Full Stack Development with MERN

Grocery web app

Introduction

- **Project Title:** Grocery web app

A grocery web app is an online platform that allows customers to browse, search, and purchase grocery items from the convenience of their devices. It typically includes features that provide a smooth shopping experience, such as product categorization, a search bar, a cart for adding items, and a secure checkout process**.**

- **Team Members: 1) Shaju Ganesh U.S (Leader)**

  2) Sadhana. M (Assistant Leader)

  3) Praveen. S

  4) Vanishree. A

2. Project Overview

- **Purpose of a Grocery Web App:**The purpose of a grocery web app is to provide a convenient, online platform for users to shop for groceries from any device, eliminating the need for in person visits to physical stores. It allows customers to browse products, add them to a virtual cart, and complete purchases, often with delivery or pickup options. This digital solution also benefits store owners and administrators by simplifying inventory management, order processing, and customer communication.
- **Convenience**: Allows users to shop for groceries anytime, anywhere.
- **Time Saving**: Reduces the time spent traveling to stores and waiting in lines.
- **Inventory Management**: Helps store owners track and manage products efficiently.
- **Enhanced Customer Experience**: Provides a seamless, user friendly shopping experience with multiple options for delivery or pickup.
- **Business Insights**: Collects data on customer preferences, order history, and feedback, helping businesses improve products and services.
- **Environmental Impact**: Reduces foot traffic and contributes to ecofriendly shopping by minimizing transportation.

Features:

User Interface (Customer)

- **User Registration/Login**: Secure access with account creation or login options for personalized shopping.
- **Product Browsing:** Categories and filters for easy browsing by product type, brand, price, etc.
- **Search Functionality:** Enables users to quickly find specific products.
- Product Details: Displays information like ingredients, price and ratings.
- **Shopping Cart:** Users can add, edit, and view selected items before purchasing.
- **Order Placement:** Options for delivery, pickup, or instore purchases.
- **Payment Integration:** Secure checkout process with payment option(credit/debit).
- **Order History:** Displays previous orders for easy reordering or tracking.
- **User Profile:** Allows users to manage personal details, addresses, and payment methods.

Admin Interface (Administrator)

- **Inventory Management:** Tools to add, remove, and update product details and stock.
- **Order Management:** Allows sellers to view, process, and update order status.
- **Customer Communication:** Options to respond to customer inquiries or feedback.
- **User and Seller Management:** Oversee registered users and vendors, handle any account related issues.
- **Product Management:** Control over categories, subcategories, and product listings.
- **Security and Privacy Controls:** Ensure data security and compliance with privacy regulations.
- **Feedback and Support:** Manage and respond to customer feedback and troubleshoot issues.

3. Architecture

Frontend:

Describe the frontend architecture using React. The frontend architecture of a grocery web app is designed to ensure a seamless, responsive, and user-friendly experience. It is typically organized in a component-based structure, especially when using frameworks like Angular or React.js. Each component handles a specific part of the user interface, making the application modular, reusable, and easier to maintain.

1. Component Based Structure

The frontend is divided into modular components, each representing a distinct section of the user interface. For example, components could include:

- **Header Component:** Contains navigation links (e.g., home, categories, cart, profile).
- **Footer Component:** Displays contact information, links, and policies.
- **Home Page Component:** The landing page that displays popular products and promotional offers.
- **Product List Component:** Shows a grid or list of products based on categories or search results.
- **Product Details Component:** Displays detailed information about a single product, including description, price, and reviews.
- **Cart Component:** Manages items added by the user for purchase, including quantity adjustments and a checkout button.
- **User Profile Component:** Allows users to view and edit personal information, addresses, and payment methods.
- **Order History Component:** Lists past purchases with order details.

2. State Management

- **Global State Management:** In a larger app, a state management solution (e.g., Redux for React or Rx for Angular) is often used to manage data shared across components, such as user authentication, cart items, and order history.
- **Local Component State:** Used for data that is specific to individual components, such as form inputs or component specific loading states.

3. Routing and Navigation

- **Client-Side Routing:** Frontend frameworks like Angular (Router Module) and React (React Router) manage routing, enabling smooth transitions between pages without full page reloads.
- **Protected Routes:** Certain routes, like profile and order history, require the user to be logged in. Authentication checks are implemented for these routes.

4. API Integration and Services

- **HTTP Service Layer:** The frontend communicates with the backend API through an HTTP service layer, using HTTP requests (GET, POST, PUT, DELETE) to retrieve and send data. This includes:
- Fetching product data for display.
- Sending cart and checkout information.
- Retrieving order history.
- Error Handling: Centralized error handling to manage any errors from the API and display meaningful messages to the user.

5. UI and Styling

- **Component Specific Styling**: CSS, SCSS, or styled components (for React) provide unique styling for each component.
- **Responsive Design:** The app is made responsive using CSS frameworks (e.g., Bootstrap, Tailwind CSS) or custom media queries to ensure optimal display on devices of all sizes.
- **Reusable UI Elements:** Elements like buttons, form fields, and modal dialogs are created as reusable components for a consistent design.

6. Form Handling and Validation

- **Form Management:** Forms are used for login, registration, address input, and checkout.
- **Form Validation:** Frontend validation (with tools like for React or Reactive Forms for Angular) ensures that required fields are filled correctly before data is submitted.

7. Authentication and Authorization

- **Token Storage:** User authentication tokens (e.g., JWT) are securely stored (in cookies or local storage) to maintain user sessions.
- **Role Based Access Control:** Features and routes can be restricted based on user roles (e.g., customers can access the cart, but only admins can access user management).

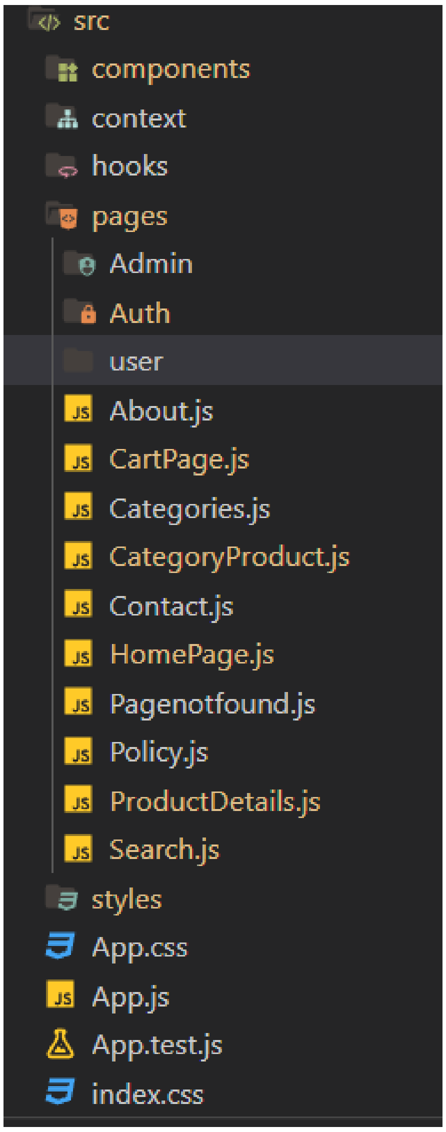
8. Optimization and Performance

- **Lazy Loading:** Components and routes are loaded on demand to reduce the initial load time.
- **Code Splitting:** Allows the app to load only the necessary code for a particular route or component, improving performance.
- **Caching:** Local caching (e.g., product data) minimizes redundant API requests, speeding up the app.

