Frontend Development with React.js

Project Documentation format

1. **Introduction**
    - **Project Title**: [Your Project Title]
    - **Team Members**: List team members and their roles.
2. **Project Overview**
    - **Purpose**: Briefly describe the purpose and goals of the project.
    - **Features**: Highlight the key features and functionalities of the frontend.
3. **Architecture**
    - **Component Structure**: Outline the structure of major React components and how they interact.
    - **State Management**: Describe the state management approach used (e.g., Context API, Redux).
    - **Routing**: Explain the routing structure if using react-router or another routing library.
4. **Setup Instructions**
    - **Prerequisites**: List software dependencies (e.g., Node.js).
    - **Installation**: Provide a step-by-step guide to clone the repository, install dependencies, and configure environment variables.
5. **Folder Structure**
    - **Client**: Describe the organization of the React application, including folders like components, pages, assets, etc.
    - **Utilities**: Explain any helper functions, utility classes, or custom hooks used in the project.
6. **Running the Application**
    - Provide commands to start the frontend server locally.
        - **Frontend**: npm start in the client directory.
7. **Component Documentation**
    - **Key Components**: Document major components, their purpose, and any props they receive.
    - **Reusable Components**: Detail any reusable components and their configurations.
8. **State Management**
    - **Global State**: Describe global state management and how state flows across the application.
    - **Local State**: Explain the handling of local states within components.
9. **User Interface**
    - Provide screenshots or GIFs showcasing different UI features, such as pages, forms, or interactions.
10. **Styling**

- **CSS Frameworks/Libraries**: Describe any CSS frameworks, libraries, or pre-processors (e.g., Sass, Styled-Components) used.
- **Theming**: Explain if theming or custom design systems are implemented.

11. **Testing**

- **Testing Strategy**: Describe the testing approach for components, including unit, integration, and end-to-end testing (e.g., using Jest, React Testing Library).
- **Code Coverage**: Explain any tools or techniques used for ensuring adequate test coverage.

12. **Screenshots or Demo**

- Provide screenshots or a link to a demo showcasing the application's features and design.

13. **Known Issues**

- Document any known bugs or issues that users or developers should be aware of.

14. **Future Enhancements**

- Outline potential future features or improvements, such as new components, animations, or enhanced styling.

PROJECT TITLE: Grocery webapp

Team Detail:

Member	**College ID**	**Role**
Sujeet Kumar Saurabh	23CS094	Frontend and Backend Developer

<u>Project Overview</u>

Purpose:-

The primary purpose of this project is to design and develop a fully functional online grocery shopping platform that offers customers a convenient way to purchase groceries from the comfort of their homes. The application aims to streamline the shopping process by integrating an intuitive user interface, robust search capabilities, and secure checkout functionality.

From a business perspective, the platform provides retailers with a scalable and efficient way to reach more customers, manage inventory, and process orders online.

From a technical perspective, the project serves as a demonstration of full-stack web development skills using the MERN stack — MongoDB for database management, Express.js for backend API handling, React for building a responsive and interactive user interface, and Node.js for server-side execution.

The goals of the project are:

1. To create a user-friendly online marketplace for grocery items.
2. To ensure cross-platform compatibility for a seamless experience on desktops, tablets, and mobile devices.
3. To implement real-time product search and filtering for faster shopping.
4. To provide secure user authentication and order management.
5. To maintain scalability for future enhancements like payment integration, order tracking, and personalized recommendations.
6. To follow industry-standard coding practices, making the application maintainable and extensible for future developers.

