1. Introduction

• **Project Title:** SHOPSMART

Team ID : LTVIP2025TMID47074

Team Size : 4

Team Leader : Bantupalli Sri Kavya

Team member : Jinaga Swati

Team member : Karri Anjali

Team member : Balla Koteswara Rao

2. Project Overview

• Purpose:
SHOPSMART is an advanced online e-commerce web application developed using the MERN (MongoDB, Express.js, React.js, Node.js) stack. The primary purpose of this project is to simulate the core features of a modern online shopping website, enabling users to explore, select, and purchase products in a user-friendly digital environment.

The system provides a complete and robust workflow for customer interaction, starting from account creation to browsing through a dynamic catalog of products. Users can filter and search products based on categories, add items to their cart, modify quantities, and place orders securely. All of this is achieved with seamless interaction between the frontend and backend services.

Beyond customer functionality, SHOPSMART also includes an integrated admin panel that allows authorized administrators to manage the entire product database, track orders, and oversee user activity. This ensures that store management is efficient and scalable. The goal is to replicate real-world ecommerce operations with a focus on smooth performance, security, and an intuitive interface.

By developing SHOPSMART, the project aims to showcase the practical implementation of full-stack development principles, API integration, state management, secure authentication, and real-time database interactions.

- **Features:**

- **User Registration and Login**
  A secure user registration and authentication system using JWT, with password hashing and validation. Only registered users can access checkout and order history.

- **Product Listing with Categories**
  Products are displayed on the homepage with category labels such as Electronics, Clothing, Accessories, etc. Each product has a thumbnail, price, and short description.

- **Search and Filter Functionality**
  Users can search for products using keywords or filter products based on price, category, or popularity. This improves the user experience by enabling faster navigation.

- **Shopping Cart and Checkout System**
  Logged-in users can add items to a cart, update quantities, and proceed to a checkout screen to finalize their purchase. Cart data is dynamically updated and synced with the backend.

- **Order Placement and Tracking**
  Once the user confirms their cart, an order is generated, stored in the database, and associated with the user. Users can track their orders through a dedicated order history page.

- **Admin Dashboard for Managing Products and Users**
  Admins have access to a separate section of the application where they can create, edit, or delete product entries, as well as manage registered users and their roles.

- **Responsive Design for All Screen Sizes**
  The UI is fully responsive, adjusting elegantly to different screen sizes including desktops, tablets, and mobile devices.

- **Secure JWT-Based Authentication**

All protected routes require valid JWT tokens for access. Authentication is implemented using middleware in the backend and context/state in the frontend. Tokens are stored securely using HTTP-only cookies.

3. Architecture

• **Frontend:**

The frontend of SHOPSMART is developed using **React.js**, one of the most popular JavaScript libraries for building interactive user interfaces. React provides a component-based architecture that ensures reusability and maintainability of code. This structure helps to scale the frontend easily as the application grows.

Routing in the application is handled using **React Router DOM**, allowing for seamless navigation between different views such as the home page, product details, cart, login/register, admin dashboard, etc. Each page is mapped to a specific route that renders the appropriate React component, and route protection is enforced through custom logic that restricts access to admin or authenticated users.

To manage the application's global state, the **React Context API** is used in combination with custom hooks. This setup handles authentication state, cart data, and user details across different components, removing the need for excessive prop drilling. The state management flow ensures that changes in user or cart status are reflected in real time across all dependent UI components.

For communication between the frontend and backend, **Axios** is used as the HTTP client library. Axios is configured with base URLs and headers using interceptors for clean and centralized API handling. Each API request from the frontend sends data to the appropriate backend route and handles the response or error accordingly, providing feedback to the user.

The component hierarchy follows a clean structure with separation between **pages**, **components**, and **utilities**. Pages represent full views, like "Home" or "Product", while components represent reusable UI blocks like "ProductCard", "Navbar", or "Footer". Utility files include API handlers and context providers.

Styling is done using a combination of CSS Modules and custom stylesheets, ensuring each component has scoped styling and consistency across the application.

The frontend is fully **responsive**, utilizing Flexbox and Grid for layout and media queries to adjust elements on smaller screen devices. This makes the UI suitable for mobile, tablet, and desktop views. Accessibility considerations such as ARIA roles, semantic HTML, and keyboard navigation are partially implemented to improve user experience.

