Full Stack Development with MERN

Project Documentation

1. INTRODUCTION:

Project Title: Grocery Web App- An E-Commerce Website

Team Members-NM ID

M.Ramya - 12183AB6ADA76CA916B4C568D4CF672E

Janani.M - E51390F019C8A6DB48E4EED6A048FD5B

R. Kiruthika - 6E175EFEAB5E941869B4E1010EC2AAA0

M.Janani - 74339937B5DF212805E4528695330643

S. Nivetha - FE96CE6ECB42C5EAF5549025D4209856

---

2. PROJECT OVERVIEW:

PURPOSE:

The grocery web app aims to simplify online grocery shopping by providing a user-friendly platform for customers to browse, select, and purchase essential items. It seeks to enhance the convenience of managing daily needs, streamline inventory management for sellers, and promote seamless customer experiences.

- Create a responsive and intuitive interface for customers.
- Facilitate efficient inventory and order management for sellers.
- Ensure secure and scalable operations using the MERN stack.
- Provide a seamless and engaging user experience with real-time updates.

FEATURES:

1.User Authentication:

- Secure login and registration system using JWT for authentication.

2.Product Management:

- Categorized product listing with search and filtering options.

- Admin dashboard for adding, updating, and removing products.

3.Shopping Cart:

- Add to cart, update quantity, and view total cost.
- Persistent cart data for logged-in users.

4. Order Processing:

- Easy checkout process with multiple payment options.
- Order tracking with real-time status updates.

5.Responsive Design:

- Fully optimized for mobile and desktop users.

6.Backend Scalability:

- Built with Node.js and Express.js for handling large data efficiently.

7.Real-Time Updates:

- Notifications for order updates and stock availability using WebSocket.

8.Database Management:

- MongoDB for efficient and scalable data storage, ensuring secure transactions.

9.Customer Support:

- Integrated chat support for resolving queries and complaints.

---

3. ARCHITECTURE:

Our Grocery Web App architecture is designed to ensure scalability, efficiency, and seamless user experiences. It integrates a responsive frontend, robust backend, and a secure, scalable database.

- ## FRONTEND:

The frontend of the Grocery Web App is built using React.js, featuring a component-based architecture that ensures modularity and reusability. It provides an interactive and responsive user interface with dynamic features like product browsing, cart management, and seamless navigation using React Router. State management is handled through the Context API or Redux to efficiently manage global states such as user data, cart items, and filters. Modern styling tools like Tailwind CSS or Bootstrap ensure the app is visually appealing and responsive across devices.

BACKEND:

The backend is powered by Node.js and Express.js, forming a robust server-sid structure that manages API endpoints for user authentication, product retrieval, cart updates, and order processing. Middleware is used for authentication with JWT, data validation, and error handling. Additionally, real-time updates for order tracking and notifications are implemented using Web Sockets or Socket.IO, ensuring a dynamic user experience.

DATABASE:

The database utilizes MongoDB to store and manage data securely and efficiently. The schema design includes collections for users, products, and orders. User data includes login credentials, profiles, and order history, while product details cover categories, prices, stock levels, and images. The order collection tracks user purchases, product details, and order statuses. CRUD operations and aggregation pipelines enable seamless data retrieval, updates, and reporting, ensuring smooth interactions between the database and the app. Together, this architecture creates a scalable, secure, and user-friendly platform for grocery shopping.

---

4. SETUP INSTRUCTION:

- ## PREREQUISITE:

Before setting up the application, ensure the following software dependencies are installed:

1. Node.js: Install the latest LTS version (recommended) from [Node.js official website](https://nodejs.org/).

- Verify installation:

```bash
node -v
npm -v
```

2. MongoDB: Install MongoDB Community Edition or use a cloud-hosted service like [MongoDB Atlas](https://www.mongodb.com/atlas).

- Verify installation:

  ```bash

  mongodb --version

  ```

3. Package Manager:

   - **NPM**: Comes with Node.js installation.

