GROCERY WEBAPP

USING MERN STACK SMART INTERNZ (NM)

PROJECT REPORT

Submitted by

JAZEEL MACAREENA P	**211121104024**
VAISHNAVI S	**211121104056**
SRIMMA T	**211121104050**
SHARMILA V	**211121104048**
HAMSA VARDHINI V	**211121104018**

BACHELORS OF ENGINEERING IN

COMPUTER SCIENCE AND ENGINEERING

MADHA ENGINEERING COLLEGE

DEPARTMENT OF COMPUTER ENGINEERING

KUNDRATHUR, CHENNAI – 600 069

NOVEMBER 2024

TABLE OF CONTENTS

ABSTRACT

The Grocery Webapp is a purpose-built e-commerce platform that revolutionizes the grocery shopping experience for both end-users and administrators. Developed from the ground up using modern web technologies, the application integrates a modular architecture to deliver a seamless, efficient, and secure shopping experience. Designed to handle all aspects of an online grocery store, it supports intuitive product browsing, streamlined purchasing workflows, and comprehensive administrative management for inventory, orders, and customer interactions.

This documentation offers a detailed walkthrough of the entire development lifecycle, encompassing architectural design, database schema modeling, backend and frontend development, RESTful API integration, testing methodologies, and deployment strategies. Emphasizing key principles such as scalability, security, and user-centered design, the project achieves a balance between robust performance and ease of use.

Additionally, the report identifies current limitations, such as token management and database optimization, while proposing strategic enhancements like advanced analytics, mobile app development, and AI-powered recommendations to elevate the platform further. This paper serves as a comprehensive guide to the Grocery Webapp, showcasing its innovative approach to modern e-commerce solutions and its potential for continued growth and improvement.

KEYWORDS

E-commerce, Online Grocery Shopping, Angular, Node.js, MongoDB, Full Stack Development, JWT Authentication, Scalable Architecture, API Documentation

LIST OF FIGURES

1. INTRODUCTION

Project Title: Grocery Webapp

The **Grocery Webapp** is a custom-built e-commerce platform designed to streamline the online grocery shopping experience for both customers and administrators. The application integrates modern technologies to provide a robust, secure, and scalable solution. It features an interactive product catalog, secure authentication, a user-friendly shopping cart, a seamless checkout process, and a comprehensive admin dashboard for managing inventory and orders. Built by a talented and collaborative team, the project showcases expertise in full-stack development, frontend and backend technologies, database management, and rigorous testing practices.

Team Members:

Full - Stack Developer: Jazeel Macareena P

- Managed the entire development process, ensuring smooth integration between the frontend and backend, scalable architecture, and efficient data flow.

Frontend Developer: Srimma T

- Created a responsive and user-friendly interface, implementing features like the product catalog, shopping cart, and checkout while ensuring mobile optimization and accessibility.

Backend Developer: Vaishnavi S

- Built secure backend systems, including APIs, order processing, and payment handling, ensuring reliable communication and robust functionality.

Database Developer: Sharmila V

- Designed a scalable database schema, optimized queries with indexing, and ensured consistent and efficient data management for users, products, and orders.

Testing Engineer: Hamsa Varshini V

- Performed comprehensive functional, integration, and performance testing to deliver a stable, error-free, and user-centric application.

2. OBJECTIVE

The Grocery Webapp project was initiated to create a cutting-edge e-commerce platform tailored specifically to the needs of the online grocery sector. The goal was to design and build an application from scratch that could deliver a seamless, intuitive, and efficient shopping experience while addressing the operational requirements of both customers and administrators. The project aimed to provide a robust, scalable, and secure platform capable of handling diverse e-commerce functionalities with precision and reliability.

For customers, the platform features secure user authentication, a highly interactive product catalog with advanced filtering and sorting options, an intuitive shopping cart, and a streamlined checkout process integrated with multiple payment options. Real-time order tracking further enhances transparency and trust. The application also prioritizes responsive design, ensuring a consistent experience across devices, including desktops, tablets, and smartphones.

For administrators, the Grocery Webapp includes a comprehensive dashboard offering tools for inventory management, sales monitoring, customer analytics, and order processing. The system is designed to scale efficiently as the database grows, incorporating features like automated stock alerts and supplier management to streamline operations. Built with modern technologies such as Angular, Node.js, Express, and MongoDB, the application employs a modular architecture that ensures flexibility, maintainability, and future adaptability.

The development process emphasized security, scalability, and performance optimization, employing industry best practices such as token-based authentication, database indexing, and efficient API design. This documentation provides a detailed account of the structured approach taken throughout the project, covering architectural planning, backend and frontend development, API design, testing methodologies, and deployment strategies. Additionally, it highlights the application's current strengths and limitations while proposing enhancements such as AI-driven product recommendations, multi-language support, and a mobile app to elevate the platform further.

The ultimate objective of the Grocery Webapp project was to create a holistic and innovative e-commerce solution that not only meets the current demands of the grocery sector but also lays the foundation for continuous evolution and scalability in a dynamic market environment.

3. PROJECT OVERVIEW

With the ever-increasing demand for online shopping platforms, particularly in the grocery sector, the Grocery Webapp project was conceptualized to provide a seamless and efficient solution for both customers and administrators. Traditional grocery shopping often lacks the convenience and accessibility modern users expect. The purpose of this project was to fill this gap by creating a user-friendly, reliable, and robust platform that simplifies the grocery shopping experience. The application not only caters to the needs of end-users by offering an intuitive and engaging interface but also empowers administrators with comprehensive tools to manage products, orders, and customer interactions efficiently.

Features and Functionality

1. **Customer Experience**:
   The Grocery Webapp prioritizes a smooth and satisfying user experience. Customers can navigate through an interactive and well-organized product catalog with advanced search and filtering options. A streamlined checkout process ensures hassle-free transactions, while the ability to view and track order history adds transparency and convenience.

2. **Admin Dashboard**:
   Administrators are equipped with powerful tools to manage the platform effectively. These include functionalities for adding, editing, or removing products, processing and monitoring orders, and accessing customer behavior analytics to make informed decisions

3. **Secure Transactions**:
   The application integrates with multiple payment gateways to ensure secure and versatile payment options. Token-based user authentication safeguards sensitive information, providing users with a trusted platform for their transactions.

4. **Responsive Design**:
   Designed with accessibility in mind, the Grocery Webapp is optimized for use across all devices, including desktops, tablets, and smartphones. The responsive architecture ensures smooth performance and a consistent user experience, regardless of the screen size or resolution.

5. **Additional Features**:
   The platform also includes advanced features such as real-time order tracking, scalable architecture for handling increasing data volumes, and predictive search to enhance product discovery. These functionalities make the application adaptable and future-ready, setting it apart from conventional e-commerce platforms.

4. SYSTEM ARCHITECTURE

Architecture Design:

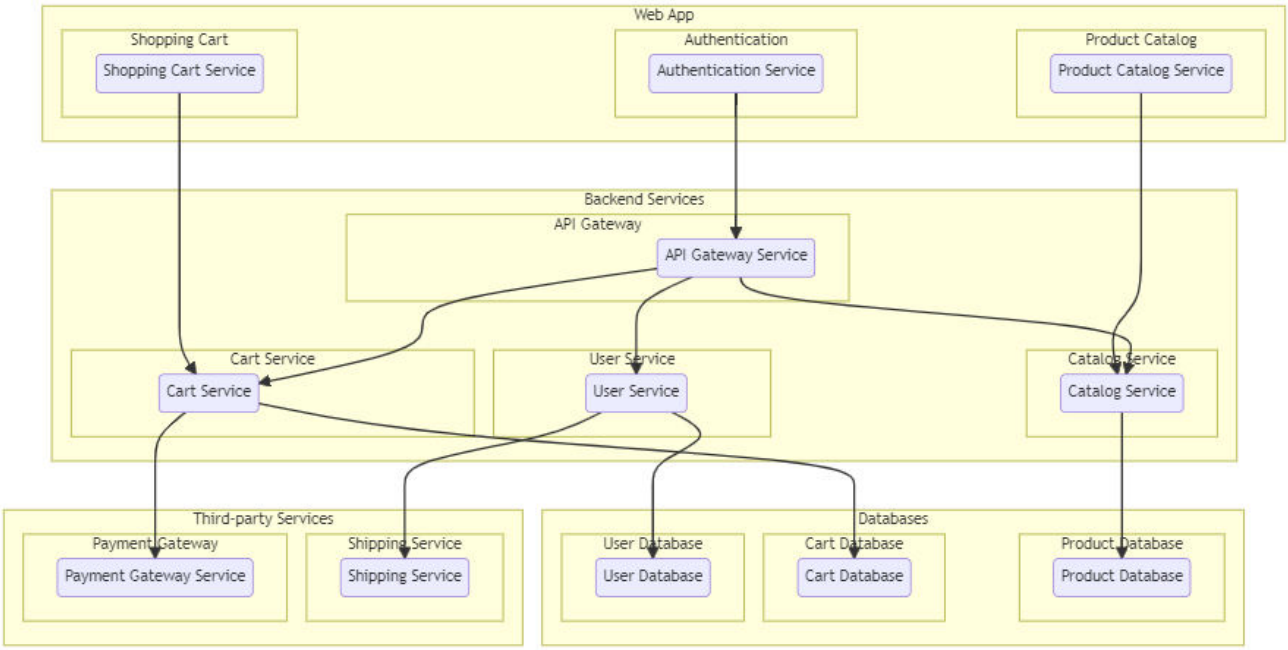

Fig. a. Architecture

The technical architecture of an flower and gift delivery app typically involves a client-server model, where the frontend represents the client and the backend serves as the server. The frontend is responsible for user interface, interaction, and presentation, while the backend handles data storage, business logic, and integration with external services like payment gateways and databases. Communication between the frontend and backend is typically facilitated through APIs, enabling seamless data exchange and functionality.

The Grocery Webapp employs a service-oriented architecture (SOA) with a clear separation of concerns:

- **Frontend Services:** Angular modules and components handle user interactions, shopping cart management, product catalog browsing, and authentication.
- **Backend Services:** Node.js services manage CRUD operations, session handling, and data processing for users, products, orders, and cart functionalities.
- **API Gateway:** A centralized API Gateway manages all requests between frontend and backend services, maintaining consistency and efficiency.
- **Third-Party Integrations:** Includes secure connections to payment processing services, such as Stripe, and shipping services.

ER Diagram (Entity-Relationship Diagram):

The technical architecture of an flower and gift delivery app typically involves a client-server model, where the frontend represents the client and the backend serves as the server. The frontend is responsible for user interface, interaction, and presentation, while the backend handles data storage, business logic, and integration with external services like payment gateways and databases. Communication between the frontend and backend is typically facilitated through APIs, enabling seamless data exchange and functionality.

- **Entities:** Key entities include User, Admin, Product, Cart, Order, Address, and Payment, each with relationships defining how data is structured and retrieved.
- **Relationships:** For instance, the User entity is linked to Orders, Cart, Address, and Wishlist, facilitating a structured approach to data retrieval and storage.

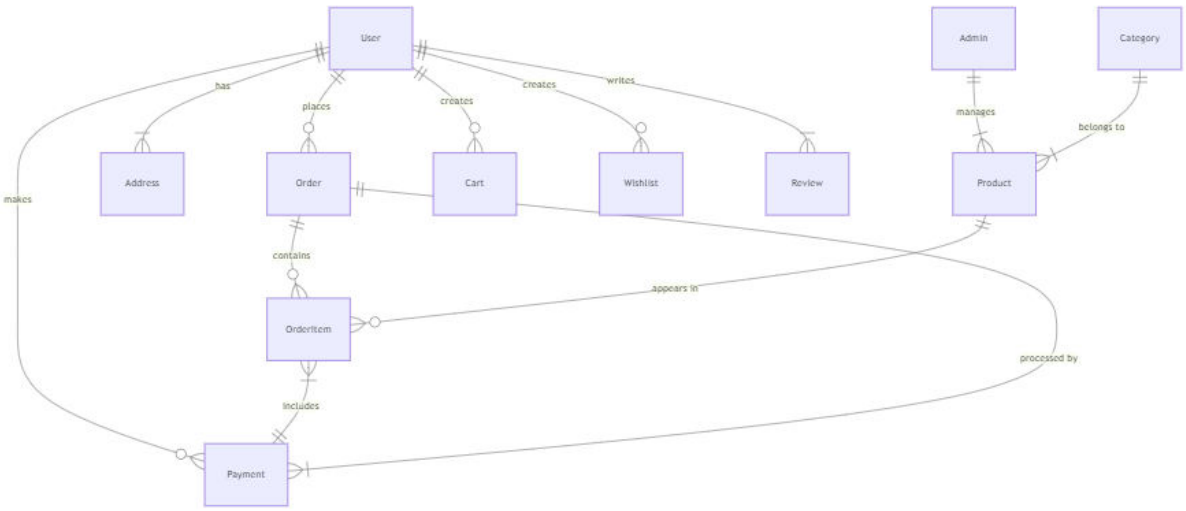

Fig. b. ER Diagram

4.1. System Implementation

4.1.1 Client-Side Structure (Angular)

- **Development Approach:** The Angular client was built from scratch, structured as a modular and component-based system with clear segregation of roles:
    - **Components:** Organized under client/src/app/components, each component handles a specific functionality, such as product listing, cart management, or user registration.
    - **Routing:** Angular's routing module supports seamless navigation between the application's main sections, utilizing role-based route guards for enhanced security.
    - **UI Design:** Leveraging Bootstrap and custom CSS, the UI is responsive, offering a consistent experience across devices.

4.1.2 Server-Side Structure (Node.js and Express.js)

- **Modular Architecture:** The backend is organized by functionality under server/src, separating concerns with individual services for each core area:
    - **Controllers:** Handle business logic for each API endpoint (e.g., productController.js for managing product-related requests).
    - **Routes:** API routes are defined in separate files, including authRoutes.js, productRoutes.js, and orderRoutes.js.
    - **Middleware:** Custom middleware, such as authMiddleware.js, checks for user authentication and enforces role-based access.

4.1.3 Database Schema (MongoDB)

- **Schema Design:** Utilizing Mongoose, schemas were defined for each entity:
    - **User Schema:** Contains user details, hashed passwords, and role designations.
    - **Product Schema:** Stores data on product categories, pricing, stock levels, and descriptions.
    - **Order Schema:** Records user orders, associated cart items, and order status.
    - **Cart Schema:** Temporarily stores items added by users before purchase, supporting CRUD operations.

5. SETUP INSTRUCTIONS

5.1 Pre-Requisites

1. **System Requirements**:
    - **Hardware:** Windows 8 or higher.
    - **Internet Bandwidth:** Minimum 30 Mbps.
    - **Browsers:** Install two modern web browsers (e.g., Chrome, Firefox).
2. **Software Setup**:
    - **Node.js and npm:** Install Node.js to run the backend and manage dependencies.
    - **MongoDB:** Set up a local or cloud-based MongoDB database.
    - **Git:** Install Git for version control.
    - **Angular CLI:** Install Angular globally for frontend development. Run:

      npm install -g @angular/cli
3. **Code Editor**:
    - Install an IDE or editor like Visual Studio Code.

Step 1: Clone the Repository

1. **Open the Terminal/Command Prompt**: Navigate to the directory where you want to save the project.
2. **Clone the Project**: Run the following command to clone the repository:

   git clone <URL>
3. **Navigate into the Project Directory**:

   cd grocery-webapp

Step 2: Frontend Setup (Angular)

1. **Navigate to the client Directory**:

   cd client
2. **Install Dependencies**: Install all necessary packages using:

   npm install
3. **Start the Development Server**: Launch the Angular application:

ng serve

- The app will run at http://localhost:4200.

4. **Optional:** Set Environment Variables:
   - Create or edit .env file for API base URLs. Example:

export const environment = {production: false,apiBaseUrl: 'http://localhost:5000/api'};

Step 3: Backend Setup (Node.js/Express)

1. **Navigate to the server Directory**: From the root project directory:

   cd server

2. **Install Dependencies**: Run:

   npm install

3. **Configure Environment Variables**:
   - Create a .env file in the server directory if not present.
   - Add the following configurations:

   PORT=5000

   MONGO_URI=mongodb://localhost:27017/groceryApp

   JWT_SECRET=your_jwt_secret_key

4. **Start the Backend Server**: Run:

   npm start

   - The backend will be accessible at http://localhost:5000.

5. **Verify MongoDB Connection**: Ensure MongoDB is running locally or adjust the MONGO_URI in the .env file to connect to a cloud instance.

Step 4: Integration

1. **Link Frontend and Backend**:
   - Confirm the Angular environment API base URL matches the backend (e.g., http://localhost:5000/api).

2. **Test the Application**:
   - Open the frontend at http://localhost:4200.
   - Perform actions like registering, logging in, browsing products, and placing orders to ensure the frontend communicates with the backend.

Step 5: Database Configuration

1. **Set Up MongoDB**:
    - Use the following MongoDB collections:
        - **Users**: Stores user information and credentials.
        - **Products**: Stores product details like name, price, and stock.
        - **Orders**: Tracks customer orders and statuses.
        - **Feedback**: Logs user feedback.
2. **Optional - MongoDB Atlas**:
    - For cloud storage, set up a free MongoDB Atlas cluster and update MONGO_URI in .env.

Step 6: Deployment

1. **Frontend Deployment**:
    - Build the Angular app for production:

      ng build --prod

    - Deploy the dist folder to a hosting service like Netlify, AWS S3, or Vercel.
2. **Backend Deployment**:
    - Host the Node.js backend on a service like Heroku, AWS EC2, or Azure.
    - Ensure the database (MONGO_URI) is accessible to the deployed server.
3. **Update Configurations**:
    - Modify Angular's .env and .env files to use production URLs and secrets.

5.2 Testing

1. **Frontend**:
    - Run Angular unit tests:

      ng test

    - Run end-to-end tests:

      ng e2e
2. **Backend**:
    - Run backend tests using a framework like Jest:

      npm test

5.3 Troubleshooting

1. **Backend Issues**:
    - Check .env for correct MONGO_URI and PORT.
    - Ensure MongoDB service is running locally or remotely.
2. **Frontend Issues**:
    - Verify apiBaseUrl in Angular's environment.ts.
    - Check console logs for CORS or API errors.
3. **Dependency Errors**:
    - Run npm install in both client and server directories.

6. FOLDER STRUCTURE

This structure assumes an Angular app and follows a modular approach. Here's a brief explanation of the main directories and files:

- **client/src/app/components:** Contains components related to the customer app, such as register, login, home, products, my-cart, my-orders, placeorder, history, feedback, product-details, and more.
- **client/src/app/modules:** Contains modules for different sections of the app. In this case, the admin module is included with its own set of components like add-category, add-product, dashboard, feedback, home, orders, payment, update-product, users, and more.
- **client/src/app/app-routing.module.ts:** Defines the routing configuration for the app, specifying which components should be loaded for each route.
- **client/src/app/app.component.ts, src/app/app.component.html, `src**.
- **server/src/db:** Manages database connectivity and schema definitions. Establishes and maintains a connection to the database (e.g., MongoDB). Contains schema definitions for various data entities, ensuring consistency in how data is stored and retrieved.
- **server/src/modules:** Organizes backend logic into feature-specific directories for modularity and maintainability. Handles request-response cycles, receiving API calls and invoking appropriate services. Defines API endpoints and maps them to corresponding controller functions. Implements the core business logic and reusable methods for a specific module.
- **server/src/routes:** Acts as a central registry for all API endpoints across modules. Combines all feature-specific routes into a single entry point. Imports and mounts routes from each module to ensure separation of concerns and easy navigation.

