Software Architecture Document for Rehnuma

---

1. Purpose

This document aims to provide an architectural overview of the application Rehnuma. It encompasses details about system architecture, design decisions, and logic to ensure the application's maintainability, scalability, and security.

2. Scope

Our application aims to help users find the most optimal and cost-friendly routes for their grocery shopping trips. We will achieve this by building a mechanism that compares prices from various supermarkets and finds the shortest, most effective paths, keeping in view the fuel costs. It will also keep track of users' location preferences.

3. Component's Overview

- **Frontend:** Built using React.js for web and mobile responsiveness.
- **Backend:** Developed with Node.js to handle API requests and business logic.
- **Optimization mechanism:** Implemented using Python for route optimization and cost calculations.
- **Database:** Utilizes MongoDB for storing user profiles, shopping lists, and inventory data.
- **APIs:** Integrates with Google Maps API for location services and various supermarket APIs for price and inventory updates.

4. Architectural Goals and Constraints

- ➢ The app should be smooth, meaning the optimized itinerary should be generated within 8-10 seconds of the user requesting it.
- ➢ The system should be up and running at all times of the day.
- ➢ The system should be able to handle the increasing number of customers and the data of the growing supermarkets without compromising on the smoothness.
- ➢ The user data(profile and location details) should be protected.
- ➢ The UI must be easy to use for users from different walks of life.
- ➢ The code should be well documented so that it can be easily maintained.
- ➢ Supermarket price and inventory data should be updated every 24 hours to give users real-time information about supermarket inventory.
- ➢ Google Maps API must be monitored to prevent exceeding free-tier limits or incurring unexpected costs.
- ➢ The system should log errors and provide users with meaningful error messages in case of failures.

5. Use-case View

- User Registration & Login
- Profile Management
- Search for Products
- View Price Comparisons
- Optimize Shopping Route
- Supermarket Data Sync
- Integration with Maps API
- Error Handling & Notifications

