FULL STACK DEVELOPMENT WITH MERN

PROJECT DOCUMENTATION

1. INTRODUCTION

- **Project Title:** Grocery Webapp
- **Team Members:**

Priyashree H (311521104035)

NM ID : 41DD7240F6D09F490A34A922558253A1

Sai Shivanie M (311521104047)

NM ID : 506BA45A3A856D2AB504D135D713E6D8

Devi Priya E (311521104010)

NM ID : 34485FAA13BFD14B58A42B958D99F94F

Srirenganathan (311521104057)

NM ID : 410823056230B1B8789459250A6780B1

2. PROJECT OVERVIEW

The online grocery app is a full-stack web application developed using the MERN stack (MongoDB, Express.js, React, and Node.js) to provide a seamless and efficient online grocery shopping experience. This platform allows customers to browse a wide range of grocery items, add them to their shopping cart, and place orders with secure payment options. The application also offers an admin panel to manage products, monitor orders, and handle user accounts, making it versatile for both end-users and administrators.

The app's frontend, developed with React, provides a responsive and interactive user interface, enabling customers to search and filter products, view detailed information, and manage their cart items in real-time. Redux is utilized for state management, ensuring consistent data flow across components, especially in handling the cart and order features. The backend, powered by Node.js and Express.js, supports RESTful APIs that handle all CRUD operations, including user authentication with JWT for secure login and data protection. MongoDB serves as the app's database, efficiently storing and managing data related to users, products, and orders. The modular design of the system allows for easy scaling and future expansion, with

functionalities like order tracking and personalized recommendations planned as potential enhancements.

Overall, this online grocery app demonstrates the power of the MERN stack in developing modern, feature-rich, and scalable e-commerce solutions. It addresses the growing demand for online grocery shopping, delivering convenience to users and efficiency to administrators.

Purpose:

The purpose of this online grocery app project is to create a convenient, efficient, and accessible platform for purchasing grocery items online. With the increasing demand for digital solutions in daily tasks, this app aims to simplify the grocery shopping process by allowing users to browse a virtual catalogue of products, add items to a cart, place orders, and make payments all from the comfort of their homes.

The project also serves to streamline the administrative side of grocery management. Through an intuitive admin panel, store managers can easily add, update, or remove products, manage inventory, and monitor orders. By implementing a secure authentication system and intuitive user interface, the app aims to ensure a seamless shopping experience for customers while offering easy management capabilities for administrators.

Ultimately, this project is intended to reduce the time, effort, and physical constraints associated with traditional grocery shopping, making it possible for users to shop anytime, from anywhere, while keeping the backend operations organized and efficient for the business.

Goals:

1. **User-Friendly Interface**: Develop an intuitive and clean interface where users can easily browse grocery items, view categories, and access detailed product information.
2. **Efficient Cart and Checkout System**: Implement a reliable shopping cart that allows users to add, update, or remove items, and a secure, streamlined checkout process to enhance the user experience.
3. **Responsive Design**: Ensure the platform works seamlessly across devices, particularly focusing on mobile responsiveness, so users can shop from their phones or tablets with ease.

4. **Inventory Management:** Include an inventory management system to keep track of product availability and update stock levels based on customer purchases.
5. **Search and Filtering Options:** Offer a robust search feature and filtering options (e.g., by category, price, or popularity) to help users find the products they need quickly.
6. **Backend Integration for Data Management:** Build a backend that handles data securely and efficiently, supporting user authentication, order processing, and possibly integration with third-party payment providers.
7. **Scalability:** Design the platform to support future growth, both in terms of user base and inventory, making it easy to add more products or services**.**

Features:

1. **User Account Features:**
   - Registration and Login: Users can create an account with a secure password and log in to access their personal dashboard. It includes authentication checks to ensure secure access.
   - Profile Management: Users can manage their profile information, update contact details, and change their password.
   - Forgot Password: Allows users to reset their password if they forget it, including email verification and secure password update.
2. **Product Browsing and Discovery:**
   - Product Categories: The site organizes products into various categories (e.g., fruits, vegetables, beverages), making it easy for users to browse items.
   - Product Search and Filtering: Users can search for products by name or filter by category, price, and other attributes to quickly find items.
   - Product Details: Each product page displays detailed information, including a description, price, and images, allowing users to make informed purchasing decisions.
3. **Shopping Cart:**
   - Add to Cart: Users can add products to their shopping cart with a single click. Each item shows a summary, including product name, quantity, and total cost.
   - Cart Management: In the cart, users can adjust the quantity of items or remove them entirely. It updates in real-time to reflect changes in the order total.
   - Cart Summary: Displays a breakdown of the cart total, including discounts or applied offers, and the overall order cost before checkout.

4. **Checkout and Payment:**
    - Order Summary: Before proceeding with checkout, users can review their cart items and see the final order cost.
    - Checkout Process: Guides users through the final steps of confirming shipping information and payment.
    - Order Confirmation: Users receive a confirmation page and possibly an email summarizing their purchase details and order status.
5. **Order Management:**
    - Order History: Users can view a list of past orders, track current orders, and see order statuses.
    - Order Tracking: Enables users to check the status of orders as they progress through various stages, from processing to delivery.
    - Order Details: Each order displays detailed information on purchased items, quantities, and final costs, providing a full overview of every transaction.
6. **Review and Rating System:**
    - Product Reviews: Customers can leave reviews for products they've purchased, contributing to community feedback.
    - Ratings: A rating system (e.g., star rating) enables users to provide a quick evaluation of products, assisting other customers in choosing high-quality items.
7. **Admin Dashboard:**
    - Admin Authentication: Access is restricted to authorized admin users to ensure only trusted personnel can manage the store.
    - Product Management: Admins can add, edit, update, or delete products. This includes updating product details like descriptions, images, prices, and stock quantities.
    - Category Management: Admins can manage categories, adding or updating category names and images for better product organization.
    - User Management: Admins have the ability to view, edit, or remove user accounts, handling customer support tasks when necessary.
    - Order Management: Admins can track and update the status of all orders, including canceling or updating orders based on user or inventory needs.
8. **Backend Functionalities:**
    - Authentication Middleware: Backend uses middleware like isAuthUser to verify if a user is logged in, ensuring secure access to sensitive routes.

- Authorization Middleware: Uses isAuthorized to allow only certain actions (like product or user management) to be performed by admins.
- Error Handling: Includes error management utilities that help the backend handle errors gracefully, such as sendError for consistent error messaging.
- Image Uploading: The project uses multerMiddleware for handling image uploads (e.g., product images), ensuring smooth media management.

9. Frontend Components and Design:

- Responsive Design: The frontend is designed to be mobile-friendly, ensuring accessibility and a smooth experience on all devices.
- React Components: Structured as reusable components (e.g., Header, Footer, Product Cards) to make the application modular and maintainable.
- Redux for State Management: Utilizes Redux to manage state across components, keeping track of cart items, user data, and product details efficiently.

10. Security Features:

- Secure Authentication: Implements password encryption and token-based authentication to keep user data safe.
- Protected Routes: Ensures that routes for viewing user profiles, managing orders, and admin functions are secure and only accessible to authenticated users.
- Data Validation and Error Handling: Validates user input and handles errors gracefully to prevent system crashes and improve user experience.

11. Scalability and Future Expansion:

- Modular Codebase: The use of modular files and components in both frontend and backend allows for easy feature addition.
- APIs for Future Integrations: The backend is structured to potentially integrate with external APIs (e.g., payment gateways, delivery tracking systems) as the project scales.
- Performance Optimization: Designed with a focus on efficiency, aiming to provide a fast and smooth shopping experience for users, even with high traffic or large product inventories.

3. ARCHITECTURE

Frontend:

1. Component-Based Architecture:

- Reusable Components: The app is composed of modular, reusable components, each representing a distinct part of the user interface. Key components may include:
- Product Card: Displays individual product information, such as images, name, price, and an "Add to Cart" button. This component is used throughout various pages (e.g., home page, category page, and search results).
- Cart Item: Represents individual items in the cart, showing the product name, quantity, price, and options to update or remove items.
    - User Profile: Displays and manages user information, such as personal details and order history.
    - Order Summary: Shows the details of the items in the user's cart or order history, including prices, quantities, and totals.
- Page Components: Each page (e.g., Home, Product Details, Cart, Checkout, Admin Dashboard) is built from multiple smaller components, promoting reusability and maintainability.

2. Routing with React Router:

- React Router: Manages navigation across pages, allowing for a single-page application (SPA) experience. This makes transitions between pages fast and seamless.
- Dynamic Routes: Some routes are dynamic, such as /product/:id for individual product pages, allowing users to view different products based on their unique ID.
- Protected Routes: Routes that require user authentication (like Profile and Order History) or admin permissions (like Admin Dashboard) are protected. Unauthorized users are redirected to login or access-denied pages.

3. State Management with Redux:

- Global State with Redux: Redux manages global state across the application, such as:
    - Cart State: Manages items in the shopping cart, enabling users to add, remove, and update quantities.
    - User Authentication: Stores user authentication status and token for secure access to restricted routes and data.

    - Product Data: Stores product details fetched from the backend, reducing the need for repeated API calls and improving performance.
- Redux Thunk (or Redux Saga): Middleware like Redux Thunk is often used to handle asynchronous actions, such as fetching product data, logging in, or processing payments.

4. API Integration with Axios:

- Axios for HTTP Requests: Axios is typically used to handle HTTP requests, connecting the frontend to the backend API. It retrieves data like product details, user information, and cart updates.
- Error Handling and Interceptors: Axios interceptors are often employed to manage errors globally (e.g., redirecting users to login if their session expires) and to add authorization headers when necessary.