- **Backend:**

The backend of SHOPSMART is built using **Node.js** with the **Express.js** framework. It provides the logic and data management necessary to support the frontend UI and performs all the operations related to authentication, data validation, user management, product CRUD operations, and order processing.

The Express application is structured in a modular format with folders such as **routes**, **controllers**, **models**, **middleware**, and **config**. This structure enables separation of concerns and better organization of backend functionality.

- **Routes** define the HTTP endpoints that the frontend interacts with, such as /api/users, /api/products, and /api/orders.

- **Controllers** handle the logic of what happens when a particular endpoint is called, such as creating a user, fetching product data, or processing an order.

- **Models** define the structure of the data using Mongoose schemas.

- **Middleware** is used for tasks like authenticating JWT tokens, checking admin access, logging, and handling errors.

Authentication in the backend is handled using **JWT (JSON Web Tokens)**. When a user logs in successfully, the server issues a signed token which the client stores in HTTP-only cookies for security. Each protected route in the backend checks the validity of this token to allow or deny access to private resources. Admin-specific routes have an additional middleware layer that checks the user's role.

Error handling is implemented using centralized middleware that captures synchronous and asynchronous errors and sends appropriate responses back to

the client in a structured format. Request data is validated using both manual checks and Mongoose validation to ensure reliability and integrity of stored data.

The backend server is configured to run on port 5000 during development and uses the dotenv package to securely load environment variables, including database URIs, secret keys, and port settings.

- **Database:**

The database layer uses **MongoDB**, a flexible NoSQL document database, to store and manage all application data including users, products, orders, and cart information. MongoDB is integrated with the backend using **Mongoose**, an ODM (Object Document Mapper) that provides a schema-based solution to model the application data.

The **User schema** stores details such as username, email, password (hashed), role (user/admin), and timestamps. The **Product schema** includes name, image URL, price, category, stock quantity, and description. The **Order schema** keeps track of user ID, ordered products, payment status, shipping details, and order history.

Mongoose also supports schema validation and pre/post hooks, which are used to hash user passwords before saving them to the database and to perform cleanup operations on cascading deletions. Relations between users and orders are modeled using references, enabling efficient data retrieval via population queries.

MongoDB can be hosted either locally for development or on **MongoDB Atlas** for production. The connection configuration is maintained in a dedicated file and imported into the main server file to establish a persistent and secure link to the database. The structure and indexing of collections are optimized for read and write operations to ensure smooth performance under high load.

The database supports dynamic data updates, which means that when a user adds a product to the cart or places an order, the corresponding documents are updated in real time. All critical operations are secured and validated before being committed to the database.

4. Setup Instructions

- Prerequisites:

- Node.js

- MongoDB

- Git

- Installation:

1. Clone the repository: git clone https://github.com/BATTULASYAMALA/ShopSmart-Your-Digital-Grocery-Store-Experience

2. Navigate to client and install dependencies: cd client && npm install

3. Navigate to server and install dependencies: cd ../server && npm install

4. Create .env files in both client and server directories for configuration.

5. Set up the MongoDB URI and JWT secret in .env.

5. Folder Structure

- Client:

- /src/components: Reusable UI elements like Header, Footer, Product Cards

- /src/pages: Individual pages such as Home, Login, Register, Product, Cart

- /src/context: Context API setup for managing state globally

- /src/api: Axios configurations and API service functions

- /src/App.js: Root component where routes and layout are handled

- Server:

- /routes: Route files for user, product, and order APIs

- /controllers: Controller files containing route logic

- /models: Mongoose schemas for Users, Products, Orders

- /middleware: Middleware for JWT auth and error handling

- /config: MongoDB connection setup and environment configuration

6. Running the Application

• Frontend:
Open a terminal, navigate to the client directory and run:
npm start

• Backend:
Open another terminal, navigate to the server directory and run: npm
start

7. API Documentation

- User APIs:

- POST /api/users/register → Register a new user

- POST /api/users/login → Authenticate and receive token

- GET /api/users/profile → Get logged-in user profile

- Product APIs:

- GET /api/products → Fetch all products

- POST /api/products → Add a new product (admin only)

- PUT /api/products/:id → Edit product by ID · 　　DELETE

  /api/products/:id → Delete product by ID

- Order APIs:

- POST /api/orders → Place an order

- GET /api/orders/myorders → View user's orders

8. Authentication

- JWT-based authentication is implemented.
- Upon login, users receive a token which is stored in HTTP-only cookies.
- Private routes are protected using middleware that checks for valid tokens.
- Role-based access is implemented for admin users.