Frontend Features:-

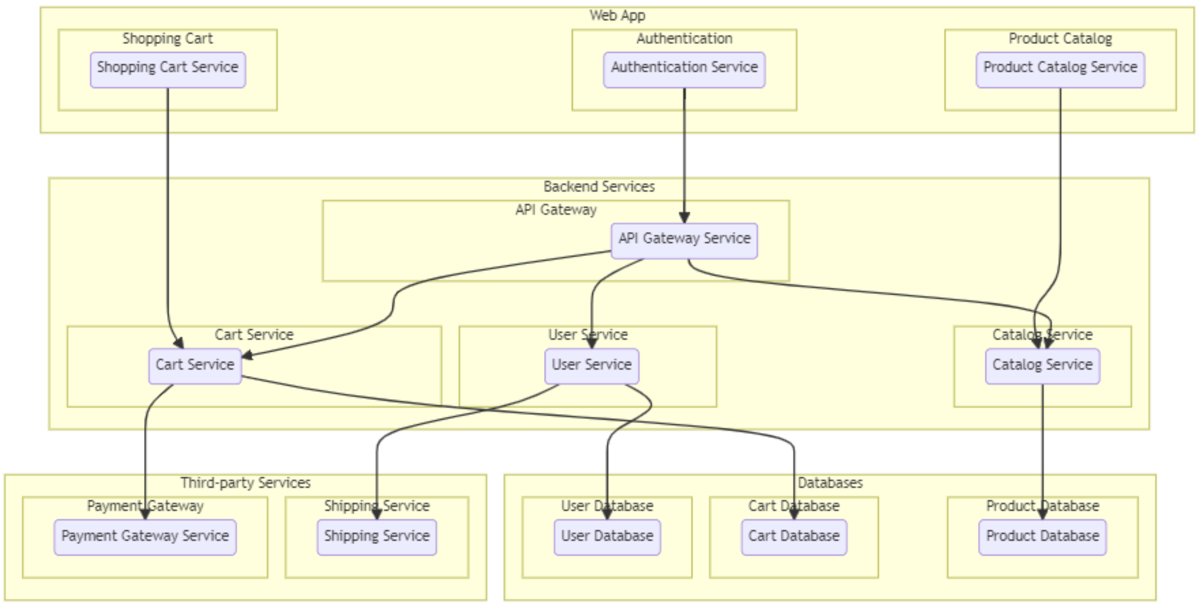
The frontend of the Grocery Webapp is designed with a focus on usability, speed, and aesthetics, ensuring an engaging shopping experience for end users while following modern web design standards.

Key Features:

1. Responsive Design – Developed using React and Tailwind CSS, the interface adapts seamlessly to different screen sizes and devices.
2. Dynamic Product Display – Products are fetched from the backend in real-time, ensuring updated information for prices, availability, and descriptions.
3. Category-based Navigation – Users can browse products by categories, making it easier to locate specific types of grocery items.
4. Real-time Search & Filtering – Instant search results with filtering options based on categories, price range, or other attributes, without reloading the page.
5. Interactive Cart System – Users can add, remove, or update product quantities directly from the cart page. Cart data persists even after a page refresh.
6. Smooth Page Transitions – Implemented with React Router for a single-page application (SPA) experience.
7. User Authentication Pages – Dedicated signup and login forms with validation feedback for better usability.
8. Order Checkout Flow – A step-by-step guided checkout process for placing orders quickly.
9. Consistent Styling & Branding – Tailored colour schemes, typography, and layout for a professional look and feel.
10. Performance Optimization – Use of lazy loading, minimized API calls, and efficient state management with React hooks to ensure fast rendering.
11. Error Handling & Feedback – Clear messages for empty carts, invalid form entries, or unavailable products to guide users effectively.