9. Testing

- Unit Testing: Components and functions are tested to ensure individual features work correctly.
- Integration Testing: Tests the interaction between components, such as adding items to the cart and proceeding to checkout.
- End to End Testing: Ensures the overall flow of the app works as expected from a user's perspective.

Component Organization



```
src
  components
  context
  hooks
  pages
    Admin
    Auth
    user
    About.js
    CartPage.js
    Categories.js
    CategoryProduct.js
    Contact.js
    HomePage.js
    Pagenotfound.js
    Policy.js
    ProductDetails.js
    Search.js
  styles
  App.css
  App.js
  App.test.js
  index.css
```

- This frontend architecture promotes modularity, scalability, and maintainability, making it easier to add new features, manage UI states, and optimize the user experience in a grocery web app.

Backend

- The backend architecture for a grocery web app using Node.js and Express.js serves as the server side framework to manage business logic, database operations, and API endpoints. It provides a secure and structured environment for handling requests, managing data, and ensuring a smooth integration with the frontend.

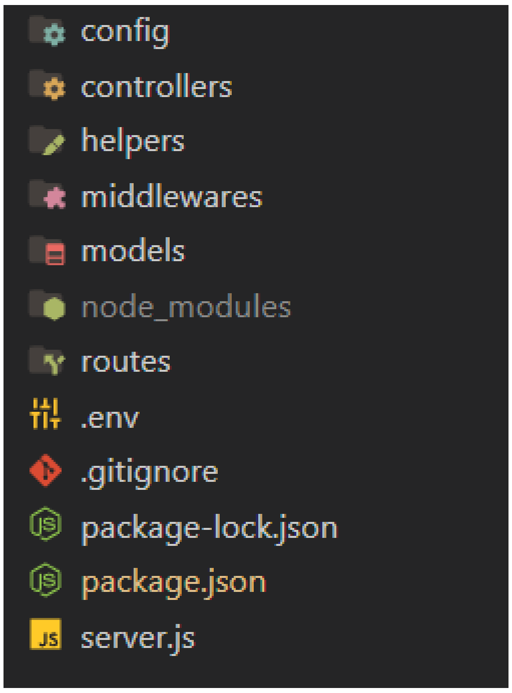
Backend Architecture of a Grocery Web App (Node.js and Express.js)

1. Server Setup with Express.js

- Express Server: Sets up the main server to handle HTTP requests and responses. This server acts as the central entry point for all client requests, such as retrieving products, managing the cart, and processing orders.
- API Routing: Organizes different functionalities into modular routes (e.g., user, product, cart, and order routes) to handle specific sections of the app.

2. Folder Structure

- A well organized folder structure ensures maintainability and scalability:



```
config
controllers
helpers
middlewares
models
node_modules
routes
.env
.gitignore
package-lock.json
package.json
server.js
```

3. Database Management (MongoDB with Mongoose)

- **MongoDB**: A NoSQL database used to store data in a flexible, schema less manner.
- **Mongoose:** An Object Data Modeling (ODM) library for MongoDB that helps structure the data and provides easy CRUD operations. Mongoose models are created for key entities like users, products, carts, and orders.

Example Models:

- User Model: Stores user data, including authentication credentials, delivery addresses, and order history.
- Product Model: Stores product details such as name, category, price, stock, and description.
- Order Model: Stores order information, including items, quantities, user ID, order status, and timestamps.
- Cart Model: Stores the current items in each user's cart.

4. API Routes

- The routes handle all the client requests and are organized based on the resource they manage (e.g., users, products, orders).
- User Routes: Handles user registration, login, profile updates, and authentication.
- Product Routes: Manages CRUD operations for products, including fetching product lists, adding new products (for admin/seller), and updating stock.
- Cart Routes: Manages user specific cart items, such as adding, updating, or removing items.
- Order Routes: Handles order creation, updating order status, and fetching order history for users.

5. Controllers (Business Logic Layer)

- Controllers manage the core logic for each route, keeping routes clean and focused on defining the endpoint itself.
- Example: The `product Controller` handles tasks like fetching product data, adding new products, and updating product stock levels, coordinating between the database and the route.

6. Middleware Functions

- Authentication: Verifies user tokens (JWT) to ensure secure access to protected routes.
- Error Handling: Centralized error handling middleware to catch and handle errors gracefully, providing meaningful responses to the client.
- Authorization: Checks user roles (e.g., admin, seller, user) to restrict access to certain actions based on permissions.

7. Environment Configuration (.env)

- Stores sensitive information like the database URI, JWT secret key, and port.
- Uses `dot env` to load environment variables, ensuring sensitive data remains secure and not hardcoded.

8. Authentication and Authorization

- JWT (JSON Web Tokens): Used for secure authentication, with tokens issued on login and stored client side (in cookies or local storage).
- Role Based Access Control: Enforces permissions so that only authorized users (e.g., admins or sellers) can access or modify specific resources.

9. Error Handling and Logging

- Global Error Handler: A middleware that captures and processes errors, sending standardized error responses.
- Logging: Use of libraries `morgan` to log requests and errors for monitoring and debugging.

10. Scalability and Performance Optimizations

- Caching: Uses tools like Redis for caching frequently requested data (e.g., product lists) to reduce database load.
- Load Balancing: Distributes traffic across multiple instances if the app grows to serve many users.
- Data Validation: Validates user inputs using libraries like `Joi` or `Express Validator` to ensure data integrity and prevent invalid data from reaching the database.
- This backend architecture using Node.js and Express.js provides a robust foundation for managing data, securing user actions, and responding to frontend requests efficiently, making it highly suited for a grocery web app.

Database

To use a carousel feature with MongoDB, the backend database can be structured to store carousel items, such as images, descriptions, and other metadata that would be displayed in each slide. Here’s a detailed look at a possible database schema and how MongoDB would interact with it.