4. Git: Install Git for version control from [Git official website](https://git-scm.com/).

   - Verify installation:

     ```bash

     git --version

     ```

5. Code Editor: Install a code editor like [Visual Studio Code](https://code.visualstudio.com/) for development.

6. Environment Variables Management: Use a tool or library like `dotenv` for managing environment variables.

- Additional Tools

  Node Package Dependencies:

  Install required dependencies for the project after cloning the repository:

  ```bash
  ```

npm install

```

Common dependencies include:

- express: Web framework.

- mongoose: MongoDB object modeling.

- dotenv: Environment variable management.

- cors, helmet: Middleware for security.

- bcrypt, jsonwebtoken: For authentication.

With these prerequisites in place, you can proceed to set up the project locally or deploy it on your desired platform.

- INSTALLATION GUIDE:

Follow these steps to set up the Grocery project on your local machine:

1. Clone the Repository

Download the project repository to your local machine:

```bash

git clone https://github.com/TSF-Solutions-24/Naan_Mudhalvan

cd Naan_Mudhalvan

```

2. Install Dependencies

Install the required Node.js dependencies on both frontend and backend:
```

```bash

npm install

```

3. Set Up Environment Variables

Create a `.env` file in the project root directory and add the following variables:

```plaintext

MONGO_URI=<your-mongodb-connection-string>

JWT_SECRET=<your-secret-key>

PORT=<desired-port-number, default: 3000>

```

- Replace `<your-mongodb-connection-string>` with your MongoDB connection URI.

- Replace `<your-secret-key>` with a strong secret key for JWT authentication.

- Optionally set a custom `<desired-port-number>` for the server.

---

4. Start the Server

Run the development server:

```bash

npm run dev

```

The server will start, and the application will be accessible at `http://localhost:<PORT>` (default: `3000`).

5. Test the Application

Open a browser or API testing tool like Postman and interact with the application to ensure everything is working correctly.

---

5. FOLDER STRUCTURE:

Client-side: React Frontend Structure

The React frontend for Grocery is designed with a modular and scalable architecture to ensure maintainability and efficiency. A typical folder structure for the frontend might look like this:

Server-side: Node.js Backend Organization

The Node.js backend for Grocery follows a structured and modular architecture to ensure scalability, maintainability, and ease of development. The backend is organized into the following directories and files:

---

6. RUNNING THE APPLICATION:

Follow these commands to start the frontend and backend servers locally:

1. Frontend:

Navigate to the frontend directory (Frontend) and start the React development server:

```bash
cd frontend
npm start
```

- The frontend server will run on `http://localhost:3000` by default.

2. Backend

Navigate to the backend directory (backend) and start the Node.js server:

```bash

cd backend

npm start

```

- The backend server will run on `http://localhost:5000` by default (or as configured in the `.env` file).

- **Ensure Both Servers are Running**

- Frontend interacts with the backend through API endpoints.

- Confirm both servers are up to experience the full functionality of the application.

---

7. API DOCUMENTATION:

The backend of the Grocery Web App provides various endpoints to handle user authentication, product management, cart operations, and order processing. Below is the detailed documentation of the available endpoints:

1. Authentication Endpoints

1.1 Register User

- Method: POST
- Endpoint: /api/auth/register
- Description: Creates a new user account.
- Parameters:
    - Body:
        - name (string) - Name of the user.
        - email (string) - Email address.
        - password (string) - Password for the account.

1.2 Login User

- Method: POST
- Endpoint: /api/auth/login
- Description: Authenticates user and provides a JWT token.
- Parameters:
    - Body:
        - email (string) - User's email.
        - password (string) - User's password.

1.3 Refresh Token

- Method: POST
- Endpoint: /api/auth/refresh
- Description: Issues a new JWT access token using a refresh token.
- Parameters:
    - Body:
        - refreshToken (string) - Refresh token for session renewal.

2. User Management Endpoints

2.1 Get User Profile

- Method: GET
- Endpoint: /api/users/profile
- Description: Retrieves the logged-in user's profile.
- Authorization: Bearer Token required.

2.2 Update User Profile

- Method: PUT
- Endpoint: /api/users/profile
- Description: Updates the logged-in user's profile information.
- Parameters:
    - Body:
        - name (string) - Updated name of the user.
        - email (string) - Updated email address.

3. Product Management Endpoints

3.1 Get All Products

- Method: GET
- Endpoint: /api/products
- Description: Retrieves a list of available products.
- Query Parameters:

    - page (number) - Page number for pagination.
    - limit (number) - Number of products per page.