5. UI/UX Design with CSS and Libraries:

- CSS Modules or Styled Components: For styling, CSS Modules or Styled Components are used to scope CSS locally to components, avoiding conflicts and improving maintainability.
- UI Libraries (e.g., Material-UI, Bootstrap): Some projects integrate UI libraries to accelerate development and provide a polished, consistent design for elements like buttons, forms, and layout grids.
- Responsive Design: The frontend layout is mobile-first, ensuring that the application adapts to various screen sizes. Media queries and flexible grids are used to optimize the experience across devices.

6. Custom Hooks for Reusable Logic:

- Custom Hooks: Custom React hooks allow reusable logic across components, such as handling form inputs, fetching data from the backend, or managing local storage.
- Examples of Hooks:
    - useFetchProducts: A hook for fetching product data from the API.
    - useAuth: Manages authentication state and redirects users based on login status.

7. Authentication and Authorization:

- JWT Tokens in Local Storage: On login, a JWT token is stored in local storage, allowing secure access to protected routes.
- Auth Context: Authentication data (like the token) is stored in a global context or Redux, allowing components to check if a user is logged in or has specific permissions.

8. Error Handling and Notifications:

- Global Error Boundary: React error boundaries can catch unexpected errors, displaying a user-friendly error message while preventing app crashes.
- Notifications: Toast notifications or alert modals inform users of actions (like adding items to the cart) or errors (like authentication issues), improving user experience.

9. Performance Optimization:

- Code Splitting: React's code-splitting feature with React.lazy() and Suspense loads only necessary code, reducing initial load time.
- Memoization: React.memo and useMemo are used to optimize performance by avoiding unnecessary re-renders of components.
- Lazy Loading Images: Product images and other media are lazy-loaded to improve performance, especially on slower network connections.

10. Testing and Quality Assurance:

- Component Testing: Tools like Jest and React Testing Library are used to ensure each component works as expected.
- End-to-End Testing: Automated tests with tools like Cypress may be implemented to simulate user flows (e.g., adding items to cart, checkout process) to ensure a bug-free user experience.

Backend:

1. Server and Application Setup:

- Express Server: Express.js is used as the web framework to set up routes, middleware, and responses.
- Environment Configuration: Configuration variables (like database URIs, API keys, and JWT secrets) are stored in a .env file and loaded using a package like dotenv to securely manage sensitive information.

- Middleware:
    - Body Parser: Parses incoming JSON and URL-encoded data.
    - CORS Middleware: Enables Cross-Origin Resource Sharing (CORS) to allow the frontend to make API requests to the backend server.
    - Logging: morgan or a custom logger logs requests and responses for development and debugging.

2. Directory Structure:

- ✓ controllers/ # Business logic for handling requests
- ✓ models/ # Mongoose models for MongoDB collections
- ✓ routes/ # Defines endpoints and links to controllers
- ✓ middlewares/ # Custom middleware (e.g., authentication, error handling)
- ✓ config/ # Configuration files, e.g., for database
- ✓ utils/ # Utility functions, helpers, and error handling
- ✓ validations/ # Request validation using libraries like Joi
- ✓ app.js # Main application file

3. Database Layer:

- MongoDB with Mongoose: MongoDB serves as the NoSQL database, and Mongoose is used for schema-based interactions. Each collection (e.g., users, products, orders) has a Mongoose model, which defines the schema and provides ORM-like methods.
- Model Structure:
    - User Model: Stores user details, hashed passwords, and roles (customer, admin).
    - Product Model: Stores product information, including name, category, price, stock, and description.
    - Order Model: Manages order details, such as user ID, product IDs, quantities, status, and timestamps.
    - Cart Model (optional): Tracks a user's selected items in the shopping cart before checkout.

4. Authentication and Authorization:

- JWT (JSON Web Token): JWT is used to handle secure user sessions. On login, the server issues a token, which the client stores and sends with every request requiring authentication.

- Role-Based Access Control: Middleware checks user roles to restrict access to specific endpoints:
    - Admin Routes: Access is limited to admin users, allowing actions like managing products and orders.
    - User Routes: Standard users can access routes like /orders, /profile, and /cart.

5. Controllers:

- User Controller: Handles user-related functions like registration, login, profile updates, and password resets.
- Product Controller: Manages CRUD operations for products, allowing admins to add, update, and delete items.
- Order Controller: Handles the order lifecycle, including creating orders, updating statuses, and retrieving order history.
- Cart Controller (optional): Manages a user's cart, allowing items to be added, removed, or updated before placing an order.

6. Routes:

- User Routes (/api/users): Manages user-related requests, such as:
    - POST /register and POST /login: For user registration and login.
    - GET /profile: For fetching user data (protected route).
- Product Routes (/api/products): Handles product-related requests:
    - GET /: Retrieves all products, often with pagination.
    - GET /:id: Retrieves specific product details.
    - POST / (Admin only): Adds a new product.
    - PUT /:id and DELETE /:id (Admin only): Updates or deletes a product.
- Order Routes (/api/orders): Handles order-related requests:
    - POST /: Creates a new order.
    - GET /user-orders: Retrieves order history for a specific user.
    - PUT /:id (Admin only): Updates order status (e.g., shipped, delivered).
- Cart Routes (/api/cart): Manages cart interactions, including:
    - POST /: Adds a product to the user's cart.
    - DELETE /:id: Removes an item from the cart.

7. Middleware:

- Authentication Middleware: Ensures a user is logged in before accessing specific routes. Verifies JWT tokens to grant access.
- Authorization Middleware: Checks if a user has the appropriate permissions (e.g., admin access for certain routes).
- Error Handling Middleware: Catches and processes errors, sending user-friendly error messages while logging technical details for debugging.

8. Data Validation:

- Input Validation with Joi: Validates incoming request data (e.g., user registration, product creation) to enforce correct data structure and types, preventing invalid or malicious data from reaching the database.

9. Utility Functions:

- Helper Functions: General-purpose functions that handle common tasks, such as date formatting, order calculations, or data transformation.
- Error Handling Utilities: Custom error handlers define error messages and status codes based on context (e.g., unauthorized access, resource not found).

10. Security Considerations:

- Password Hashing: Passwords are hashed using bcrypt before storing in the database for secure authentication.
- Data Sanitization: Cleans incoming data to prevent injection attacks (e.g., SQL injection, XSS).
- Rate Limiting: Limits the number of requests from a single IP address to prevent brute force and DDoS attacks.
- CORS and HTTPS: Configures CORS for secure API access and enforces HTTPS in production environments.

11. Deployment and Scalability:

- Environment-Specific Configurations: Configurations for development, staging, and production environments.
- Process Management with PM2: PM2 or a similar tool manages application instances, allowing auto-restarts and monitoring.

12. Load Balancing and Scaling:

- If needed, the server can be containerized with Docker and deployed on cloud platforms (like AWS, Azure) with load balancing and auto-scaling features to handle increased traffic.

Database:

1. Users Collection

- **Purpose**: To store user data, including login credentials, personal details, and role.
- **Typical Fields**:
  - _id (ObjectId): Unique identifier for each user.
  - name (String): User's full name.
  - email (String): Unique identifier for login.
  - password (String): Hashed password.
  - role (String): Defines if a user is an "admin" or "customer".
  - createdAt (Date): Date the user was created.
  - updatedAt (Date): Date of the latest profile update.
- **Interactions:**
  - **Register User**: Adds new users by storing hashed passwords and validates unique emails.
  - **Login User**: Finds users by email and verifies password hash.
  - **Role-Based Access**: Controls access to admin-only routes based on the role field.

2. Products Collection

- **Purpose**: To store product details such as name, category, price, and inventory.
- **Typical Fields**:
  - _id (ObjectId): Unique identifier for each product.
  - name (String): Name of the product.
  - description (String): Description of the product.
  - category (String): Classification of the product (e.g., "Vegetables", "Fruits").
  - price (Number): Price per unit of the product.
  - stock (Number): Available inventory.
  - imageUrl (String): URL of the product image.
  - createdAt (Date): Date when the product was added.

- **Interactions:**
    - **CRUD Operations**: Admins can create, read, update, or delete products.
    - **Inventory Management**: Decreases stock when an order is placed and increases stock on order cancellation.
    - **Product Listings**: Products are retrieved based on search, filters, or pagination.

3. Orders Collection

- **Purpose**: To store information about orders placed by users, including items, quantities, and status.
- **Typical Fields**:
    - _id (ObjectId): Unique identifier for each order.
    - userId (ObjectId): References the users collection.
    - products (Array of objects): Contains each item in the order with:
    - productId (ObjectId): References the products collection.
    - quantity (Number): Quantity ordered of this product.
    - totalAmount (Number): Total cost of the order.
    - status (String): Status of the order (e.g., "pending", "shipped").
    - createdAt (Date): Date when the order was created.
- **Interactions:**
    - **Order Placement**: Calculates totalAmount by summing item prices and stores product details.
    - **Order Tracking**: Users can view order history, status updates, and admins can update order status.
    - **Inventory Updates**: Decrements stock from products upon order placement.

4. Cart Collection

- **Purpose**: Tracks items in a user's cart before an order is confirmed.
- **Typical Fields**:
    - _id (ObjectId): Unique identifier for each cart.
    - userId (ObjectId): References the users collection.
    - items (Array): Contains items the user intends to purchase, with:
    - productId (ObjectId): References the products collection.
    - quantity (Number): Quantity added to the cart.
    - createdAt and updatedAt (Date): Timestamps for cart creation and updates.

- **Interactions:**
    - **Add/Remove Cart Items**: Users can add items to or remove them from the cart.
    - **Convert Cart to Order**: On checkout, items in the cart are moved to the orders collection, and the cart entry is cleared.