1. Database Schema for Carousel

Assume we’re creating a carousel that stores the following for each slide:

- **Image URL**: Path or link to the image.
- **Title**: A short title for the slide.
- **Description**: Brief text shown on the slide.
- **Active**: Boolean value indicating whether the slide is active.
- **Order**: A number representing the order of slides.

2. Database Interactions with MongoDB

Create a Carousel Item

To add a new slide to the carousel, you’d create a new entry in the MongoDB database.

javascript

```
const addCarouselItem = async (itemData) => {

  try {

    const newItem = new CarouselItem(itemData);

    await newItem.save();

    console.log('Carousel item added successfully:', newItem);

  } catch (error) {

    console.error('Error adding carousel item:', error);

  }

};
```

Sample itemData:

javascript

Copy code

```
{
  imageUrl: "https://example.com/image.jpg",
  title: "Welcome Slide",
  description: "This is the first slide in the carousel",
  active: true,
  order: 1
}
```

Retrieve All Carousel Items

Fetch all carousel items to display them in the front-end carousel component.

javascript

```
const getCarouselItems = async () => {
  try {
    const items = await CarouselItem.find().sort({ order: 1 });
    return items;
  } catch (error) {
    console.error('Error retrieving carousel items:', error);
  }
};
```

Update a Carousel Item

To modify an existing carousel item, such as updating the image or title:

javascript

```
const updateCarouselItem = async (id, updateData) => {

  try {

    const updatedItem = await CarouselItem.findByIdAndUpdate(id, updateData, {
new: true });

    console.log('Carousel item updated:', updatedItem);

  } catch (error) {

    console.error('Error updating carousel item:', error);

  }

};
```

Delete a Carousel Item

To delete a carousel item from MongoDB:

javascript

```
const deleteCarouselItem = async (id) => {

  try {

    await CarouselItem.findByIdAndDelete(id);

    console.log('Carousel item deleted successfully');

  } catch (error) {

    console.error('Error deleting carousel item:', error);

  }

};
```

3. Frontend Integration Example

On the front end, the carousel component would retrieve these items from the backend API and render them. Here’s a basic example using JavaScript/HTML:

```
html

<div id="carousel" class="carousel">

  <!-- Carousel items will be populated here -->

</div>

<script>

  async function loadCarouselItems() {

    try {

      const response = await fetch('/api/carousel-items');

      const items = await response.json();

  const carouselContainer = document.getElementById('carousel');

      items.forEach((item) => {

        const slide = document.createElement('div');

        slide.className = `carousel-item ${item.active ? 'active' : ''}`;

        slide.innerHTML = `  <img src="${item.imageUrl}" alt="${item.title}">

          <div class="carousel-caption">

            <h5>${item.title}</h5>

            <p>${item.description}</p>

          </div>

        `;

        carouselContainer.appendChild(slide);

      });

    } catch (error) {
      console.error('Error loading carousel items:', error);
    }
  }
  loadCarouselItems();
</script>
```

4. API Endpoints

For a complete setup, here are example API endpoints:

- **POST /api/carousel-items** – To create a new carousel item.
- **GET /api/carousel-items** – To fetch all carousel items.
- **PUT /api/carousel-items/:id** – To update an existing carousel item.
- **DELETE /api/carousel-items/:id** – To delete a carousel item.

These endpoints enable full CRUD (Create, Read, Update, Delete) operations on the carousel items in MongoDB.

This setup lets you dynamically manage carousel items stored in MongoDB, which can then be displayed and managed efficiently on your frontend interface.

Setup Instructions

Prerequisites

To set up a carousel feature integrated with MongoDB, here is a list of the software dependencies you'll need:

1. Node.js

- **Purpose**: Server-side JavaScript runtime that enables the backend environment.
- **Version**: Latest stable version (usually v14 or higher).
- **Installation**: Download from Node.js
- **Dependency Management**: `npm` (Node Package Manager) is bundled with Node.js and is used to install other dependencies.

2. MongoDB

- **Purpose**: NoSQL database to store carousel items data, such as images, titles, and descriptions.
- **Version**: Latest stable version (e.g., MongoDB 6.0 or higher).
- **Installation**: Download MongoDB or use a managed service like MongoDB Atlas if a cloud database is preferred.

3. Mongoose

- **Purpose**: An Object Data Modeling (ODM) library for MongoDB and Node.js that makes it easier to define schemas and manage MongoDB data.
- **Installation**: Install via npm:
- bash
- Copy code
- npm install mongoose

4. Express.js

- **Purpose**: Web application framework for Node.js that simplifies server setup and routing.
- **Installation**: Install via npm:
- bash
- Copy code
- npm install express

5. Frontend Dependencies (optional if using a frontend framework)

If your project involves a frontend framework like React, Vue, or Angular, additional dependencies might be required, such as:

- **Bootstrap (optional)**: For carousel styling if using Bootstrap's carousel component.

  ```
  bash
  Copy code
  npm install bootstrap
  ```

- **Axios or Fetch API**: For making HTTP requests from the frontend to the backend API.

6. Dotenv

- **Purpose**: Module to load environment variables from a .env file for sensitive information like database credentials.
- **Installation**:
- bash
- Copy code
- npm install dotenv

Installation

Here’s a step-by-step guide to help you clone a Node.js project, install dependencies, and set up environment variables for your carousel feature with MongoDB.

Step 1: Clone the Project Repository

- **Open a Terminal** on your computer.
- **Navigate to the directory** where you want to clone the project.
- Use the following command to clone the repository (replace <repository-url> with the actual Git URL)
- git clone <repository-url>
- **Navigate into the project folder:**
- cd <project-folder>

Step 2: Install Node.js and MongoDB (If Not Installed)

- **Download Node.js** from the official website: Node.js Download.
- **Install MongoDB** by downloading it from the official website: MongoDB Download.
- Alternatively, you can sign up for a **MongoDB Atlas** account to use a cloud-hosted MongoDB instance.

Step 3: Install Project Dependencies

- **Check if Node.js and npm are installed** by running:
- bash
- Copy code
- node -v
- npm -v
- You should see version numbers if they are installed correctly.
- **Install dependencies** by running the following command in the project folder:
- bash
- Copy code
- npm install
- This will install all required libraries listed in the package.json file, such as express, mongoose, cors, dotenv, etc.