3.2 Add a Product

- Method: POST
- Endpoint: /api/products
- Description: Allows admins to add a new product to the catalog.
- Authorization: Bearer Token required (Admin role).
- Parameters:
    - Body:
        - name (string) - Name of the product.
        - price (number) - Price of the product.
        - description (string) - Product description.
        - category (string) - Product category.

4. Cart Endpoints

4.1 Add to Cart

- Method: POST
- Endpoint: /api/cart/add
- Description: Adds a product to the user's cart.
- Authorization: Bearer Token required.
- Parameters:
    - Body:
        - productId (string) - ID of the product to add.
        - quantity (number) - Quantity of the product.

4.2 Get Cart Items

- Method: GET
- Endpoint: /api/cart
- Description: Retrieves items in the logged-in user's cart.
- Authorization: Bearer Token required.

4.3 Remove from Cart

- Method: DELETE
- Endpoint: /api/cart/remove
- Description: Removes a product from the user's cart.
- Parameters:
    - Body:
        - productId (string) - ID of the product to remove.

5. Order Management Endpoints

5.1 Place an Order

- Method: POST
- Endpoint: /api/orders
- Description: Places a new order based on the user's cart.
- Authorization: Bearer Token required.

5.2 Get User Orders

- Method: GET
- Endpoint: /api/orders
- Description: Retrieves the order history of the logged-in user.
- Authorization: Bearer Token required.

General Notes

- Authorization: Most endpoints require a valid JWT token in the Authorization header as Bearer <token>.
- Response Format: All responses follow a JSON format with fields for data, message, and status.
- Error Handling: Endpoints return appropriate HTTP status codes (400, 401, 404, etc.) along with descriptive error messages for debugging and clarity**.**

---

8. AUTHENTICATION AND AUTHORIZATION IN GROCERY

The Grocery Web App uses JWT (JSON Web Token) for managing authentication and authorization. This mechanism ensures secure access to user-specific resources by verifying users' identities and managing their permissions. Below is a detailed explanation of how the system handles authentication and authorization:

1. Authentication

JWT Authentication Flow

User Registration:
When a new user registers, their details (e.g., name, email, and password) are securely stored in the database. The password is hashed using bcrypt to ensure data security and prevent unauthorized access in case of breaches.

Login:
During login, the user provides their email and password. The system verifies the credentials against the stored data. If they are valid, a JWT token is generated and sent to the client. This token is signed using a secure key stored in the server environment (JWT_SECRET).

JWT Token:
The JWT contains encoded information such as the user ID and role, which helps in identifying and authenticating the user in subsequent requests. On the client side, the token is typically stored in local Storage or session Storage for use during the session.

2. Authorization

Protected Routes:

To restrict access to certain resources, such as managing the cart or placing orders, the backend uses middleware to verify the JWT sent in the request headers. If the token is valid, the user is granted access; otherwise, the request is denied.

Authorization Middleware:

A custom middleware (e.g., authMiddleware.js) ensures that each incoming request contains a valid token. The token must be included in the HTTP request headers under Authorization: Bearer <token>.

3. Token Expiry and Refresh

Token Expiry:
JWT tokens have an expiration time (exp) encoded in them. When the token expires, users must log in again to regain access.

Refresh Token Mechanism:
To minimize disruptions caused by token expiration, the system implements a refresh token flow:

1. Upon login, two tokens are issued:
    - Access Token (JWT): Short-lived, used for authorization.
    - Refresh Token: Longer expiry, used to renew access tokens.
2. When the access token expires, the client sends the refresh token to the server.
3. The server validates the refresh token and issues a new access token, maintaining user sessions without requiring reauthentication.

4. Session Management

While JWT is stateless and does not maintain server-side sessions, the refresh token can be stored in a secure HTTP-only cookie. This enhances security by protecting against cross-site scripting (XSS) attacks. Storing the refresh token in cookies ensures that it is not accessible via JavaScript.

