1. INTRODUCTION

- ## Project Name

  Grocery WebApp

  Team Members:

  1) Satyajeet Zende

  2) Jagdish Jagdale

  3) Atharv Salvi

  4) Satyajit Patil

2.Project Overview

- ## Purpose

The purpose of the **GroceryWebApp** is to provide a convenient, efficient, and user-friendly platform for purchasing groceries online. It aims to eliminate the need for physical store visits by allowing customers to browse products, add them to a cart, make secure payments, and track orders from any device.

For store owners or administrators, the application serves as a digital management system to easily update product details, manage inventory, process orders, and handle customer queries. By integrating modern technologies like **Angular, Node.js, Express.js, and MongoDB**, the system ensures smooth performance, secure data handling, and scalability.

- ## Features

The Grocery Web App offers a complete online grocery shopping experience with separate functionalities for customers and administrators. Customers can easily register or log in to their accounts, browse products through advanced search, filters, and sorting options, and add items to their cart or wishlist. The platform supports a secure checkout process with multiple

payment options and provides real-time order tracking along with access to order history. Users can also share feedback by rating and reviewing products. On the admin side, the app includes powerful management tools such as adding, updating, or removing products, handling customer orders, updating their statuses, managing customer accounts, and monitoring inventory with low-stock alerts. Technically, the app is designed with a responsive, mobile-friendly interface, secure data storage in MongoDB, RESTful API integration for seamless frontend–backend communication, and role-based access control to ensure that customers and admins have access only to the features relevant to them.

3. Architecture

- ## Component Structure

1. AppComponent (Root Component)

- **Purpose:** The main entry point of the application, containing the global layout and router outlet.
- **Contains:** Header, footer, navigation bar.
- **Interaction:** Hosts the router to load different feature components.

2. HeaderComponent & FooterComponent

- **Purpose:** Common UI across all pages (logo, navigation links, search bar, cart icon, login/register button).
- **Interaction:**
    - HeaderComponent communicates with CartService to show the number of items in the cart.
    - Navigation triggers route changes.

3. HomeComponent

- **Purpose:** Displays featured products, categories, and promotional banners.

- **Interaction:** Fetches product data from ProductService.

4. ProductListComponent

- **Purpose:** Displays products by category or search results.
- **Interaction:**
    - Uses ProductService to get product data from the backend.
    - Passes selected product ID to ProductDetailComponent.

5. ProductDetailComponent

- **Purpose:** Shows detailed product information (price, description, stock availability).
- **Interaction:**
    - Receives product ID via route params.
    - Can add product to cart via CartService or to wishlist via WishlistService.

6. CartComponent

- **Purpose:** Displays cart items, quantity update, and remove options.
- **Interaction:**
    - Uses CartService to fetch and update cart items.
    - Navigates to CheckoutComponent.

7. CheckoutComponent

- **Purpose:** Handles delivery details, payment method selection, and order confirmation.
- **Interaction:**
    - Uses OrderService to place an order in the backend.
    - Clears cart after successful order.

8. OrderHistoryComponent

- **Purpose:** Displays the user's past orders and their statuses.

- **Interaction:** Fetches order data from OrderService.

9. LoginComponent / RegisterComponent

- **Purpose:** Handles authentication.
- **Interaction:**
    - Uses AuthService to log in or register users.
    - Stores JWT token for secure API calls.

10. AdminModule (Separate section for admins)

AdminProductComponent

- Add, edit, delete products (uses ProductService).

AdminOrderComponent

- View and update order status (OrderService).

AdminCustomerComponent

- View customer details and manage accounts.

Service Layer (Shared)

- **ProductService:** Handles product API calls.
- **CartService:** Manages cart state.
- **OrderService:** Handles order operations.
- **AuthService:** Manages login, registration, and JWT.
- **WishlistService:** Manages wishlist items.

Interaction Flow Example:

1. User clicks on a product in ProductListComponent.
2. Angular router loads ProductDetailComponent with product ID.
3. ProductDetailComponent fetches product info from ProductService.
4. User clicks "Add to Cart" → CartService updates cart → HeaderComponent updates cart count.

5. User proceeds to checkout → CheckoutComponent sends data via OrderService.
6. Admin sees new order in AdminOrderComponent.

- ## State Managment

In the **GroceryWebApp**, state management is handled using **Angular's built-in services** and the **RxJS BehaviorSubject** for reactive data sharing. The application maintains a centralized service for managing key states such as the shopping cart, user authentication status, product data, and order details.

When a user interacts with the application—such as adding a product to the cart, logging in, or updating profile details—the relevant service updates its state, and all components subscribed to that service automatically receive the updated data in real-time. This appr oach ensures a **single source of truth** for application data, avoids unnecessary API calls, and keeps components lightweight by delegating logic to services.

This method is simpler and more performance-friendly than heavy state management libraries like Redux, making it ideal for small-to-medium-sized Angular applications while still providing scalability for future feature expansions.

- ## Routing

The **GroceryWebApp** uses **Angular Router** to handle navigation between different views without reloading the entire page. The routing configuration is defined in the app-routing.module.ts file, where each route is mapped to a specific Angular component. This approach ensures smooth transitions, improves performance, and provides a single-page application (SPA) experience.

The routing structure is organized into **user** and **admin** sections to manage role-based access:

1. **Public Routes**
    - / → **HomeComponent** – Displays featured products and categories.
    - /products → **ProductListComponent** – Shows all available products with filtering options.
    - /product/:id → **ProductDetailComponent** – Displays detailed product information.
    - /login → **LoginComponent** – Allows users/admins to log in.
    - /register → **RegisterComponent** – Handles new user registration.
2. **User Routes (Protected)**
    - /cart → **CartComponent** – Shows products added to the shopping cart.
    - /checkout → **CheckoutComponent** – Handles order placement and payment.
    - /orders → **OrderHistoryComponent** – Displays past orders for the logged-in user.
3. **Admin Routes (Protected)**
    - /admin → **AdminDashboardComponent** – Main admin control panel.
    - /admin/products → **ManageProductsComponent** – Add, update, or delete products.
    - /admin/orders → **ManageOrdersComponent** – View and manage customer orders.
    - /admin/customers → **ManageCustomersComponent** – View registered customers and manage accounts.