Step 4: Configure Environment Variables

- **Create a .env file** in the root of your project directory:
- bash
- Copy code
- touch .env
- Open the .env file and add the following variables. Replace the placeholders with actual values:
- plaintext
- Copy code
- PORT=5000
- MONGODB_URI=<your-mongodb-connection-string>
- **PORT**: The port on which your server will run (e.g., 5000).
- **MONGODB_URI**: The MongoDB connection string. If using MongoDB locally, this could be something like mongodb://localhost:27017/carouselDB. If using MongoDB Atlas, use the connection string provided in your Atlas account, and replace <username> and <password> with your credentials.

Step 5: Verify and Run the Server

- In your project directory, open the server.js (or app.js, depending on your project) and check if it uses dotenv to load environment variables. If not, add the following line at the top of the file:
- javascript
- Copy code
- require('dotenv').config();
- **Run the server** with the following command:
- bash
- Copy code
- npm start
- If using nodemon for development, you can start the server with:
- bash
- Copy code
- nodemon server.js
- **Check the console** for messages. You should see a message indicating the server is running and connected to MongoDB if the setup is successful.

Step 6: Test the API Endpoints

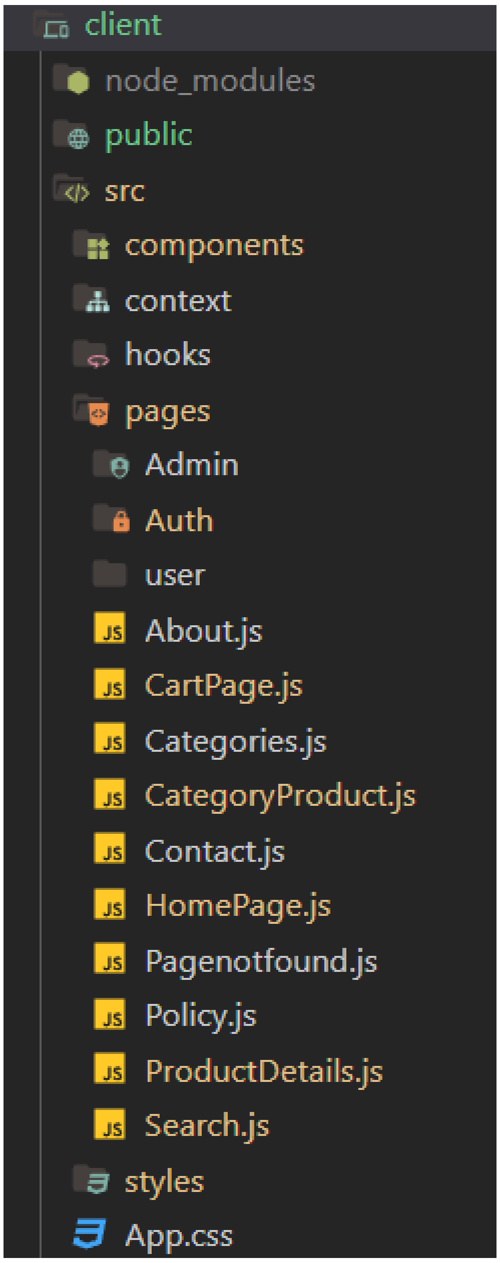
- Use a tool like **Postman** or **curl** to test your API endpoints.
- For example, to fetch all carousel items, you might send a GET request to:
- bash
- Copy code
- http://localhost:5000/api/carousel-items
- Step by step guide to clone, install dependencies, and set up the environment variables.

Folder Structure

Client

In a project with a React frontend for a carousel feature connected to MongoDB, the React application's structure should be organized to manage components, services, and API interactions effectively. Here's a breakdown of how the project structure might look, including essential folders and files.

1. Project Folder Structure



```
client
  node_modules
  public
  src
    components
    context
    hooks
    pages
      Admin
      Auth
      user
      About.js
      CartPage.js
      Categories.js
      CategoryProduct.js
      Contact.js
      HomePage.js
      Pagenotfound.js
      Policy.js
      ProductDetails.js
      Search.js
    styles
    App.css
```

2. Folder and File Descriptions

- **Root Directory (my-carousel-app/)**
- **public/index.html**: The main HTML file where the React app is rendered. It contains a root <div> where the React components mount.
- **src/index.js**: The entry point for the React application. It imports the root component (App.js) and renders it to the index.html file.
- **src/App.js**: The main application component that renders the overall app layout.
- **src/App.css**: Styling for the main App component.
- **Components Folder (src/components/)**
- This folder organizes reusable React components. In this example, it includes a Carousel folder, containing all the files necessary for the carousel functionality.
- **src/components/Carousel/**: This folder contains the carousel component files.
- **Carousel.js**: The main component that handles the carousel layout, slide transitions, and renders each slide.
- **CarouselItem.js**: A child component that represents each individual slide in the carousel. It receives props from Carousel.js to render the image, title, and description.
- **Carousel.css**: Styles specific to the carousel components.
- **Services Folder (src/services/)**
- The services folder holds files related to external API calls, keeping data-fetching logic separate from components.
- **src/services/api.js**: This file contains functions to fetch carousel data from the backend, making the code reusable and easy to maintain. Here, we'll define functions to handle API requests like getCarouselItems, addCarouselItem, etc.
- **Environment Variables File (.env in src/)**
- Environment variables can be used for storing the backend API URL or other configuration settings.
- **.env**:
- Define the API base URL:
- REACT_APP_API_URL=http://localhost:8080/api

3. Code Overview

- **Carousel.js (Main Carousel Component)**
- javascript
- Copy code

```
import React, { useEffect, useState } from 'react';
import CarouselItem from './CarouselItem';
import './Carousel.css';
import { getCarouselItems } from '../services/api';

function Carousel() {
  const [items, setItems] = useState([]);

  useEffect(() => {
    // Fetch carousel items from the API
    const fetchItems = async () => {
      try {
        const data = await getCarouselItems();
        setItems(data);
      } catch (error) {
        console.error("Error fetching carousel items:", error);
      }
    };
    fetchItems();
  }, []);

  return (
    <div className="carousel">
      {items.map((item, index) => (
        <CarouselItem
          key={item._id}
          active={index === 0}
          title={item.title}
          description={item.description}
          imageUrl={item.imageUrl}
        />
      ))}
    </div>
  );
}
export default Carousel;
```

- **CarouselItem.js (Individual Carousel Item Component)**
- javascript
- Copy code
- import React from 'react';
- import './Carousel.css';
- function CarouselItem({ title, description, imageUrl, active }) {
-   return (
-     <div className={`carousel-item ${active ? 'active' : ''}`}>
-       <img src={imageUrl} alt={title} />
-       <div className="carousel-caption">
-         <h5>{title}</h5>
-         <p>{description}</p>
-       </div>
-     </div>
-   );
- }
- export default CarouselItem;
- **api.js (API Services)**
- javascript
- Copy code
- import axios from 'axios';
- const API_URL = process.env.REACT_APP_API_URL;
- // Fetch all carousel items
- export const getCarouselItems = async () => {
-   try {
-     const response = await axios.get(`${API_URL}/carousel-items`);
-     return response.data;
-   } catch (error) {
-     throw error;
-   }
- };

