organized backend system, ensuring smooth communication between the frontend and the database while promoting scalability and ease of development.