9. User Interface

- Clean and modern interface using CSS Flexbox and Grid.
- Components styled with CSS and utility classes.
- Fully responsive layout for both mobile and desktop views.
- Navigation bar, product cards, modals, form validations, and interactive UI.

10. Testing

- **Manual Testing of Major User Flows**
Manual testing was conducted thoroughly across all critical functionalities of the application. These include user registration and login, product browsing, cart operations, checkout, order placement, and admin functionalities like product management. Each flow was tested for both valid and invalid input data to ensure the UI handles errors gracefully and the backend responds appropriately.

- **API Testing Using Thunder Client and Postman**
All backend API endpoints were individually tested using API testing tools like Thunder Client and Postman. These tools allowed for rapid testing of RESTful routes including user authentication, product listing, order processing, and admin-level operations. Different HTTP methods (GET, POST, PUT, DELETE) were tested with correct and incorrect headers, tokens, and body payloads to confirm the behavior under various edge cases.

- **Form Validations and Input Testing**
All input fields across the application (login forms, registration, product creation, etc.) were tested to ensure client-side validation works correctly. Fields were tested for required inputs, type constraints (e.g., number fields), and validation feedback.

- **Unit Testing Using Jest**

Jest was used to implement basic unit tests for selected utility functions and context handlers. These include tests for cart calculations, authentication logic, and component rendering. Though limited in scope, the unit tests demonstrate the structure and potential for future test scalability.

• **Token Expiry and Authorization Testing**

JWT authentication was tested for both valid and expired tokens. Protected routes were verified by removing or manipulating the token to ensure proper 401/403 error responses. Admin-only routes were tested with both user and admin roles to ensure proper access control.

• **Error Handling and Edge Case Testing**

Error responses from the backend were evaluated for appropriate HTTP status codes and consistent messaging. Tests included submitting blank data, invalid product IDs, and unauthorized access attempts.

• **Responsiveness Testing**

Manual testing was conducted across various screen sizes and devices using browser dev tools to ensure that the UI maintains consistency and functionality across resolutions.