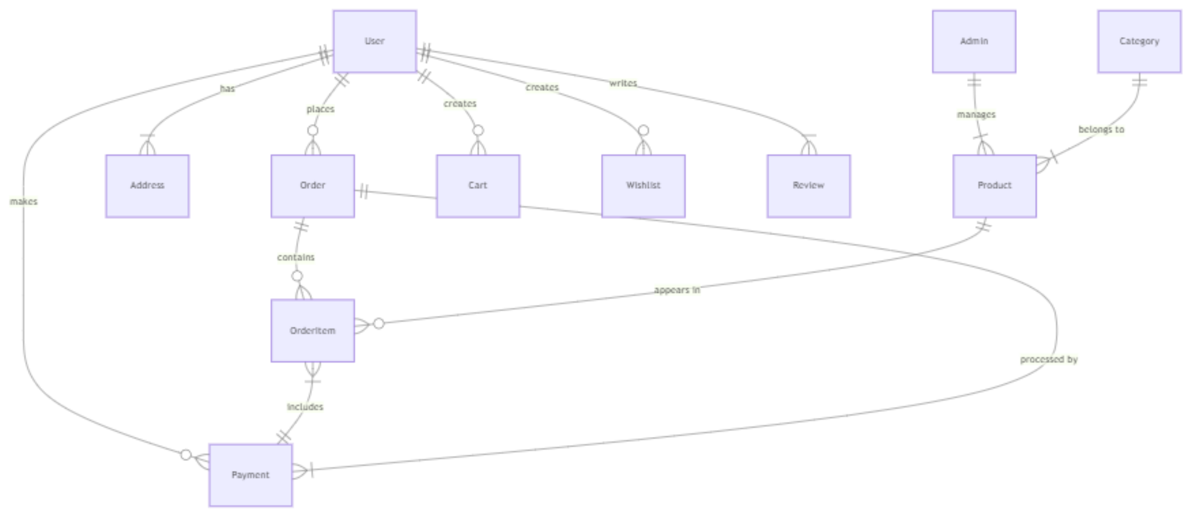
Architecture

Technical 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.

ER Diagram:



The Entity-Relationship (ER) diagram for an flower and gift delivery app visually represents the relationships between different entities involved in the system, such as users, products, orders, and reviews. It illustrates how these entities are related to each other and helps in understanding the overall database structure and data flow within the application.

Pre-requisites:-

To develop a full-stack Grocery web app using AngularJS, Node.js, and MongoDB, there are several prerequisites you should consider. Here are the key prerequisites for developing such an application:

Node.js and npm: Install Node.js, which includes npm (Node Package Manager), on your development machine. Node.js is required to run JavaScript on the server side.

- Download: https://nodejs.org/en/download/
- Installation instructions: https://nodejs.org/en/download/package-manager/

MongoDB: Set up a MongoDB database to store hotel and booking information. Install MongoDB locally or use a cloud-based MongoDB service.

- Download: https://www.mongodb.com/try/download/community
- Installation instructions: https://docs.mongodb.com/manual/installation/

Express.js: Express.js is a web application framework for Node.js. Install Express.js to handle server-side routing, middleware, and API development.

- Installation: Open your command prompt or terminal and run the following command: npm install express

Angular: Angular is a JavaScript framework for building client-side applications. Install Angular CLI (Command Line Interface) globally to create and manage your Angular project.

Install Angular CLI:

- Angular provides a command-line interface (CLI) tool that helps with project setup and development.
- Install the Angular CLI globally by running the following command:

  npm install -g @angular/cli

Verify the Angular CLI installation:

- Run the following command to verify that the Angular CLI is installed correctly: ng version

You should see the version of the Angular CLI printed in the terminal if the installation was successful.

Create a new Angular project:

- Choose or create a directory where you want to set up your Angular project.
- Open your terminal or command prompt.
- Navigate to the selected directory using the cd command.
- Create a new Angular project by running the following command: ng new client Wait for the project to be created:
- The Angular CLI will generate the basic project structure and install the necessary dependencies

Navigate into the project directory:

- After the project creation is complete, navigate into the project directory by running the following command: cd client

Start the development server:

- To launch the development server and see your Angular app in the browser, run the following command: ng serve / npm start
- The Angular CLI will compile your app and start the development server.
- Open your web browser and navigate to http://localhost:4200 to see your Angular app running.

You have successfully set up Angular on your machine and created a new An- gular project. You can now start building your app by modifying the generated project files in the src directory.

Please note that these instructions provide a basic setup for Angular. You can explore more ad- vanced configurations and features by referring to the official Angular documentation: https://angular.io

HTML, CSS, and JavaScript: Basic knowledge of HTML for creating the structure of your app, CSS for styling, and JavaScript for client-side interactivity is essential.

Database Connectivity: Use a MongoDB driver or an Object-Document Mapping (ODM) library like Mongoose to connect your Node.js server with the MongoDB database and perform CRUD (Create, Read, Update, Delete) operations.