4.1 Architecturally Significant Use Cases

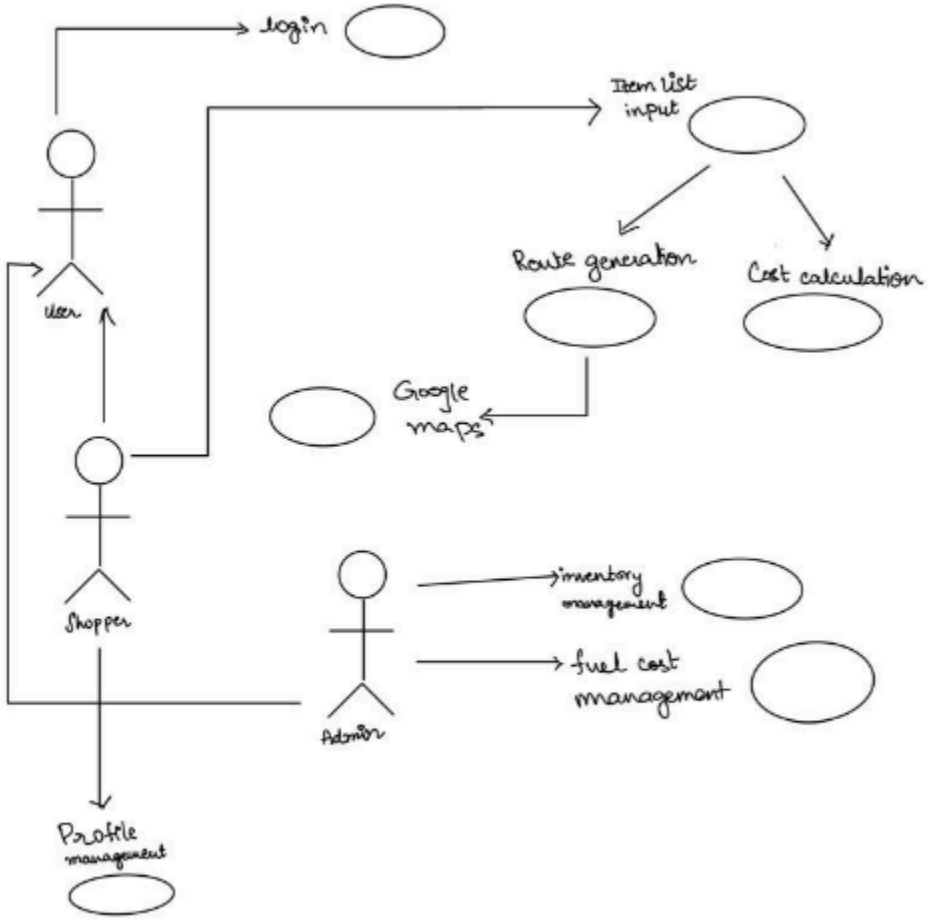


Figure 1: Architecturally significant use-case

4.1.1 User Registration & Login:

Users can sign up and log in using email or social authentication.

4.1.2 Profile Management:

Users can update location settings.

4.1.3 Search for Products:

Users can input their grocery items which will be searched across multiple supermarkets.

4.1.4 View Price Comparisons:

The system retrieves real-time or periodically updated prices from different stores.

4.1.5 Optimize Shopping Route:

The system calculates and suggests the most cost-effective shopping route considering distance, fuel costs, and total spending.

4.1.6 Supermarket Data Sync:

Periodic updates fetch inventory and price details from supermarkets.

4.1.7 Integration with Maps API:

Location services suggest optimal routes and travel estimates.

4.1.8 Error Handling & Notifications:

The system notifies users about unavailable products, price fluctuations, or changes in store data.

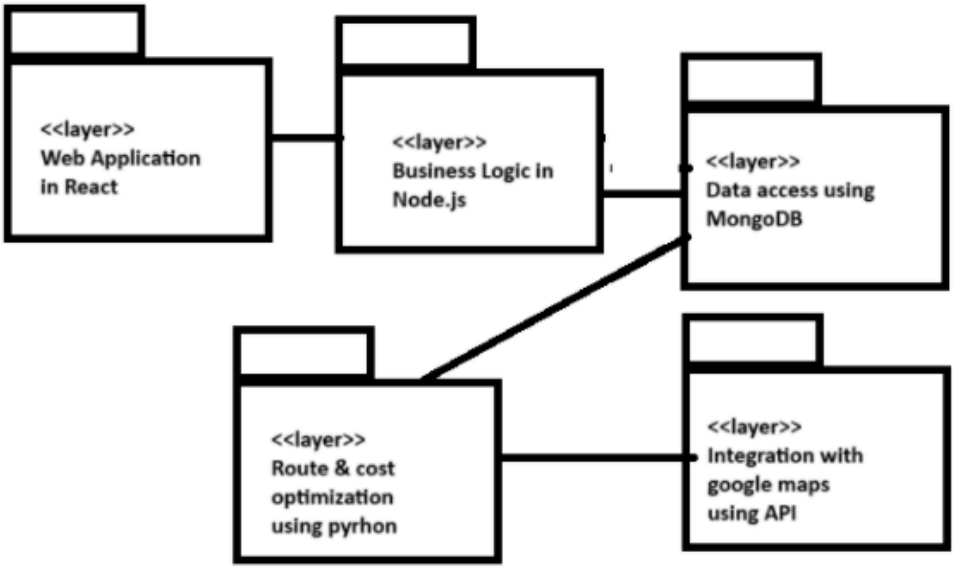
5. Logical View

A description of the logical view of the architecture. Describes the most important classes, their organization in service packages and subsystems, and the organization of these subsystems into layers.