11. Screenshots or Demo

Live demo: https://shopsmart-15.netlify.app/
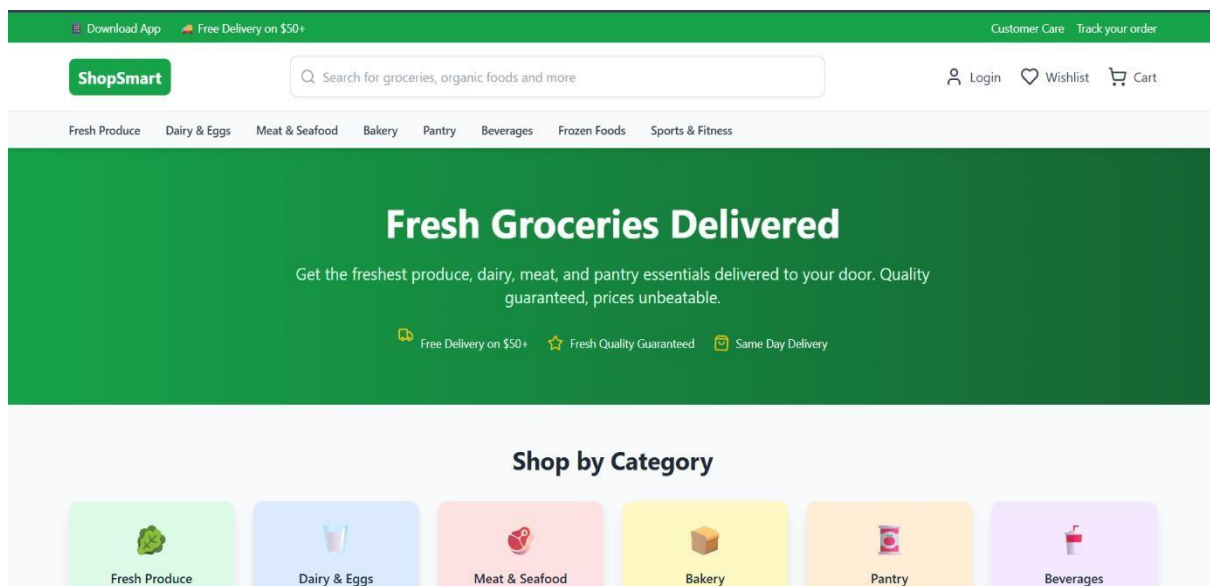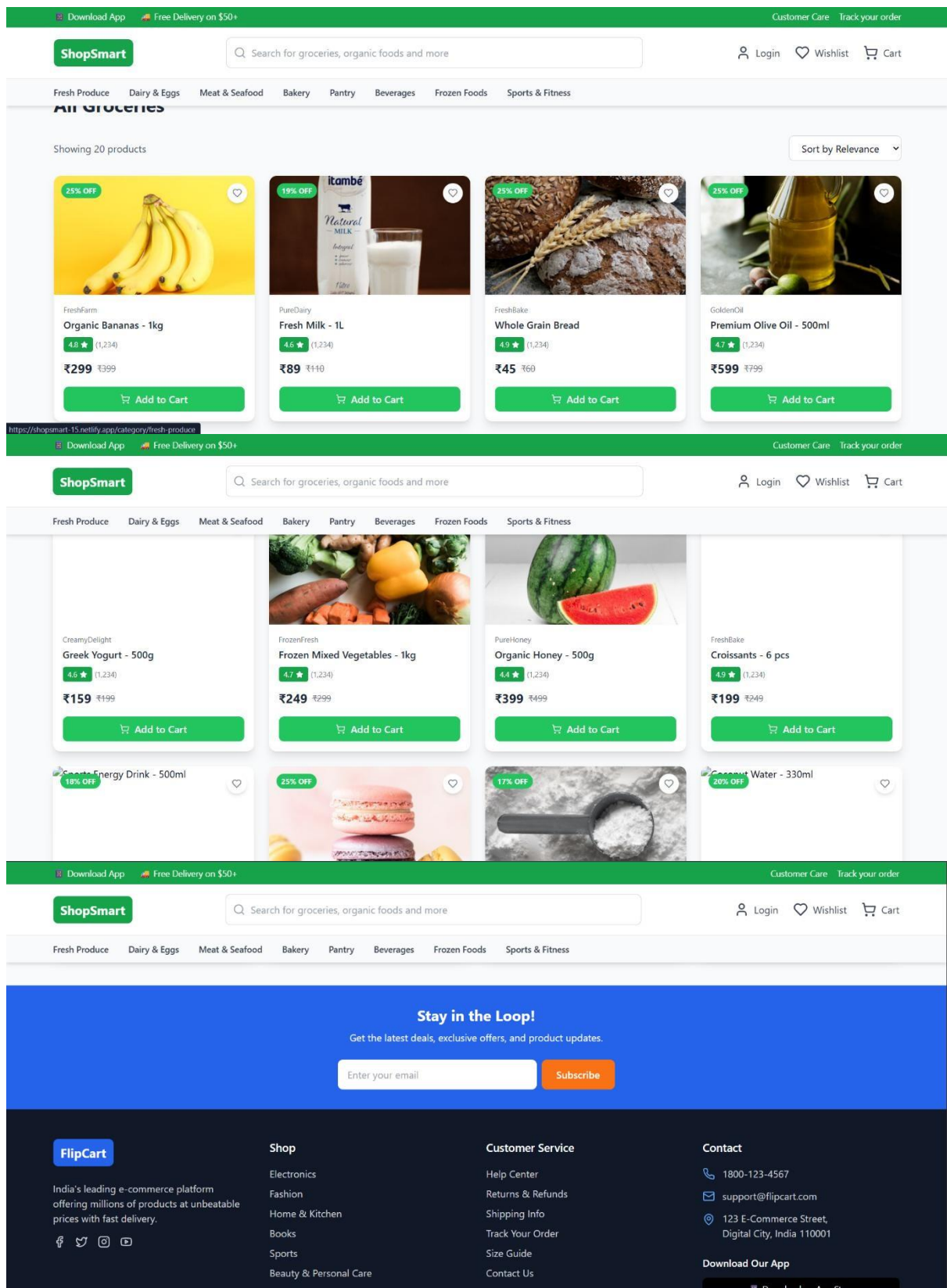
12. Known Issues

• Some responsiveness issues may occur on older devices
• Lack of full test coverage for edge cases
• Product filtering may lag with very large data sets

13. Future Enhancements

• Payment gateway integration (Stripe/Razorpay)
• Add wishlist functionality
• Improve search with autocomplete and suggestions
• Add product reviews and ratings
• Pagination and sorting
• Admin analytics dashboard with charts