Front-end Framework: Utilize Angular to build the user-facing part of the application, including products listings, booking forms, and user interfaces for the admin dashboard.

Version Control: Use Git for version control, enabling collaboration and tracking changes throughout the development process. Platforms like GitHub or Bitbucket can host your repository.

- Git: Download and installation instructions can be found at: https://git-scm.com/downloads

Development Environment: Choose a code editor or Integrated Development Environment (IDE) that suits your preferences, such as Visual Studio Code, Sublime Text, or WebStorm.

- Visual Studio Code: Download from https://code.visualstudio.com/download
- Sublime Text: Download from https://www.sublimetext.com/download
- WebStorm: Download from https://www.jetbrains.com/webstorm/download

To Connect the Database with Node JS go through the below provided link:

• Link: https://www.section.io/engineering-education/nodejs- mongoosejs-mongodb/

Installation:

To run the existing grocery-web app project downloaded from github:

Follow below steps:

1. Clone the Repository:

- Open your terminal or command prompt.
- Navigate to the directory where you want to store the grocery-webapp app.
- Execute the following command to clone the repository:

git clone https://github.com/Sujeet8042/Grocery-webapp

2. Install Dependencies:

- Navigate into the cloned repository directory:

cd grocery-webapp

- Install the required dependencies by running the following command:

npm install

3. Start the Development Server:

- To start the development server, execute the following command:

npm run dev or npm run start

- The e-commerce app will be accessible at http://localhost:5100 by default. You can change the port configuration in the .env file if needed.

4. Access the App:

- Open your web browser and navigate to http://localhost:5100.
- You should see the grocery-webapp app's homepage, indicating that the installation and setup were successful.

Video Tutorial Link to clone the project:
- https://drive.google.com/file/d/1uLULSx0heidWUYRRUlm_QbmLfaO8oBRf/view?usp=sharing

Project Repository Link: https://github.com/Sujeet8042/Grocery-webapp

Congratulations! You have successfully installed and set up the grocery-webapp app on your local machine. You can now proceed with further customization, development, and testing as needed.

Folder Structure

Client:

The client folder contains the React.js code that powers the user-facing interface of the Grocery Webapp. Its structure is organized for scalability, reusability, and maintainability.

1. src/ (Source Code Root)

This is where all the application logic and UI code for the frontend resides.

2. components/

- Contains reusable UI building blocks that are shared across different pages.
- Examples:
    - Navbar.jsx – The top navigation bar for the app.
    - Footer.jsx – The bottom footer displayed on all pages.
    - ProductCard.jsx – Displays individual product details in card format.
    - SearchBar.jsx – Allows users to search for products.
- Purpose: Keeps the UI modular so changes in one component don't affect the whole app.

3. pages/

- Holds page-level components that correspond to routes in the app.
- Each file in this folder represents a full screen/page the user can visit.
- Examples:
    - Home.jsx – Displays featured products and promotions.
    - Products.jsx – Shows the product listing page.
    - Cart.jsx – Displays products added to the cart.
    - Checkout.jsx – Handles order placement.
- Purpose: Separates complete views from small UI components.

4. assets/

- Stores static files used by the frontend.
- Examples:

    - Images: Product images, banners, icons (logo.png, banner.jpg).
    - Styles: CSS/SCSS files for custom styling.
- Purpose: Keeps design-related files organized and separate from logic.

5. App.js

- The root React component that holds the main structure of the app.
- Responsible for:
    - Setting up routes (if using react-router-dom).
    - Wrapping components in layout containers.

6. index.js

- Entry point for the React application.
- Mounts the <App /> component to the HTML DOM (public/index.html).
- Loads global styles and libraries.

7. routes/ *(if implemented)*

- Contains routing configuration using react-router-dom.
- Example:
    - AppRoutes.js – Defines URL paths and maps them to components/pages.

8. context/ or store/ *(if implemented)*

- Holds global state management logic (like Context API or Redux).
- Example:
    - CartContext.js – Manages cart state across pages.