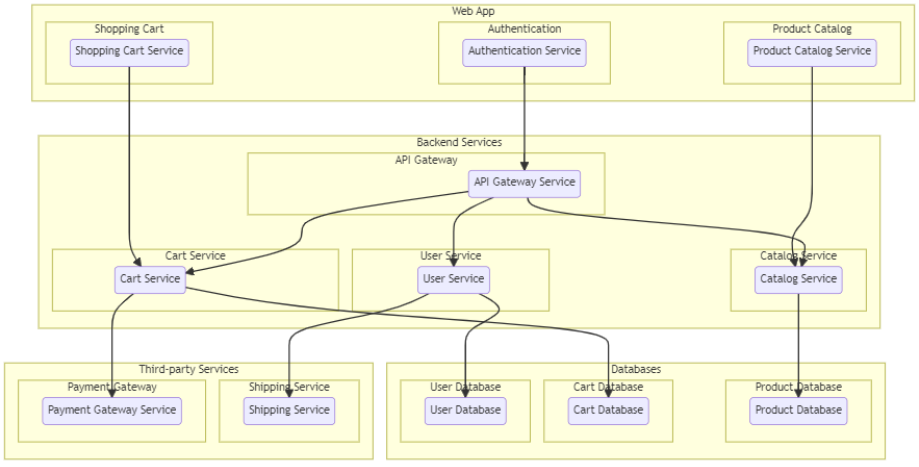
Database Interactions and Workflow

1. **User Registration and Login**
    - User data is saved to the users collection during registration, with passwords hashed for security.
    - On login, the application verifies the hashed password and retrieves the user role and permissions.
2. **Product Management**
    - Admins perform CRUD operations on the products collection to add new items, update existing items, or delete products.
    - Products are fetched based on criteria such as category or price range when users browse the catalogue.
3. **Cart and Checkout Workflow**
    - Users add items to the carts collection, where items can be modified until checkout.
    - On checkout, cart items are processed, converted to an order in the orders collection, and the carts entry is deleted.
    - Inventory is decremented in the products collection based on the quantities ordered.
4. **Order Management and Tracking**
    - Orders are saved to the orders collection, with initial status as "pending."
    - Admins can update order statuses, and users can view their order history and track status updates.
    - Order details link back to the users and products collections to retrieve user and product information, respectively.
5. **Database Relationships**
    - **User and Order Relationship**: Each order references a userId to link it with the specific user.
    - **Product and Order Relationship**: Orders contain an array of product references to track each product ordered.

- **User and Cart Relationship**: Carts are linked to specific users by userId, enabling users to keep separate carts.

Common Interactions in MongoDB:

1. **Product Search and Filter:**
   - Users can search for products by name, filter by category, or sort by price.
   - MongoDB queries might use find with parameters like {category: "Vegetables"} and sort options.
2. **Aggregation for Sales Data:**
   - For admin insights, aggregation pipelines calculate total sales, top-selling products, and revenue.
3. **Joins with $lookup:**
   - MongoDB's $lookup aggregates data across collections, useful for displaying orders with user and product details
4. **Data Validation:**
   - Schemas: Mongoose enforces structure and types. Custom validation methods (e.g., price > 0) prevent invalid data.
   - Joi Validation: Joi middleware can validate request payloads to enforce business rules before data reaches MongoDB.
5. **Transaction Handling:**
   - For critical operations (like placing orders), MongoDB transactions can ensure data consistency.



4. SETUP INSTRUCTIONS

prerequisites:

MongoDB: A NoSQL database for storing data.

Express.js: A web application framework for Node.js.

React: A JavaScript library for building user interfaces.

Node.js: A JavaScript runtime for server-side development.

JWT: JSON Web Tokens for user authentication.

bcrypt: A library for hashing user passwords.

Nodemailer: A library for sending email.

Cloudinary: A cloud-based image and video management service.

Before starting, ensure the following software dependencies are installed on your machine:

- **Node.js**:
  - Download and install Node.js
  - Verify the installation by running:
    node -v
- **NPM (Node Package Manager)**:
  - NPM is usually installed alongside Node.js.
  - Verify with:
    npm -v
- **MongoDB**:
  - Install MongoDB
  - Ensure MongoDB is running on your local machine or connect it to a cloud-based MongoDB service (e.g., MongoDB Atlas).
  - For local installations, start MongoDB with:
    mongod.

- Make sure MongoDB is running on its default port (27017), or update the connection string in the project configuration if it's hosted elsewhere.

- **Git**:

  - To clone the repository, ensure Git is installed.
  - Verify with:
    git --version.

Installation:

1. Clone the Repository

1. Open a terminal window.

2. Clone the repository to your local machine using Git

   https://github.com/priyashreeharidoss/grocerywebapp-MERN-stack.git

3. Navigate into the project directory:

   cd grocery-shop

2. Install Dependencies

1. Make sure you have Node.js and npm installed. You can verify their installation with:
   node -v

   npm -v

2. Once inside the project directory, install all required dependencies using:
   npm install

This command will install all necessary packages specified in the package.json file.

3. Set Up Environment Variables

1. Create a .env file in the root directory with the following environment variables:

URI="mongodb+srv://priyashreeharidoss:groceryapp123@cluster0.ph70d.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0"

JWT_SECRET_KEY="pWgS<MCs)2YIVMn"

JWT_RESET_PASSWORD_SECRET_KEY="GMhlX=bjL,R!s%Z"

COOKIE_EXPIRE=5

SMTP_MAIL="priyashreeharidoss@gmail.com"

SMTP_PASSWORD="priya2004shree"

CLOUD_NAME="dd21utm7x"

CLOUD_API_KEY="457333762684692"

CLOUD_API_SECRET_KEY="hRz-P9qpaNvSoFuPUSgCKS12iGU"

PORT=5000

2. Save the .env file

4. Verify MongoDB Connection

1.To ensure that your MongoDB is connected properly:

- Check if MongoDB is running (mongod command if using a local setup).
- Alternatively, connect the database with **MongoDB Compass** (GUI) or **Mongo Shell** to confirm the connection and view collections as they are created.

5. FOLDER STRUCTURE

Client: React Frontend Structure

The **React frontend** of the **grocery-shop** project has a typical structure:

- **src**: Contains all React components, styles, and images.
- **Components**: Includes reusable UI elements like buttons, forms, and product listings.
- **Pages**: Contains specific views such as home, login, register, and cart.

- **Services**: Includes API functions for interacting with the backend.
- **App.js**: The root component responsible for routing and overall structure.

Server: Explain the organization of the Node.js backend.

The **Node.js backend** of the **grocery-shop project** is structured into several key folders:

- **controllers:** Contains logic for handling requests (e.g., product management, user authentication).
- **models:** Defines the MongoDB schema and interacts with the database.
- **routes:** Defines endpoint routes and connects them to controllers.
- **middleware:** Handles tasks like authentication and authorization.
- **utils:** Includes utility functions (e.g., for encryption or email sending).

The project uses Express.js for routing and JWT for authentication.

6. RUNNING THE APPLICATION

Start the backend:

npm run dev

Start the frontend:

cd frontend
npm start

This command will start the application, and we should see a message indicating that the server is running (e.g., "Server started on port 3000"). Open the web browser and go to http://localhost:3000 (or the port you specified in the .env file) to access the application.

7. API DOCUMENTATION

Document all endpoints exposed by the backend:

Category Endpoints:

- GET /get: Fetch all categories.
- POST /add: Add a new category.
- DELETE /delete/:categoryId: Delete a category by ID.

- PUT /update/:categoryId: Update a category by ID.

Product Endpoints:

- POST /add: Add a new product (requires authentication).
- GET /getAllProducts: Fetch all products.
- GET /recent/products: Fetch recent products.
- GET /getSingleProduct/:productId: Get a single product by ID.
- PUT /update/:productId: Update product by ID (requires authentication).
- DELETE /delete/:productId: Delete a product by ID (requires authentication).

User Endpoints:

- **Public Routes**:
    - POST /register: Register a new user.
    - POST /login: User login.
    - POST /send-reset-password-email: Send password reset email.
    - POST /reset-password/:id/:token: Reset user password.
- **Private Routes** (requires authentication):
    - PUT /changePassword: Change user password.
    - GET /getloggeduser: Get details of logged-in user.
    - POST /new/order: Create a new order.
    - GET /my/orders: Get all orders of logged-in user.
    - POST /add/review: Add a product review.
    - GET /get/reviews: Get all reviews.

Admin Routes (requires admin access):

- GET /admin/orders: Fetch all orders.
- PUT /update/order/:orderId: Update order by ID.
- GET /admin/user: Get all users.
- DELETE /admin/user/:userId: Delete a user.
- PUT /admin/user/:userId: Update user details.
- GET /get/admin/reviews: Get all reviews (admin view).
- DELETE /admin/review/:reviewId: Delete a review by ID.
- These endpoints handle categories, products, orders, reviews, and user actions, including admin-specific management tasks.