- ## Prerequisites

  1) Node.js

  2) npm

  3)AngularCLI

  4)MangoDb

  5)CodeEditor

  6)WebBrowser

  7)Git

- ## Installation

To develop a full-stack Grocery web app using AngularJS, Node.js, and MongoDB, there are several prerequisites you should consider. Here are the key prerequisites for developing such an application:

Node.js and npm: Install Node.js, which includes npm (Node Package Manager), on your development machine. Node.js is required to run JavaScript on the server side.

- Download: https://nodejs.org/en/download/
- Installation instructions: https://nodejs.org/en/download/package-manager/

MongoDB: Set up a MongoDB database to store hotel and booking information. Install MongoDB locally or use a cloud-based MongoDB service.

- Download: https://www.mongodb.com/try/download/community
- Installation instructions: https://docs.mongodb.com/manual/installation/

Express.js: Express.js is a web application framework for Node.js. Install Express.js to handle server-side routing, middleware, and API development.

- Installation: Open your command prompt or terminal and run the following command: npm install express

Angular: Angular is a JavaScript framework for building client-side applications. Install Angular CLI (Command Line Interface) globally to create and manage your Angular project.

Install Angular CLI:

- Angular provides a command-line interface (CLI) tool that helps with project setup and development.
- Install the Angular CLI globally by running the following command:

npm install -g @angular/cli

Verify the Angular CLI installation:

- Run the following command to verify that the Angular CLI is installed correctly: ng version

You should see the version of the Angular CLI printed in the terminal if the installation was successful.

Create a new Angular project:

- Choose or create a directory where you want to set up your Angular project.
- Open your terminal or command prompt.
- Navigate to the selected directory using the cd command.
- Create a new Angular project by running the following command: ng new client Wait for the project to be created:
- The Angular CLI will generate the basic project structure and install the necessary dependencies

Navigate into the project directory:

- After the project creation is complete, navigate into the project directory by running the following command: cd client

Start the development server:

- To launch the development server and see your Angular app in the browser, run the following command: ng serve / npm start
- The Angular CLI will compile your app and start the development server.
- Open your web browser and navigate to http://localhost:4200 to see your Angular app running.

You have successfully set up Angular on your machine and created a new Angular project. You can now start building your app by modifying the generated project files in the src directory.

Please note that these instructions provide a basic setup for Angular. You can explore more ad- vanced configurations and features by referring to the official Angular documentation: https://angular.io

HTML, CSS, and JavaScript: Basic knowledge of HTML for creating the structure of your app, CSS for styling, and JavaScript for client-side interactivity is essential.

Database Connectivity: Use a MongoDB driver or an Object-Document Mapping (ODM) library like Mongoose to connect your Node.js server with the MongoDB database and perform CRUD (Create, Read, Update, Delete) operations.

Front-end Framework: Utilize Angular to build the user-facing part of the application, including products listings, booking forms, and user interfaces for the admin dashboard.

Version Control: Use Git for version control, enabling collaboration and tracking changes throughout the development process. Platforms like GitHub or Bitbucket can host your repository.

- Git: Download and installation instructions can be found at: https://git-scm.com/downloads

Development Environment: Choose a code editor or Integrated Development Environment (IDE) that suits your preferences, such as Visual Studio Code, Sublime Text, or WebStorm.

- Visual Studio Code: Download from https://code.visualstudio.com/download
- Sublime Text: Download from https://www.sublimetext.com/download
- WebStorm: Download from https://www.jetbrains.com/webstorm/download

To Connect the Database with Node JS go through the below provided link:

• Link: https://www.section.io/engineering-education/nodejs- mongoosejs-mongodb/

5.Folder Structure

- ### Client

The **GroceryWebApp** frontend is structured using the Angular framework, organized for scalability and maintainability.

1. client/src/app

This is the main application directory containing all Angular components, modules, and routing logic.

2. components/

Contains **reusable UI components** used across the app:

- **feedback/** – Component for collecting and displaying user feedback.
- **footer/** – Application footer section, visible on all pages.
- **header/** – Navigation header with links to main sections.
- **history/** – Displays the user's order history.
- **home/** – Homepage of the application.
- **landing-page/** – Landing page for first-time visitors.

- **loader-spinner/** – A loading indicator shown during data fetch or processing.
- **login/** – User login form.
- **my-cart/** – Shopping cart display and management.
- **my-orders/** – List of previous and current orders.
- **not-found/** – 404 page for invalid routes.
- **place-order/** – Order placement and checkout page.
- **product-details/** – Detailed view of a selected product.
- **register/** – User registration form.

3. modules/

Contains **feature-specific modules** for better separation of concerns.

Admin Module (modules/admin)

Includes admin-specific functionality for managing products and categories.

- **components/**
    - **add-categories/** – Add new product categories.
    - **add-products/** – Add new products to the store.
    - **admin-dashboard/** – Main admin control panel.
    - **dashboard/** – Overview of admin metrics and stats.
    - **feedback/** – View feedback from users.
    - **footer/** – Footer for admin pages.
    - **header/** – Navigation header for admin pages.
    - **home/** – Admin home page.

- ## Client

1. Helper Functions

- **formatDate(date)** – Converts a raw date (ISO or timestamp) into a user-friendly format (e.g., DD-MM-YYYY).
- **calculateTotal(cartItems)** – Sums up all product prices in the cart, factoring in quantity.
- **validateEmail(email)** – Checks if the email entered follows a valid format using regex.

2. Utility Classes

- **TextUtils** – Contains string manipulation methods like capitalizeFirstLetter() or truncateText().
- **ApiEndpoints** – A constant object storing all API URLs in one place, making it easier to update.

3. Custom Hooks (React projects)

- **useFetch(url)** – A reusable hook to fetch data from an API and manage loading/error states.
- **useAuth()** – Manages authentication state and provides methods like login() and logout().
- **useDebounce(value, delay)** – Delays updates to avoid unnecessary API calls during fast typing.

6.Running the Application

- **Commands to start Frontend locally**

  1️⃣ **Navigate to your frontend folder**

  cd frontend

  2️⃣ **Install dependencies** (only needed the first time or when package.json changes)

  npm install

  3️⃣ **Start the development server**

  npm start

  4️⃣ **Access in browser**

  Once it compiles successfully, open:

  http://localhost:3000 (React default)
  http://localhost:4200 (Angular default)

7.Component Documentation

- **Key Components**

1. Navbar

Purpose:
Provides navigation links for the application, allowing users to move between different pages (Home, Products, Cart, Login, etc.).