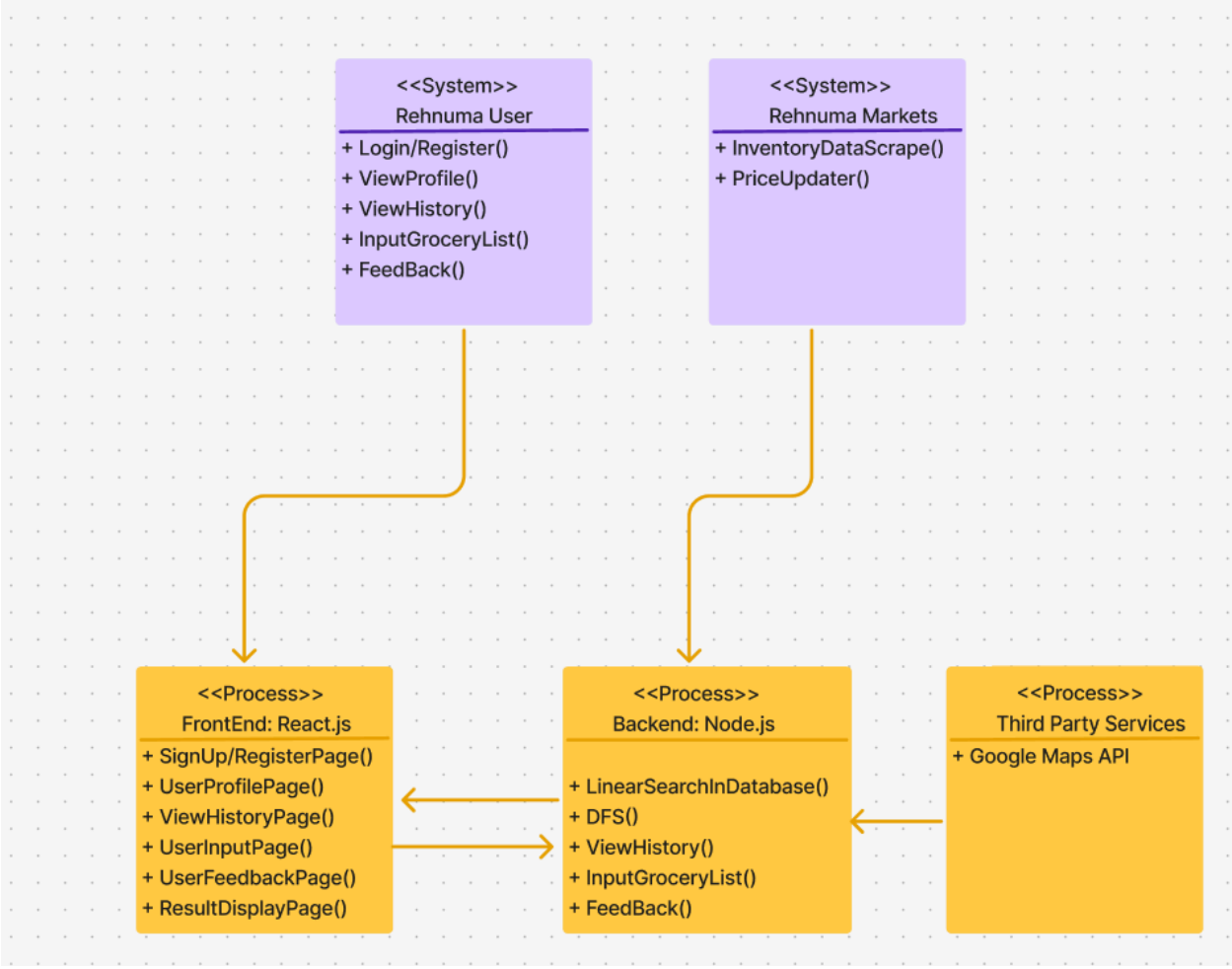
1. **User Interface Module:**
    - Handles user interactions and input validation.
    - Provides a responsive and intuitive UI using React.js.
2. **Authentication Module:**
    - Manages user registration, login, and session handling.
    - Uses OAuth/JWT for secure authentication and authorization.
3. **Shopping List and Profile Management Module:**
    - Manages CRUD operations for shopping lists and user profiles.
    - Stores data securely in MongoDB collections.
4. **Price Comparison and Inventory Module:**
    - Retrieves product prices and inventory from multiple supermarkets.
    - Ensures real-time data consistency using caching and scheduled updates.
5. **Route Optimization Module:**
    - Implements cost-effective route planning and fuel cost calculation.
    - Utilizes Python-based dynamic programming algorithms.