4. Adding Styles (Carousel.css)

- css
- Copy code
- .carousel {
-   display: flex;
-   overflow: hidden;
-   position: relative;
- }
-
- .carousel-item {

- min-width: 100%;
- transition: transform 0.5s ease;
- }
-
- .carousel-caption {
- position: absolute;
- bottom: 20px;
- left: 20px;
- color: white;
- }

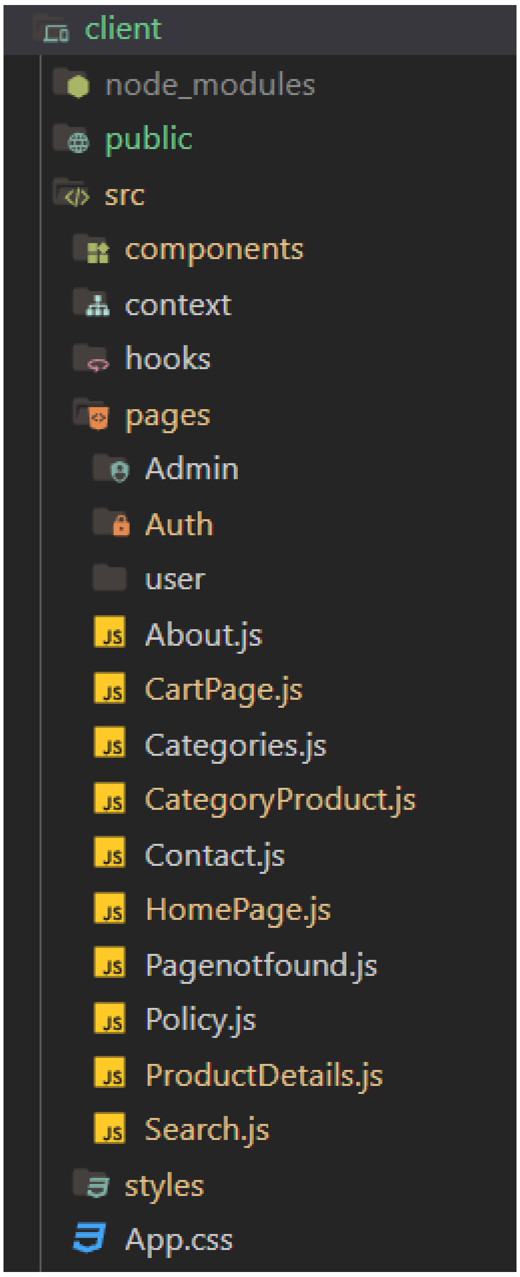
5. Environment Configuration (.env)

- In your .env file in the project root, set up the API URL so it can be accessed globally:
- plaintext
- Copy code
- REACT_APP_API_URL=http://localhost:5000/api
- This .env file ensures that the API base URL can be easily changed without modifying multiple files.

Server

- The Node.js backend for a carousel feature with MongoDB should be organized to handle routing, data handling, and API responses effectively. Here's an ideal structure and explanation of the backend organization, with key files and folders.

1. Project Folder Structure



```
client
  node_modules
  public
  src
    components
    context
    hooks
    pages
      Admin
      Auth
      user
      About.js
      CartPage.js
      Categories.js
      CategoryProduct.js
      Contact.js
      HomePage.js
      Pagenotfound.js
      Policy.js
      ProductDetails.js
      Search.js
    styles
    App.css
```

2. Folder and File Descriptions

- **Root Directory** (carousel-backend/)
- server.js: The main entry point of the application where the Express server is configured and initialized.
- .env: Environment variables, such as MongoDB connection string and server port, are stored here for easy access and security.
- .gitignore: Specifies files and directories, like node_modules and .env, that should not be included in version control.
- **Config Folder** (config/)
- This folder stores configuration files.
- db.js: Handles the connection to MongoDB. It connects to the database using mongoose and exports the connection for use in server.js.
- **Controllers Folder** (controllers/)
- The controllers folder holds the business logic of the application, keeping the code clean and modular.
- carouselController.js: Defines functions to handle requests related to carousel items, such as fetching all items, adding a new item, updating an item, and deleting an item. Each function corresponds to a specific API endpoint in carouselRoutes.js.
- **Models Folder** (models/)
- The models folder contains Mongoose schemas and models, defining how data is structured in MongoDB.
- CarouselItem.js: Defines the Mongoose schema for a carousel item. This includes fields like title, description, and imageUrl, and any validations or default values.
- **Routes Folder** (routes/)
- This folder organizes route definitions, keeping them separate from the business logic.
- carouselRoutes.js: Defines the RESTful API routes for the carousel items, such as GET, POST, PUT, and DELETE requests. It maps each route to a corresponding function in the carousel Controller.

3. Code Overview

- server.js **(Main Server File)**
- javascript
- import express from "express";
- import colors from "colors";
- import dotenv from "dotenv";
- import morgan from "morgan";
- import connectDB from "./config/db.js";
- import authRoutes from "./routes/authRoute.js";

```
• import categoryRoutes from "./routes/categoryRoutes.js";
• import productRoutes from "./routes/productRoutes.js";
• import cors from "cors";
•
• //configure env
• dotenv.config();
•
• //databse config
• connectDB();
•
• //rest object
• const app = express();
•
• //middelwares
• app.use(cors());
• app.use(express.json());
• app.use(morgan("dev"));
•
• //routes
• app.use("/api/v1/auth", authRoutes);
• app.use("/api/v1/category", categoryRoutes);
• app.use("/api/v1/product", productRoutes);
•
• //rest api
• app.get("/", (req, res) => {
•   res.send("<h1>Welcome to ecommerce app</h1>");
• });
•
• //PORT
• const PORT = process.env.PORT || 8080;
•
• //run listen
• app.listen(PORT, () => {
•   console.log(
•     `Server Running on ${process.env.DEV_MODE} mode on port
  ${PORT}`.bgCyan
•       .white
•   );
• });});
```

4. Environment Configuration (.env)

- Add MongoDB connection details and server port in the .env file:
- PORT=8080
- MONGO_URI= mongodb+srv://shaju002:******@shaju.g53ml.mongodb.net/Grocery

Running the Application

Frontend

To start the React frontend in the client directory, follow these steps:

- **Navigate to the Client Directory**:
  - Open your terminal, and move into the `client` directory (where your React frontend is located):

    ```
    cd client
    ```

- **Install Dependencies (If Not Already Installed)**:
  - Before starting the app, ensure that all dependencies listed in `package.json` are installed:

    ```
    npm install
    ```

- **Start the React Development Server**:
  - Once dependencies are installed, start the React app:

    ```
    npm start
    ```

- **Access the Frontend**:
  - By default, the React app should open automatically in your default web browser, typically on http://localhost:3000.
  - If it doesn't open automatically, you can manually go to http://localhost:3000 in your browser.