Utilities:-

The utilities section (or similar folder like src/utils/) contains **helper functions, custom hooks, and utility classes** that simplify common tasks and promote code reuse:

- **Helper Functions**
  Small, reusable JavaScript functions for tasks such as formatting prices, validating forms, or filtering products.
  Example: formatCurrency.js – formats numbers into currency format.
- **Custom Hooks**
  React hooks that encapsulate reusable logic, such as API fetching or state management.
  Example: useFetchProducts.js – handles fetching products from the backend.
- **Configuration Files**
  Files that store API endpoints, constants, or environment variables for easy access.
  Example: config.js – contains BASE_URL for API calls.

Running the Application:-

To run the frontend of the Grocery Webapp locally, follow these steps:

1. Navigate to the Client Directory
   Open a terminal or command prompt, then move into the client folder of the project:

```
cd client
```

2. **Install Dependencies**
   Ensure all required packages are installed:

```
npm install
```

3. **Start the Development Server**
   Run the following command to start the frontend:

```
npm run dev
```

4. **Access the Application**
   Once the server is running, open your browser and navigate to:

```
http://localhost:5173
```

Component Documentation:-

1. Key Components:-

These are the main React components responsible for rendering specific pages or core functionalities of the Grocery Webapp.
Each key component handles a distinct feature, often integrating API calls, state management, and UI rendering.

Examples:

- **Navbar.jsx**
  - **Purpose:** Provides site-wide navigation, including links to Home, Products, Cart, and Profile.
  - **Props:**
    - user *(object)* — Contains logged-in user details for showing profile info.
    - onLogout *(function)* — Handles user logout action.
- **ProductList.jsx**
  - **Purpose:** Displays a grid/list of grocery items fetched from the backend.
  - **Props:**
    - products *(array)* — List of product objects with name, price, and image.
    - onAddToCart *(function)* — Callback for adding items to the shopping cart.
- **Cart.jsx**
  - **Purpose:** Shows all products added to the cart, allowing quantity adjustments or item removal.
  - **Props:**
    - cartItems *(array)* — Items in the cart with quantity and price details.
    - onCheckout *(function)* — Initiates the checkout process.
- **Checkout.jsx**
  - **Purpose:** Handles billing details, payment options, and order confirmation.
  - **Props:**
    - orderSummary *(object)* — Contains cart total, taxes, and final price.

2. Reusable Components:-

Reusable components are designed for **consistency and efficiency** across the project.
They follow the DRY (Don't Repeat Yourself) principle, meaning the same UI logic can be used in multiple places without rewriting code.

Examples:

- **Button.jsx**
  - **Purpose:** Renders styled buttons for different actions (Add to Cart, Checkout, Submit).
  - **Props:**
    - label *(string)* — Text shown on the button.
    - onClick *(function)* — Action to perform when clicked.
    - type *(string)* — Defines style variant (primary, secondary, danger).
- **InputField.jsx**
  - **Purpose:** Provides a consistent styled input element for forms.
  - **Props:**
    - type *(string)* — Input type (text, number, email, password).
    - value *(string)* — Current input value.
    - onChange *(function)* — Updates state when the input changes.
    - placeholder *(string)* — Placeholder text.
- **ProductCard.jsx**
  - **Purpose:** Displays product image, name, price, and an "Add to Cart" button.
  - **Props:**
    - product *(object)* — Contains all product details.
    - onAddToCart *(function)* — Handles adding a product to the cart.

State Management:-

1. Global State

Global state refers to data that needs to be accessed and updated by multiple components across the application.
Instead of passing props down many levels (prop drilling), global state is stored in a centralized store using Context API or state management libraries like Redux, Zustand, or Recoil.