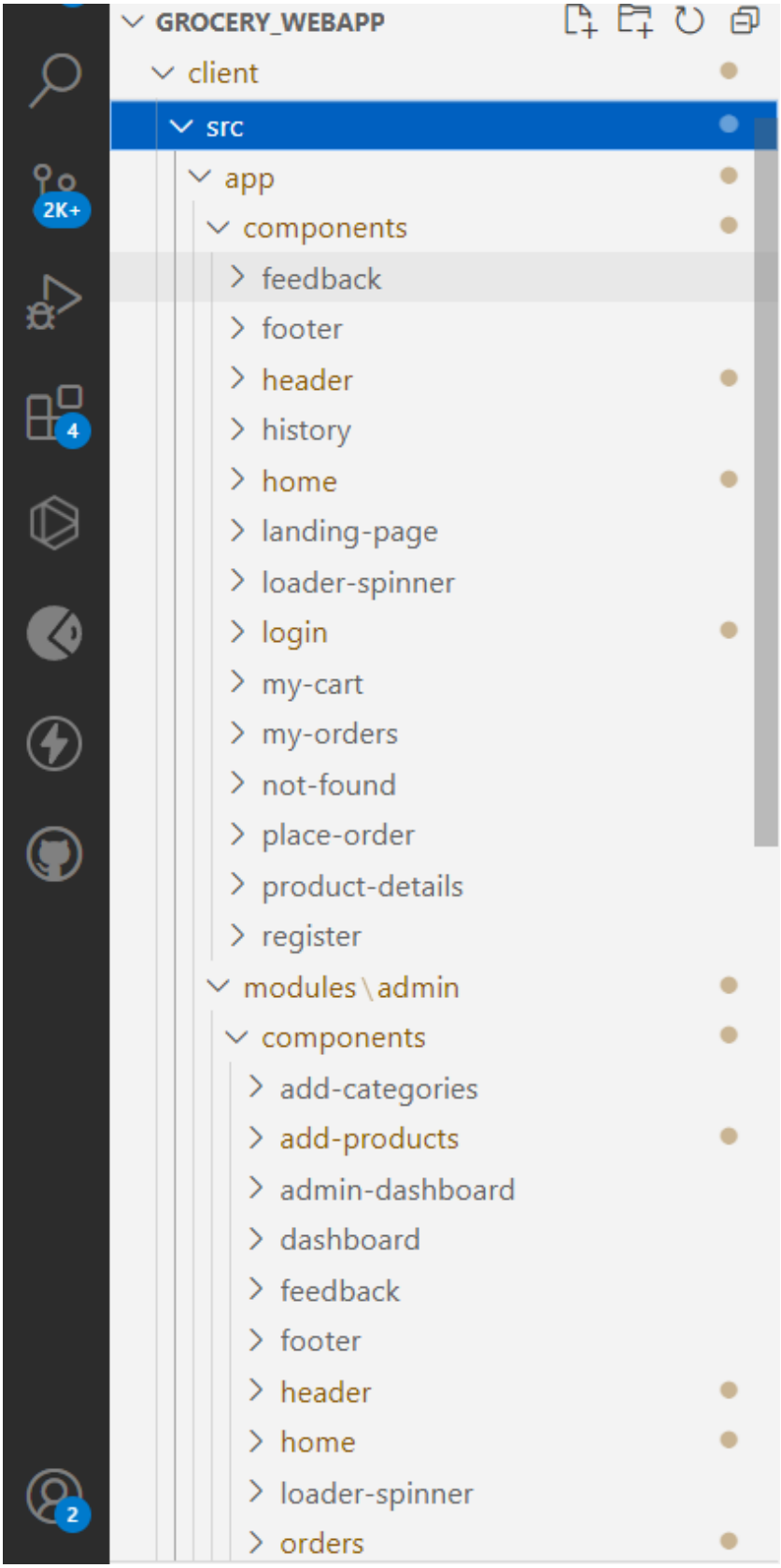

Fig. 1. Structure

7. RUNNING THE APPLICATION

Step 1: Start MongoDB

1. **Local Database**:
    - Open your terminal and start the MongoDB server.
    - If MongoDB is installed locally, run:

      mongod

    - Ensure the database is running on the default port (27017) or update the connection string (MONGO_URI) in your .env file.
2. **Cloud Database (Optional)**:
    - If using a cloud-based MongoDB like **MongoDB Atlas**, ensure your MONGO_URI in the .env file is correctly configured with the cluster's connection string.

Step 2: Start the Backend Server

1. **Navigate to the Backend Directory**:
    - In your terminal, move to the server folder:

      cd server
2. **Start the Server**:
    - Run the backend server using:

      npm start
    - This will start the server on the port specified in .env (default: 5000).
    - The backend will be accessible at http://localhost:5000.
3. **Verify Backend**:
    - Open a browser or tool like Postman to test the health of the API.
    - Visit:

      http://localhost:5000/api/health
    - You should see a confirmation message (if a health check route is implemented).

Step 3: Start the Frontend Application

1. **Navigate to the Frontend Directory**:
    - Open a new terminal and navigate to the client folder:

      cd client
2. **Start the Angular App**:
    - Run:

      ng serve
    - This will start the Angular development server on http://localhost:4200.
3. **Verify Frontend**:
    - Open your web browser and visit:

      http://localhost:4200
    - You should see the grocery web app's homepage.

Step 4: Verify Integration

1. **Ensure Frontend-Backend Communication**:
    - Perform user actions like:
        - Registering or logging in.
        - Browsing products.
        - Adding items to the cart.
        - Placing an order.
    - These actions should send requests to the backend (http://localhost:5000).
2. **Check for Errors**:
    - Open the browser's developer console (press F12) and look for API-related errors (e.g., 404 or CORS issues).
    - If needed, adjust the apiBaseUrl in client/src/environments/environment.ts to match the backend URL.

Step 5: (Optional) Adjust Ports

- If the default ports (4200 for frontend, 5000 for backend) conflict with other services:
    1. Change the frontend port by adding this flag:

       ng serve --port=4300

2. Change the backend port in the .env file:

PORT=5500

Step 6: Running Both Servers Simultaneously

To simplify running both servers, use the following steps:

1. Open **two terminals**:
    - One for the backend (server) and one for the frontend (client).
2. Start both servers as described above:
    - In one terminal: npm start (backend).
    - In the other terminal: ng serve (frontend).

Access the Application

- **Frontend URL:** http://localhost:4200 (or the port you specified).
- **Backend API:** http://localhost:5000 (or the port in .env).

8. API DOCUMENTATION

8.1 Authentication Endpoints

User Signup

- **Endpoint:** /api/auth/signup
- **Method:** POST
- **Description:** Registers a new user by collecting name, email, and password. It validates the input, hashes the password using bcrypt for security, and stores the user data in the database. A confirmation message is returned along with the user's basic information.
- **Request Body:**

```
{
 "name": "Jane Doe",
 "email": "jane.doe@example.com",
 "password": "securepassword"
}
```

- **Response:**

```
{
 "message": "User successfully registered.",
 "user": {
  "id": "64a7b12c8f8a1234567e8c9",
  "name": "Jane Doe",
  "email": "jane.doe@example.com"
 }
}
```

User Login

- **Endpoint:** /api/auth/login
- **Method:** POST
- **Description:** Authenticates a user by comparing the submitted credentials with stored records. Upon successful login, it generates a JWT token for secure access to protected routes.
- **Request Body:**

{
"email": "jane.doe@example.com",
"password": "securepassword"
}

- **Response:**

{
"message": "Login successful.",
"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ...",
"user": {
"id": "64a7b12c8f8a1234567e8c9",
"name": "Jane Doe"
}
}

8.2 Product Endpoints

Get All Products

- **Endpoint:** /api/products
- **Method:** GET
- **Description:** Fetches a list of all available products with details such as name, price, stock, and category. Supports optional query parameters for filtering (e.g., by category or price range).
- **Response:**

[
{
"id": "64a7b13d8f8a1234567e8d1",
"name": "Banana",
"price": 1.5,
"category": "Fruits",
"stock": 100,
"image": "https://example.com/banana.jpg"
},

```
 {
  "id": "64a7b13d8f8a1234567e8d2",
  "name": "Almond Milk",
  "price": 3.0,
  "category": "Dairy",
  "stock": 50,
  "image": "https://example.com/almondmilk.jpg"
 }
]
```


Get Product by ID

- **Endpoint:** /api/products/:id
- **Method:** GET
- **Description:** Fetches detailed information about a specific product by its ID.
- **Response:**


```
{
 "id": "64a7b13d8f8a1234567e8d1",
 "name": "Banana",
 "description": "Fresh organic bananas sourced from local farms.",
 "price": 1.5,
 "category": "Fruits",
 "stock": 100,
 "image": "https://example.com/banana.jpg"
}
```


8.3 Cart Endpoints

Add Item to Cart

- **Endpoint:** /api/cart
- **Method:** POST
- **Description:** Adds a specified quantity of a product to the user's cart. The cart updates dynamically with the total price calculation.

- **Request Body:**

```
{
 "productId": "64a7b13d8f8a1234567e8d1",
 "quantity": 2
}
```

- **Response:**

```
{
 "message": "Item successfully added to cart.",
 "cart": {
  "items": [
   {
    "productId": "64a7b13d8f8a1234567e8d1",
    "quantity": 2,
    "price": 3.0
   }
  ],
  "totalPrice": 3.0
 }
}
```

8.4 Order Endpoints

Place Order

- **Endpoint:** /api/orders
- **Method:** POST
- **Description:** Processes the user's cart into an order. The API validates the shipping address, calculates the total, and initiates order fulfillment.
- **Request Body:**

```
{
 "address": "123 Main Street, Springfield",
 "paymentMethod": "Credit Card"
}
```

- **Response:**


```
{
 "message": "Order placed successfully.",
 "order": {
  "id": "64a7b12c8f8a1234567e8e0",
  "status": "Processing",
  "items": [
   {
    "productId": "64a7b13d8f8a1234567e8d1",
    "quantity": 2,
    "price": 3.0
   }
  ],
  "totalPrice": 3.0,
  "address": "123 Main Street, Springfield"
 }
}
```

9.AUTHENTICATION AND AUTHORIZATION

9.1 Authentication Process

Authentication ensures that only legitimate users gain access to the platform by validating their credentials. The system implements secure practices to protect user data and maintain integrity.

User Signup

1. **Input Validation**:
    - Every registration request is validated to ensure the input adheres to pre-defined rules:
        - **Name**: Should not be empty and should not exceed a certain length.
        - **Email**: Validated against a standard email format (e.g., example@domain.com).
        - **Password**: Must meet strength requirements, such as a minimum length, inclusion of uppercase letters, numbers, and special characters.
    - Invalid inputs are rejected with detailed error messages.
2. **Password Hashing**:
    - Plain text passwords are never stored.
    - The app uses **bcrypt**, a reliable hashing library, to encrypt user passwords before saving them in the database. Even if the database is compromised, hashed passwords remain unreadable.
3. **JWT Token Generation**:
    - After successful registration, the system generates a **JWT token**.
    - The token contains:
        - **User ID**: The unique identifier of the newly created user.
        - **Expiration Time**: Tokens expire after a defined duration (e.g., 1 hour) to prevent misuse.
        - **Signature**: Signed using a secret key to ensure the token cannot be forged.
    - The token is sent in the response, allowing the user to start a session immediately.

User Login

1. **Credential Verification**:
    - The email provided by the user is matched against records in the database.

- The hashed password stored in the database is compared with the password submitted during login.

2. **Session Establishment**:
   - If the credentials are valid, the system generates a new JWT token for the session.
   - Tokens can be securely stored on the client-side using **LocalStorage** or **HTTPOnly cookies**.
3. **Error Handling**:
   - Specific error messages are returned for:
     - Incorrect email or password.
     - Inactive accounts.

9.2 Authorization

Authorization ensures users can only access resources and functionalities they are allowed to.

Protected Routes

- Routes such as viewing the cart, placing orders, or accessing the admin panel are protected using middleware that checks the validity of JWT tokens.
- **Client-Side:** The token is included in the Authorization header as Bearer <token>.
- **Server-Side:** Middleware verifies the token before allowing access.

Role-Based Access Control (RBAC)

- **General Users**:
  - Can browse products, manage their cart, place orders, and view order history.
- **Admins**:
  - Have elevated privileges to manage products, monitor orders, and analyze sales data.

10. USER INTERFACE

The User Interface (UI) of the Grocery WebApp is designed for simplicity, responsiveness, and ease of navigation. Each page is tailored to provide a seamless user experience.

10.1 Home Page

1. **Purpose**:
    - Introduces users to the platform, showcasing its offerings.
    - Serves as the entry point for browsing products and categories.
2. **Features**:
    - **Navigation Bar**:
        - Links to Home, Products, Cart, and Login/Signup.
        - Dynamic links for logged-in users (e.g., Profile, Order History).
    - **Promotional Banner**:
        - Highlights ongoing sales, discounts, or featured products.
    - **Categories Section**:
        - Displays product categories like "Fruits," "Vegetables," and "Dairy" in a visually appealing grid layout.
    - **Search Bar**:
        - Allows users to search for products by name or keywords.

10.2 Product Listing Page

1. **Purpose**:
    - Displays a list of available products with options to filter and sort them.
2. **Features**:
    - **Filters**:
        - Filter by category, price range, or availability.
    - **Sorting**:
        - Options to sort products by price (low to high, high to low) or popularity.
    - **Product Cards**:

- Compact and visually appealing cards display:
    - Product image.
    - Name and price.
    - Stock availability.
    - "Add to Cart" button.

10.3 Product Details Page

1. **Purpose**:
    - Provides detailed information about a selected product.
2. **Features**:
    - **Images**:
        - High-quality, zoomable images.
    - **Description**:
        - Includes details like ingredients, weight, and nutritional value (if applicable).
    - **Stock Availability**:
        - Real-time updates on product availability.
    - **Interaction**:
        - Quantity selector.
        - Reviews section with ratings and user comments.

10.4 Cart Page

1. **Purpose**:
    - Allows users to review and manage selected items before proceeding to checkout.
2. **Features**:
    - **Item List**:
        - Shows all added products with:
            - Name, image, price and Quantity selector..
            - Option to remove items.
    - **Summary**:
        - Displays subtotal, taxes, and total price dynamically.

- **Checkout Button**:
  - Takes users to the checkout process.

10.5 Admin Dashboard

1. **Purpose**:
   - A control panel for managing the app's content and monitoring performance.
2. **Features**:
   - **Product Management**:
     - Add, update, and delete products.
     - Real-time inventory tracking.
   - **Order Monitoring**:
     - View order statuses (e.g., Pending, Shipped, Delivered).
     - Update statuses as orders are processed.
   - **Analytics**:
     - Charts for sales performance, top products, and user activity.

10.6 Checkout Page

1. **Purpose**:
   - Finalizes the purchase process by collecting necessary details.
2. **Features**:
   - **Shipping Details Form**:
     - Captures user name, address, and contact number.
   - **Order Summary**:
     - Displays items with total cost and estimated delivery time.
   - **Payment Options**:
     - Supports multiple payment gateways like Stripe or PayPal.

10.7 Error/404 Page

1. **Purpose**:

- Handles invalid URLs or application errors gracefully.

2. **Features**:
   - **Design**:
     - Friendly error message (e.g., "Oops! Page Not Found").
     - A link to return to the home page.
   - **Error Codes**:
     - **400**: Bad Request.
     - **401**: Unauthorized Access.
     - **404**: Resource Not Found.

10.8 Error Handling

Frontend Error Handling

1. **Network Issues**:
   - Displays messages like "Network Error. Please check your connection."
2. **Form Validation**:
   - Inline messages highlight invalid fields (e.g., "Email is required").

Backend Error Handling

1. **Global Error Handling Middleware**:
   - Catches errors across all routes.
   - Returns structured responses:

```
{
 "statusCode": 500,
 "message": "Internal Server Error"
}
```

2. **Logging**:
   - Uses libraries like **Winston** to log errors for debugging.

11. TESTING AND VALIDATION

Testing and validation are critical phases of the Grocery WebApp development lifecycle, ensuring its functionality, performance, security, and user satisfaction. A well-structured and exhaustive testing plan provides confidence in the application's ability to meet business and user expectations under various scenarios.

11.1 Testing Strategies

Unit Testing

- **Objective**: Unit testing isolates and tests individual components or functions to ensure they perform as expected. It focuses on specific pieces of functionality, like adding an item to the cart or calculating the total price in the backend.
- **Scope**:
    - Verifying isolated functions, methods, and classes in both the frontend and backend.
    - Testing utility functions such as calculating discounts, validating forms, and generating order summaries.
- **Tools**:
    - **Frontend**: Jasmine and Karma for Angular or Jest for React.
    - **Backend**: Mocha and Chai for Node.js APIs.
- **Examples**:
    - Test the addItemToCart method to ensure it correctly updates the cart with the right quantities and prices.
    - Validate the calculateTotalPrice function to confirm accurate price summation with taxes and discounts.

Sample Test Case:

```
describe('Cart Service', () => { it('should add an item to the cart', () => {const cart = new Cart();

cart.addItem({ id: '101', name: 'Bananas', price: 1.5 });

expect(cart.items.length).toBe(1);

expect(cart.items[0].name).toBe('Bananas');

});

});
```

Integration Testing

- **Objective**: Integration testing verifies that different modules work together as expected. It ensures seamless communication between the frontend and backend, as well as between the database and APIs.
- **Scope**:
    - Testing interactions such as data flow between the product catalog API and the product listing page.
    - Verifying workflows like user login, cart management, and order placement.
- **Tools**:
    - Postman for API testing.
    - Cypress for end-to-end integration scenarios.
- **Examples**:
    - Test that adding a product to the cart reflects correctly in the backend cart database.
    - Ensure that the login process generates a valid JWT token, which is then used to authenticate protected routes.

Sample Integration Flow:

1. Simulate user login via /api/auth/login.
2. Use the generated token to fetch the cart data from /api/cart.
3. Verify the cart contents and ensure proper communication between the frontend and backend.

Functional Testing

- **Objective**: Functional testing ensures the application operates according to its functional requirements. It validates user interactions, including browsing products, managing carts, and placing orders.
- **Scope**:
    - Testing core functionalities such as authentication, product search, and checkout.
    - Ensuring admin operations, like adding products and updating order statuses, work as expected.
- **Tools**:
    - Selenium for browser automation.

    - Protractor for Angular-based frontend testing.
- **Examples**:
    - Verify that users can filter products by categories, sort by price, and add selected items to the cart.
    - Ensure that the "Forgot Password" feature sends a password reset email with the correct link.

Functional Test Case:

- **Test**: Adding a product to the cart.
- **Steps**:
    1. Search for a product.
    2. Click the "Add to Cart" button.
    3. Verify the product appears in the cart.
    4. Check the updated total price.