Backend

To start the backend server in the server directory, follow these steps:

- **Navigate to the Server Directory**:
  - Open your terminal and move to the `server` directory (where your Node.js backend is located):

    ```
    cd server
    ```

- **Install Dependencies (If Not Already Installed)**:
  - Before starting the server, make sure all dependencies listed in `package.json` are installed:

    ```
    npm install
    ```

- **Set Up Environment Variables**:
  - Ensure that your `.env` file is correctly configured with essential variables, like your MongoDB URI and server port:

    ```
    PORT=8080
    MONGO_URI=
    mongodb+srv://shaju002:******@shaju.g53ml.mongodb.net/Grocery
    ```

- **Start the Backend Server**:
  - Once dependencies are installed and environment variables are set, start the server:

    ```
    npm start
    ```

- **Confirm the Server is Running**:
  - You should see output in the terminal, such as:

    ```
    Server is running on port 8080
    Connected to MongoDB
    ```

  - By default, the backend will be accessible on http://localhost:5000.

API Documentation

Here's a documentation of all the RESTful API endpoints exposed by the backend for managing carousel items. Each endpoint interacts with the MongoDB database to perform CRUD operations on carousel items.

1. Base URL

All endpoints are prefixed by the base URL:

http://localhost:8080/api/v1/carousel-items

2. Endpoints

a. Get All Carousel Items

- **URL**: /
- **Method**: GET
- **Description**: Fetches all carousel items from the database.
- **Response**:
  - **Success**: Returns a JSON array of all carousel items.
  - **Error**: Returns an error message if unable to retrieve items.
- **Example Request**:

```
http

GET http://localhost:8080/api/v1/carousel-items/
```

- **Example Response**:

```
json

Copy code

[

  {

    "_id": "60f71b34c2b9d72d7e7639a4",

    "title": "Sample Carousel Item",

    "description": "This is a sample carousel item.",

    "imageUrl": "https://example.com/image.jpg",

    "createdAt": "2023-10-01T10:00:00Z",
    "updatedAt": "2023-10-01T10:00:00Z"
  }
]
```

b. Get a Single Carousel Item

- **URL**: /:id
- **Method**: GET
- **Description**: Fetches a single carousel item by its unique ID.
- **Parameters**:
  - **id** (in URL): The unique ID of the carousel item to fetch.
- **Response**:
  - **Success**: Returns a JSON object of the requested carousel item.
  - **Error**: Returns an error message if the item is not found.
- **Example Request**:

http

Copy code

```
GET http://localhost:8080/api/carousel-items/60f71b34c2b9d72d7e7639a4
```

c. Create a New Carousel Item

- **URL**: /
- **Method**: POST
- **Description**: Adds a new carousel item to the database.
- **Request Body**:
  - **title** (string): The title of the carousel item.
  - **description** (string, optional): A description for the carousel item.
  - **imageUrl** (string): The URL of the image for the carousel item.
- **Response**:
  - **Success**: Returns the newly created carousel item object.
  - **Error**: Returns an error message if the item could not be created.
- **Example Request**:

http

```
POST http://localhost:8080/api/carousel-items/
```

```
Content-Type: application/json

{
  "title": "New Carousel Item",
  "description": "This is a description for the new item.",
  "imageUrl": "https://example.com/new-image.jpg"
}
```

- **Example Response**:

```
json
Copy code
{
  "_id": "60f71b34c2b9d72d7e7639a5",
  "title": "New Carousel Item",
  "description": "This is a description for the new item.",
  "imageUrl": "https://example.com/new-image.jpg",
  "createdAt": "2023-10-01T11:00:00Z",
  "updatedAt": "2023-10-01T11:00:00Z"
}
```

d. Update an Existing Carousel Item

- **URL**: /:id
- **Method**: PUT
- **Description**: Updates an existing carousel item in the database.
- **Parameters**:
  - **id** (in URL): The unique ID of the carousel item to update.
- **Request Body**:
  - Fields to update (e.g., title, description, imageUrl).
- **Response**:
  - **Success**: Returns the updated carousel item object.
  - **Error**: Returns an error message if the item could not be updated.

Example Request:

http

```
PUT http://localhost:5000/api/carousel-items/60f71b34c2b9d72d7e7639a5

Content-Type: application/json

{

  "title": "Updated Carousel Item Title"

}
```

Example Response:

json

Copy code

```
{

  "_id": "60f71b34c2b9d72d7e7639a5",

  "title": "Updated Carousel Item Title",

  "description": "This is a description for the new item.",

  "imageUrl": "https://example.com/new-image.jpg",

  "createdAt": "2023-10-01T11:00:00Z",

  "updatedAt": "2023-10-01T12:00:00Z"

}
```

e. Delete a Carousel Item

- **URL**: /:id
- **Method**: DELETE
- **Description**: Deletes a carousel item from the database by its ID.
- **Parameters**:
  - **id** (in URL): The unique ID of the carousel item to delete.
- **Response**:
  - **Success**: Returns a success message.
  - **Error**: Returns an error message if the item could not be deleted.

- **Example Request**:

  http

  Copy code

  ```
  DELETE http://localhost:5000/api/carousel-items/60f71b34c2b9d72d7e7639a5
  ```

- **Example Response**:

  json

  Copy code

  ```
  {
    "message": "Item deleted successfully"
  }
  ```

Authentication

To handle authentication and authorization in this project, you would typically use JSON Web Tokens (JWT) for a secure and stateless authentication process. Below is an explanation of how it can be implemented, along with details on tokens, sessions, and middleware used to protect routes.

1. Authentication with JSON Web Tokens (JWT)

JWT is a popular choice for authentication in REST APIs due to its simplicity and stateless nature. JWT tokens can be generated upon user login, which allows users to access protected routes without storing any session data on the server.

Steps for JWT Authentication:

- **User Login**: When a user logs in with a valid email and password, the backend verifies the credentials. If they are valid, the server generates a JWT for the user.
- **Token Creation**: The JWT token is created using a secret key stored in environment variables. The token includes encoded information (like the user ID) and has an expiration time for added security.
- **Token Storage**: The generated token is sent to the client (typically stored in localStorage or a secure cookie).
- **Token Validation**: For each request to a protected route, the client includes the token in the request header (Authorization: Bearer <token>). The server then verifies the token's authenticity and allows or denies access to the resource.