Props:

- user*(object)* – current logged-in user details (optional).
- onLogout*(function)* – callback function to handle user logout.

2. ProductCard

Purpose:
Displays a single product's details such as name, price, image, and an "Add to Cart" button.

Props:

- product*(object)* – contains product details (id, name, image, price, description).
- onAddToCart*(function)* – function to add the product to the cart.

3. ProductList

Purpose:
Renders a grid/list of ProductCard components by mapping over product data.

Props:

- products*(array)* – list of product objects to display.
- onAddToCart*(function)* – function to add a selected product to the cart.

4. Cart

Purpose:
Shows all products added to the cart with the option to increase, decrease, or remove items.

Props:

- cartItems*(array)* – products currently in the cart.
- onRemoveItem*(function)* – remove product from the cart.
- onUpdateQuantity*(function)* – change product quantity in the cart.

5. AdminDashboard

Purpose:
Provides an interface for admins to manage products, view orders, and track customers.

Props:

- products*(array)* – list of products for management.
- orders*(array)* – list of orders placed by customers.
- onAddProduct*(function)* – add a new product.
- onUpdateProduct*(function)* – update product details.
- onDeleteProduct*(function)* – delete a product.

6. LoginForm

Purpose:
Allows users to log in with email and password.

Props:

- onLogin*(function)* – handles the login process when the form is submitted

Reusable Components

1. Button Component (Button.jsx)

- **Purpose**:
  A customizable button used throughout the application for actions like “Add to Cart,” “Login,” “Checkout,” and “Submit.”
- **Props**:
  - label*(string)* – Text to display inside the button.
  - onClick*(function)* – Function to be executed on click.
  - type*(string)* – Button type ("button", "submit", "reset").

    - variant*(string)* – Style type (e.g., "primary", "secondary", "danger").
    - disabled*(boolean)* – Disables the button if true.
- **Example Usage**:

```
<Button label="Add to Cart" onClick={handleAddToCart} variant="primary" />
```

2. Input Field Component (InputField.jsx)

- **Purpose**:
  A reusable form input field with label and error message support.
- **Props**:
    - label*(string)* – Field label text.
    - type*(string)* – Input type ("text", "email", "password", "number").
    - value*(string)* – Current value of the input.
    - onChange*(function)* – Callback function for handling input changes.
    - placeholder*(string)* – Placeholder text.
    - error*(string)* – Error message to display.
- **Example Usage**:

```
<InputField
  label="Email"
  type="email"
  value={email}
  onChange={(e) => setEmail(e.target.value)}
  error={emailError}
/>
```

3. Modal Component (Modal.jsx)

- **Purpose**:
  Displays content in an overlay dialog box for confirmations, product details, or warnings.
- **Props**:
  - isOpen*(boolean)* – Whether the modal is visible.
  - onClose*(function)* – Function to close the modal.
  - title*(string)* – Title of the modal.
  - children*(node)* – Custom JSX content to render inside.
- **Example Usage**:

```
<Modal isOpen={showDeleteConfirm} onClose={closeModal} title="Confirm Delete">
  <p>Are you sure you want to delete this product?</p>
</Modal>
```

4. Loader Component (Loader.jsx)

- **Purpose**:
  A visual loading spinner used while fetching API data.
- **Props**:
  - size*(string)* – Size of the loader ("small", "medium", "large").
  - color*(string)* – Loader color.
- **Example Usage**:

```
{loading && <Loader size="large" color="#28a745" />}
```

5. Card Component (Card.jsx)

- **Purpose**:
  A styled container for displaying product information, categories, or user details.
- **Props**:
  - title*(string)* – Title text of the card.
  - image*(string)* – URL of the image.
  - description*(string)* – Short description text.
  - children*(node)* – Any additional elements inside the card.
- **Example Usage**:

```
<Card
  title={product.name}
  image={product.image}
  description={product.description}
>
  <Button label="View Details" onClick={() => openDetails(product)} />
</Card>
```

8.State Mangment

Global State

In the Grocery Web App, **global state** is primarily handled using **AngularContext API** with the useReducer hook to manage complex state changes.
This ensures that important data such as **user authentication**, **cart items**, and **wishlist** is available across all components without repetitive prop drilling.

Global State Structure

The main global states include:

1. **Auth State** – Stores logged-in user info, role (User/Admin), and JWT token.
2. **Cart State** – Keeps track of items added to the cart, quantities, and total price.
3. **Wishlist State** – Stores products that the user has marked for later.
4. **Product State** – Maintains the fetched product list for browsing.
5. **Order State** – Holds order details during checkout and tracking.

How State Flows Across the Application

1. Initialization

- On app load, the **Auth Context** checks localStorage for an existing JWT token and user data.
- If found, it sets the user as logged in and fetches related data (cart, wishlist).

2. User Interaction

- When a **user adds a product to the cart**, the **Cart Context** updates its state using a reducer function (ADD_TO_CART) and syncs it to the backend.

- If the user **logs out**, the **Auth Context** clears the stored token, and Cart/Wishlist states are reset.

3. Data Fetching

- The **Product Context** fetches product data from the backend API and updates the state so it's accessible to **ProductList**, **ProductDetails**, and **Search** components.

4. Checkout & Orders

- During checkout, the **Order Context** sends the cart data to the backend for order creation.
- Once confirmed, the cart is cleared, and order details are stored for tracking.

Example State Flow Diagram

User Logs In → AuthContext updates → Fetch user data → CartContext & WishlistContext populated → User adds product to cart → CartContext updates → Backend syncs cart → User checks out → OrderContext sends order to backend → Order history updated.

Local State

Local state refers to the state that is **managed within a single component** and is not shared across the entire application. It is used for **UI-specific data** and **temporary values** that do not require global access.