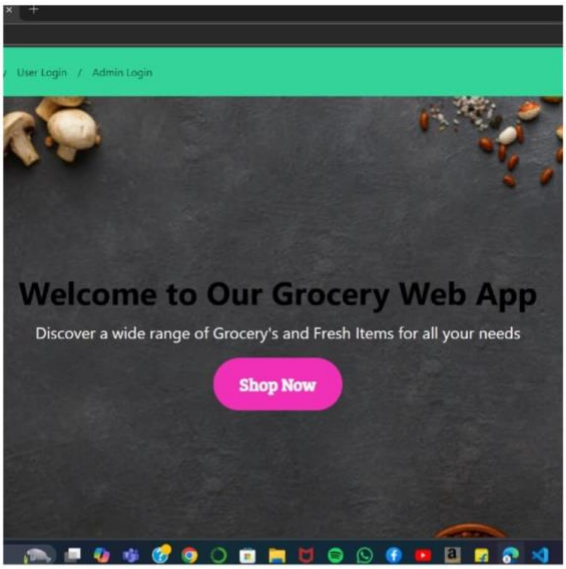
Summary of Authentication Flow

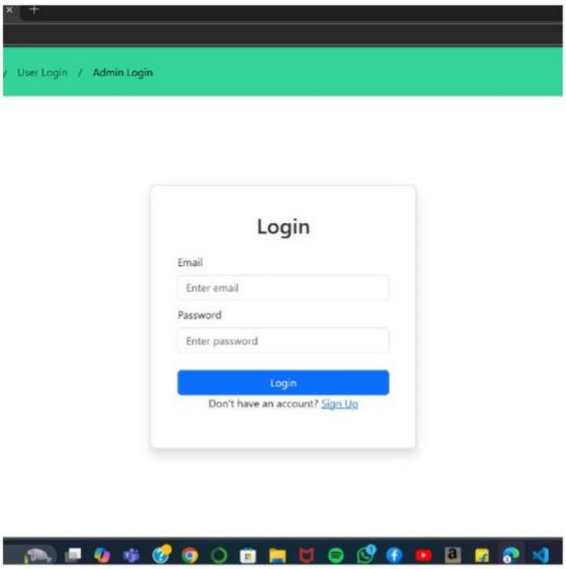
- User Registration: User details are stored securely, with passwords hashed using bcrypt.
- Login: Credentials are validated, and a JWT is issued.
- Token Usage: JWT is included in subsequent requests to access protected routes.
- Token Expiry: Users either log in again or use a refresh token to get a new access token.

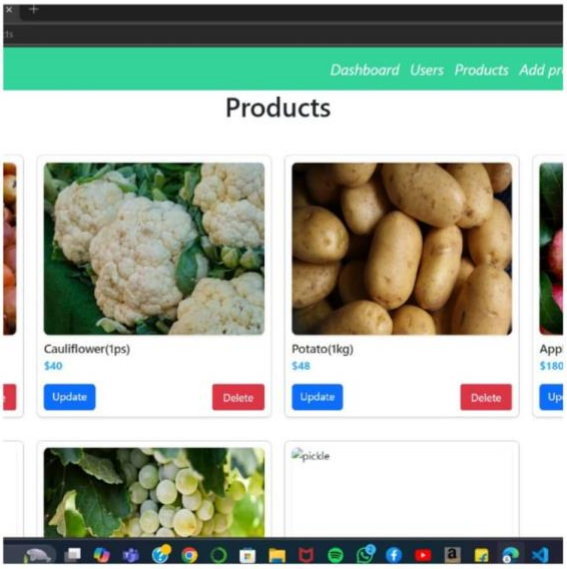
This authentication and authorization approach ensures a secure, scalable, and user-friendly experience for the Grocery Web App, maintaining the confidentiality of user data while enabling smooth interactions.

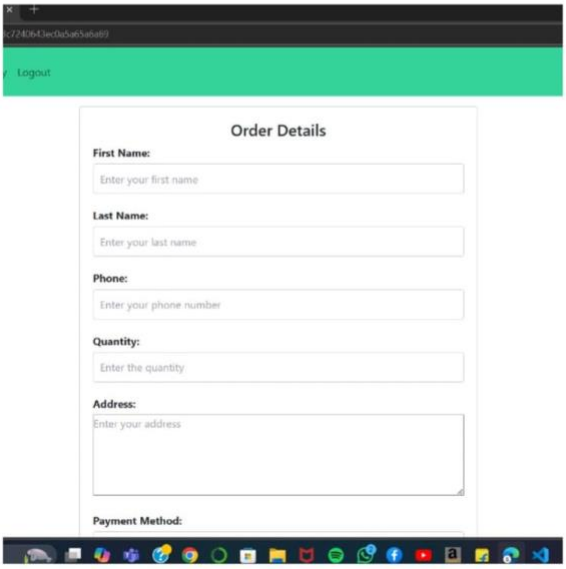
---

9. USER INTERFACE:









10. Testing

The testing strategy for the Grocery Web App ensures its robustness, reliability, and seamless performance across both the frontend and backend. The approach includes unit testing, integration testing, and end-to-end testing, with an emphasis on functionality, usability, and security.