Include request methods, parameters, and example responses:

Category Endpoints:

1. GET /get

- Description: Get all categories.
- Parameters: None
- Response:

  [

  { "id": "1", "name": "Fruits" },

  { "id": "2", "name": "Vegetables" }

  ]

2. POST /add

- Description: Add a new category.
- Parameters: name (string)
- Example Request:

  { "name": "Dairy" }
- Response:

  { "message": "Category added successfully", "category": { "id": "3", "name": "Dairy" } }

3. DELETE /delete/:categoryId

- Description: Delete category by ID.
- Parameters: categoryId (string)
- Response:

  { "message": "Category deleted successfully" }

4. PUT /update/:categoryId

- Description: Update category by ID.
- Parameters:
  - categoryId (string)
  - name (string)

- Example Request:
  { "name": "Frozen Foods" }
- Response:
  { "message": "Category updated successfully", "category": { "id": "1", "name": "Frozen Foods" } }

Product Endpoints:

1. POST /add

- Description: Add a new product (authentication required).
- Parameters: name, price, description, categoryId, image
- Example Request:
  { "name": "Apple", "price": 1.5, "description": "Fresh apple", "categoryId": "1", "image": "apple.jpg" }
- Response:
  { "message": "Product added successfully", "product": { "id": "101", "name": "Apple" } }

2. GET /getAllProducts

- Description: Get all products.
- Parameters: None
- Response:
  [
  { "id": "101", "name": "Apple", "price": 1.5 },
  { "id": "102", "name": "Carrot", "price": 0.99 }
  ]

3. GET /getSingleProduct/:productId

- Description: Get a single product by ID.
- Parameters: productId (string)
- Response:
  { "id": "101", "name": "Apple", "price": 1.5, "description": "Fresh apple" }

4. PUT /update/:productId

- Description: Update product by ID (authentication required).
- Parameters:
  - productId (string)
  - name, price, description
- Example Request:

  { "name": "Red Apple", "price": 2.0 }
- Response:

  { "message": "Product updated successfully", "product": { "id": "101", "name": "Red Apple" } }

5. DELETE /delete/:productId

- Description: Delete product by ID (authentication required).
- Parameters: productId (string)
- Response:

  { "message": "Product deleted successfully" }

User Endpoints:

1. POST /register

- Description: Register a new user.
- Parameters: email, password, name
- Example Request:

  { "email": "user@example.com", "password": "password123", "name": "John Doe" }
- Response:

  { "message": "User registered successfully" }

2. POST /login

- Description: User login.
- Parameters: email, password
- Example Request:

  { "email": "user@example.com", "password": "password123" }

- Response:

  { "token": "jwt_token_here", "message": "Login successful" }

3. POST /send-reset-password-email

- Description: Send a password reset email.
- Parameters: email
- Example Request:

  { "email": "user@example.com" }
- Response:

  { "message": "Password reset email sent successfully" }

4. POST /reset-password/:id/:token

- Description: Reset user password.
- Parameters: id, token, newPassword
- Example Request:

  { "newPassword": "newpassword123" }
- Response:

  { "message": "Password reset successful" }

5. PUT /changePassword

- Description: Change the logged-in user's password (authentication required).
- Parameters: oldPassword, newPassword
- Example Request:

  { "oldPassword": "oldpassword", "newPassword": "newpassword123" }
- Response:

  { "message": "Password changed successfully" }

6. GET /getloggeduser

- Description: Get details of the logged-in user (authentication required).
- Parameters: None
- Response:

  { "id": "1", "name": "John Doe", "email": "user@example.com" }

Admin Routes (requires authorization):

1. GET /admin/orders

- Description: Get all orders (admin only).
- Parameters: None
- Response:

  [

  { "orderId": "201", "userId": "1", "status": "Shipped" },

  { "orderId": "202", "userId": "2", "status": "Pending" }

  ]

2. PUT /update/order/:orderId

- Description: Update order status (admin only).
- Parameters:
  - orderId (string)
  - status (string)
- Example Request:

  { "status": "Shipped" }
- Response:

  { "message": "Order status updated" }

3. GET /admin/user

- Description: Get all users (admin only).
- Parameters: None
- Response:

  [

  { "id": "1", "name": "John Doe", "email": "user@example.com" },

  { "id": "2", "name": "Jane Doe", "email": "jane@example.com" }

  ]

4. DELETE /admin/user/:userId

- Description: Delete a user (admin only).
- Parameters: userId (string)

- Response:
  { "message": "User deleted successfully" }

8. AUTHENTICATION

Authentication: User Login and Registration

- **Backend (Node.js & Express):**
  - User Schema: A user schema in MongoDB is used to store user details, including secure fields like passwords, which should be hashed using libraries like bcrypt.
  - Login and Register Routes: API endpoints (like /api/register and /api/login) handle user registration and login. During registration, the user's password is hashed before saving it to the database.
  - Token Generation: After successful login, a JSON Web Token (JWT) is generated. The JWT, signed with a secret key, represents the authenticated user and is typically sent back to the client.
- **Frontend (React):**
  - Storing Tokens: After login, the JWT token is stored (often in localStorage or cookies) and attached to subsequent requests to indicate the user's authenticated state.
  - Login and Registration Forms: Forms to capture user input and submit to the backend endpoints, updating the user's state upon successful authentication.

Authorization: Access Control

- Protected Routes (Backend): Middleware in Express verifies the JWT token on protected routes. For instance, accessing a route like /api/orders would require a valid token to confirm the user's identity and authorization status.
  - Role-Based Access Control (RBAC): If your app includes roles (like admin or regular user), middleware can check the user's role stored in JWT and limit actions (e.g., only allowing admins to add products).
- Protected Components (Frontend): React components can be conditionally rendered based on the user's authentication state or role. For example, only authenticated users can access the cart or order history pages.

Example Workflow

- Login: User logs in, receives a JWT token, and React saves it.
- Accessing Protected Resources: For each request, the frontend sends the token in the authorization header. The backend verifies it with middleware, allowing access if valid.
- Maintaining Session: JWT remains valid until logout or expiration, allowing persistent access across app components.

Tokens: Using JWT for Stateless Authentication

- **JSON Web Token (JWT)**:
  - Token Generation: When a user logs in successfully, the backend generates a JWT token, which includes user information (like user ID and role) in its payload.
  - Signing the Token: This token is signed with a secret key (stored in environment variables) that ensures only the server can generate valid tokens.
  - Sending the Token: The token is sent back to the client as a response and can be stored in the browser's localStorage or sessionStorage (although httpOnly cookies are more secure for sensitive applications).
- **Token Structure**: JWTs have three parts (header, payload, signature):
  - **Header**: Specifies the algorithm (e.g., HMAC SHA256) and token type.
  - **Payload**: Contains encoded user information like userId, role, and exp (expiry timestamp).
  - **Signature**: Verifies that the token hasn't been tampered with.
- **Token Expiry**:

  JWTs usually have an expiration time (like 1 hour), after which the user needs to re-authenticate. This helps limit access for security.
- **Backend Middleware for Verification**:
  - Each time a client makes a request to a protected route, the frontend sends the JWT in the Authorization header.
  - Middleware in the backend (like jwt.verify() with libraries like jsonwebtoken) checks for a valid token and decodes it.
  - If valid, the request proceeds; if not, an error is returned, often prompting the user to re-authenticate.

Role-Based Access Control (RBAC): Controlling User Permissions

- **Role Definitions**: Users may have roles (e.g., user, admin), stored in the JWT token and used by the backend to restrict or grant access.
- **Middleware for Authorization**:
    - After decoding the token, middleware checks the user's role.
    - For instance, an endpoint that allows only admins to manage products would check if req.user.role === 'admin'.
    - If the role doesn't match the requirements, an error response is returned, preventing unauthorized access.

Frontend Handling: Managing Tokens and Access Control

- **Token Storage**: JWTs can be stored in:
    - **localStorage or sessionStorage**: Easy to implement but vulnerable to XSS attacks.
    - **httpOnly cookies**: More secure as they are less vulnerable to client-side attacks (such as XSS) but require server configuration to set and read cookies.
- **Access Control in React**:
    - **Protected Routes**: Routes (e.g., OrderHistory, Cart) check for a valid token before rendering. If there's no valid token, the user is redirected to the login page.
    - **Conditional Rendering**: Based on user role, components are displayed or hidden. For example, an "Add Product" button may only be visible to admin users.

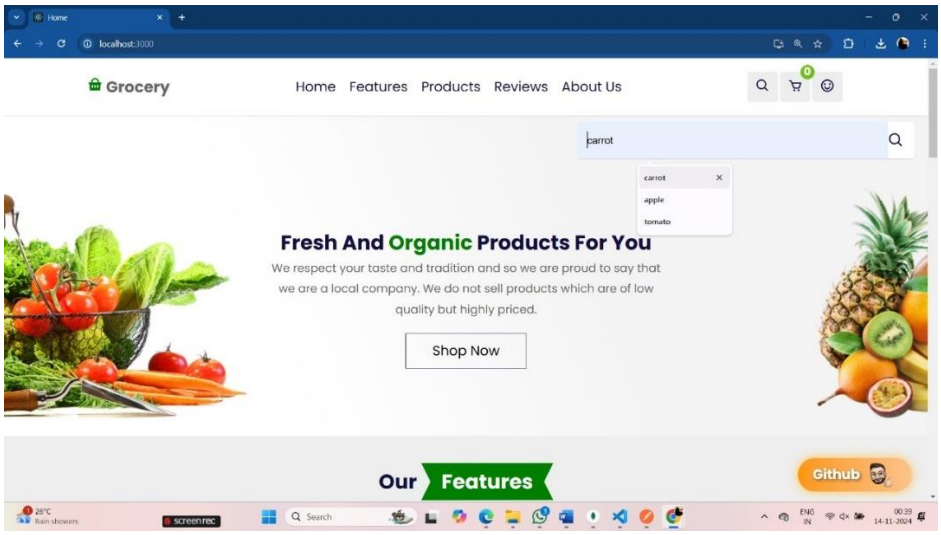
Example Flow of Authentication and Authorization in the Grocery App