Performance Testing

- **Objective**: Performance testing evaluates the app's behavior under various loads, ensuring it meets response time, scalability, and reliability expectations.
- **Scope**:
    - Testing API response times during peak traffic.
    - Simulating thousands of concurrent users to assess system capacity.
- **Tools**:
    - Apache JMeter for stress and load testing.
    - Locust for distributed performance testing.
- **Examples**:
    - Measure the time taken to fetch product data when 1,000 users are accessing the app simultaneously.
    - Ensure the database performs efficiently during bulk order placements.

Security Testing

- **Objective**: Security testing identifies vulnerabilities to protect user data, prevent unauthorized access, and ensure secure transactions.

- **Scope**:
    - Testing for vulnerabilities such as SQL injection, XSS (Cross-Site Scripting), and CSRF (Cross-Site Request Forgery).
    - Validating password hashing, token-based authentication, and secure API communication.
- **Tools**:
    - OWASP ZAP for penetration testing.
    - Burp Suite for security vulnerability assessment.
- **Examples**:
    - Attempt SQL injection attacks on /api/products to verify database security.
    - Test the validity of expired or tampered JWT tokens in protected routes.

User Acceptance Testing (UAT)

- **Objective**: UAT ensures the application meets end-user expectations and business requirements. It involves real users performing tasks to validate the app's usability and functionality.
- **Scope**:
    - Testing end-to-end workflows like registration, browsing, and checkout.
    - Gathering feedback on the user interface, navigation, and overall experience.
- **Process**:
    - Select a group of test users.
    - Provide predefined tasks like placing an order or updating profile details.
    - Collect feedback and identify any usability issues.

11.2 Validation Plan

Input Validation

- **Frontend**:
    - Enforces constraints like valid email formats and password complexity.
    - Uses client-side validation to provide immediate feedback to users.
- **Backend**:
    - Validates all inputs to ensure data integrity and prevent malicious payloads.
    - Rejects invalid requests with descriptive error messages.

Example Backend Validation:

```
if (!email.includes('@')) {
return res.status(400).json({ message: 'Invalid email format.' });
}
```

API Validation

- APIs validate request bodies against schemas to ensure data consistency.
- Tools like Joi or Yup are used for server-side validation.

Example Joi Schema for Product Creation:

```
const productSchema = Joi.object({name: Joi.string().min(3).required(),
price: Joi.number().greater(0).required(),
category: Joi.string().required()
});
```

UI Validation

- Tests the responsiveness and accessibility of the application across devices.
- Ensures consistent design and layout, with error messages displayed in an understandable manner.

11.3 Error Handling

Frontend Errors

1. **Validation Errors**:
    - Display inline messages for fields like email and password.
    - Example: "Password must be at least 8 characters long."
2. **Network Errors**:
    - Show user-friendly messages for connectivity issues.
    - Example: "Unable to connect. Please check your internet connection."

Backend Errors

1. **Global Error Handling**:
    - Captures unhandled exceptions and returns standardized error responses.
2. **Error Messages**:
    - Include HTTP status codes with meaningful descriptions.
    - Example for unauthorized access:

```
{
  "status": 401,
  "message": "Access denied. Please log in."
}
```

11.4 Reporting and Metrics

- **Bug Tracking**: Use tools like Jira to document bugs and track resolution progress.
- **Test Coverage Reports**: Measure the percentage of code tested using tools like Istanbul or nyc.
- **Performance Metrics**: Record response times, server throughput, and error rates during stress tests.

12. SCREENSHOTS AND CODING

client/src/app/components/feedback

- **Purpose:** Allows customers to share their opinions, suggestions, or complaints about their shopping experience.
- **Details:** This component likely contains a form with fields for star ratings, comments, and optional fields such as a subject line. It validates inputs, sends data to the backend via an API, and may show a success or error message after submission. It can also include a history view where users see their previously submitted feedback.

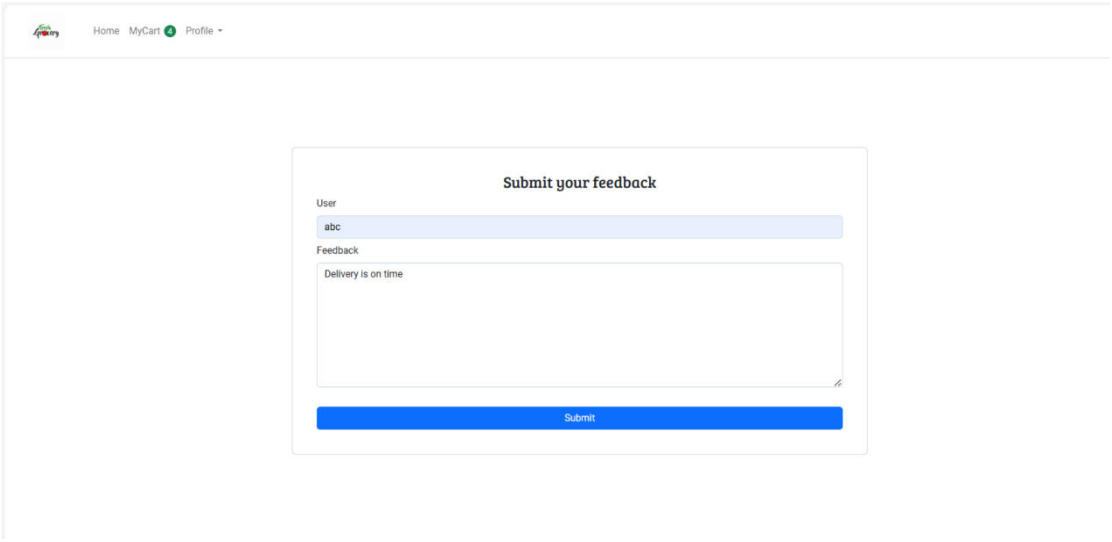

Fig. 2. Feedback

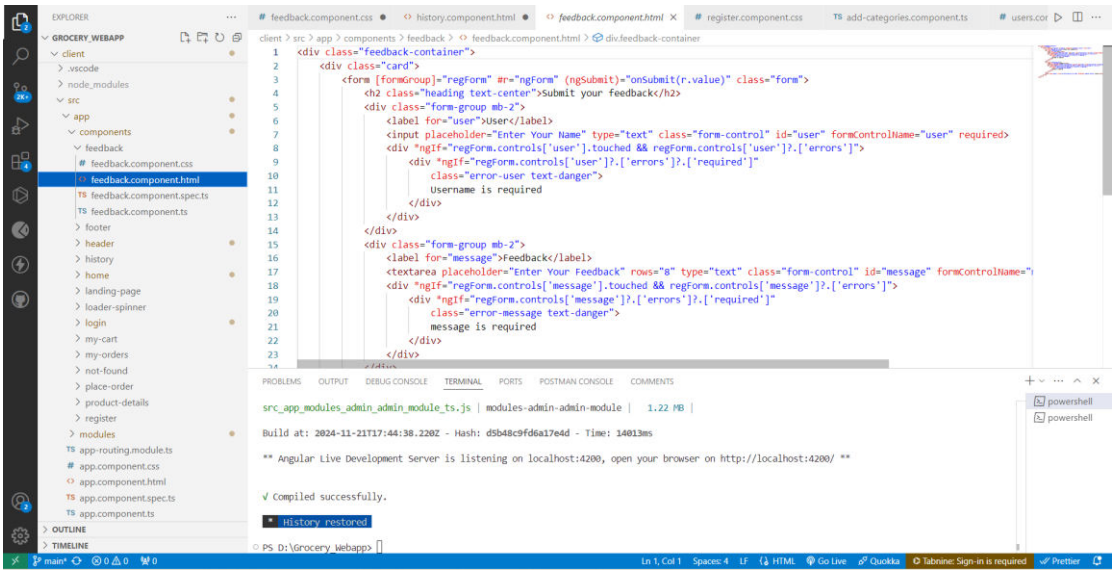

Fig. 19. feedback.component.html

client/src/app/components/footer

- **Purpose:** Ensures uniformity across all pages by displaying important details at the bottom.
- **Details:** The footer might include sections like "About Us," "Contact Us," and "Terms & Conditions." It could dynamically adjust based on user role (e.g., admin-specific links when logged in). It may also include a mini newsletter subscription form for customers to stay updated.

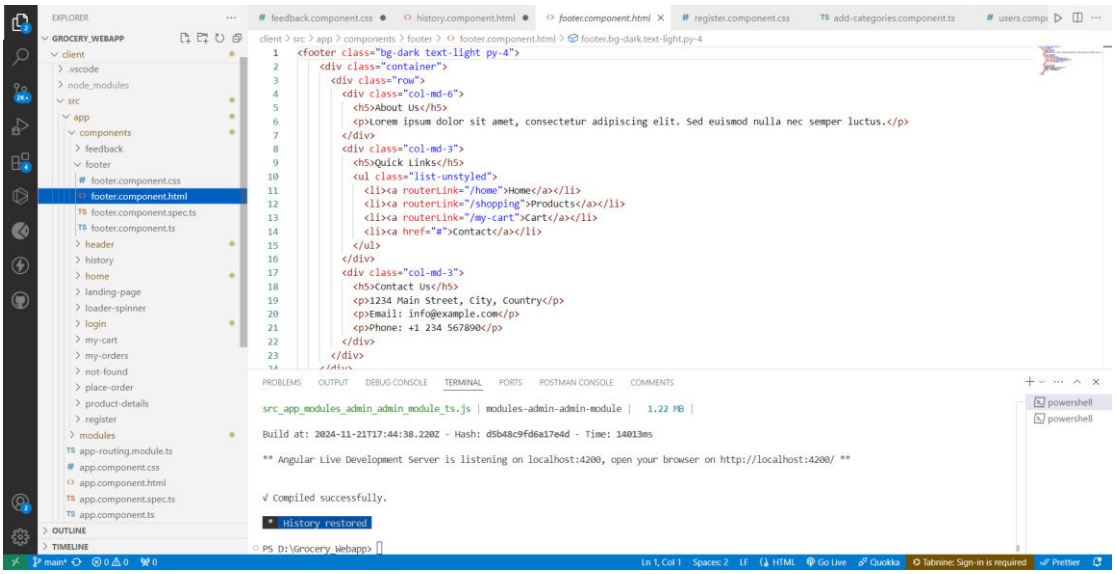

Fig. 20 footer.component.html

client/src/app/components/header

- **Purpose:** Offers primary navigation across the app.
- **Details:** The header could include a dynamic menu showing links such as Home, Products, and My Cart. It might have a search bar connected to the product database, allowing customers to filter products by name, category, or price. For logged-in users, it might display a dropdown for profile management.

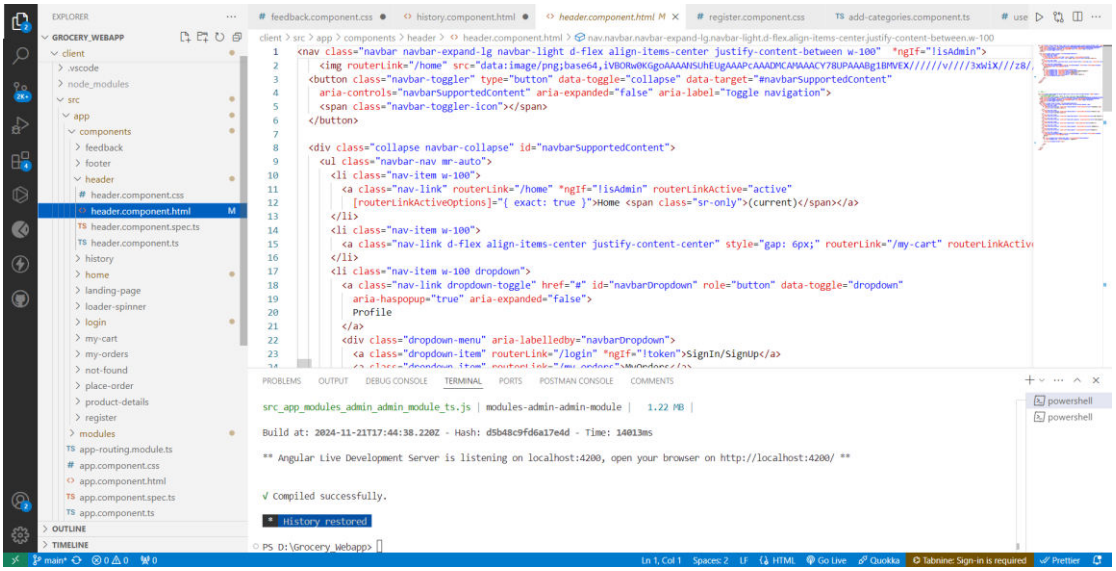

Fig. 21. header.component.html

client/src/app/components/history

- **Purpose:** Gives users an overview of their activity within the app.
- **Details:** This component may track orders placed, items viewed, and actions like cancellations or refunds. It could include filters (e.g., show only refunded orders) and allow users to download receipts or invoices.

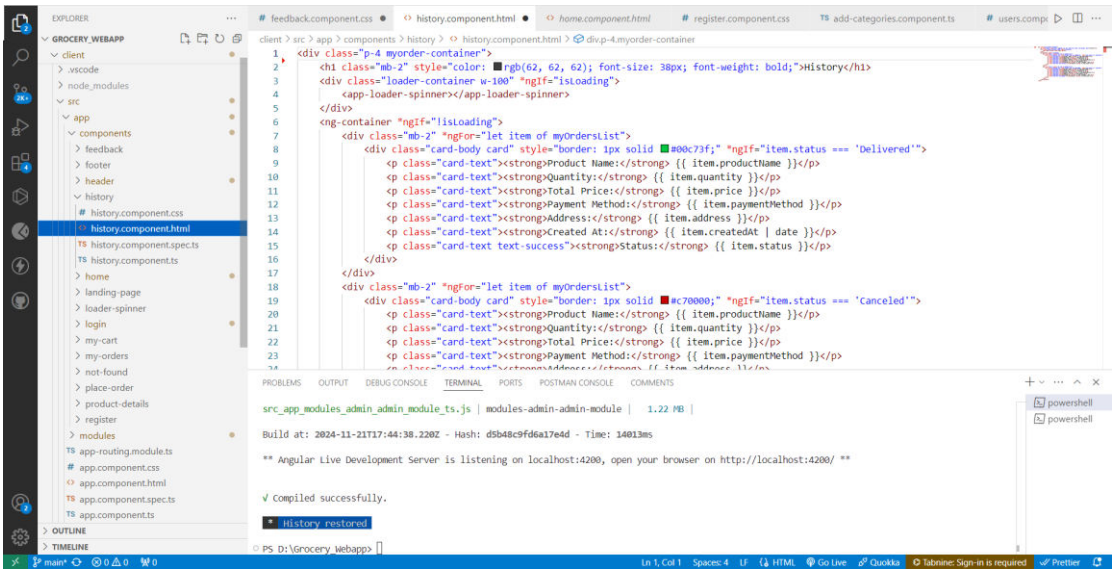

Fig. 22. history.component.html

client/src/app/components/home

- **Purpose:** Acts as the initial entry point for the app's shopping experience.
- **Details:** The homepage may feature a dynamic carousel for promotions or seasonal offers, trending products, and quick-access buttons to popular categories. It's designed to captivate the user and drive them deeper into the app by showcasing curated recommendations powered by backend data.

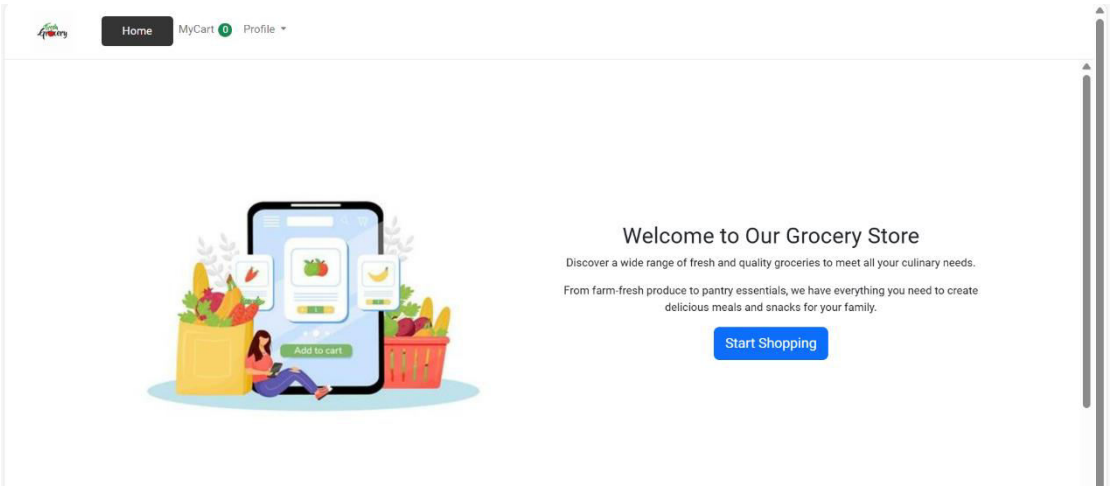

Fig. 3. Home

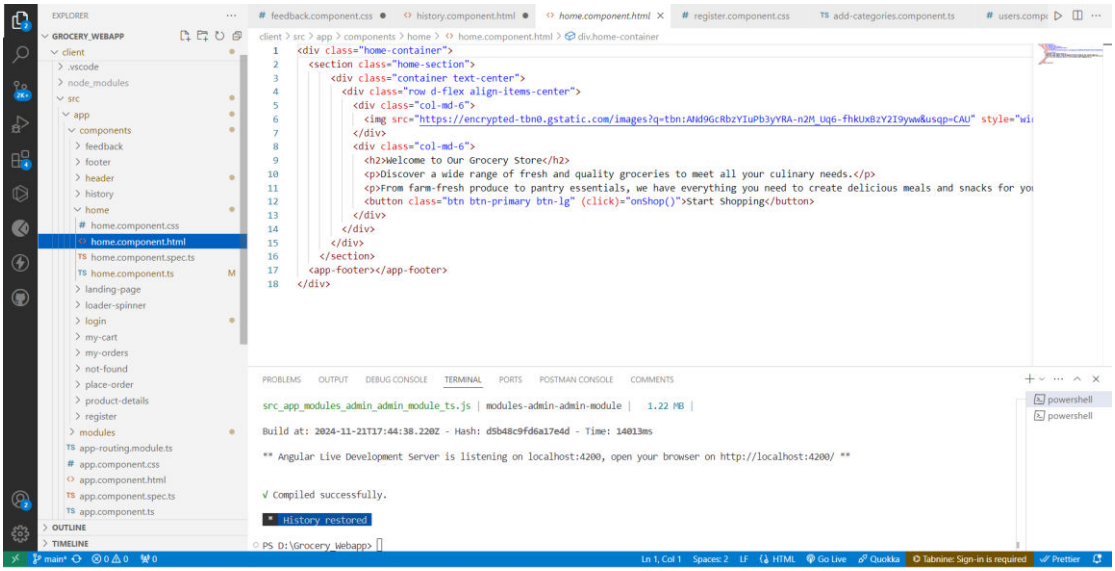

Fig. 23. home.component.html

client/src/app/components/landing-page

- **Purpose:** Introduces first-time users to the app.
- **Details:** This is usually designed to engage potential customers with a clean, eye-catching layout. It may showcase benefits like "Free Delivery for Orders Above $50," highlight the ease of shopping with a few steps (browse, add to cart, place order), and include testimonials.