Example of JWT Token Generation:

javascript

Copy code

```
// In authController.js



const jwt = require('jsonwebtoken');

exports.login = async (req, res) => {

  const { email, password } = req.body;

  // Validate user credentials (assuming User model has been set up)

  const user = await User.findOne({ email });
  if (!user || !(await user.comparePassword(password))) {

    return res.status(401).json({ message: 'Invalid credentials' });

  }

  // Generate JWT with user ID

  const token = jwt.sign({ id: user._id }, process.env.JWT_SECRET, {

    expiresIn: '1h', // Token expiration time

  });

  res.json({ token });

};
```

2. Authorization

Authorization ensures that only users with the correct permissions can access specific routes. JWTs also play a role in authorization by including user role information or other identifiers within the token payload.

Protecting Routes with Middleware:

- **JWT Verification**: Middleware is created to verify that the token sent in the Authorization header is valid.
- **User Role Checks**: After validating the token, the server can check the user’s role or ID, allowing or denying access to specific resources based on permissions.

Example Middleware for Protecting Routes:

javascript

Copy code

```
// In authMiddleware.js

const jwt = require('jsonwebtoken');

const protect = (req, res, next) => {

  const token = req.headers.authorization?.split(" ")[1]; // Extract token from 'Bearer <token>'
  if (!token) {
    return res.status(401).json({ message: 'No token, authorization denied' });
  }
  try {
    // Verify token and decode payload
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded; // Attach decoded user data to request object
    next();
  } catch (error) {
    res.status(401).json({ message: 'Token is not valid' });
  }
};
module.exports = protect;
```

Applying the Middleware:

You can apply this protect middleware to any route you want to secure. For example, only authenticated users can create, update, or delete carousel items.

javascript

```
// In carouselRoutes.js
const express = require('express');
const router = express.Router();
const protect = require('../middleware/authMiddleware');
const carouselController = require('../controllers/carouselController');
// Protected route - only authenticated users can access
router.post('/', protect, carouselController.addCarouselItem);
router.put('/:id', protect, carouselController.updateCarouselItem);
router.delete('/:id', protect, carouselController.deleteCarouselItem);
```

3. Session Management

Since JWT is stateless, there is no session data stored on the server. Each request to the server requires the client to include a valid JWT token. This eliminates the need for traditional server sessions and makes the application more scalable.

4. Token Expiration and Refresh

For added security, tokens are set to expire after a specific duration (e.g., 1 hour). When a token expires, users are required to re-authenticate. If needed, a refresh token mechanism can be implemented:

- **Access Token**: Short-lived token used for actual API requests.
- **Refresh Token**: Long-lived token used to request a new access token after the previous one expires.

5. Storing Tokens on the Client-Side

- **Local Storage**: Simple and persistent but vulnerable to cross-site scripting (XSS) attacks. Useful for storing tokens in development environments.
- **HTTP-Only Cookies**: More secure, as they cannot be accessed via JavaScript, making them resistant to XSS attacks.

User Interface

To showcase different UI features effectively, here are some suggestions for screenshots or GIFs you could create to highlight the user experience and functionality:

1. Home Page

- **Screenshot/GIF**: Show the homepage layout, highlighting the main navigation, featured sections, and any banners or carousel sliders.
- **Description**: This gives an overview of the app’s primary layout and first impression.

2. Carousel or Featured Items

- **Screenshot/GIF**: Show the carousel slider in action, cycling through featured items or promotions.
- **Description**: Emphasizes the visual appeal and interactive elements that draw users in.

3. Product Catalog

- **Screenshot/GIF**: Display the product catalog page, with a grid of items, search functionality, filters, and sorting options.
- **Description**: Shows how users can browse, search, and filter products for easy navigation.

4. Product Details Page

- **Screenshot**: Highlight the product detail page, showcasing the product image, price, description, and add-to-cart button.
- **Description**: Demonstrates the information users have at their fingertips when deciding on a purchase.

5. Shopping Cart

- **Screenshot/GIF**: Show the shopping cart page, displaying items added, quantity adjustments, and total price.
- **Description**: Illustrates how users can view, update, or proceed to checkout with their selected items.

6. User Authentication (Login/Signup)

- **Screenshot/GIF**: Capture the login or signup form, including error handling for incorrect inputs.

- **Description**: Shows the authentication process and highlights any validation features for a smooth user experience.

7. Checkout Process

- **GIF**: Display the steps involved in the checkout process, from entering shipping details to placing an order.
- **Description**: Walks users through completing a purchase, showcasing ease of use and security of transactions.

8. Admin Dashboard

- **Screenshot**: Display the admin dashboard, showing key functionalities like managing products, orders, and viewing analytics.
- **Description**: Highlights backend management features that support efficient store operations.

9. Responsive Design

- **Screenshot/GIF**: Show the app on both desktop and mobile views to demonstrate responsive design.
- **Description**: Highlights adaptability to various screen sizes for a consistent user experience.

Testing

Testing Strategy for the Project

A robust testing strategy is essential to ensure that the application functions as expected and provides a smooth user experience. The strategy will focus on both **frontend** and **backend** testing, as well as ensuring overall system integration. Here's a comprehensive testing strategy:

1. Types of Testing

a. Unit Testing:

- **Goal**: To test individual components or functions for correctness.
- **Tools**:
  - **Frontend**: Jest and React Testing Library
  - **Backend**: Mocha, Chai, and Supertest
- **Description**: Each small function or unit in the application is tested in isolation. For example, testing React component rendering, function outputs, or API routes.

b. Integration Testing:

- **Goal**: To test how different parts of the application work together, ensuring that data flows correctly between components and the backend.
- **Tools**:
  - **Frontend**: React Testing Library, Jest
  - **Backend**: Mocha, Chai, Supertest
- **Description**: Tests interactions between different modules, such as testing the login form with the authentication API to verify data flow.

c. End-to-End (E2E) Testing:

- **Goal**: To test the complete flow of the application from the user's perspective, simulating real user interactions with the frontend and backend.
- **Tools**: Cypress, Selenium
- **Description**: E2E testing ensures that the entire application functions as expected when integrated, including interaction with the backend API, form submissions, and UI rendering.

d. Performance Testing:

- **Goal**: To ensure the application performs well under load, especially the backend server.
- **Tools**:
  - **Frontend**: Lighthouse (built-in to Chrome DevTools)

    - **Backend**: Apache Jmeter, Artillery
- **Description**: Performance testing ensures that the app can handle multiple concurrent users without issues. It's especially important for high-traffic e-commerce applications.

e. User Interface (UI) Testing:

- **Goal**: To check the visual elements, layout, and responsiveness across different devices and screen sizes.
- **Tools**: Cypress, Puppeteer, Percy
- **Description**: UI testing checks that the UI behaves correctly on different screen sizes, devices, and browsers, making sure the design is consistent.

f. Security Testing:

- **Goal**: To identify vulnerabilities in the application.
- **Tools**: OWASP ZAP, Postman for API testing
- **Description**: Security testing ensures that the app is protected against common attacks such as SQL injection, XSS, CSRF, etc., and that sensitive data (like passwords) is stored and transmitted securely.

g. Regression Testing:

- **Goal**: To verify that new changes (e.g., feature additions, bug fixes) haven't broken existing functionality.
- **Tools**: Jest, Cypress
- **Description**: After making updates to the app, regression tests are run to ensure that existing features continue to work as expected.