1. Frontend Testing (React)

Testing Strategy

- **Unit Testing:**
  Individual React components are tested in isolation to verify their behavior. This includes ensuring components render properly based on given props and state, as well as confirming that user interactions (e.g., button clicks or form inputs) trigger the expected functionality.
- **Integration Testing:**
  These tests validate the interaction between multiple components, focusing on workflows like form submissions, API requests, and state management across the application. For instance, checking that adding an item to the cart updates the cart correctly.
- **End-to-End Testing:**
  Comprehensive tests simulate real-world user scenarios, from browsing products to adding items to the cart, checking out, and receiving a confirmation. This ensures smooth interactions between the frontend and backend while maintaining overall functionality.

Tools Used

- **React Testing Library:**
  React Testing Library, paired with Jest, is used to write frontend tests. The focus is on simulating real user actions and verifying outcomes, rather than testing internal implementation details.

2. Backend Testing (Node.js and Express)

Testing Strategy

- **Unit Testing:**
  Backend endpoints and functions are tested in isolation to ensure they handle specific inputs and outputs correctly. Examples include validating input data for login and ensuring API endpoints return the correct HTTP status codes and responses.

- **Integration Testing:**
  Backend integration tests validate the interaction between routes, middleware, and the database. For instance, testing that adding an item to the cart creates the appropriate entry in the database and returns the expected response.
- **Performance Testing:**
  Tests are conducted to ensure the backend can handle high volumes of concurrent user requests, ensuring the app remains responsive under load.

Tools Used

- **Jest and Supertest:**
  Jest is used for unit tests, while Supertest is used to simulate HTTP requests and validate backend API responses.

3. Continuous Integration and Deployment (CI/CD)

The testing process is fully integrated into the CI/CD pipeline to maintain consistent quality with every code update.

- **GitHub Actions:**
  Automated tests are triggered on every push or pull request using GitHub Actions. This ensures that new code changes are thoroughly tested before being merged into the main branch. Any failures are flagged early in the development process.
- **Environment Consistency:**
  Tests are executed in a consistent environment, replicating the production setup to catch potential issues before deployment**.**

---

11. Screenshots or Demo

- Provide screenshots or a link to a demo to showcase the application.

12. KNOWN ISSUES

1. Product Title and Description Alignment Issue

Description:
When a user browses or searches for products, the product titles and descriptions on the product listing page are not properly aligned. This misalignment causes the text to appear inconsistent or cluttered, negatively impacting the user experience and making it harder to read product details.

Steps to Reproduce:

- Log in or browse as a guest user.
- Navigate to the product listing page.

- Search for any product category (e.g., "Fruits" or "Vegetables").
- Observe the misalignment of product titles and descriptions on the displayed results.

Expected Behavior:
Product titles and descriptions should be aligned correctly, with uniform spacing and formatting, ensuring a clean and visually appealing layout.

Current Workaround:
Users can zoom out or resize the browser window to adjust the layout temporarily. However, this is not a permanent solution.

Status:
The issue has been acknowledged and is currently being worked on. A fix will be implemented in the upcoming update to improve the alignment and presentation of product details.

---

13. FUTURE ENHANCEMENT:

1. Personalized Shopping Experience:

Integrating machine learning algorithms to analyze user behavior, preferences, and purchase history. This will enable personalized product recommendations, tailored discounts, and suggestions for frequently purchased or related items. Features like shopping list generation, dynamic pricing offers, and reminders for restocking essentials will enhance the user experience.

2. Accessibility Features:

Adding accessibility features to ensure an inclusive shopping experience for all users. This includes screen reader compatibility for visually impaired users, keyboard navigation for improved usability, and providing alternative text for product images to meet accessibility standards. These enhancements aim to make the platform more user-friendly and accessible to a wider audience.