6. **API Integration Module:**
    - Integrates with external services like Google Maps API.
    - Manages API requests and responses securely.

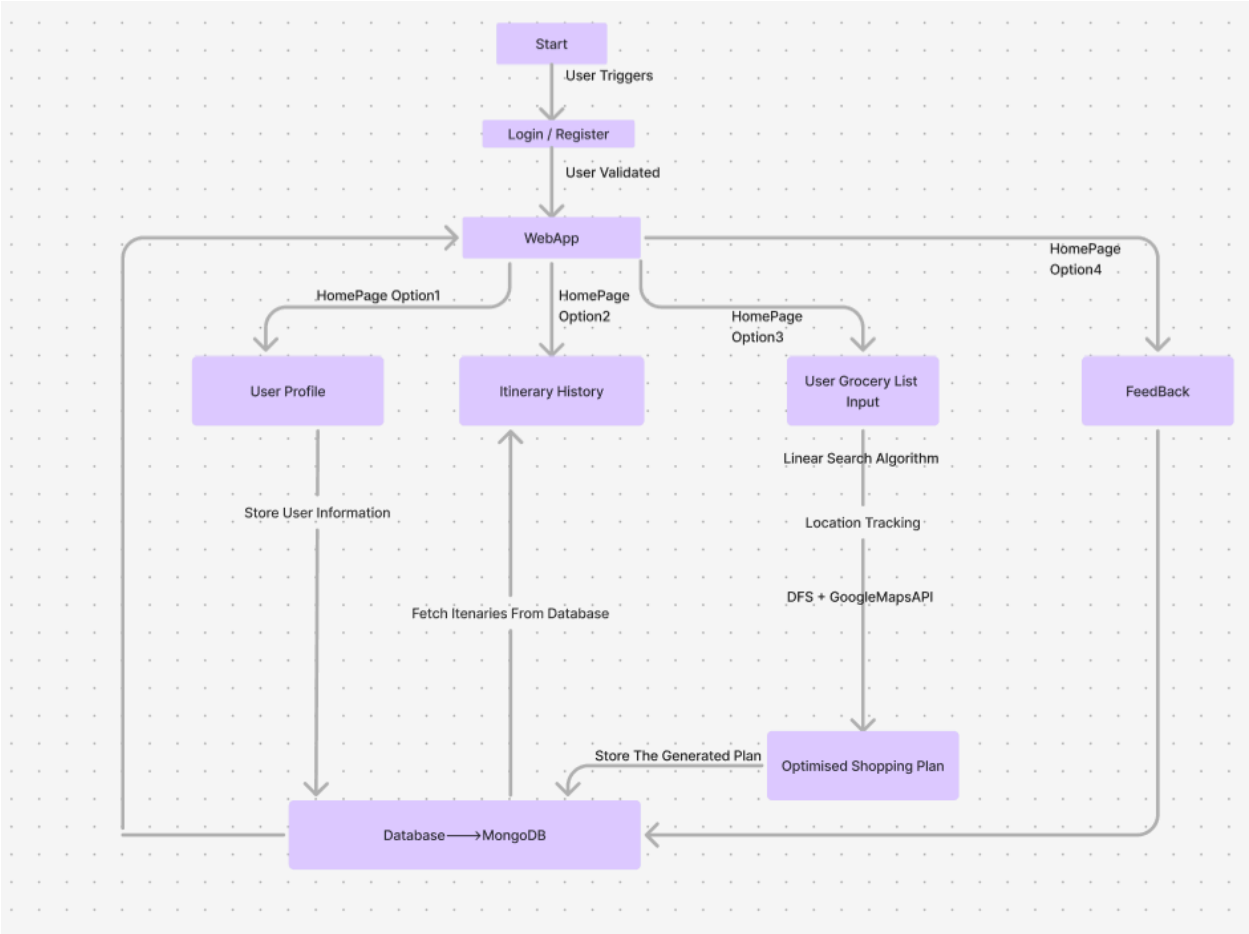
5.1 Architecture Overview – Package and Subsystem Layering