Usage in Grocery Web App

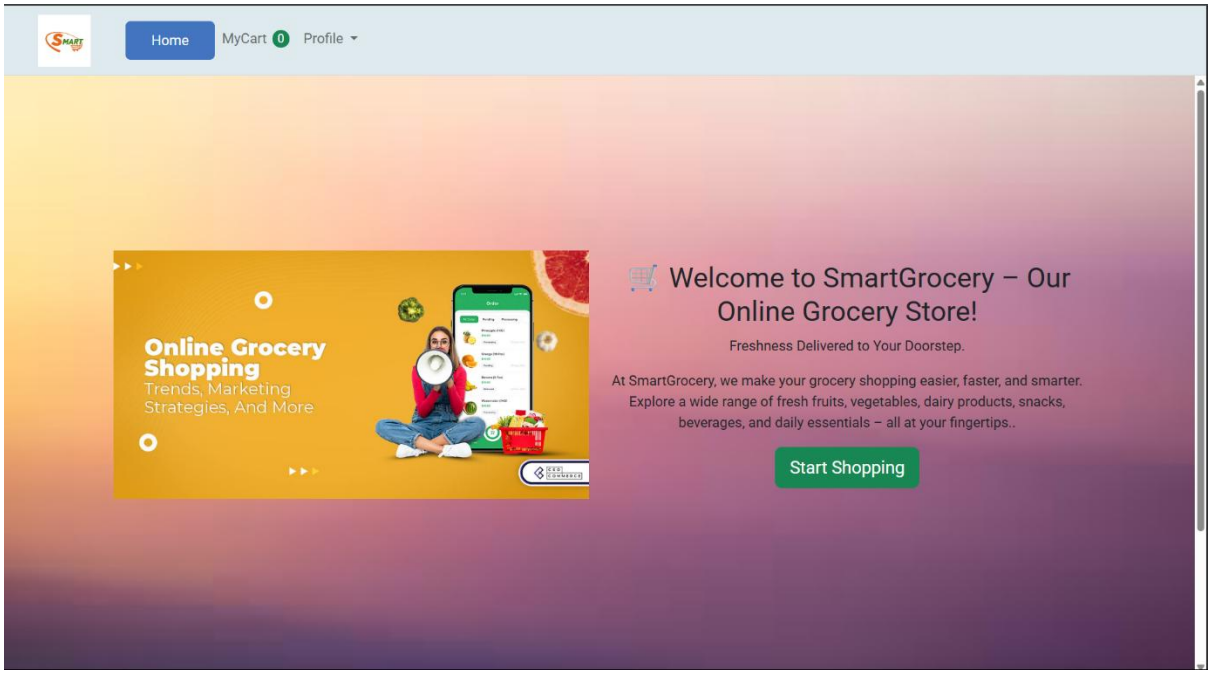
1. **Form Inputs**
    - Used in login, signup, add product, and feedback forms to store values before submission.
    - Example: email, password, productName, feedbackMessage.
2. **Modal & Dialog Visibility**

- Used to toggle modals for actions like **Add to Cart**, **View Product Details**, or **Confirm Delete**.
- Example: isModalOpen, showDeleteConfirm.

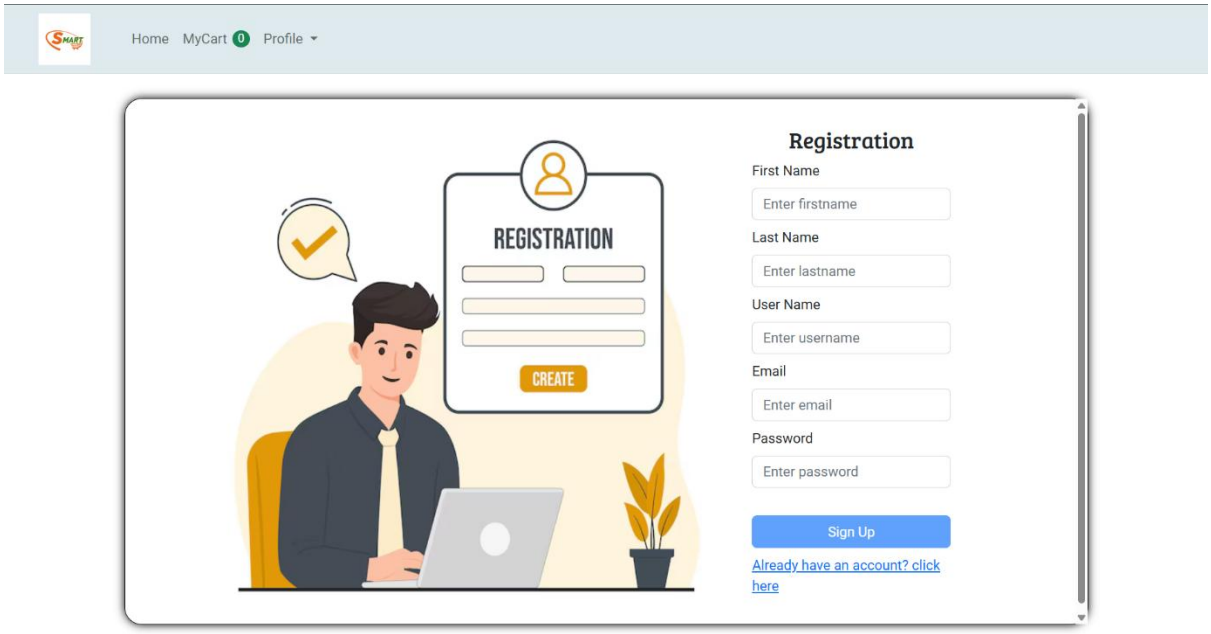
3. **Filtering & Sorting**
    - Used to store temporary filter and sort preferences for products.
    - Example: selectedCategory, sortByPrice.
4. **Pagination**
    - Used to manage the current page of product listings.
    - Example: currentPage, itemsPerPage.
5. **Temporary UI States**
    - For showing loaders, success/error messages.
    - Example: isLoading, errorMessage