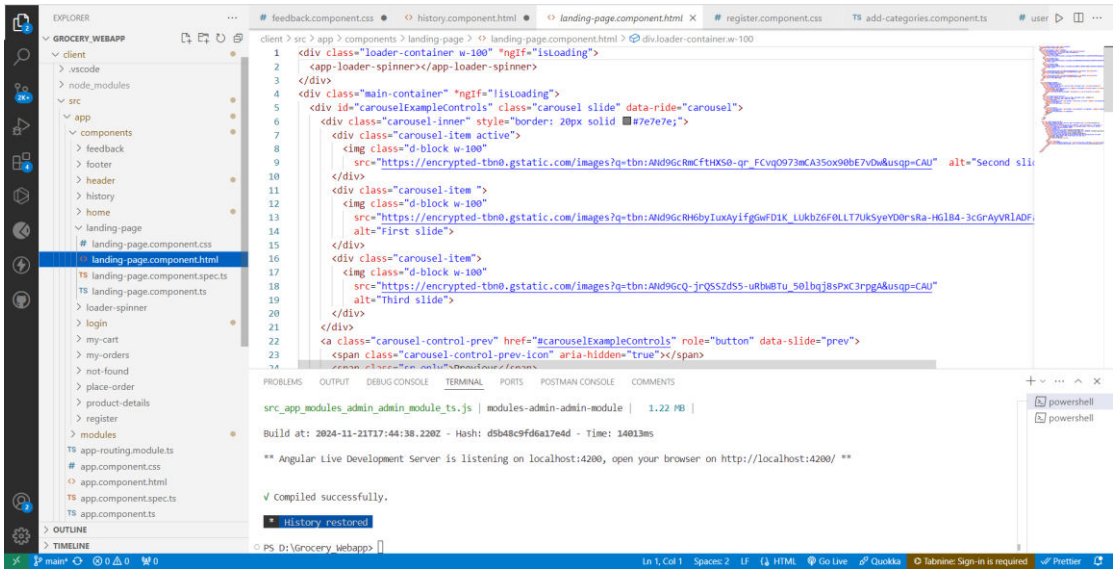

Fig. 24. landing-page.component.html

client/src/app/components/loader-spinner

- **Purpose:** Indicates progress when the app is fetching data or processing actions.
- **Details:** It appears as a spinner or progress bar during delays caused by API calls, database queries, or form submissions. It ensures the app doesn't seem unresponsive, offering visual feedback to users.

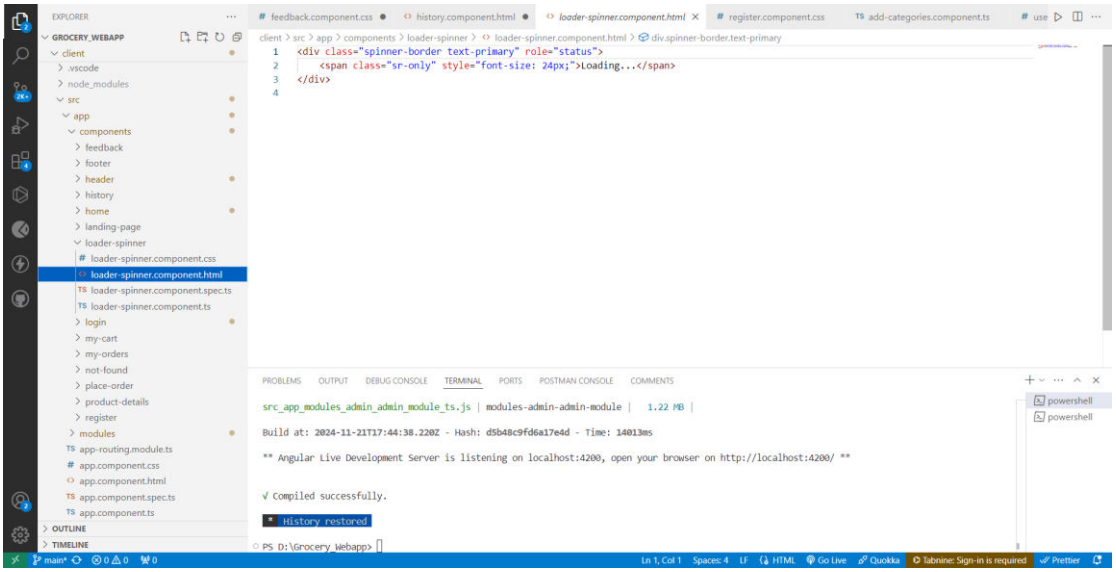

Fig. 25. loader-spinner.component.html

client/src/app/components/login

- **Purpose:** Handles the secure login process for customers.
- **Details:** Contains fields for email and password, with validation rules like email format checks and password strength indicators. It sends credentials to the backend via an API and processes tokens (e.g., JWT) for authentication. It may also include a "Forgot Password" feature.

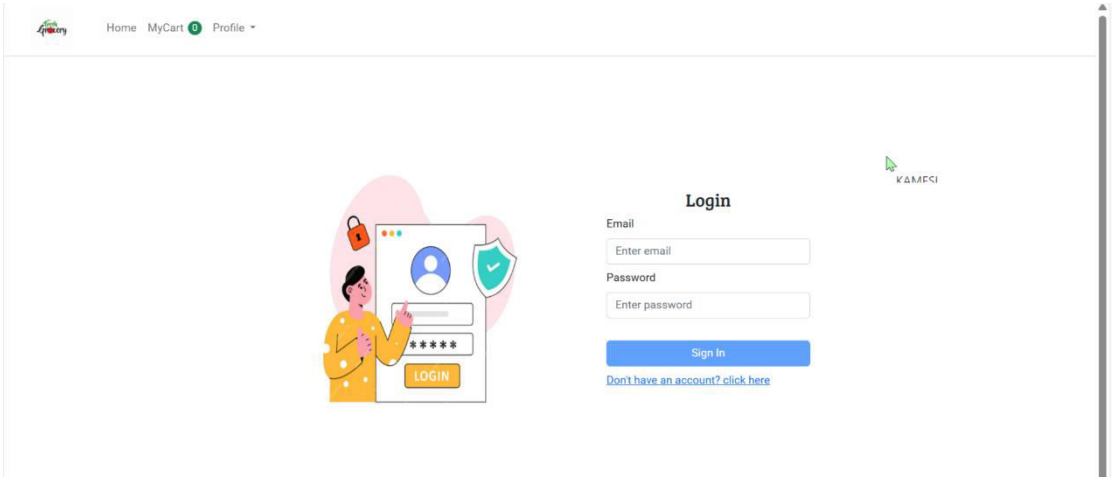

Fig. 4. Login

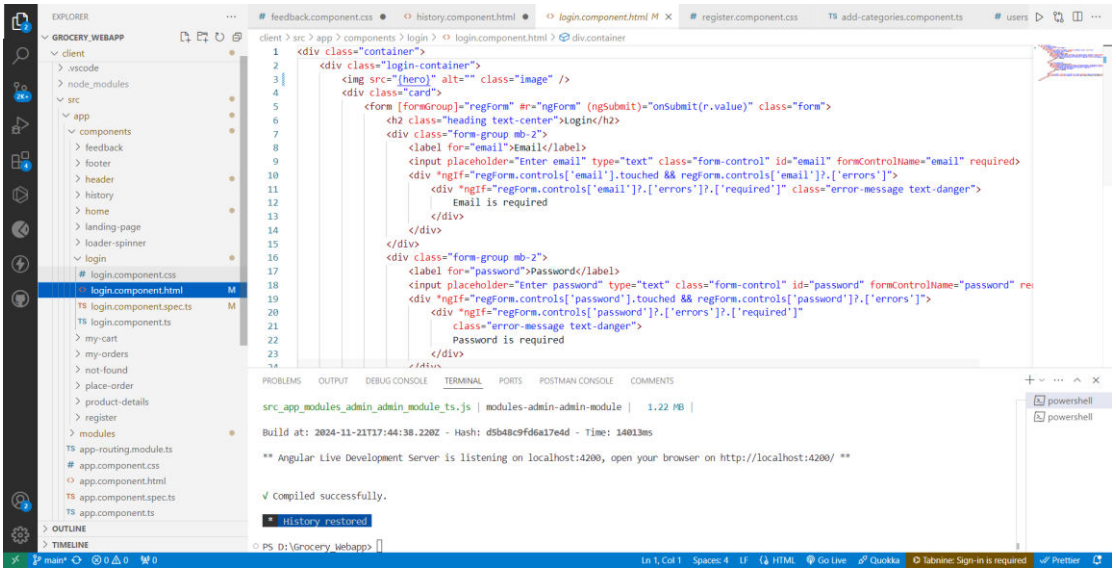

Fig. 26. login.component.html

client/src/app/components/my-cart

- **Purpose:** Displays items the user is preparing to purchase.
- **Details:** Shows each product with details like name, price, quantity, and subtotal. It might allow users to apply promo codes or coupons, calculate taxes dynamically, and display a grand total. Includes buttons like "Update Quantity," "Remove Item," and "Proceed to Checkout."

Fig.5. My-cart

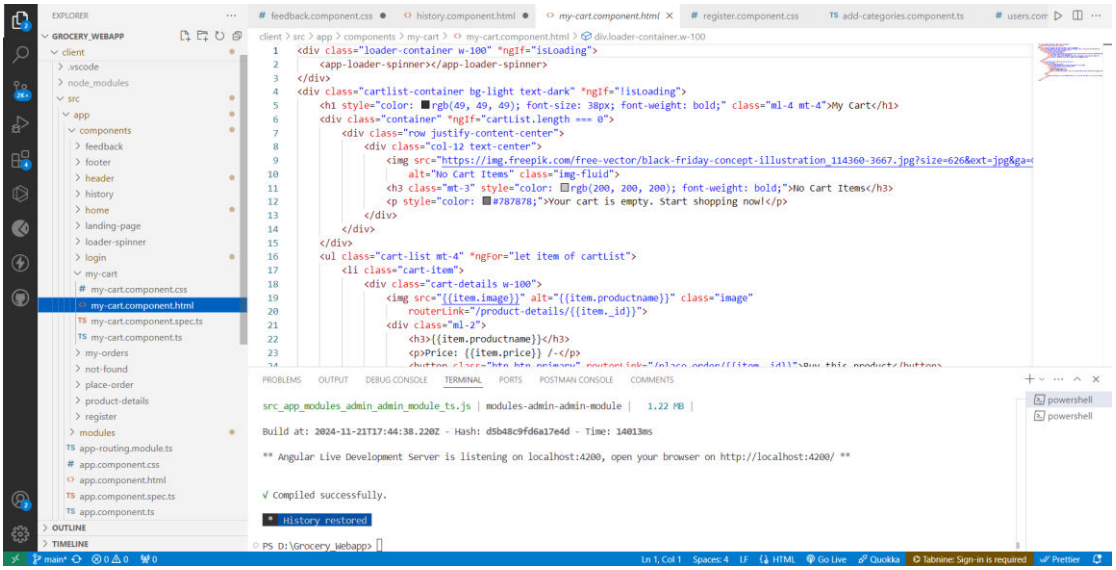

Fig. 27. My-cart.component.html

client/src/app/components/my-orders

- **Purpose:** Tracks all orders placed by the user.
- **Details:** Displays a paginated list of past orders with fields like order number, date, total amount, and status (e.g., Pending, Shipped, Delivered). Users can click on an order to view details or request support for issues like returns or replacements.

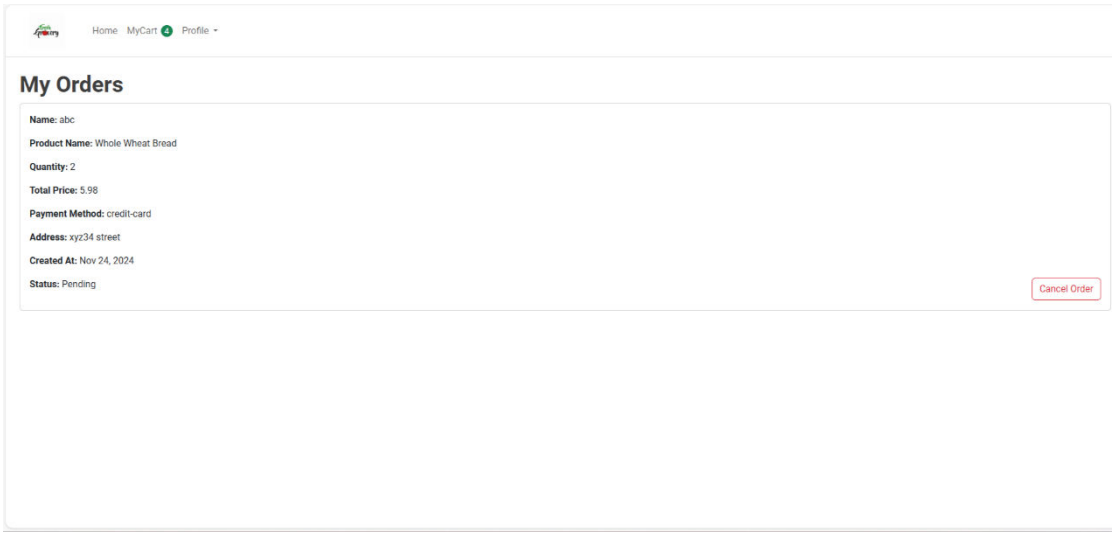

Fig. 6. My-orders

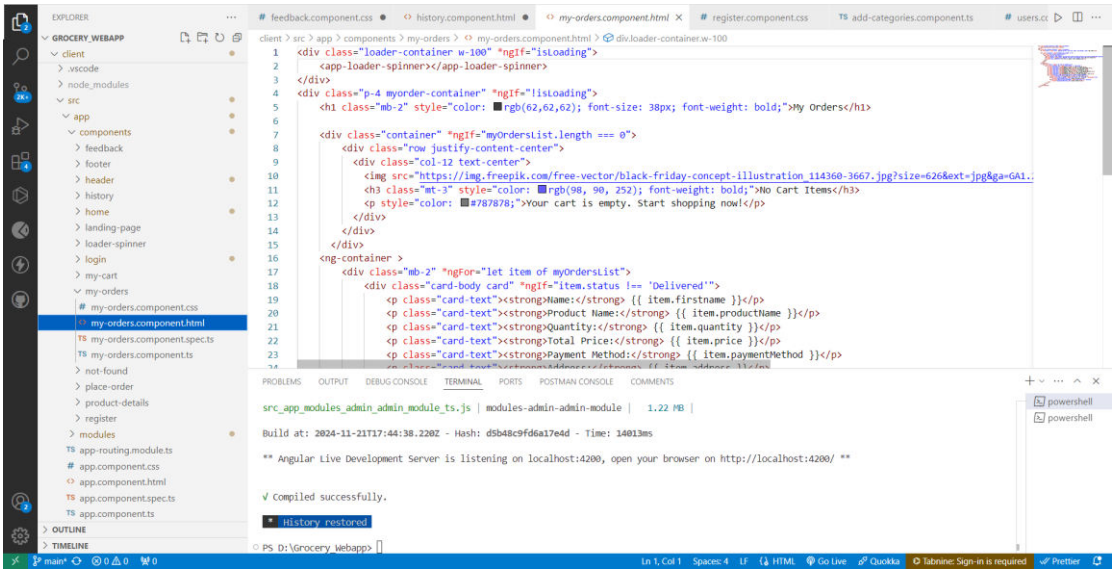

Fig. 28. orders.component.html

client/src/app/components/not-found

- **Purpose:** Handles invalid or unknown URLs.
- **Details:** Displays a custom 404 error message, possibly with creative visuals to keep the tone light. It might include a "Back to Home" button or a search bar for users to find their intended destination.

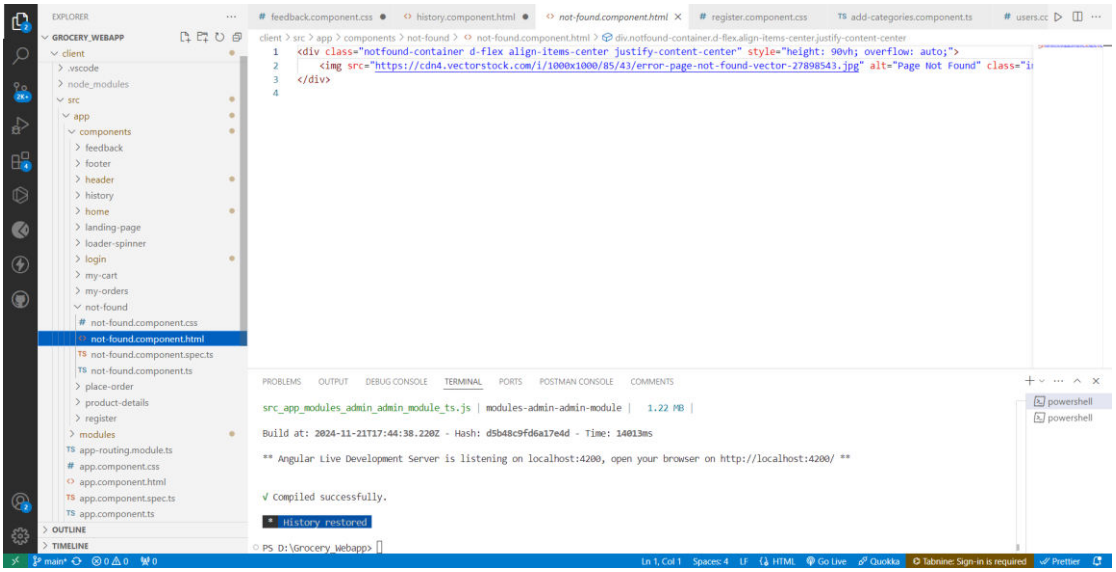

Fig. 29. Not-found.component.html

12.12 client/src/app/components/place-order

- **Purpose:** Handles the checkout process to finalize purchases.
- **Details:** This multi-step component includes selecting a delivery address, choosing a payment method (e.g., credit card, PayPal), and reviewing the order summary. It validates user inputs and sends the data to the backend to create an order record.