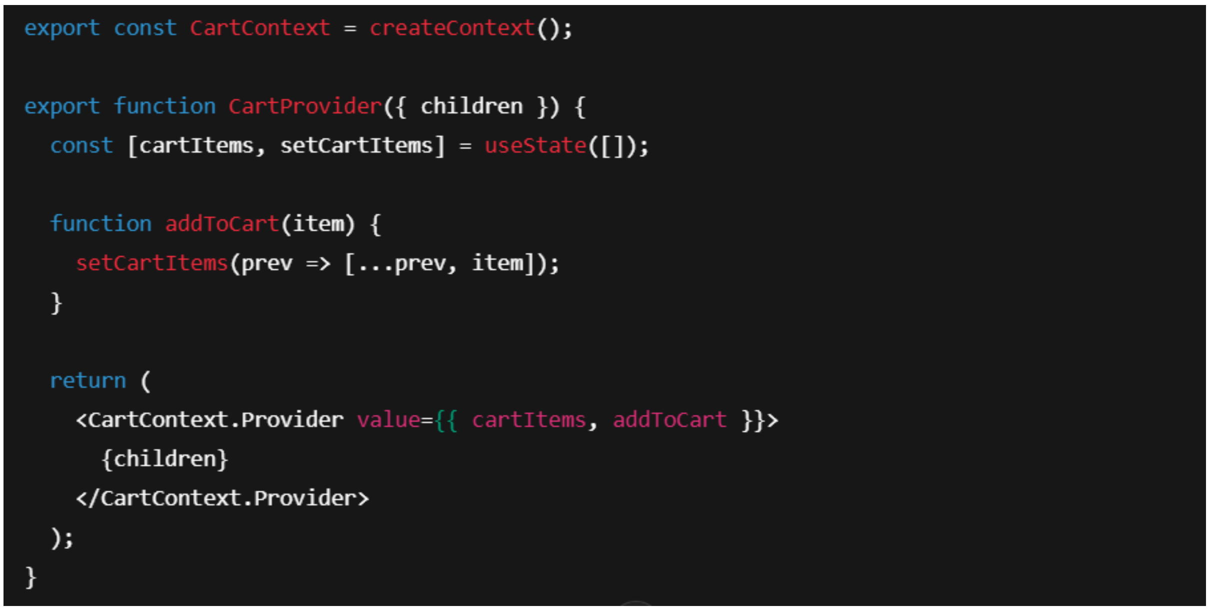
In the Grocery Webapp example:

- What is stored in global state?
    - Authentication info → Logged-in user data (username, token).
    - Shopping Cart → Products added to the cart, quantities, and prices.
    - Wishlist (if implemented) → Products marked for later purchase.
    - Theme settings → Light/Dark mode preferences.

How it flows:

1. Global store setup → For example, using React.createContext() to create a CartContext.
2. Provider component wraps the whole app (<App>), making the state available to all children.
3. Components read global state using hooks like useContext(CartContext) or Redux's useSelector().
4. Components update global state using actions like addToCart(product) or logoutUser().

Example with Context API:



```jsx
export const CartContext = createContext();

export function CartProvider({ children }) {
  const [cartItems, setCartItems] = useState([]);

  function addToCart(item) {
    setCartItems(prev => [...prev, item]);
  }

  return (
    <CartContext.Provider value={{ cartItems, addToCart }}>
      {children}
    </CartContext.Provider>
  );
}
```

This ensures state is shared across components like ProductList, Cart, and Checkout without repetitive prop passing.

2. Local State

Local state is **specific to a single component** and is not shared across the application. It's used for **temporary data or UI behaviour** that doesn't affect other components.

In the Grocery Webapp example:

- **Search bar input** → Text entered by the user in Navbar before hitting search.
- **Form handling** → Billing form fields inside Checkout.jsx.
- **Modal visibility** → Boolean value to show/hide a login popup.
- **Dropdown menus & filters** → Category selection in the product listing page.

How it flows:

1. **Stored in a component's useState() hook**.
2. **Only available inside that component**.
3. **Passed as props to child components if needed** (but not upwards unless using a callback).