9.User Interface

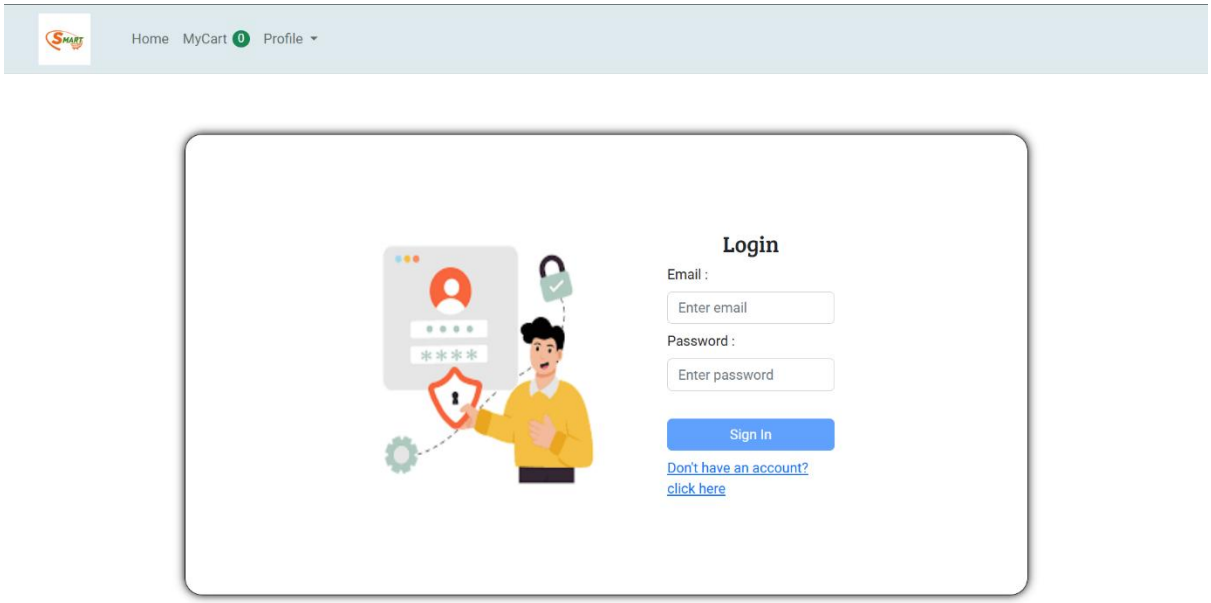
1)Landing Page



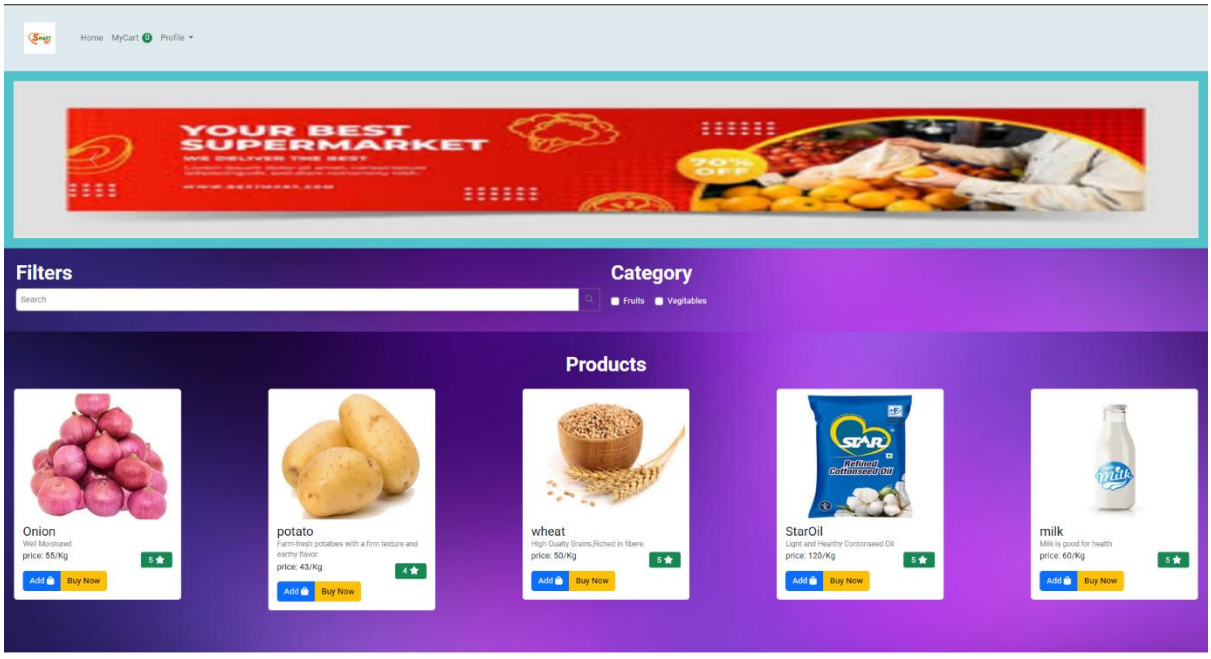
2)Register Page



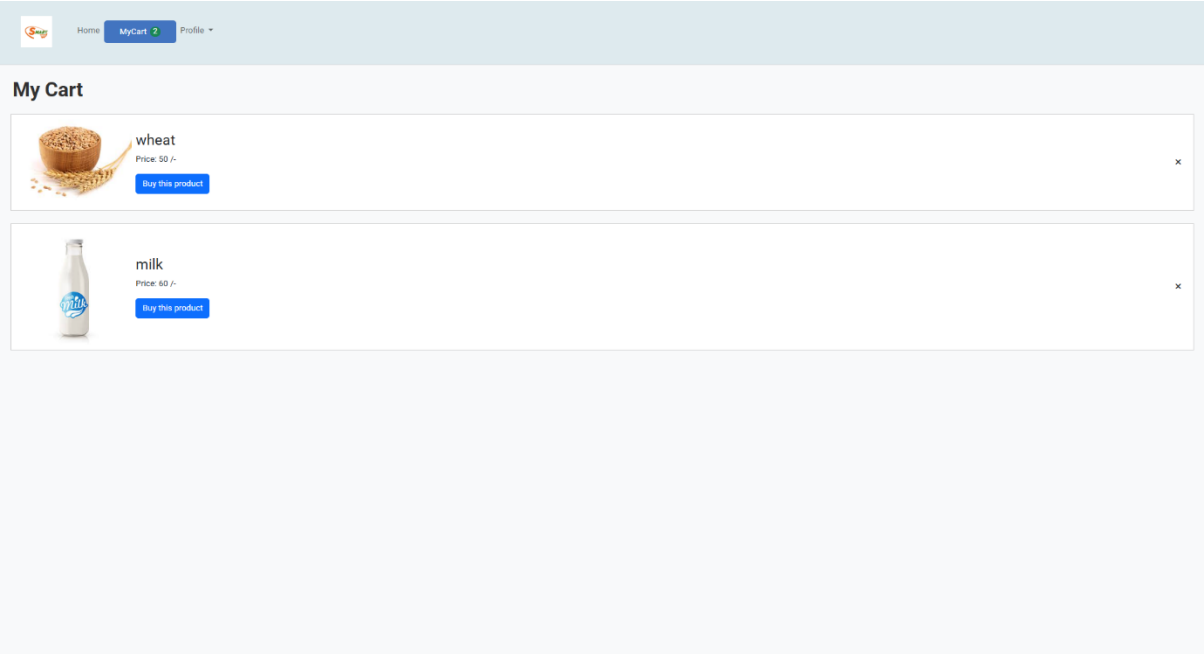
3)Login page



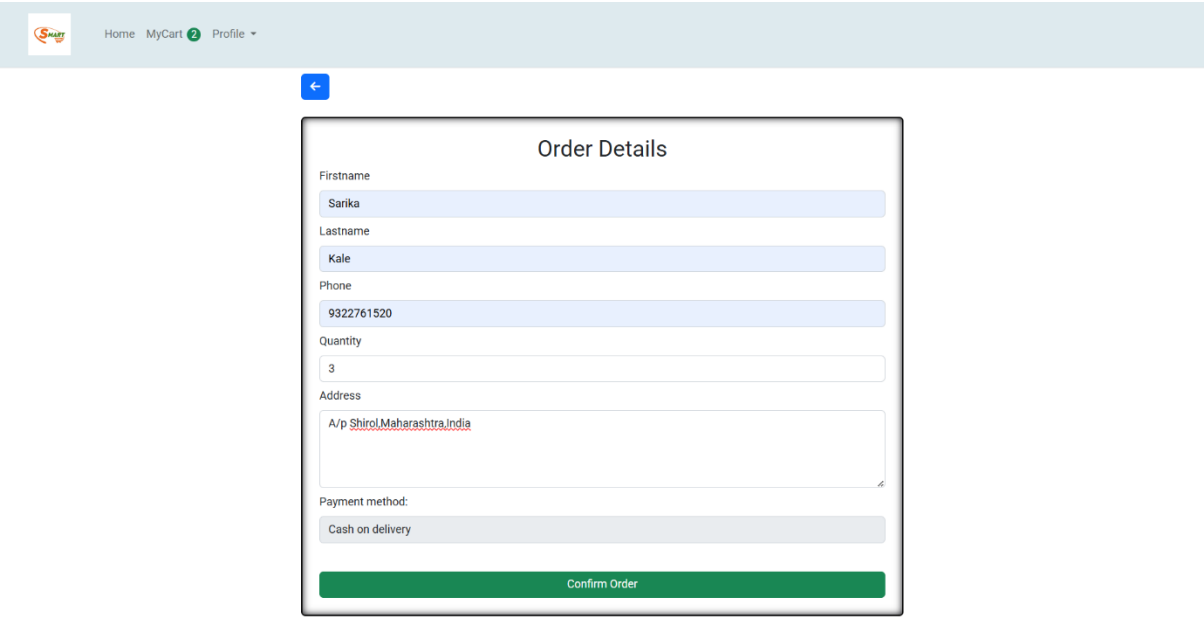