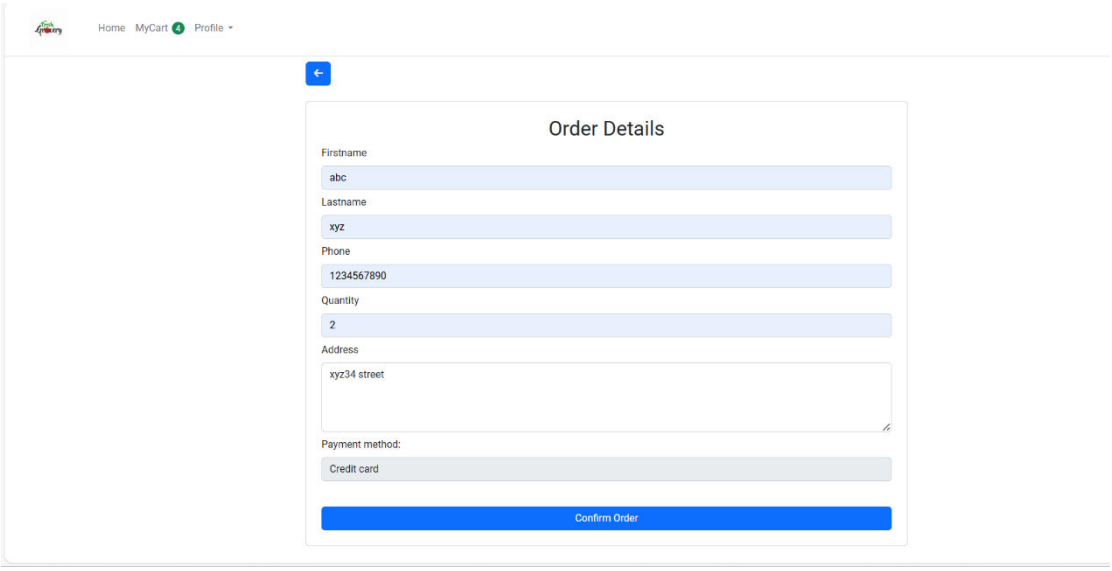

Fig. 7. Placed-order

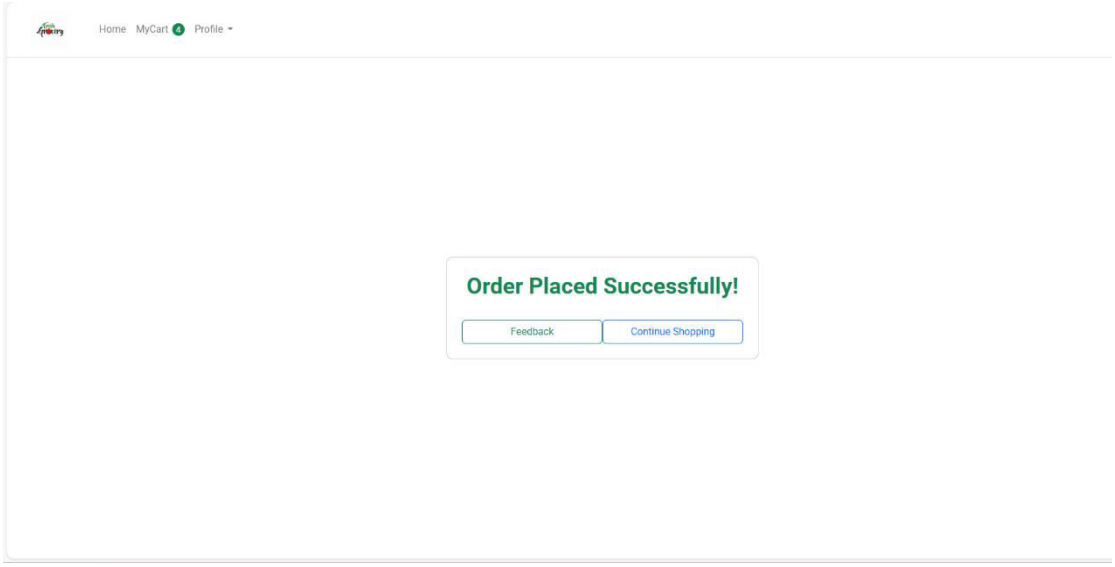

Fig.8. Order-placed

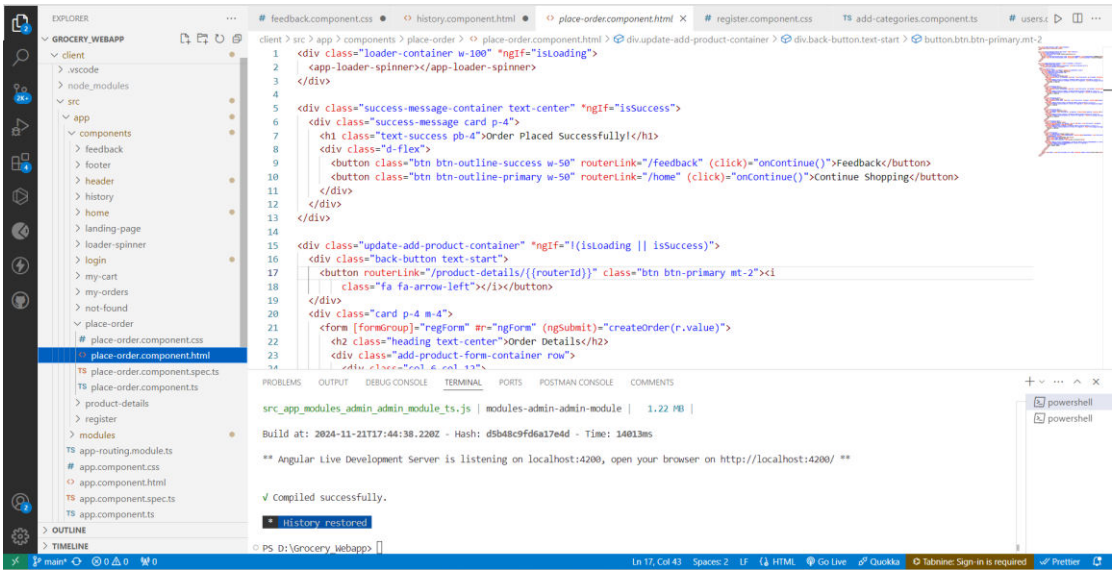

Fig. 30. Place-order.component.html

client/src/app/components/product-details

- **Purpose:** Displays comprehensive information about a single product.
- **Details:** This page includes product images, descriptions, ratings, reviews, price, availability, and related products. It may allow users to add the product to their cart or wishlist and includes features like a review submission form.
-

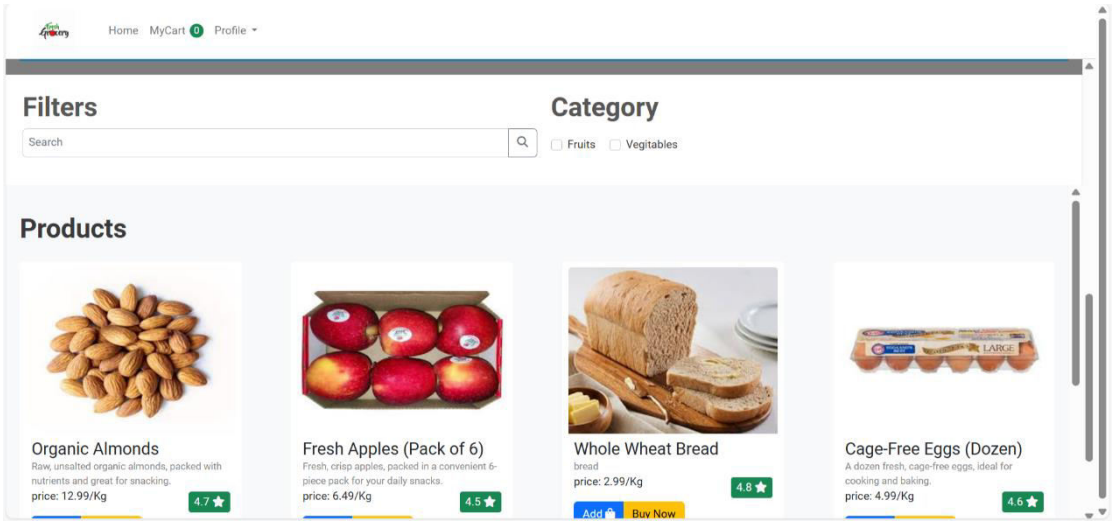

Fig.9. Product-details

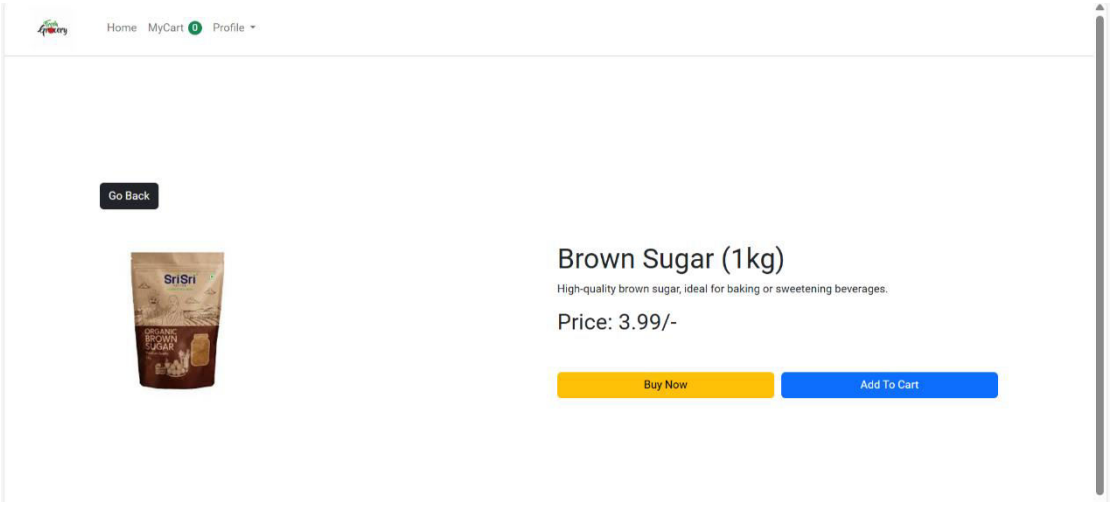

Fig.10. Product-preview

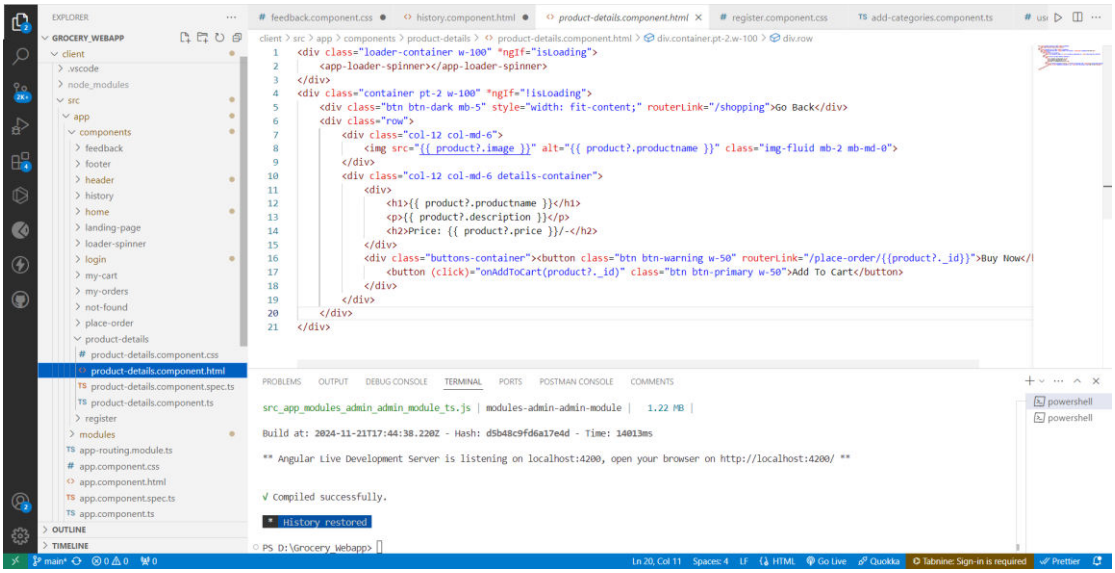

Fig. 31. product-details.component.html

client/src/app/components/register

- **Purpose:** Allows new users to create accounts.
- **Details:** A form-based component that collects user information like name, email, password, and optional details like a phone number. It validates inputs and sends a request to the backend for account creation. Includes security measures like captcha or email verification links.

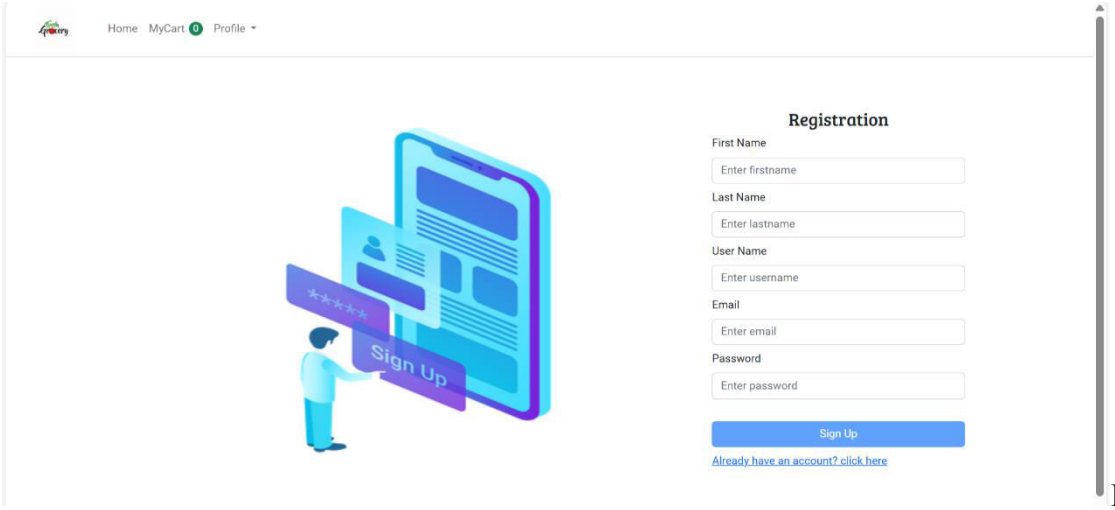

Fig.11. Register

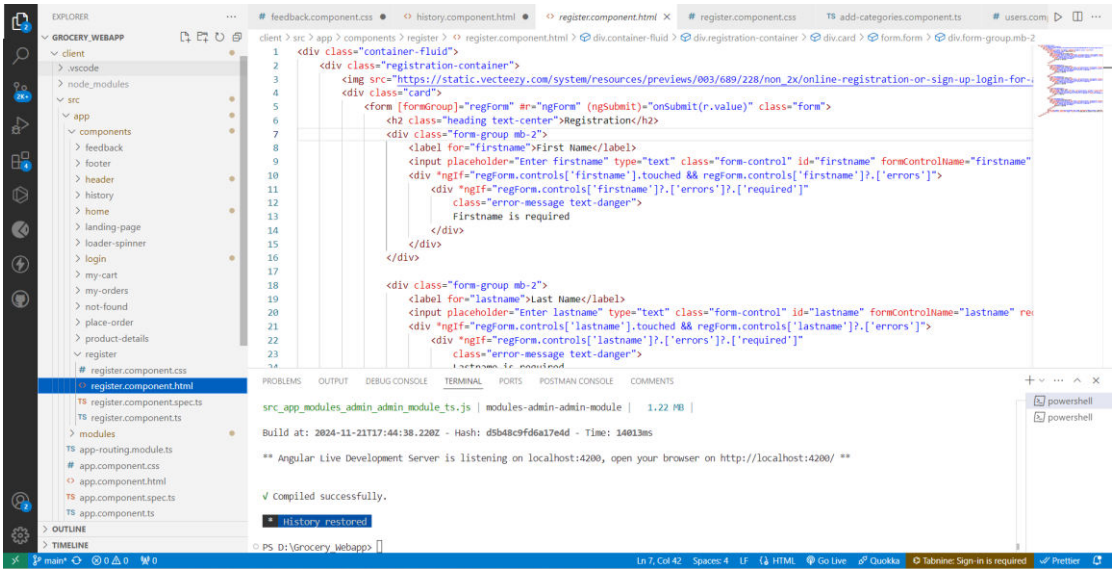

Fig. 32. register.component.html

12.15client/src/app/modules/admin/components/add-categories

- **Purpose:** Enables admins to create and manage product categories.
- **Details:** A form-based interface where admins can input category names, upload icons or images, and assign descriptions. The component integrates with the backend to store category information and displays a list of existing categories for easy management.

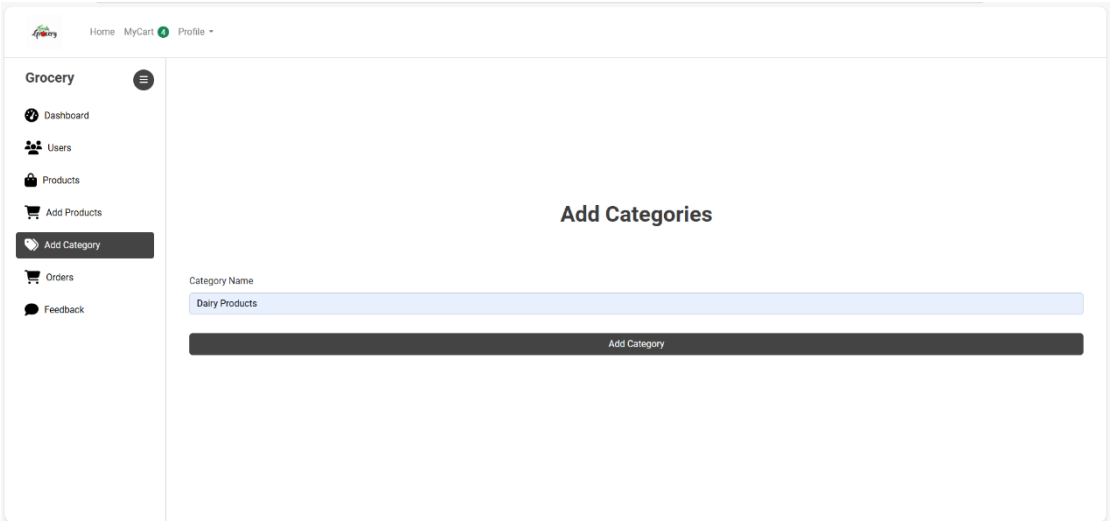

Fig. 12. Add- categories

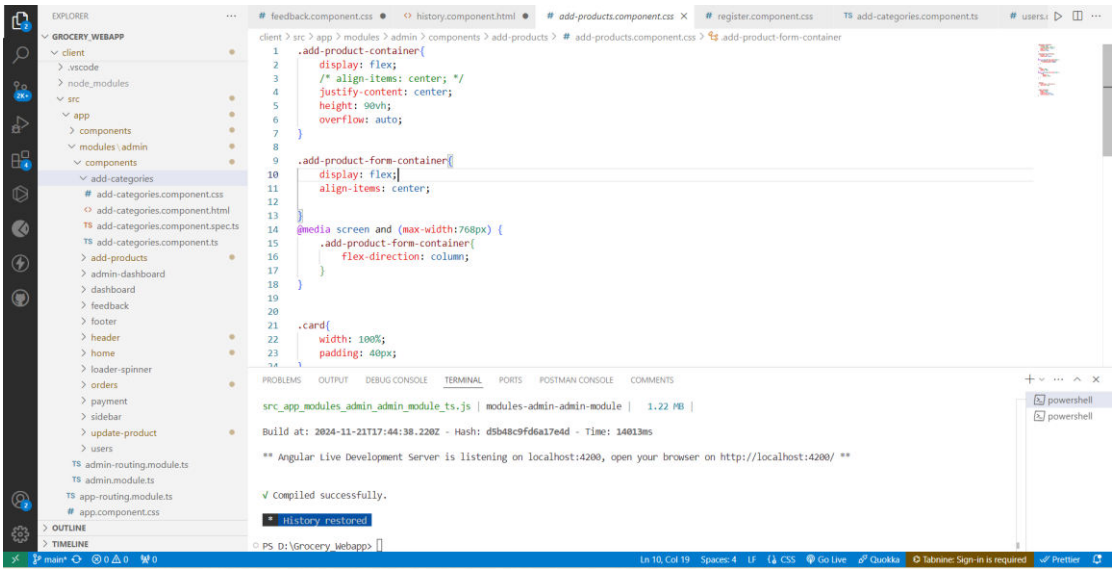

Fig. 33. Add-categories.component.html

client/src/app/modules/admin/components/add-products

- **Purpose:** Allows admins to add new products to the inventory.
- **Details:** The component includes multiple fields like product name, price, description, stock quantity, category selection (dropdown), and image upload. Sends data to the backend and displays confirmation upon successful addition.