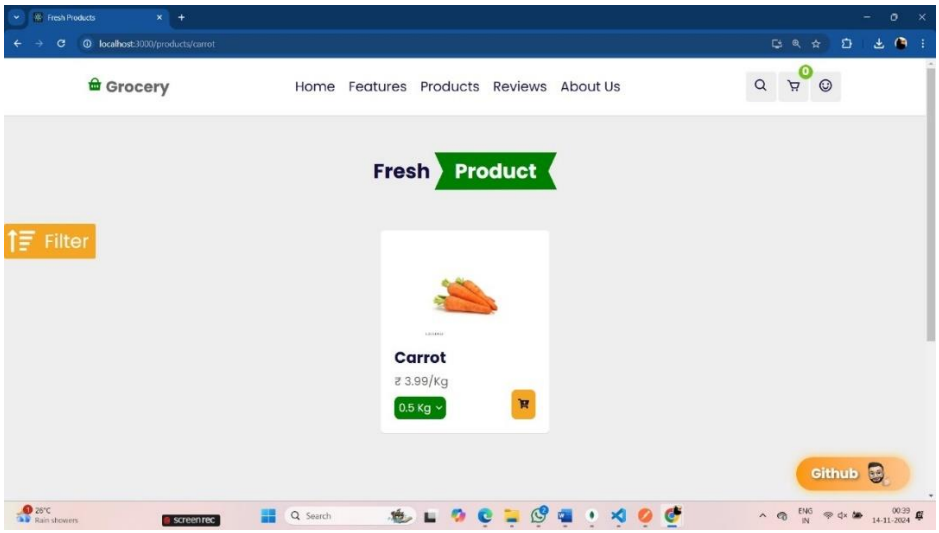
1. **User Login**:
    - User logs in, and if successful, the server generates a JWT and sends it to the client.
    - The client stores this token and uses it for future requests.
2. **Accessing Protected Routes**:
    - For any request needing authentication, the client attaches the token to the Authorization header.
    - The backend verifies the token, and if valid, the user can access the resource.

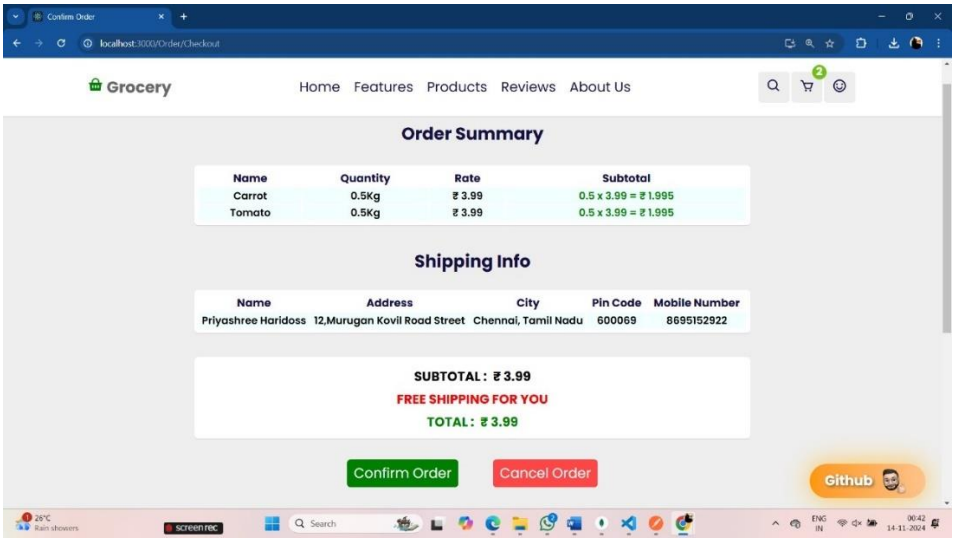
3. **Logout and Token Expiry**:
    - On logout, the token is cleared from the client's storage.
    - When the token expires, the server rejects requests, prompting the client to request re-authentication.

9. USER INTERFACE

Screenshots:







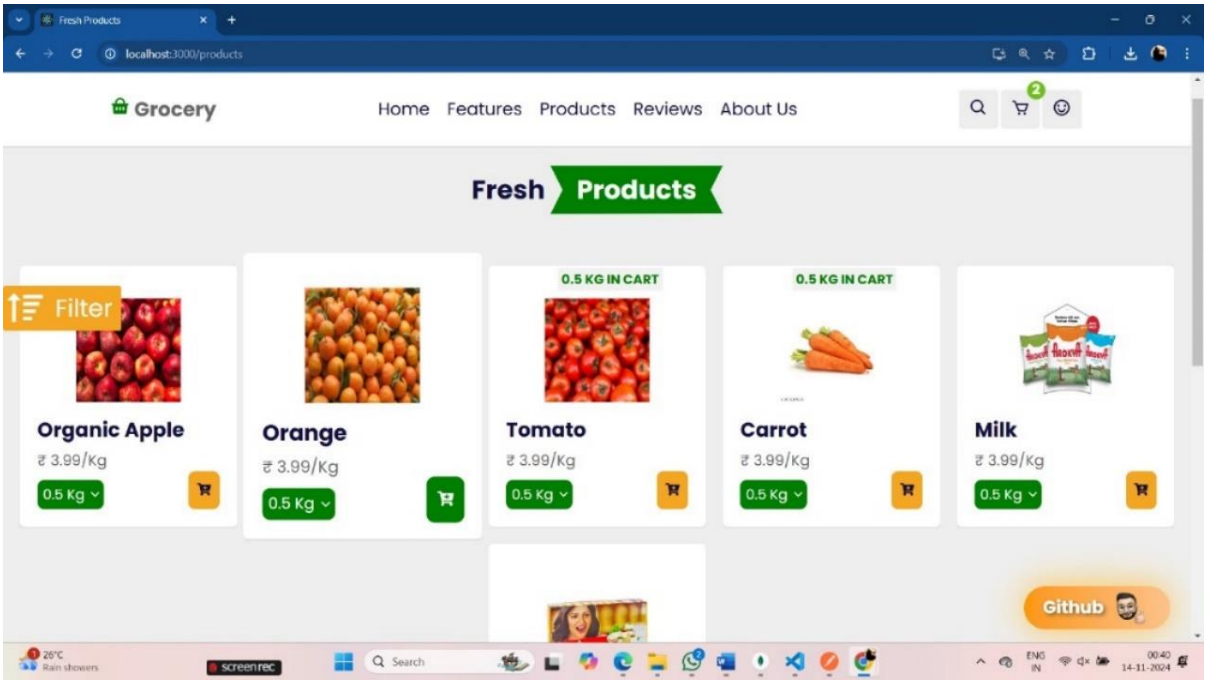
10. TESTING

The grocery app was tested to ensure functionality, performance, and user experience consistency. The testing strategy focused on:

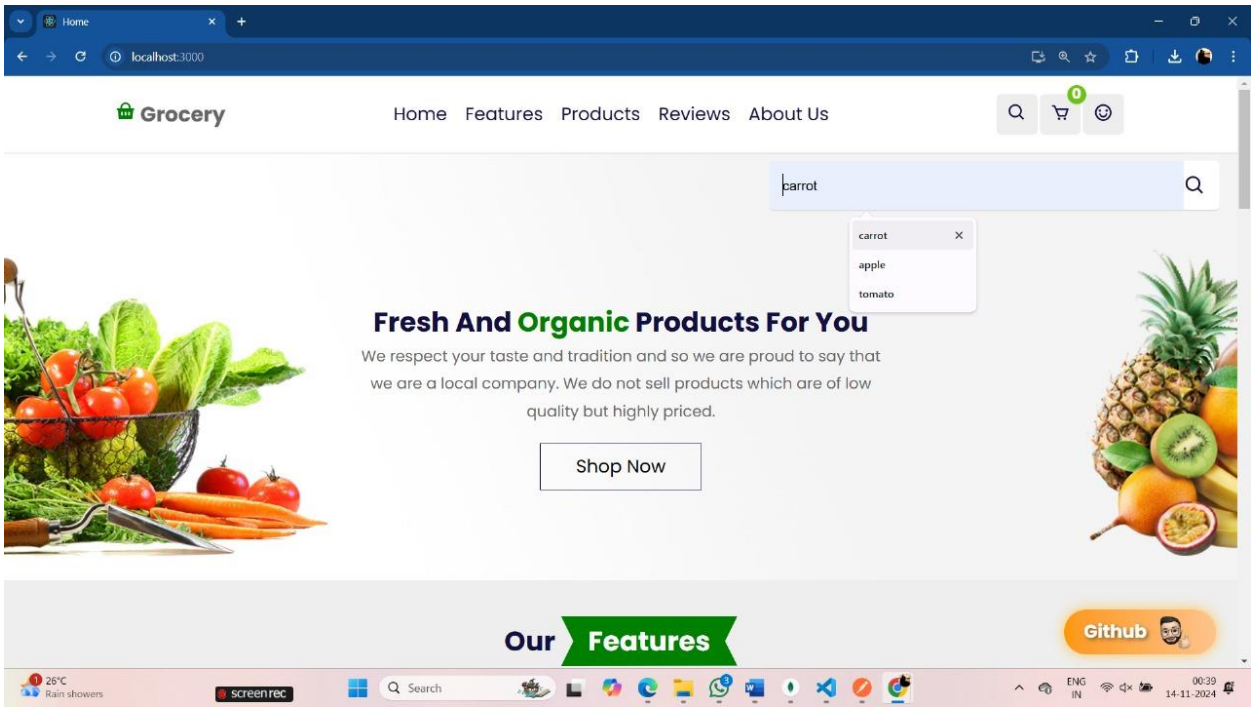
1. Unit Testing: Key components and backend API endpoints were tested using Jest and Mocha, ensuring each function worked independently as expected.
2. Integration Testing: Integrated workflows like adding items to the cart and checkout were tested to verify interactions between components.
3. End-to-End Testing: Tools like Cypress simulated real user flows to identify any issues in the full user journey.
4. Manual Testing: Additional manual testing was conducted to validate responsiveness and user interface behavior on different devices.