Example:

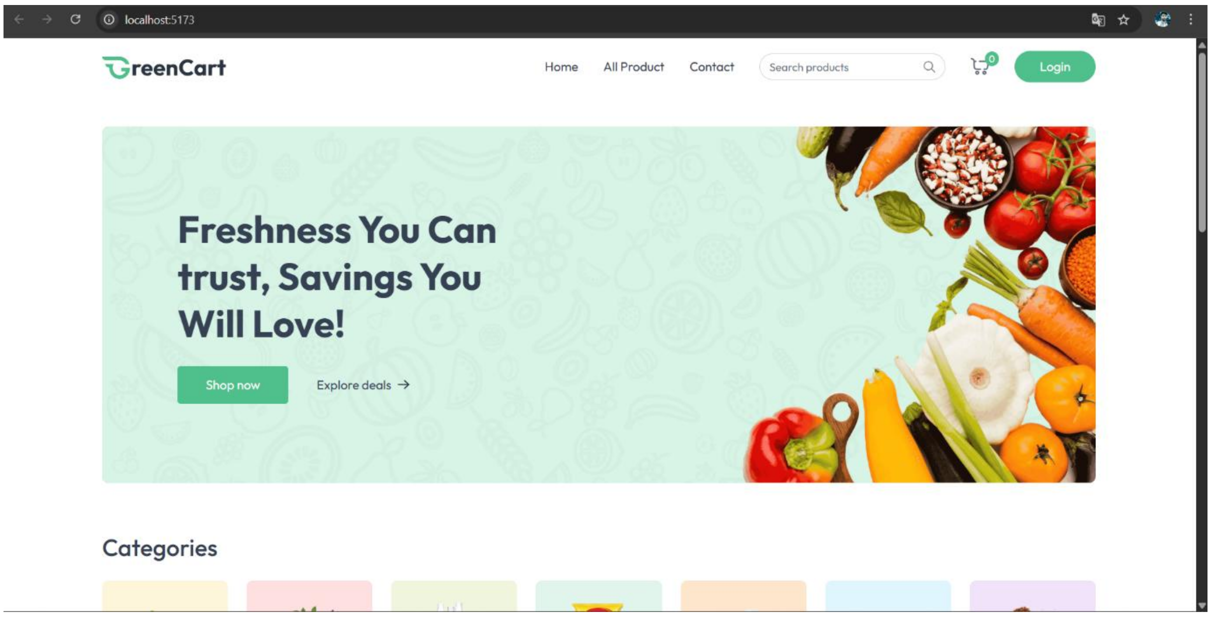
```
function SearchBar() {
  const [query, setQuery] = useState("");

  function handleSearch() {
    console.log("Searching for:", query);
  }

  return (
    <div>
      <input
        type="text"
        value={query}
        onChange={(e) => setQuery(e.target.value)}
        placeholder="Search products..."
      />
      <button onClick={handleSearch}>Search</button>
    </div>
  );
}
```

Here, query is local state because it’s only relevant inside the SearchBar component.