4)Shopping Page



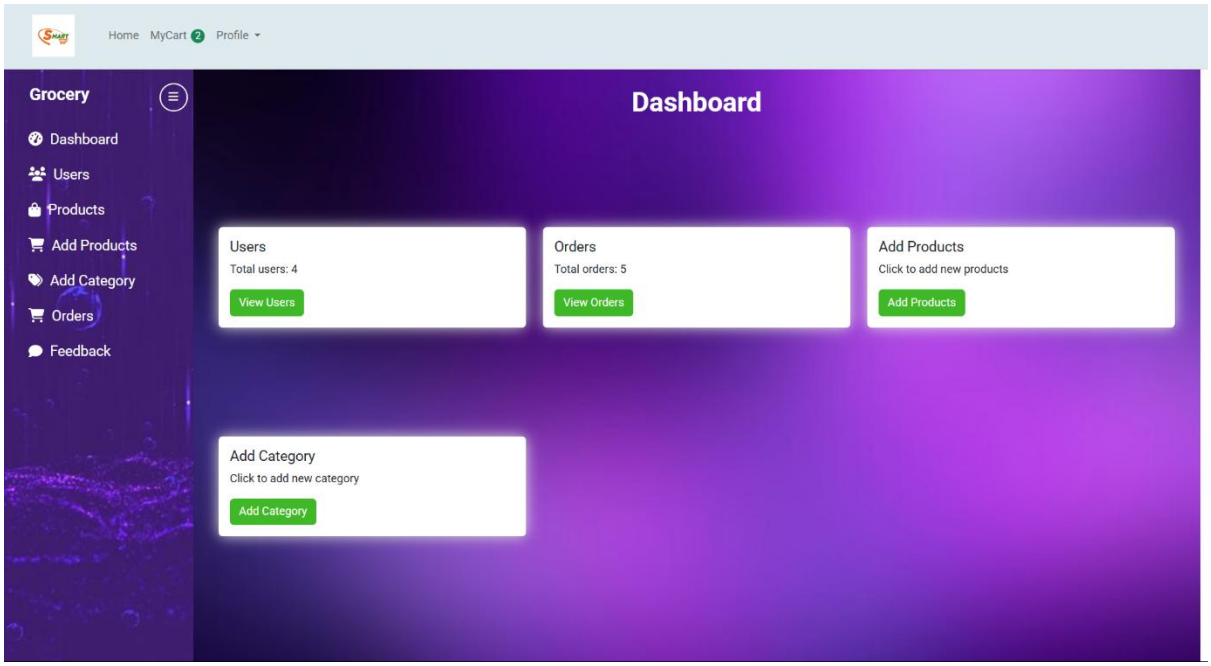
5)Cart Page



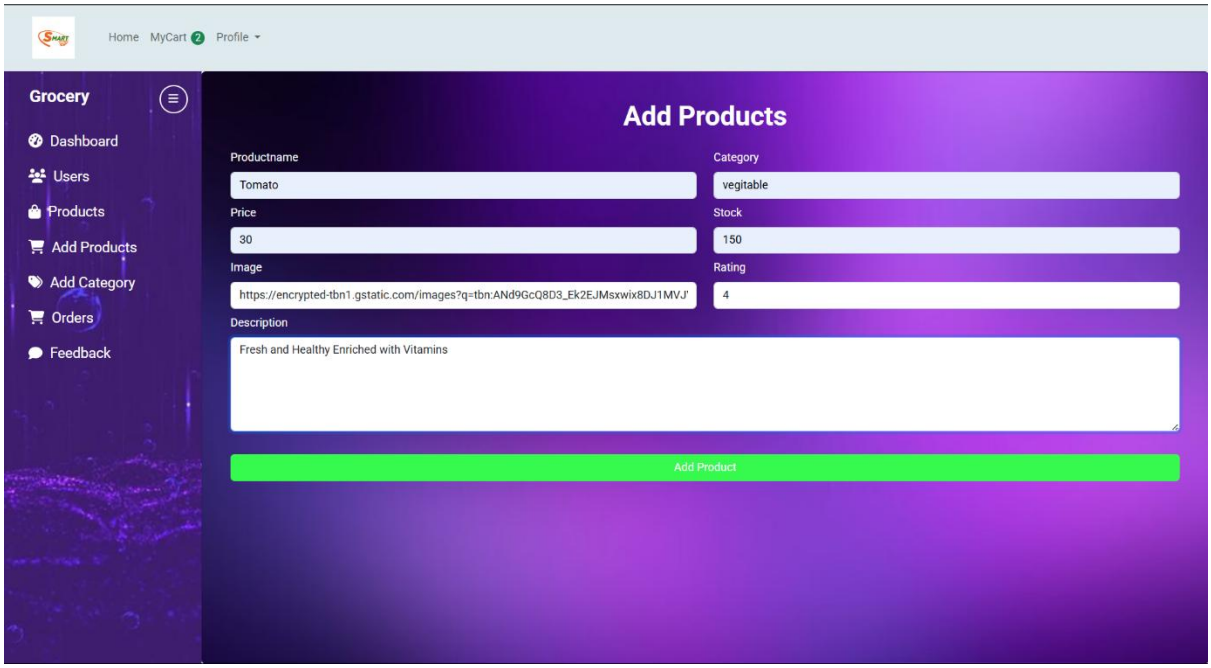
6)Place Order



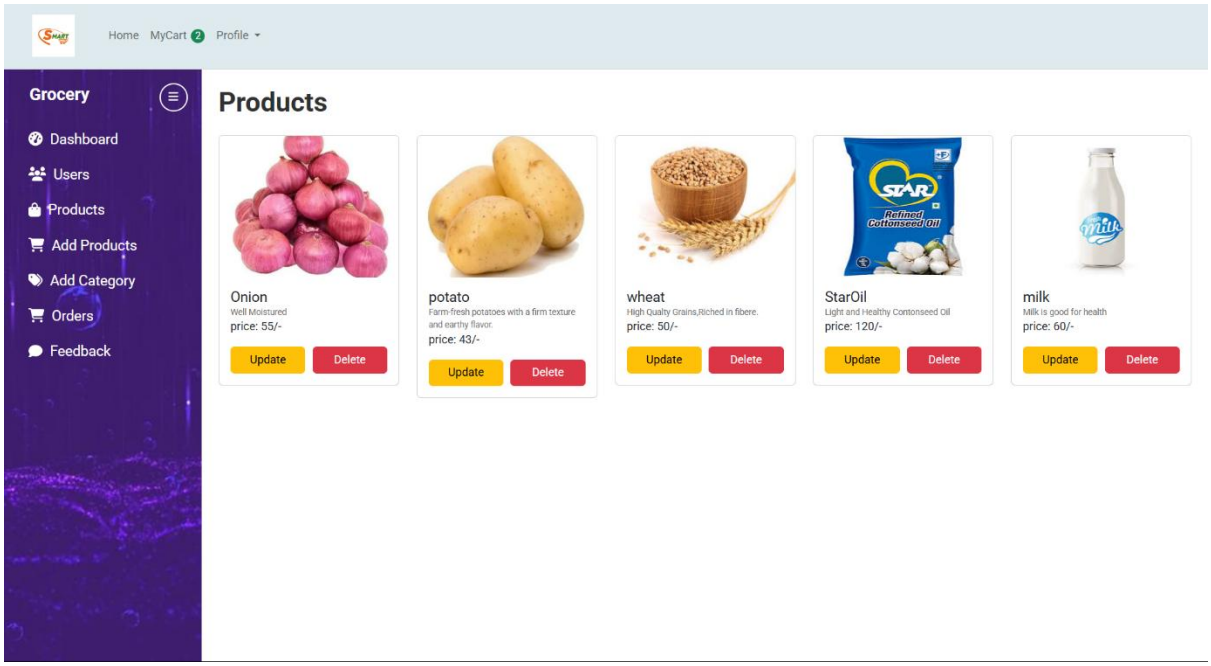
7)Admin Dashboard