---

2. Tools Used in the Project

Frontend Testing Tools:

1. **Jest**:
    - **Purpose**: A JavaScript testing framework used to run unit and integration tests.
    - **Use Case**: Jest is used for unit tests of React components, checking if they render properly and function as expected.
2. **React Testing Library**:
    - **Purpose**: A library for testing React components in a way that simulates real user interactions.
    - **Use Case**: React Testing Library helps simulate and test user behavior on the React components, such as submitting forms, clicking buttons, and checking if the UI updates correctly.

3. **Cypress**:
    - **Purpose**: An end-to-end testing framework that runs in the browser and allows full interaction with the application.
    - **Use Case**: Cypress is used to test complete workflows in the application, such as logging in, adding items to the cart, and completing the checkout process.

Backend Testing Tools:

1. **Mocha**:
    - **Purpose**: A feature-rich JavaScript testing framework used for backend unit testing.
    - **Use Case**: Mocha is used to write backend tests for individual API routes, such as checking the functionality of POST, GET, PUT, and DELETE endpoints.
2. **Chai**:
    - **Purpose**: An assertion library for Node.js, used in combination with Mocha for testing.
    - **Use Case**: Chai provides syntax for writing expectations and assertions in backend tests, making the tests more readable.
3. **Supertest**:
    - **Purpose**: A supercharged HTTP request testing library for Node.js.
    - **Use Case**: Supertest is used to simulate API requests (GET, POST, PUT, DELETE) in backend tests, and it integrates well with Mocha and Chai for validating API responses.
4. **Postman**:
    - **Purpose**: A popular API testing tool used for sending requests to API endpoints and checking responses.
    - **Use Case**: Postman is used for manual API testing to ensure the correctness and functionality of backend routes before automating the tests.

---

4. Test Coverage and Continuous Integration (CI)

Test Coverage:

- **Goal**: To ensure that all critical parts of the codebase are tested, reducing the chance of bugs and regressions.
- **Tools**: Jest and Istanbul (via jest –coverage) for generating test coverage reports.
- **Description**: The test suite should cover major functionalities, such as user authentication, CRUD operations, and payment flows.

Continuous Integration (CI):

- **Goal**: To automatically run tests on every code commit and merge to ensure the codebase remains functional.
- **Tools**: GitHub Actions, Travis CI, CircleCI
- **Description**: CI ensures that the tests run on each pull request or commit, catching issues early. If any tests fail, the CI system alerts developers to fix the issues before merging the changes into the main branch.

---

4. Testing Process

1. **Write Tests**:
   - Developers write unit, integration, and E2E tests for frontend and backend features.
2. **Run Tests Locally**:
   - Developers run tests locally using Jest, Mocha, or Cypress before committing code.
3. **CI/CD Pipeline**:
   - The code is pushed to a version control system (like GitHub). CI tools automatically trigger tests and deploy the app if all tests pass.
4. **Test Reporting**:
   - The test results are displayed on the CI dashboard, providing a clear view of test coverage and failures.

---

5. Example of Running Tests Locally

- **Frontend** (React):

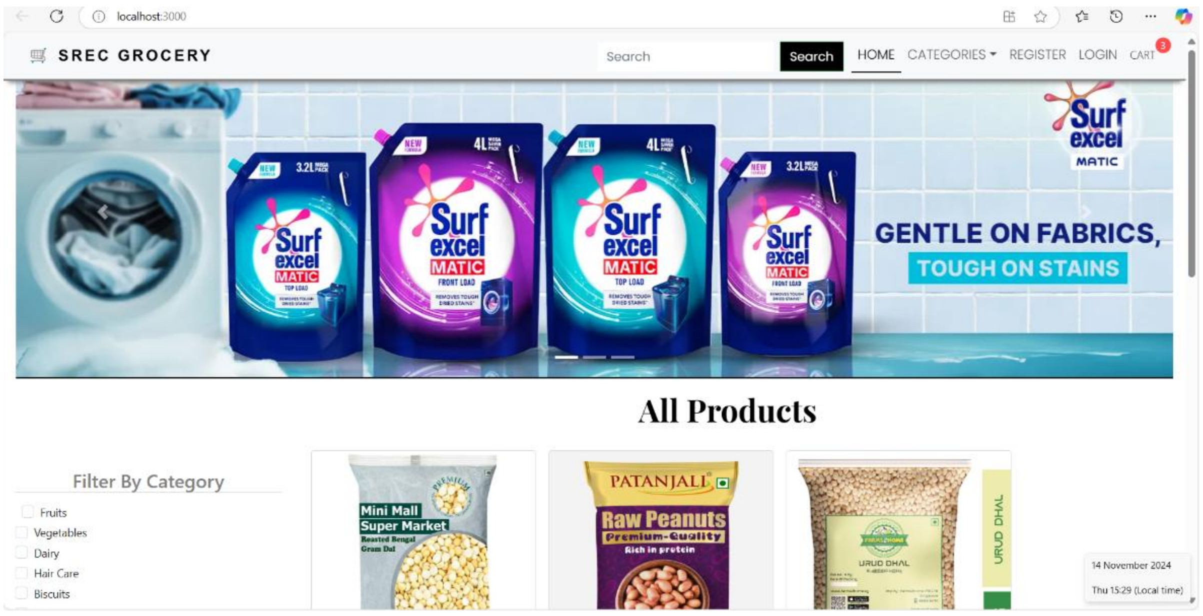
```
npm start
```

- **Backend** (Node.js):

```
npm run server
```

11. Screenshots or Demo

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



12. Known Issues

Online payment issue in paypal and there is developer error.

13. Future Enhancements

- Outline potential future features or improvements that could be made to the project.