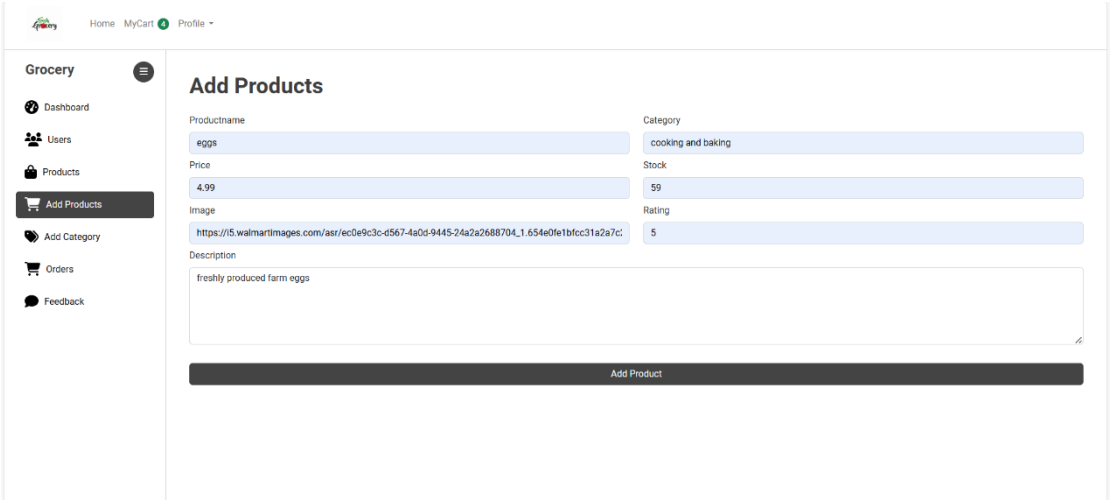

Fig. 13. Add-Products

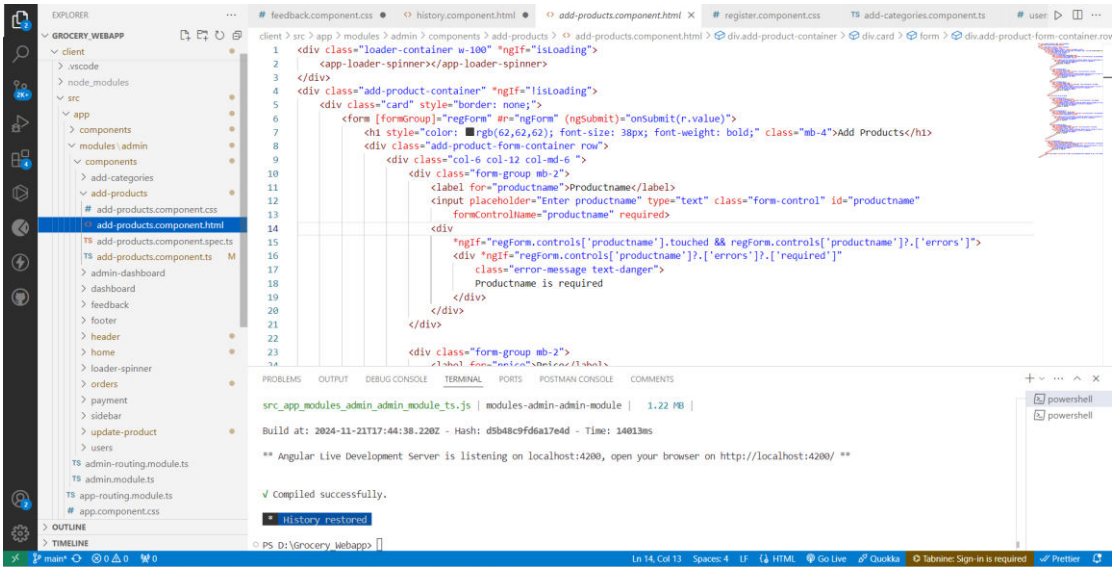

Fig. 34. Add-products.component.html

client/src/app/modules/admin/components/admin-dashboard

- **Purpose:** Serves as a centralized hub for admin activities.
- **Details:** Displays business-critical metrics like total revenue, active users, daily orders, and low-stock alerts. Includes visualizations like bar charts, line graphs, and KPIs, helping admins track performance at a glance.

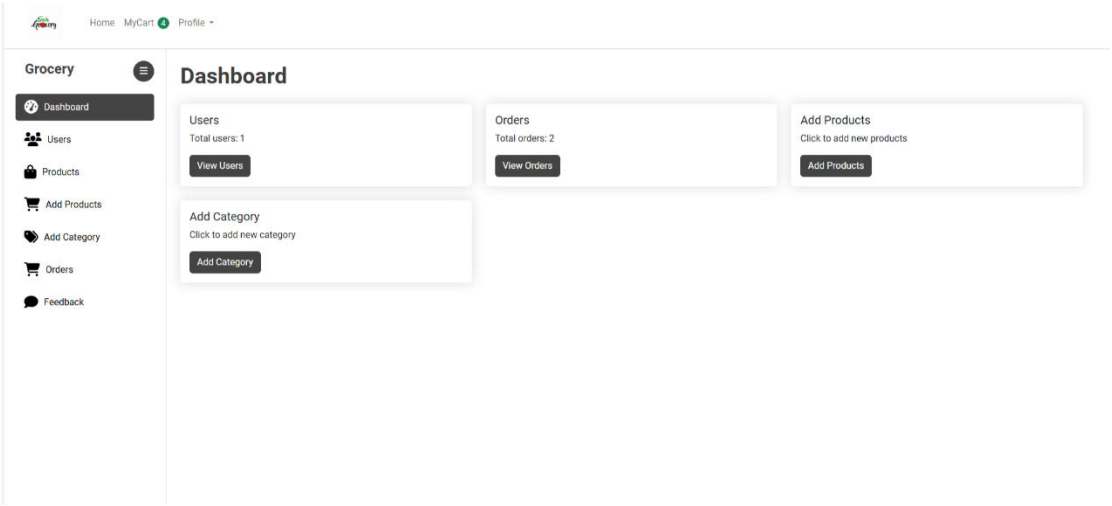

Fig. 14. Admin-dashboard

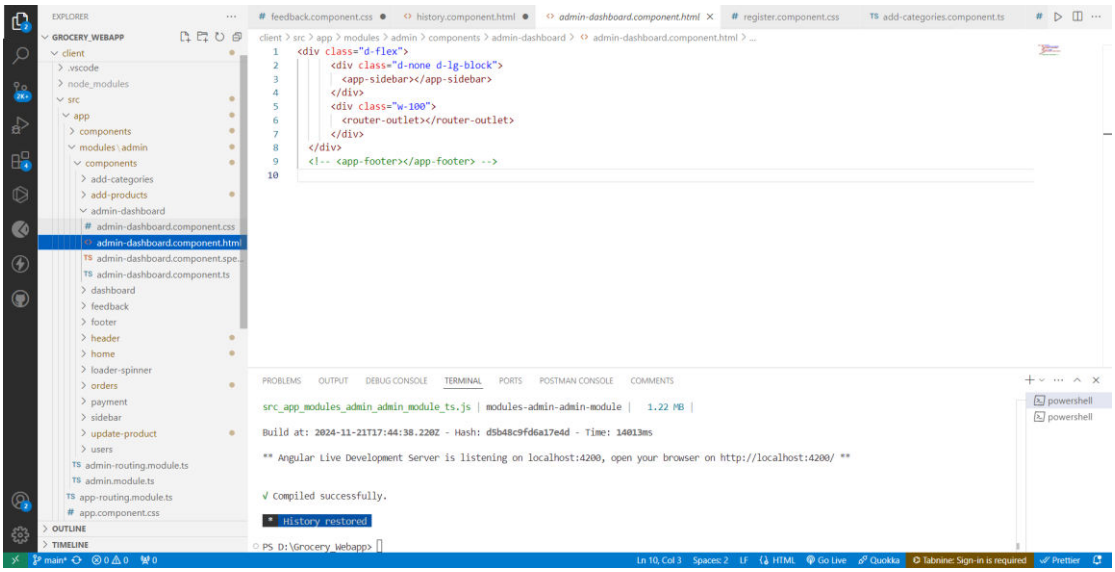

Fig. 35. Admin-dashboard.component.html

client/src/app/modules/admin/components/dashboard

- **Purpose:** Offers a more detailed view of admin tasks.
- **Details:** Highlights specific activities like recent orders, pending payments, or flagged feedback. May include sortable tables and action buttons for quicker decision-making.

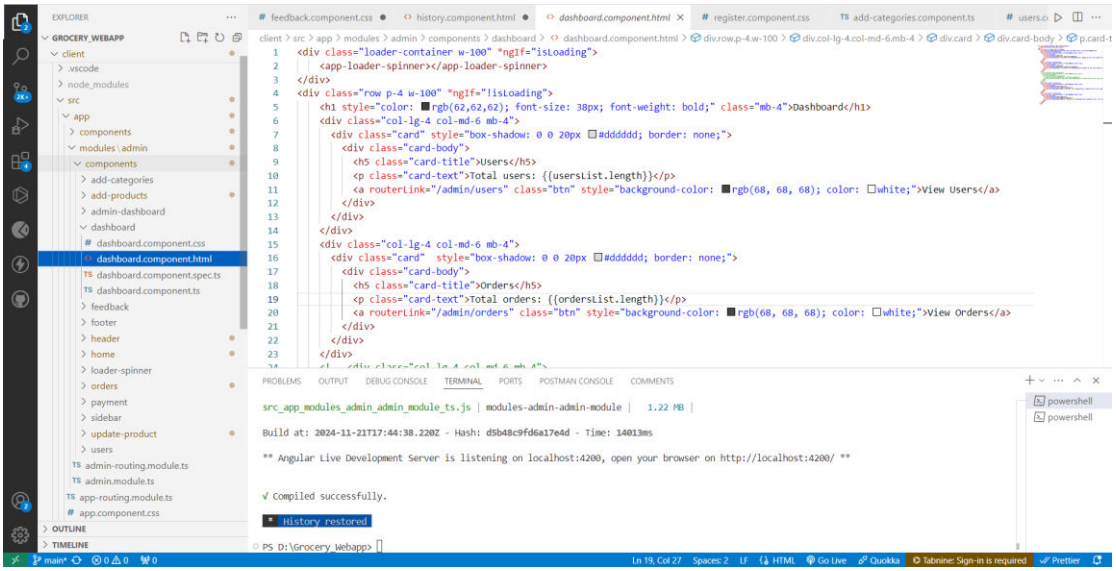

Fig. 36. Dashboard.component.html

client/src/app/modules/admin/components/feedback

- **Purpose:** Displays customer feedback for admin review.
- **Details:** A list view of feedback submitted by customers, categorized by ratings or tags. Allows admins to respond directly or escalate issues for further action.

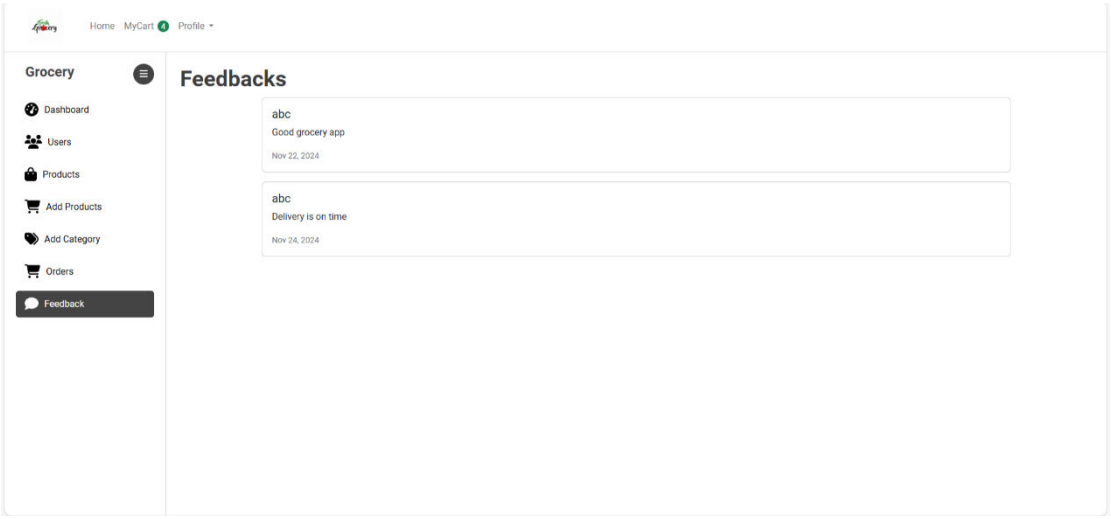

Fig. 15. Feedback

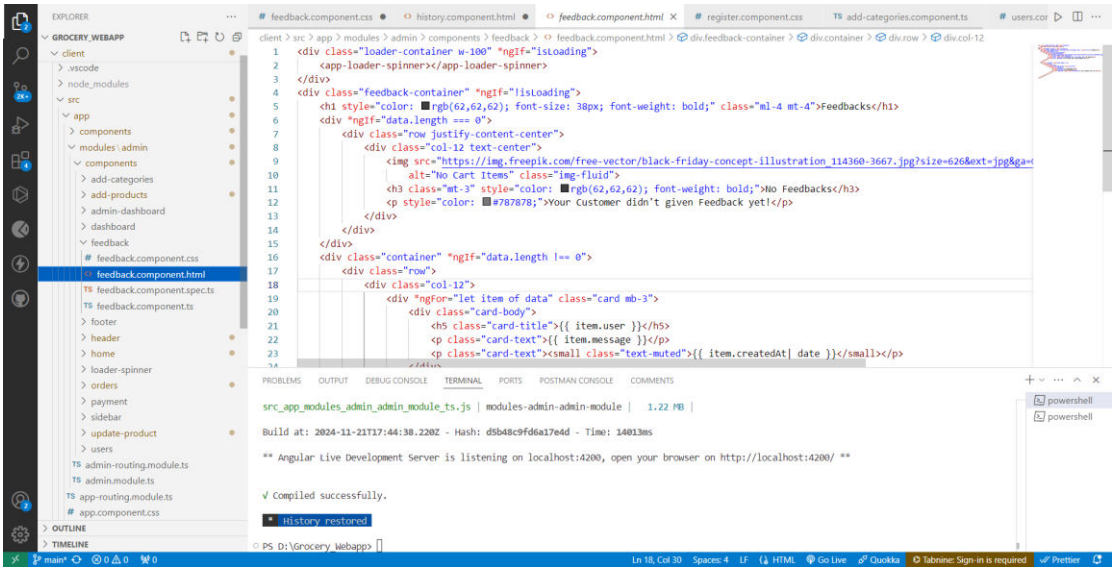

Fig. 37. feedback.component.html

client/src/app/modules/admin/components/footer

- **Purpose**: Provides admin-specific quick links and relevant information at the bottom of the admin dashboard.
- **Details**: The footer in the admin module might differ from the customer-facing footer by including admin-oriented links and shortcuts.
  Sections may include:
  - **Quick Access Links**: To settings, analytics, or system logs.
  - **Copyright Notice**: Information specific to the admin portal.
  - **Contact Support**: A link for admin technical support or troubleshooting documentation.

  Ensures consistent design across all admin pages, reinforcing the app's branding.

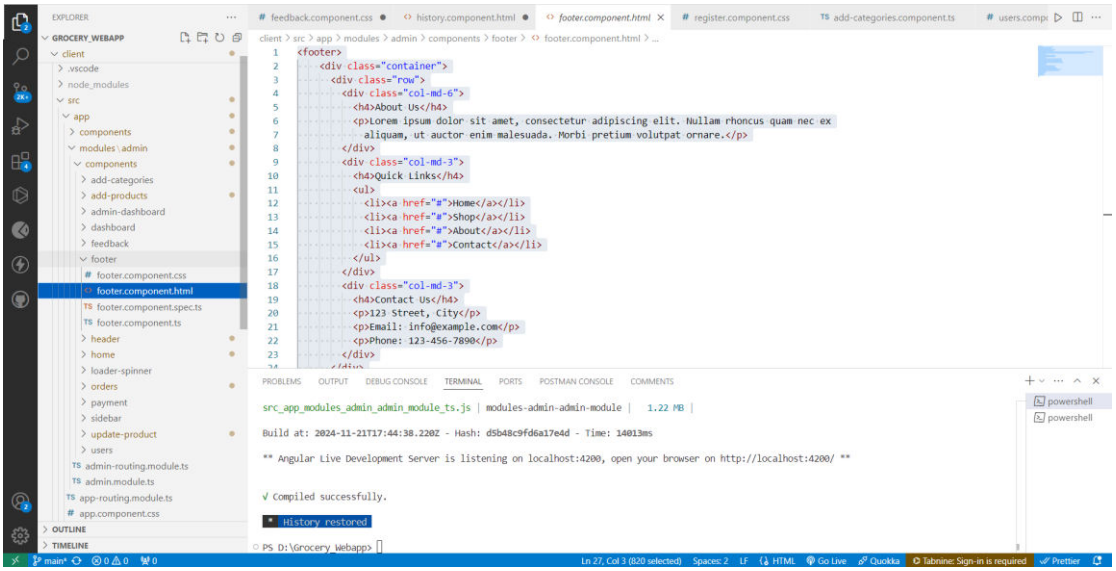

Fig. 38. Footer.component.html

client/src/app/modules/admin/components/header

- **Purpose**: Serves as a navigation bar for the admin module, providing quick access to critical tools and dashboards.
- **Details**: The header includes:
  - A **menu dropdown** or icons linking to different sections like Products, Orders, Users, and Analytics.
  - An **admin profile dropdown** for settings, logout, or switching accounts.
  - Notifications section to display system alerts or pending actions (e.g., low stock alerts, user feedback requiring attention).

  Could also include a **search bar** for locating products, orders, or categories quickly.