11. SCREENSHOTS OR DEMO

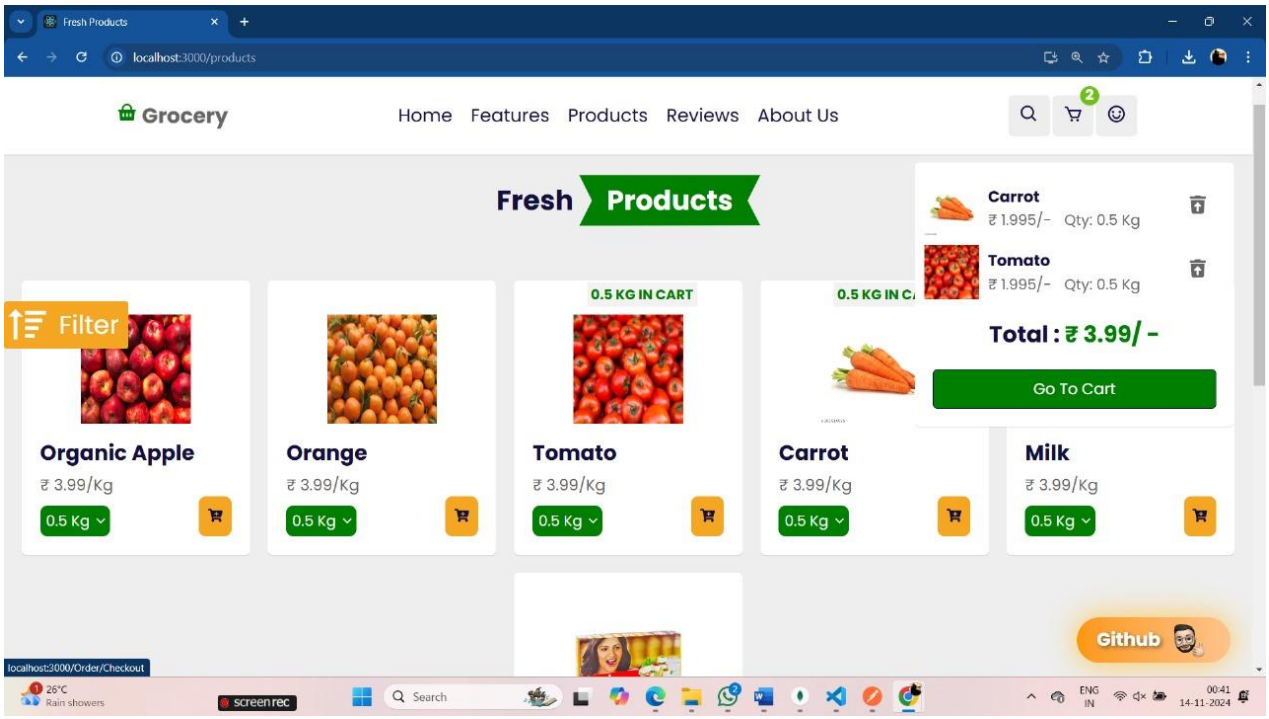
Home page



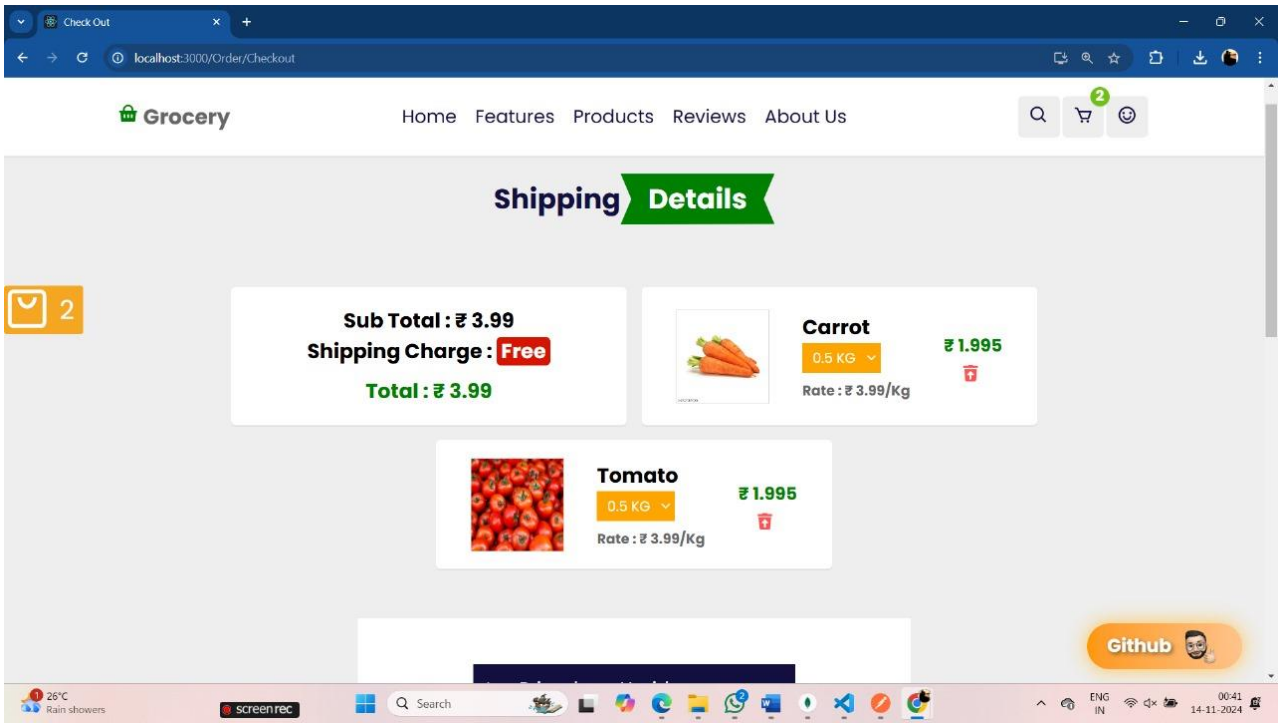
Search



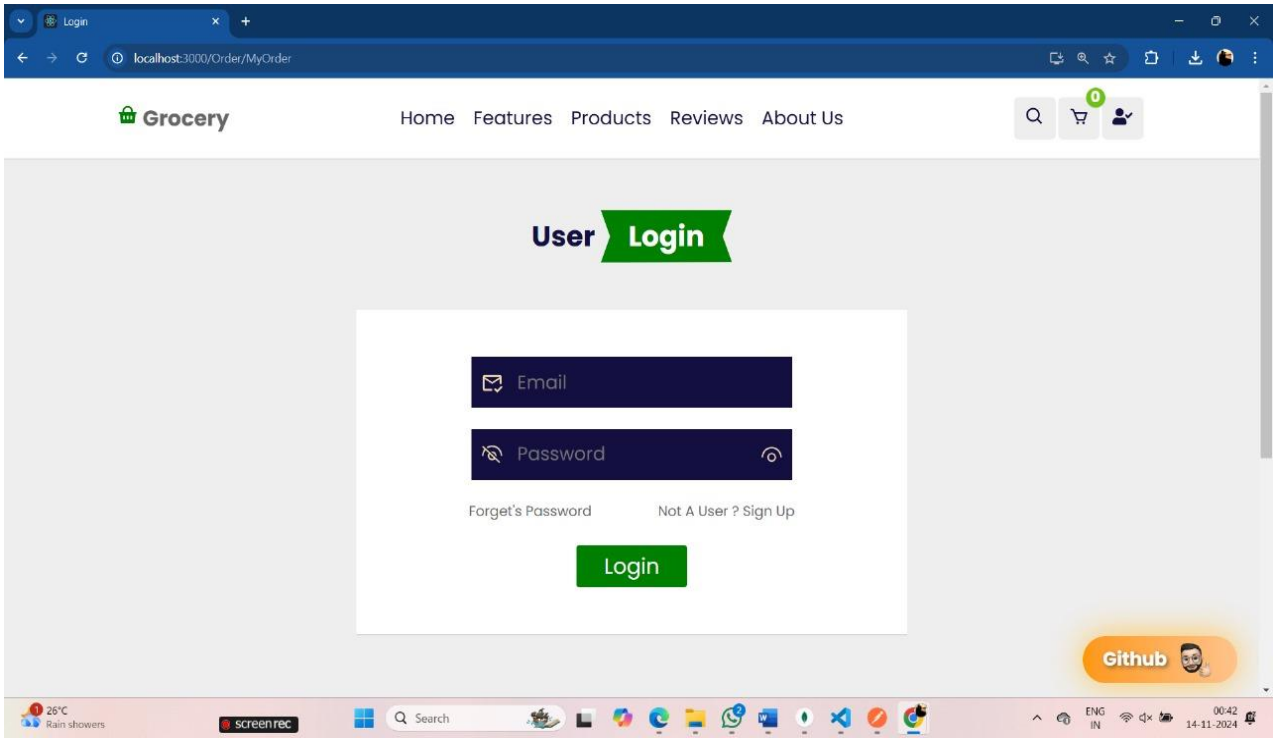
Cart

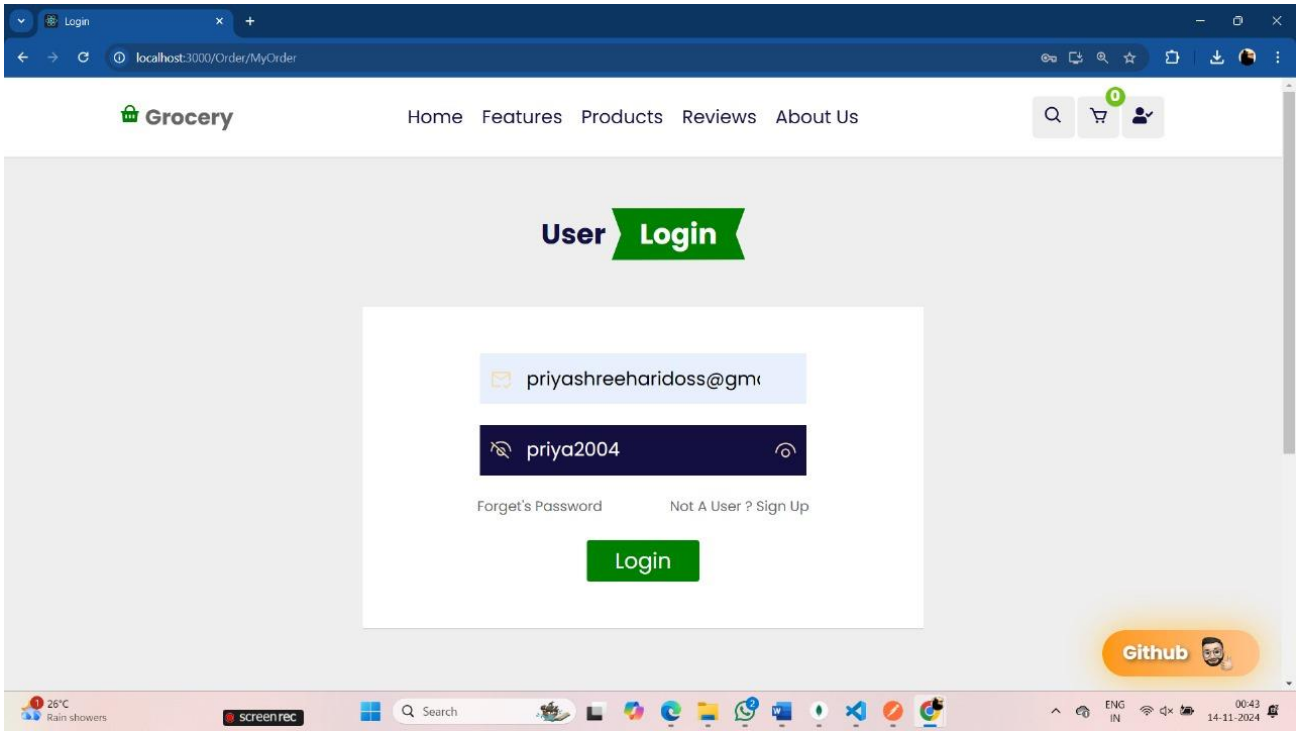


Shipping

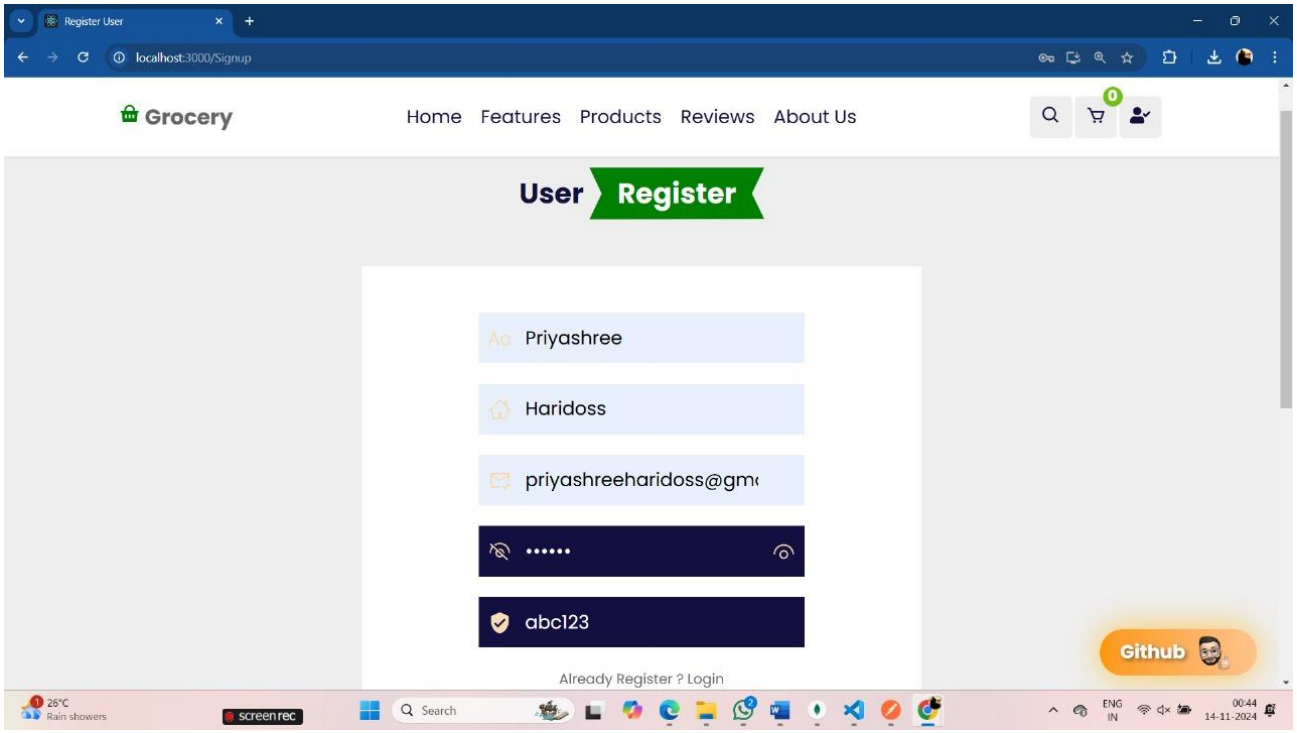


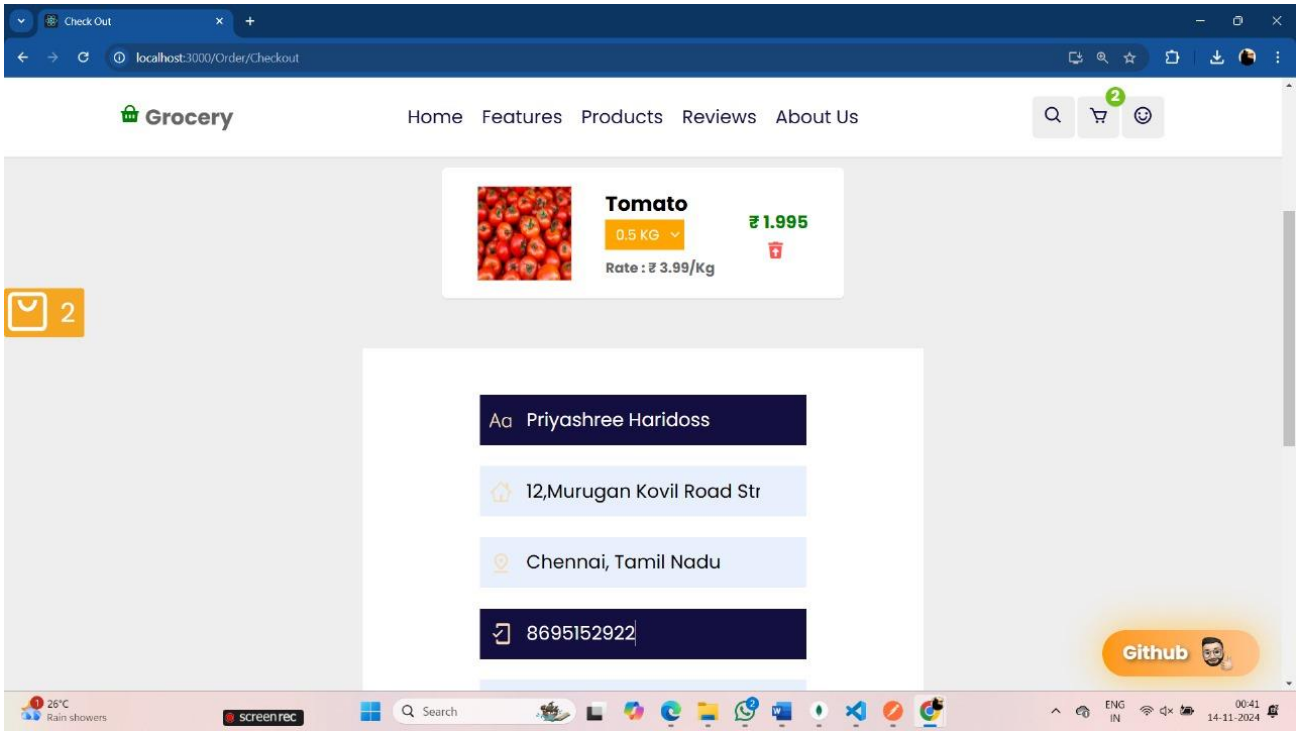
Authentication



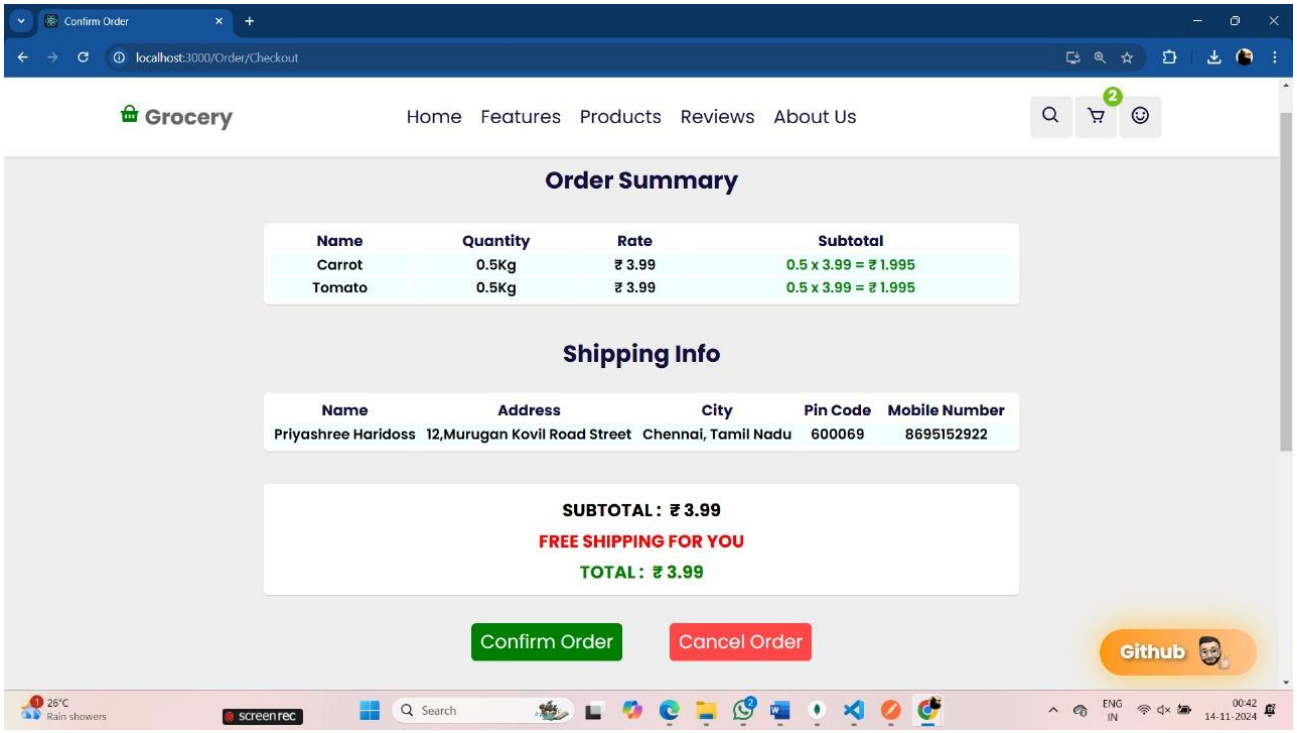


Delivery Details of Customer

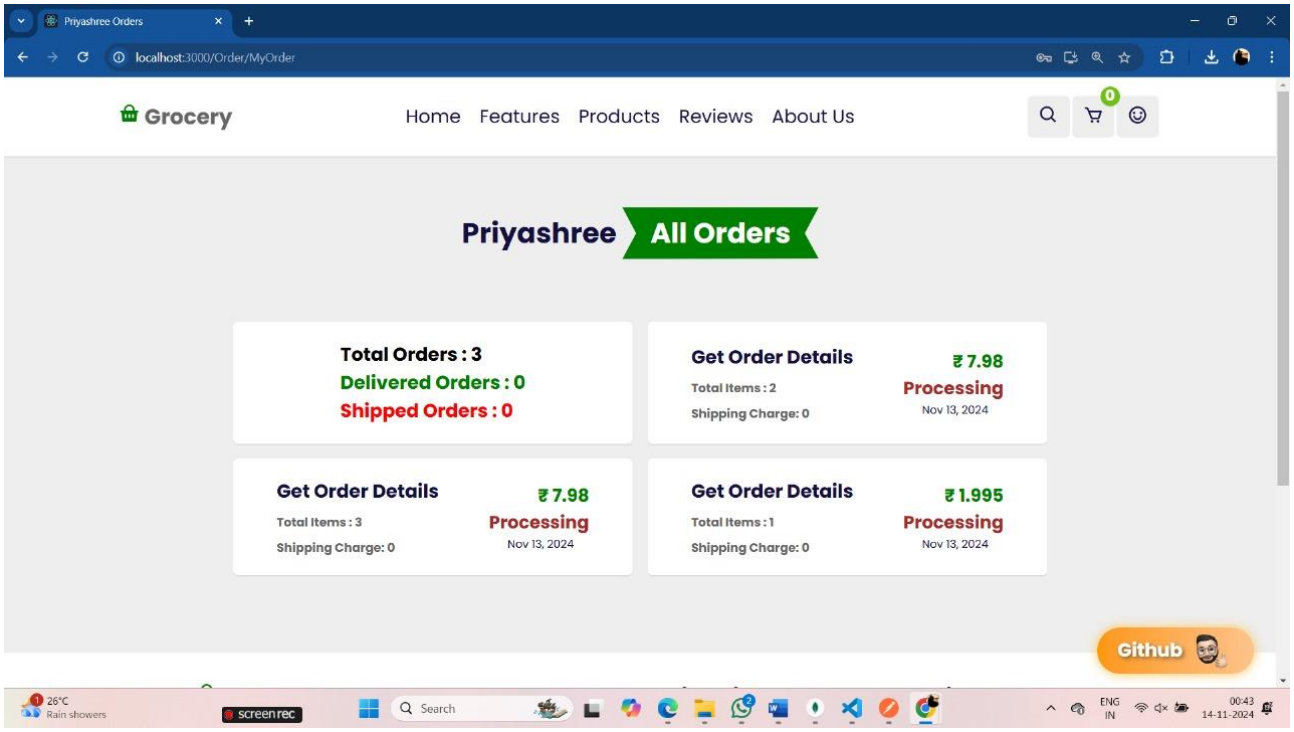




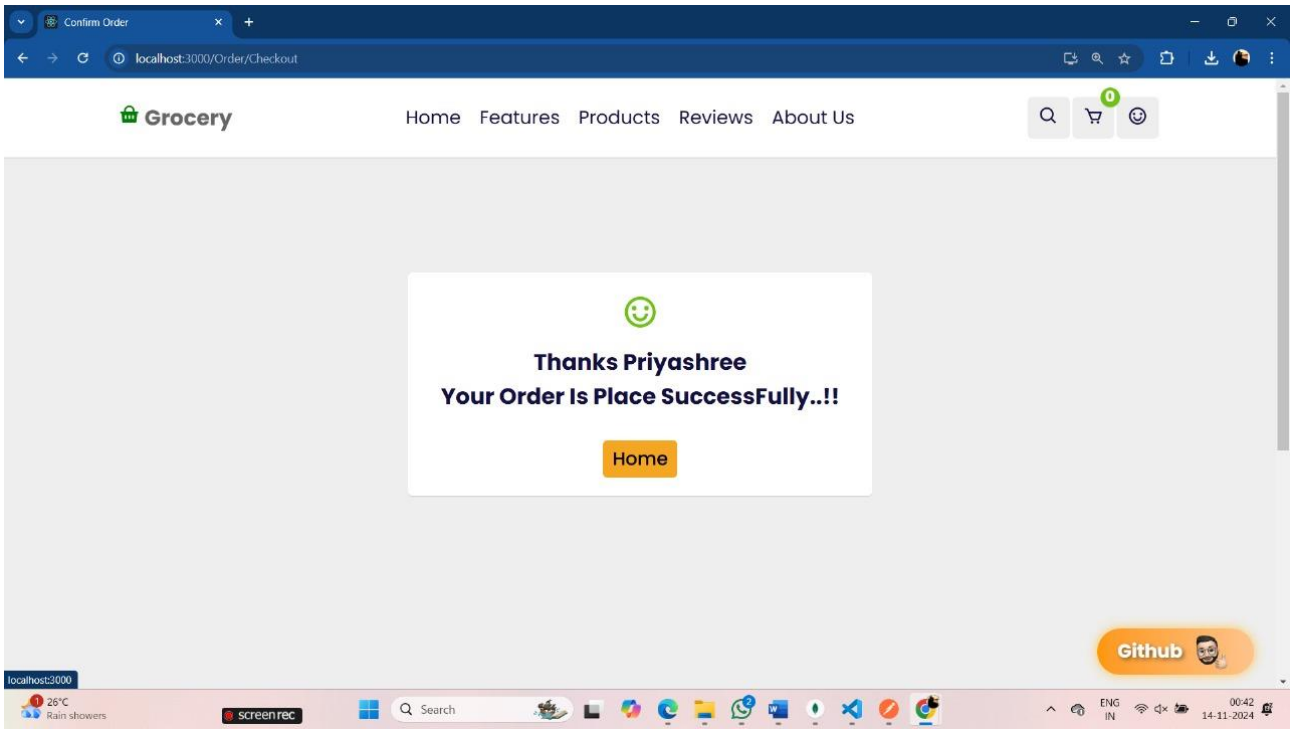
Order Summary



Order Details



Order Confirmation



12. KNOWN ISSUES

1. **Search and Filter Limitations**: The search functionality may be limited to basic keywords, and advanced filters (e.g., by category or price range) are not fully implemented, potentially impacting the user's shopping experience.
2. **Token Expiry Handling**: Users may need to manually log in again if their session expires, as there is no automatic token refresh implemented.
3. **Responsiveness**: Some UI elements may not display optimally on smaller screens, affecting the mobile shopping experience.
4. **Error Messages**: Error messages for failed actions (e.g., login or checkout failures) are not fully descriptive, which may lead to confusion.
5. **Performance:** Large datasets in product listings can cause slow load times without efficient pagination.

13. FUTURE ENHANCEMENTS

1. **Advanced Search and Filtering**: Adding more sophisticated filters (e.g., by category, brand, and price) to improve the shopping experience.
2. **Wishlist Functionality**: Allowing users to save items for future purchases.
3. **Real-Time Notifications**: Implementing notifications for order updates, stock changes, and discounts.
4. **Recommendation System**: Integrating personalized recommendations based on user purchase history.
5. **Improved Mobile Optimization**: Enhancing mobile responsiveness and performance for a smoother experience on all devices.
6. **Payment Gateway Integration**: Adding support for popular payment gateways like PayPal and Stripe.