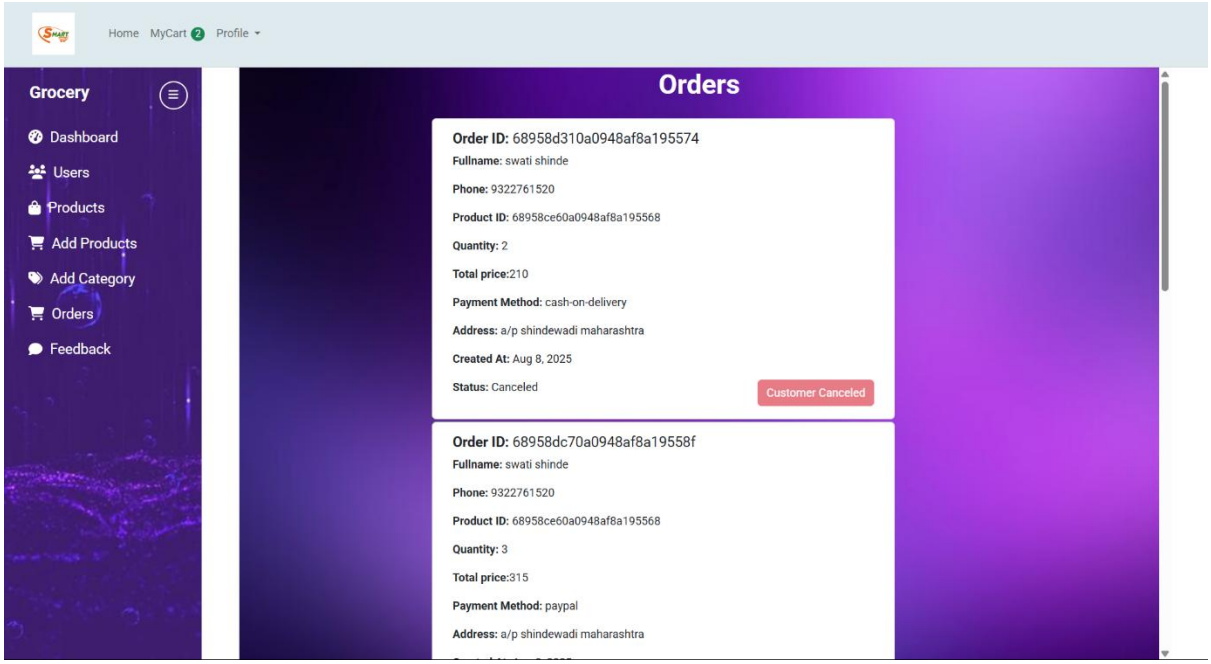
8)AddProduct Page



9)Product Page



10)Orders



10.Styling

Css Framework/Libraries

In the Grocery Web App, **Bootstrap** is used as the primary CSS framework for styling.