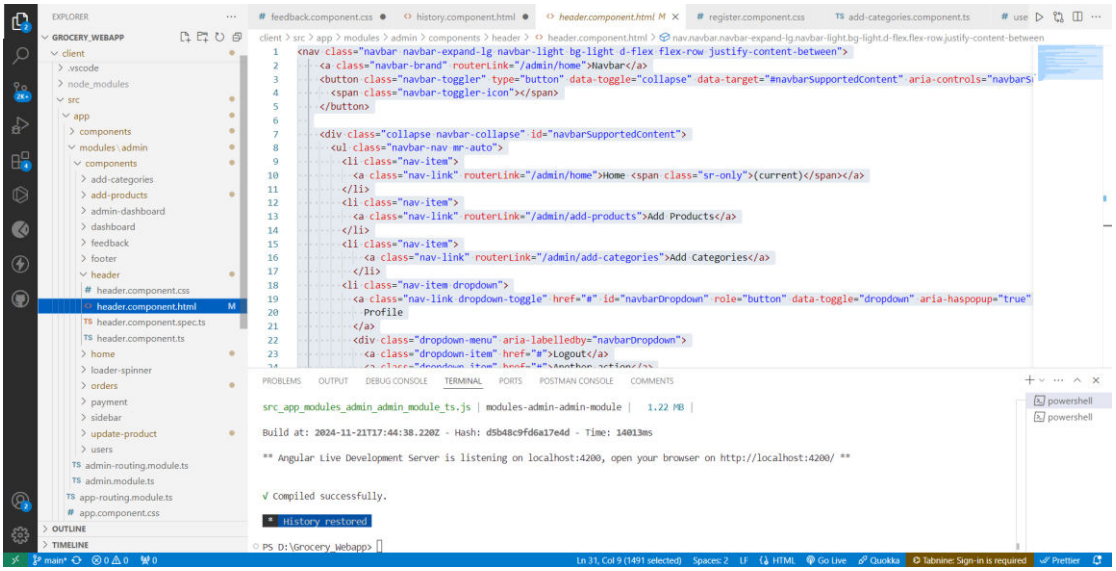

Fig. 39. Header.component.html

client/src/app/modules/admin/components/home

- **Purpose**: Acts as the landing page for admins upon logging into the admin panel.
- **Details**: The home page typically displays an **overview of the admin portal**, including:
  - **Key Metrics**: Total revenue, active users, pending orders.
  - **Quick Links**: Buttons to create new products, update stock, or monitor user activity.
  - **Recent Activity Feed**: Summarizes the latest actions (e.g., most recent orders, added products).
  - **Widgets/Charts**: Visual dashboards showing performance trends over time (e.g., daily sales chart, order breakdown by category).

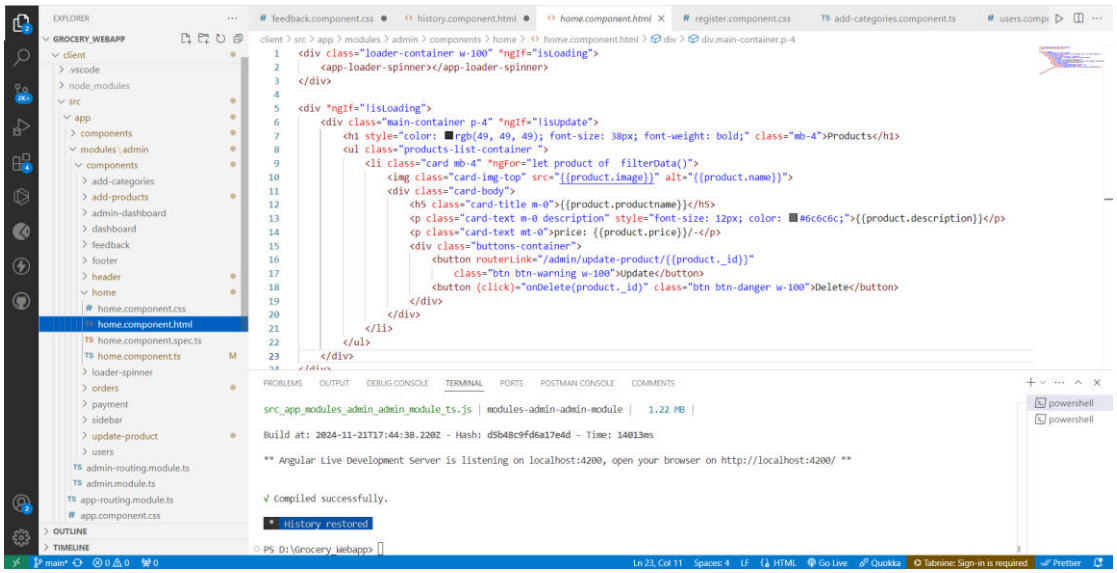

Fig. 40. home.component.html

client/src/app/modules/admin/components/loader-spinner

- **Purpose**: Enhances user experience by providing a loading animation during backend operations or data fetching.
- **Details**: Used to **prevent unresponsiveness** during:
    - Loading order or product data from the backend.
    - Submitting large forms (e.g., adding a new product or bulk updating stock levels).
    - Generating analytics or visual charts.

  Features a simple spinner or animation to visually indicate progress.Ensures a smooth and professional admin experience during heavy data interactions.

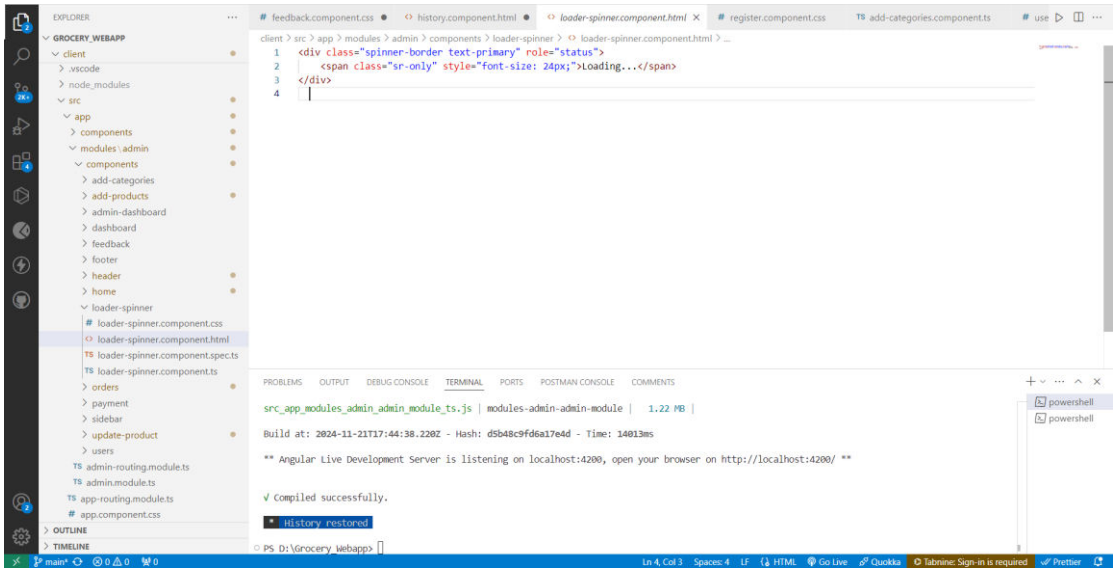

Fig. 41. loader-spinner.component.html

client/src/app/modules/admin/components/orders

- **Purpose:** Manages all customer orders.
- **Details:** Displays a comprehensive list of all orders with sortable columns for date, status, customer name, and amount. Includes actions like updating the order status (e.g., "Shipped"), generating invoices, or viewing detailed breakdowns.

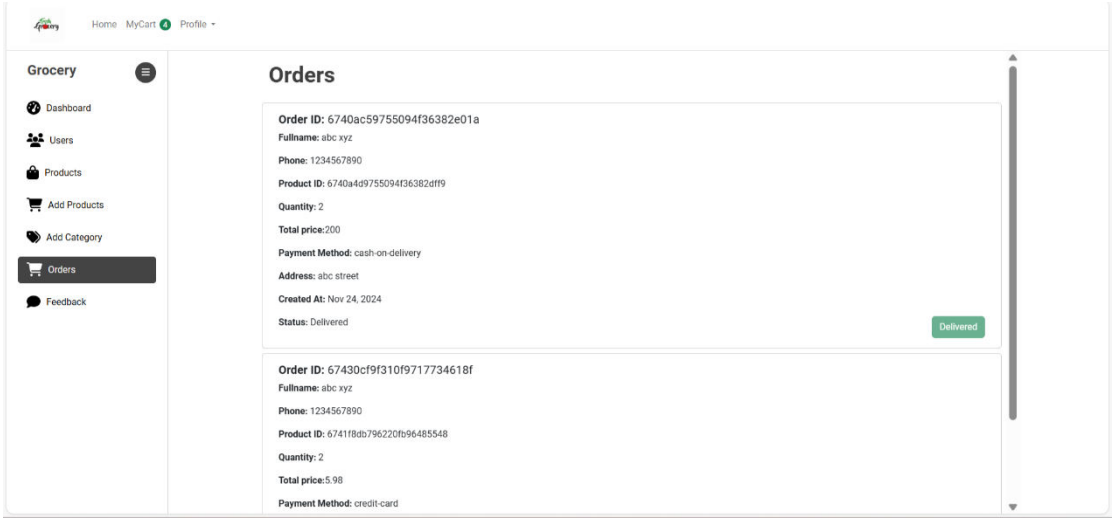

Fig. 16. Orders

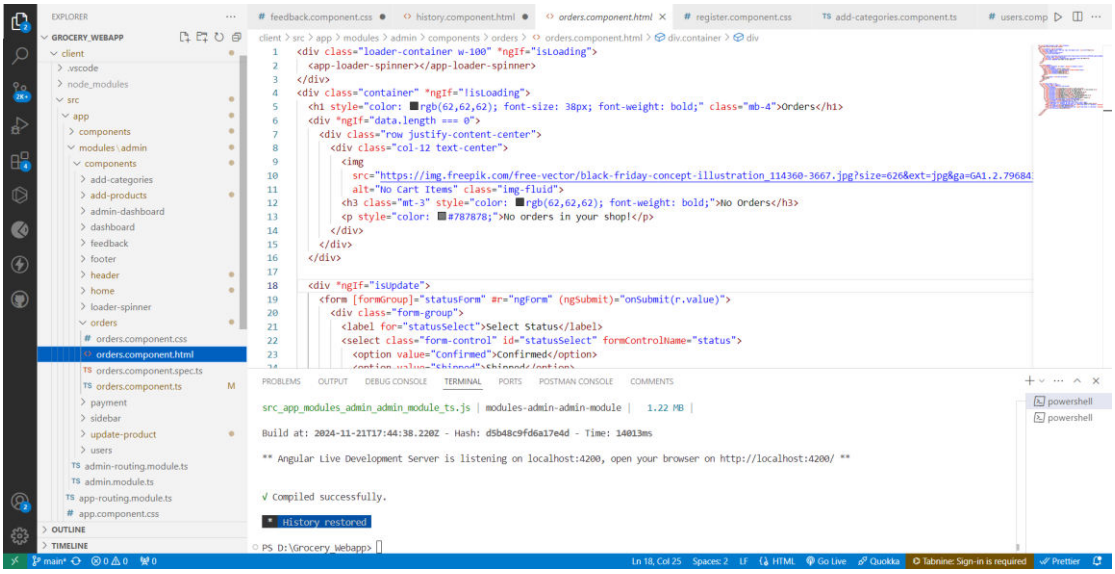

Fig. 42. Orders.component.html

client/src/app/modules/admin/components/payment

- **Purpose:** Tracks payment transactions made by customers.
- **Details:** Provides a detailed view of all payments, including transaction IDs, amounts, and payment methods. It may flag failed transactions for manual review and reconciliation.

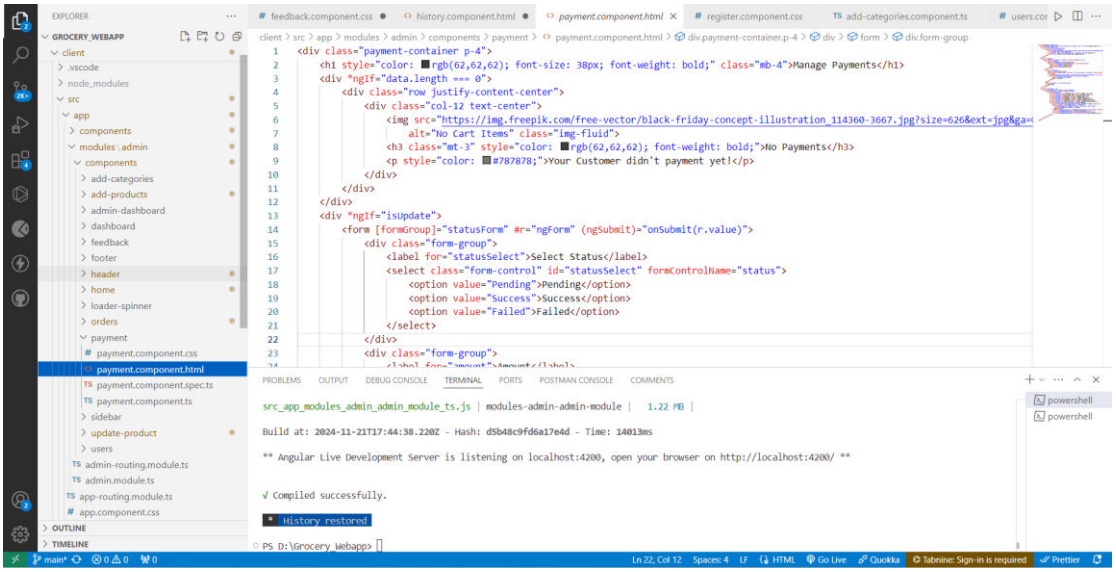

Fig. 43. payment.component.html

client/src/app/modules/admin/components/sidebar

- **Purpose:** Provides easy navigation for admins.
- **Details:** Includes links to all major admin functionalities, such as Dashboard, Products, Orders, Feedback, and Analytics. May include role-based navigation visibility.

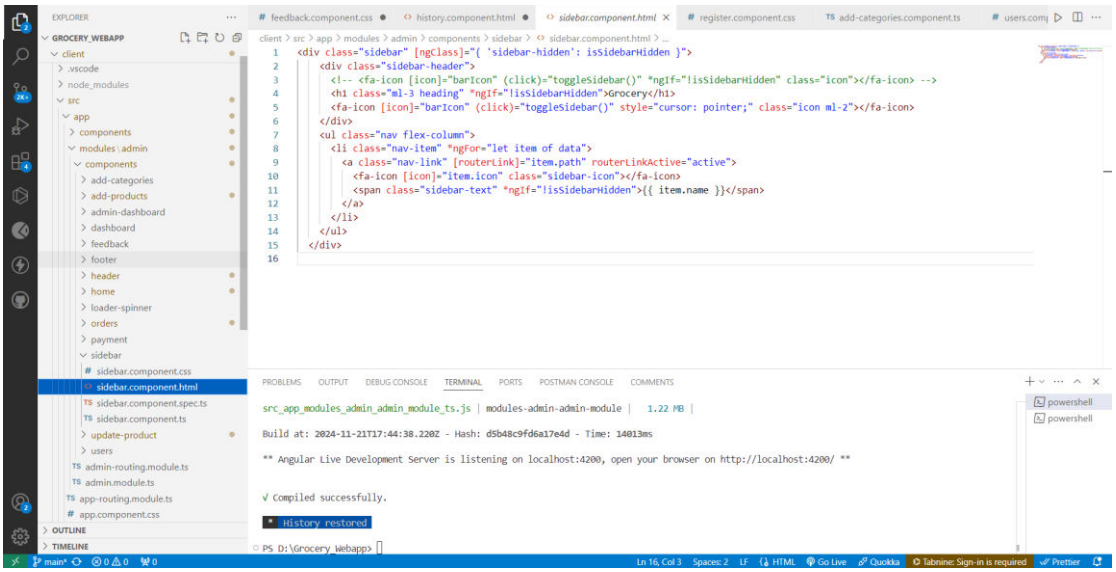

Fig. 44. sidebar.component.html

client/src/app/modules/admin/components/update-product

- **Purpose:** Enables admins to edit and update product details effectively.

- **Details:** Pre-fills product details for easy modification of name, price, stock, category, and description. Uses loader-spinner during backend data fetching and submission. Validates inputs in real-time and shows error alerts for invalid data or backend issues. Ensures smooth product updates with a responsive and user-friendly design.
-

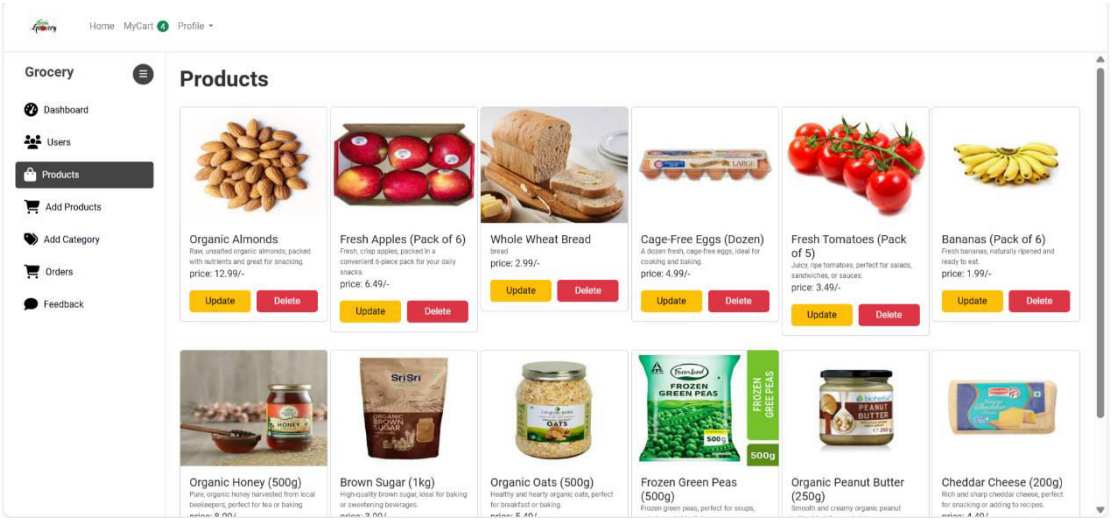

Fig. 17. Update-product

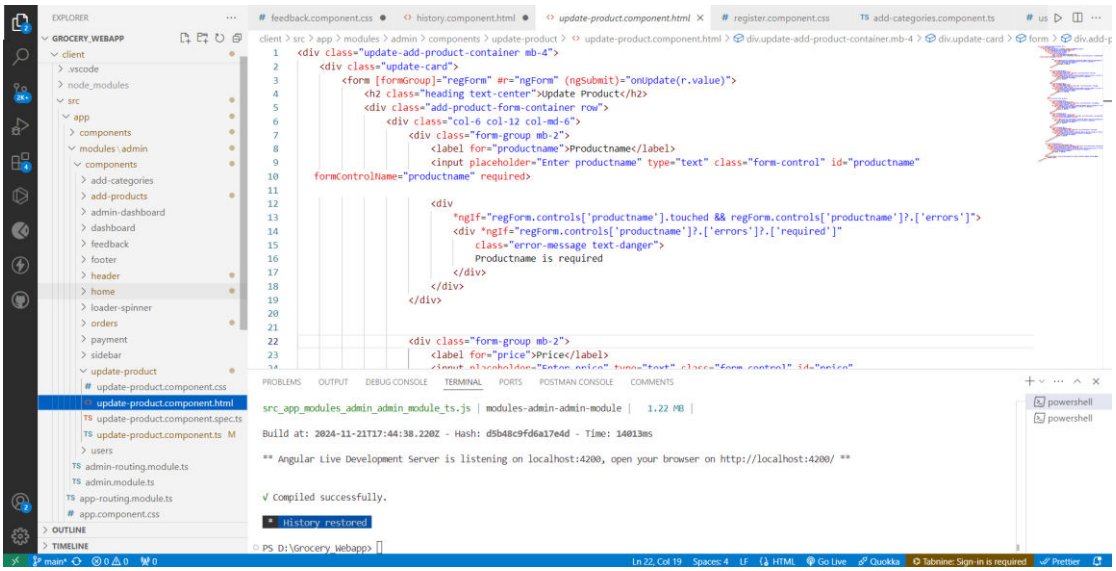

Fig. 45. Update-product.component.html

client/src/app/modules/admin/components/users

- **Purpose**: Manages the administrative functions related to users of the platform.
- **Details**: The **user management component** allows admins to:

- View all registered users (customers, other admins).
- Filter users by roles, activity status, or registration date.
- Perform actions like suspending a user, upgrading a role (e.g., customer to moderator/admin), or resetting passwords.