6. UML DIAGRAM:

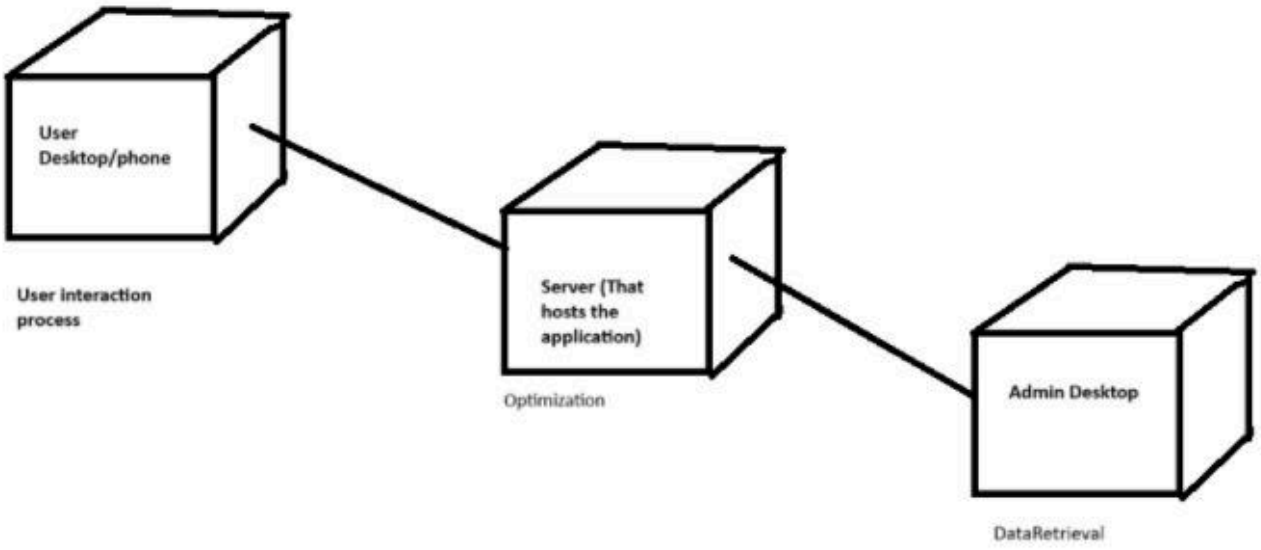


7. APPLICATION FLOW DIAGRAM:



8. Deployment View

A description of the deployment view of the architecture describes the various physical nodes for the most typical platform configurations. Also describes the allocation of tasks (from the Process View) to the physical nodes.



8.1 User Desktop/Phone

User will be able to log in as well as give input of their grocery list.

8.2 Admin Desktop

Admin will manage/update the data on the website. This data will include updated supermarket prices and fuel rates.

8.3 Server

The domain where our site will be hosted and will be available to multiple users

9. Size and Performance

The selected software architecture ensures that the system meets key performance and scalability requirements:

- The system shall support up to 2000 simultaneous users interacting with the central database and 500 concurrent users on localized servers.
- Data retrieval from external sources, such as APIs and legacy databases, shall have a latency of no more than 10 seconds.
- 80% of all transactions must be completed within 2 minutes to ensure system responsiveness.
- The client-side application shall require less than 20 MB of disk space and operate efficiently within 32 MB of RAM.

To meet these requirements, the architecture follows a distributed client-server model, where:

- Clients operate on local machines with lightweight processing needs.
- Servers handle computationally heavy tasks such as data retrieval, optimization, and processing.
- Caching mechanisms are used to reduce redundant computations and speed up frequent queries.

10. Quality

The software architecture supports key quality requirements for usability, reliability, and maintainability:

- The user interface shall adhere to modern UI/UX standards and be designed for ease of use with minimal training required.
- The system shall be available 24/7, with downtime not exceeding 4% over a given period.
- The Mean Time Between Failures (MTBF) shall exceed 300 hours, ensuring reliability.

- Built-in online help and documentation will provide step-by-step guidance and definitions of system-related terms.
- System updates and patches shall be easily downloadable from the server to ensure users have access to the latest features without manual installation.