Why Bootstrap is used

- **Responsive Design** – Ensures the UI adapts seamlessly to desktops, tablets, and mobile devices without writing a lot of custom media queries.
- **Predefined Components** – Offers a wide range of ready-to-use components such as **navbar, cards, buttons, modals, forms, and grids**, which speed up development.
- **Grid System** – The 12-column grid layout helps create flexible and consistent layouts across all pages.
- **Utility Classes** – Provides utility classes for margin, padding, colors, typography, and alignment without writing extra CSS.
- **Cross-Browser Compatibility** – Works consistently across all modern browsers.

Implementation in the Project

1. **Installation** (if not using CDN)

npm install bootstrap

2. Or add via CDN in index.html:

<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">

3. **Import in AngularApp**

Open angular.json and find the "styles" and "scripts" arrays under the build options for your project.
Add Bootstrap's CSS and JS paths from node_modules:

```
"styles": [
  "node_modules/bootstrap/dist/css/bootstrap.min.css",
  "src/styles.css"
],
"scripts": [
  "node_modules/bootstrap/dist/js/bootstrap.bundle.min.js"
]
```

11.Testing

- ## Testing Stratergy

The project follows a **multi-level testing strategy** to ensure component quality, application stability, and reliable user experiences. Testing is divided into **unit**, **integration**, and **end-to-end (E2E)** phases.

1. Unit Testing

Purpose:

- Validate the logic of individual Angular components, services, and pipes in isolation.

Tools Used:

- **Jasmine** – Test framework.
- **Karma** – Test runner for executing unit tests in a browser environment.

Approach:

- Each component has a dedicated .spec.ts file.

- Use Angular TestBed to configure a test module that mirrors the actual module dependencies.
- Mock services and HTTP calls to isolate logic.

Example:

```
it('should create the ProductCard component', () => {
  const fixture = TestBed.createComponent(ProductCardComponent);
  const component = fixture.componentInstance;
  expect(component).toBeTruthy();
});
```

---

2. Integration Testing

Purpose:

- Verify how multiple Angular components and services work together.

Tools Used:

- **Angular Testing Library** – Helps simulate real user interactions rather than focusing on internal implementation details.

Approach:

- Test flows like product search, cart addition, and checkout process where multiple components interact.
- Use mock API responses to simulate backend communication.

Example:

```
it('should display product list when API returns data', async () => {
  mockProductService.getProducts.and.returnValue(of(mockProducts));
  fixture.detectChanges();
  const items = fixture.nativeElement.querySelectorAll('.product-item');
  expect(items.length).toBe(mockProducts.length);
});
```

3. End-to-End (E2E) Testing

Purpose:

- Validate the **entire** application flow from the user's perspective.

Tools Used:

- **Protractor** (default Angular E2E testing tool) or **Cypress** for more modern E2E testing.

Approach:

- Test critical workflows such as **User Login → Product Selection → Cart → Checkout → Payment Success**.
- Ensure UI changes reflect backend state changes.

- ## Code Coverage

Code Coverage in Angular Grocery Web App

Purpose

Code coverage is used to measure how much of the application's source code is executed when running tests.
It ensures:

- All critical components, services, and modules are tested.
- Dead/unreachable code is minimized.
- Better maintainability and fewer production bugs.

Tools Used

- **Karma + Jasmine** (Default Angular Testing Stack)
- **Istanbul** (via Angular CLI) for coverage reports

How It Works

Angular CLI comes with built-in support for coverage reports using Istanbul. When tests are run with the --code-coverage flag, Istanbul instruments the code to determine:

- **Statements Coverage** → % of statements executed
- **Branches Coverage** → % of control flow branches tested
- **Functions Coverage** → % of functions called
- **Lines Coverage** → % of lines executed

Commands

ng test --code-coverage

Coverage Report Location

After running the command:

- Coverage report is generated inside:

/coverage/<project-name>/

- Open index.html in the /coverage folder to view the detailed report in a browser.

Coverage Targets

Defined in angular.json or enforced via CI/CD pipeline:

- **Statements**: ≥ 80%
- **Branches**: ≥ 75%
- **Functions**: ≥ 80%
- **Lines**: ≥ 80%

12.Screenshots or Demo

- ### Link of Demo Video

Video Link:https://www.youtube.com/embed/--0gp64-AJA

github Link: https://github.com/satya6408/GrosaryWebApp

13.Known Issues

- ### Known bugs or issues

In the current version of the Grocery Web App, there is a known issue where the **product image does not update immediately after the admin uploads a new image** for an existing product.

- **Cause:** The frontend product list uses cached API responses and does not automatically re-fetch the updated image until a full page refresh is performed.
- **Impact:** Users may temporarily see the old product image instead of the updated one.
- **Workaround:** Perform a hard refresh (Ctrl + F5) or clear the browser cache to view the updated image.
- **Planned Fix:** Implement cache-busting by appending a unique query parameter to image URLs when updated.

14.Future Enhancement

- ## New Features

1. **Advanced Search & Filters** – Implement category-wise, price-range, and rating-based product filtering for better user experience.
2. **AI-based Product Recommendations** – Use machine learning to suggest products based on user purchase history and browsing patterns.
3. **Progressive Web App (PWA) Support** – Enable offline browsing and push notifications for order updates.
4. **Multi-language Support** – Provide the app in multiple languages to cater to a wider audience.
5. **Subscription & Auto-Reorder** – Allow users to set recurring orders for frequently purchased items.
6. **Real-time Order Tracking** – Integrate live delivery status updates with Google Maps API.
7. **Payment Gateway Expansion** – Add more payment options, including UPI AutoPay and Wallet integrations.
8. **Wishlist Sharing** – Allow users to share their wishlist with friends or family.