Displays details such as:

- User ID, Name, Email, Role, Last Login.
- Account status (Active, Suspended).

May include functionality to:

- **Create New Admins**: For delegating tasks.
- **Monitor User Activity**: Check recent purchases or feedback history.

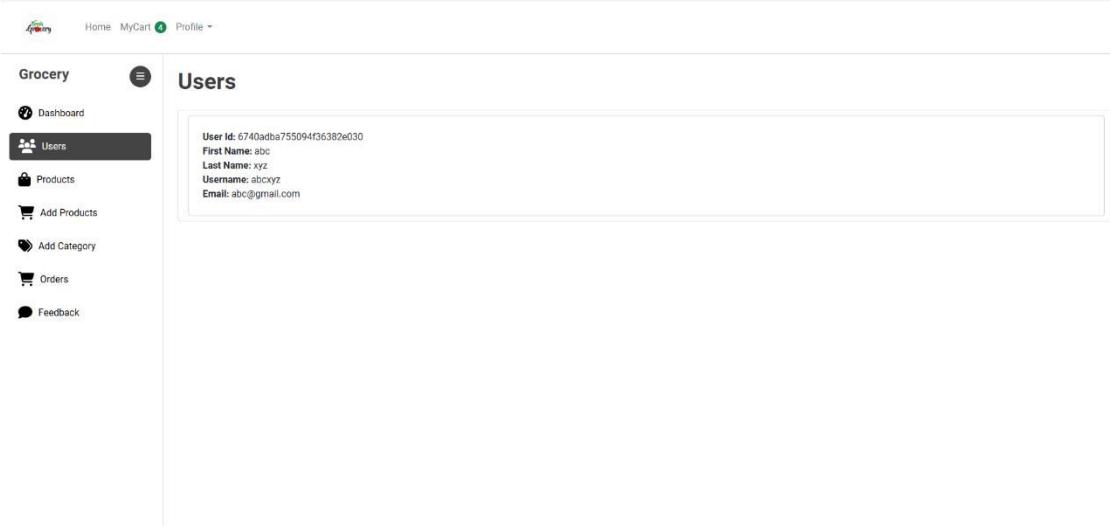

Fig. 18. Users

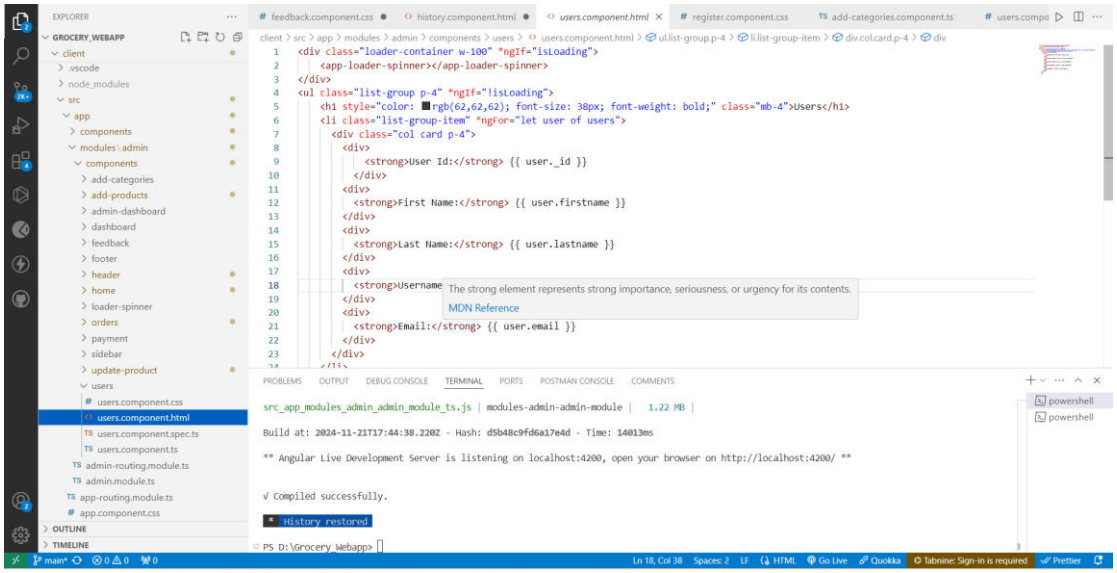

Fig. 46. Users.component.html

server/src/db

- **Purpose**: Manages database connectivity and configuration. Establishes and maintains the connection to MongoDB for the application.
- **Details**: Contains scripts to connect to the MongoDB instance (local or cloud-based like MongoDB Atlas). Includes error handling for connection issues to ensure the app doesn't crash.

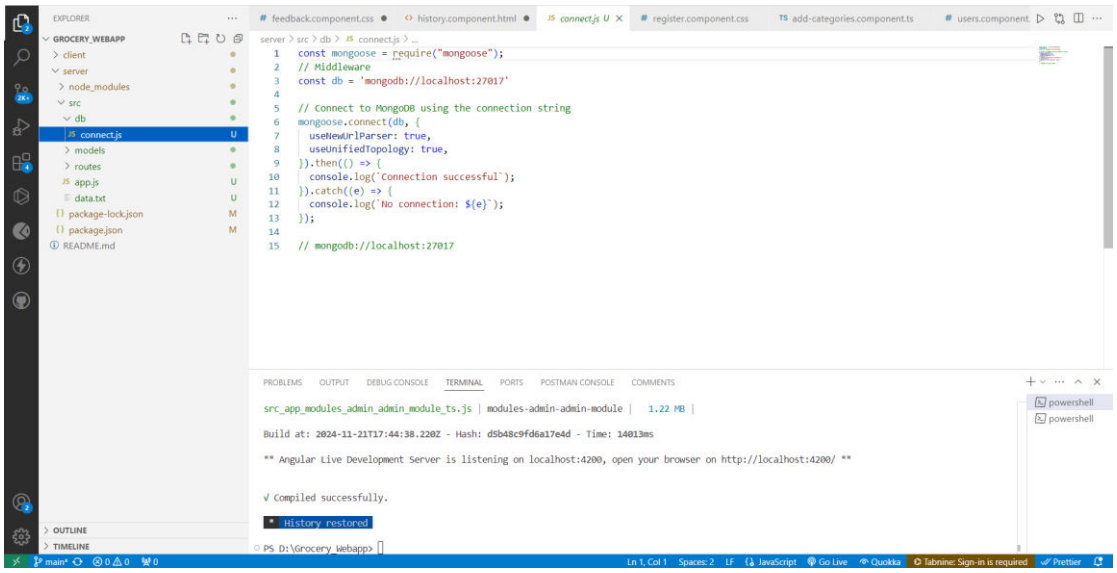

Fig. 47. Connect.js

server/src/models

- **Purpose**: Defines schemas and models for the application's data structure in MongoDB.
- **Details**: Includes Mongoose schemas for different collections like User, Product, Order, and Cart. Enforces consistency by defining field types, default values, and validation rules.

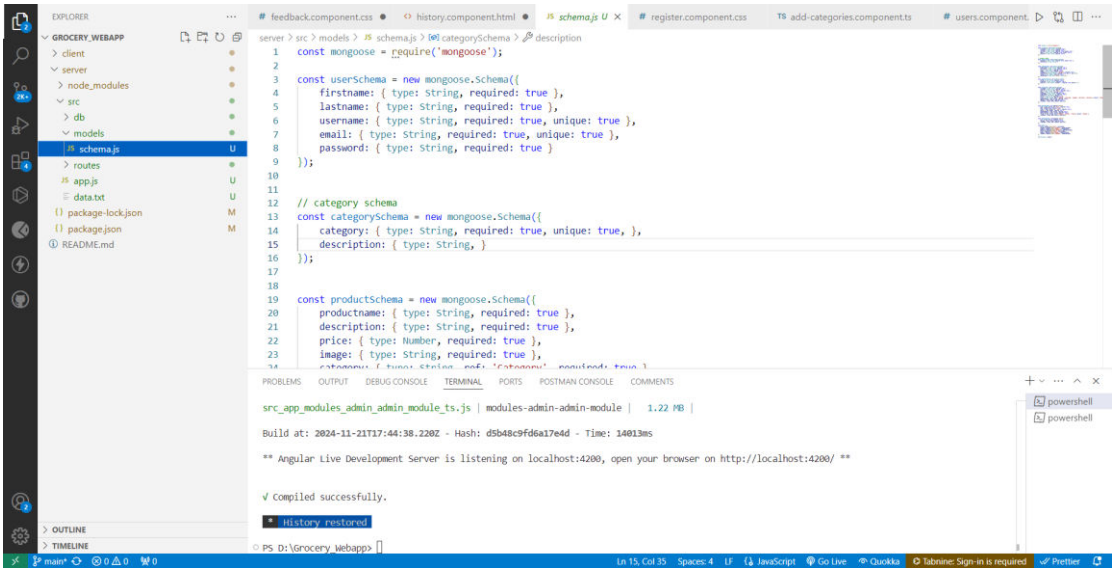

Fig. 48. Schema.js

server/src/routes

- **Purpose**: Defines API endpoints for client-server communication.
- **Details**: Organizes routes by feature (e.g., auth, products, orders). Maps HTTP methods (GET, POST, PUT, DELETE) to controller functions that handle business logic. Combines feature-specific routes into a centralized entry point (e.g., index.js).

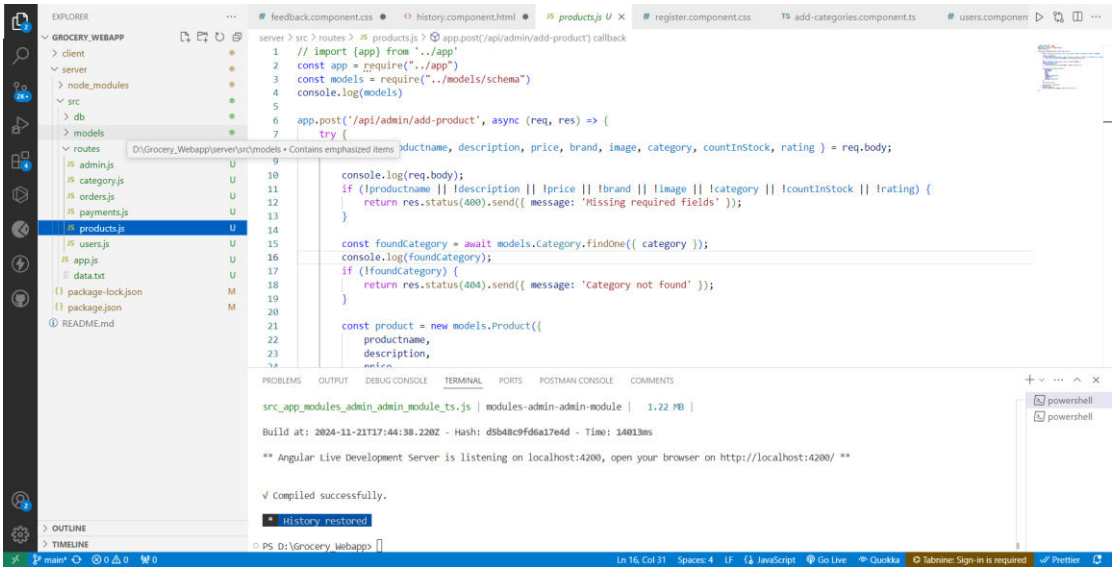

Fig. 49. Products.js

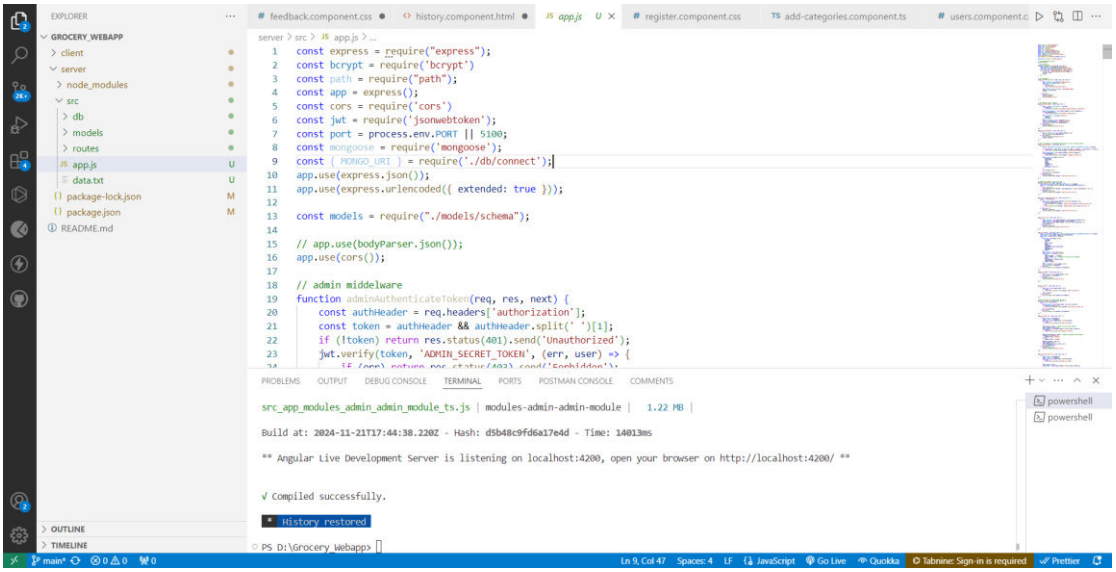

Fig. 50. App.js

13. KNOWN ISSUES

The **Product Listing Page** of the Grocery Web Application operates reliably, but several areas require optimization to enhance functionality and user experience. One notable issue is **token expiration management**, where users are logged out upon session expiration. This disrupts extended browsing sessions and could be improved by implementing a silent token refresh mechanism, allowing seamless session renewals without requiring user action.

The current **product filtering and sorting** features are limited in scope and performance. Filters can become slow as the database grows, and users are restricted to sorting by a single criterion. Adding database indexing for commonly queried fields like category and price, along with enabling advanced multi-criteria sorting options, would greatly improve the responsiveness and usability of the filtering system.

Mobile responsiveness also presents challenges, as layouts often overlap or misalign on smaller screens, leading to a poor user experience. Refactoring the design with CSS media queries and responsive frameworks such as Bootstrap can address these issues. Additionally, **image optimization** is lacking, with high-resolution images increasing page load times. Implementing lazy loading and responsive image delivery would improve performance across devices, especially on mobile networks.

Pagination inconsistencies also hinder user experience, as the system does not dynamically update when new products are added. Transitioning to an infinite scrolling model or dynamically updating pagination would resolve this issue. Furthermore, the lack of a **wishlist feature** limits user engagement by not allowing users to save products for later. Adding a wishlist functionality would enhance the convenience and retention of users.

Finally, **CORS configuration** issues occasionally block legitimate requests during development or testing phases, and **error messages** are generic and uninformative. Refining CORS policies to whitelist trusted origins and implementing descriptive error messages would improve usability and transparency. Addressing these known issues would significantly enhance the scalability, usability, and overall performance of the Product Listing Page.

14. FUTURE ENHANCEMENTS

The Grocery Web Application is well-built, but additional features can expand its capabilities and improve user experience. Key areas for enhancement include:

1. **Product Recommendation Engine**: Introducing a machine learning-based system to suggest products based on user behavior and purchase history. This would increase user engagement and drive sales.
2. **Mobile App Development**: Creating a cross-platform mobile app using frameworks like React Native or Flutter to provide a seamless shopping experience for mobile users.
3. **Advanced Admin Analytics**: Developing dashboards to offer detailed insights into sales, customer preferences, and product performance, enabling admins to make data-driven decisions.
4. **Multiple Payment Gateways**: Adding support for payment options like Apple Pay, Google Pay, and cryptocurrency to offer flexibility and improve checkout experiences.
5. **Inventory Management System**: Enhancing inventory features with automated stock alerts and supplier management for efficient stock handling.
6. **Real-Time Order Tracking**: Allowing users to track orders with live updates and estimated delivery times, boosting transparency and customer trust.
7. **Voice Search Integration**: Enabling voice search functionality to make product discovery more accessible and user-friendly.
8. **Loyalty Program**: Implementing a rewards system where users earn points for purchases, fostering customer retention and brand loyalty.
9. **AI-Powered Chatbot**: Deploying a chatbot to handle common inquiries and provide instant support, reducing customer service workloads.
10. **Multi-Language Support**: Adding language options to cater to a diverse user base, improving inclusivity and accessibility.

These enhancements will significantly enhance the application's functionality, scalability, and user experience, ensuring it meets evolving user expectations and industry standards.

CONCLUSION

The grocery website project demonstrates the successful implementation of modern web technologies to deliver a scalable, responsive, and user-centric application. By combining the power of the MERN stack and Angular, the project effectively addresses the complexities of e-commerce functionality, such as inventory management, real-time cart updates, and secure user authentication.

The backend, developed with Node.js, Express.js, and MongoDB, ensures efficient data handling and API functionality, while the front-end technologies—React for the MERN version and Angular for the alternative approach—offer seamless navigation, dynamic UI components, and a visually appealing interface.

This project also emphasizes the importance of clean code structure, reusable components, and efficient state management, ensuring that future enhancements and maintenance are hassle-free. Additionally, incorporating features like category filtering, search functionality, and a checkout page adds to its professionalism and market readiness.

Overall, this project stands as a comprehensive demonstration of full-stack and Angular development expertise, reflecting an ability to meet user expectations and business requirements in the competitive e-commerce space.

REFERENCES

React Documentation - https://reactjs.org/docs

Node.js Documentation - https://nodejs.org/en/docs

Express.js Guide - https://expressjs.com/en/guide

MongoDB Documentation - https://www.mongodb.com/docs

JWT Authentication - https://jwt.io/introduction

Cypress Testing - https://www.cypress.io

Bootstrap (for responsive design) - https://getbootstrap.com/docs

Postman (for API testing) - https://www.postman.com

MERN Stack Tutorials - https://www.freecodecamp.org/news/mern-stack-tutorial

Full-Stack React, TypeScript, and Node" by David Choi

MERN Quick Start Guide" by Eddy Wilson Iriarte Koroliova

Learning React: Modern Patterns for Developing React Apps" by Alex Banks and Eve Porcello

Node.js Design Patterns" by Mario Casciaro

MongoDB: The Definitive Guide" by Kristina Chodorow and Michael Dirolf

Pro MERN Stack: Full Stack Web App Development with Mongo, Express, React, and Node" by Vasan Subramanian