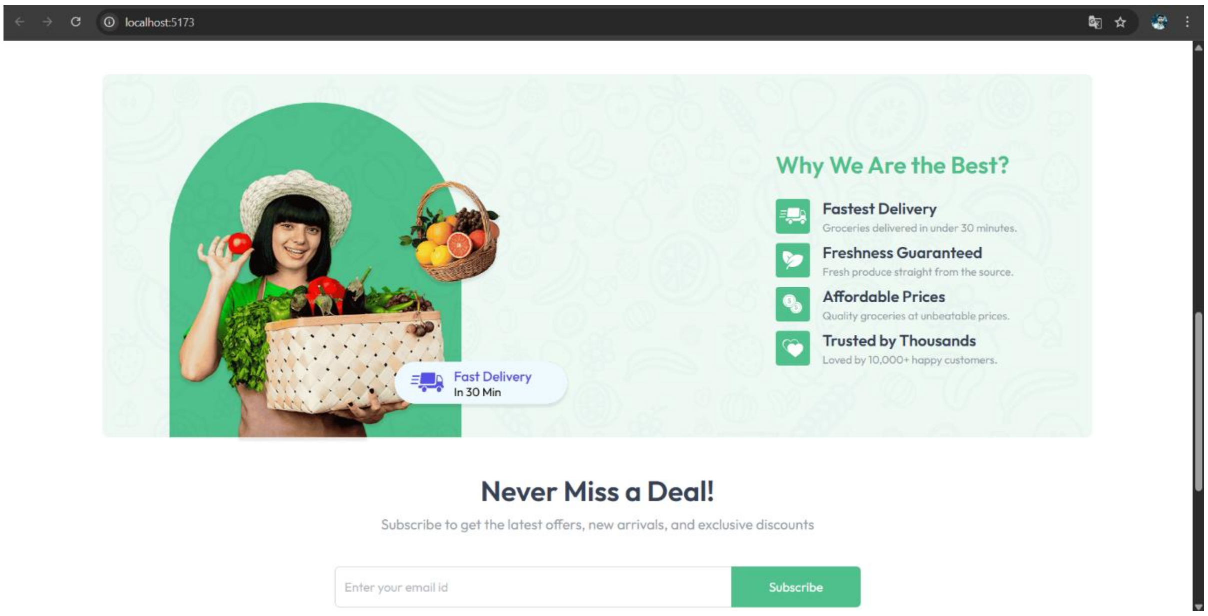
User Interface:-

This application provides a clean, responsive, and user-friendly interface, ensuring smooth navigation and accessibility across devices.
Below are the major UI features with corresponding screenshots :--

1. Home Page



Freshness You Can trust, Savings You Will Love!



Why We Are the Best?

Fastest Delivery
Groceries delivered in under 30 minutes.

Freshness Guaranteed
Fresh produce straight from the source.

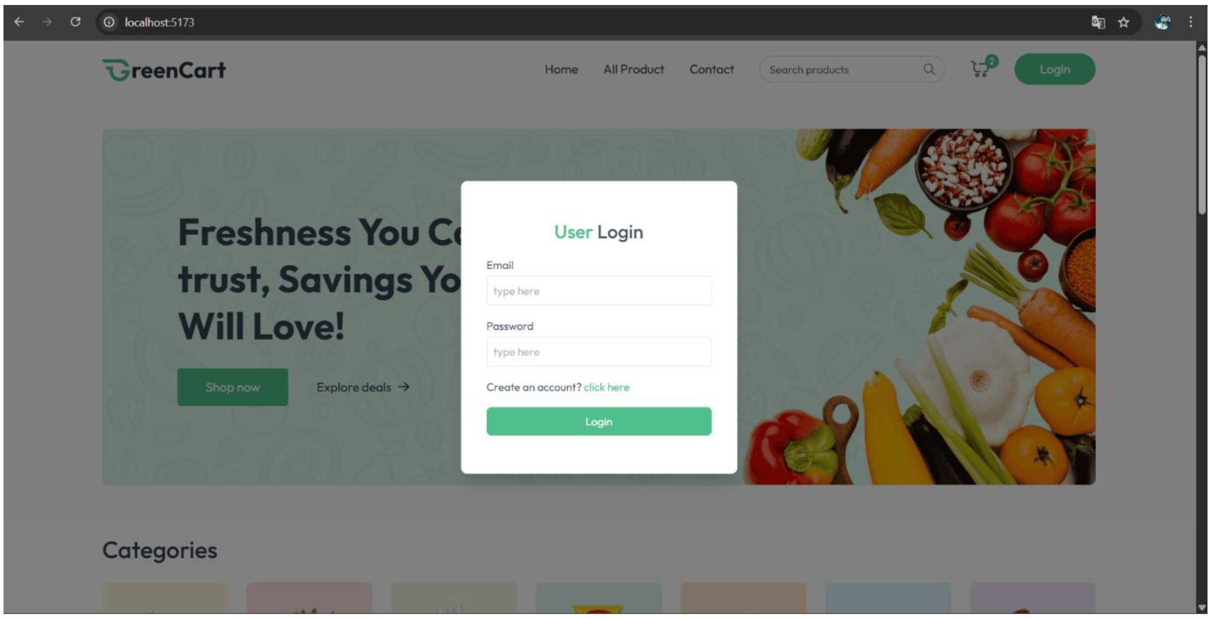
Affordable Prices
Quality groceries at unbeatable prices.

Trusted by Thousands
Loved by 10,000+ happy customers.

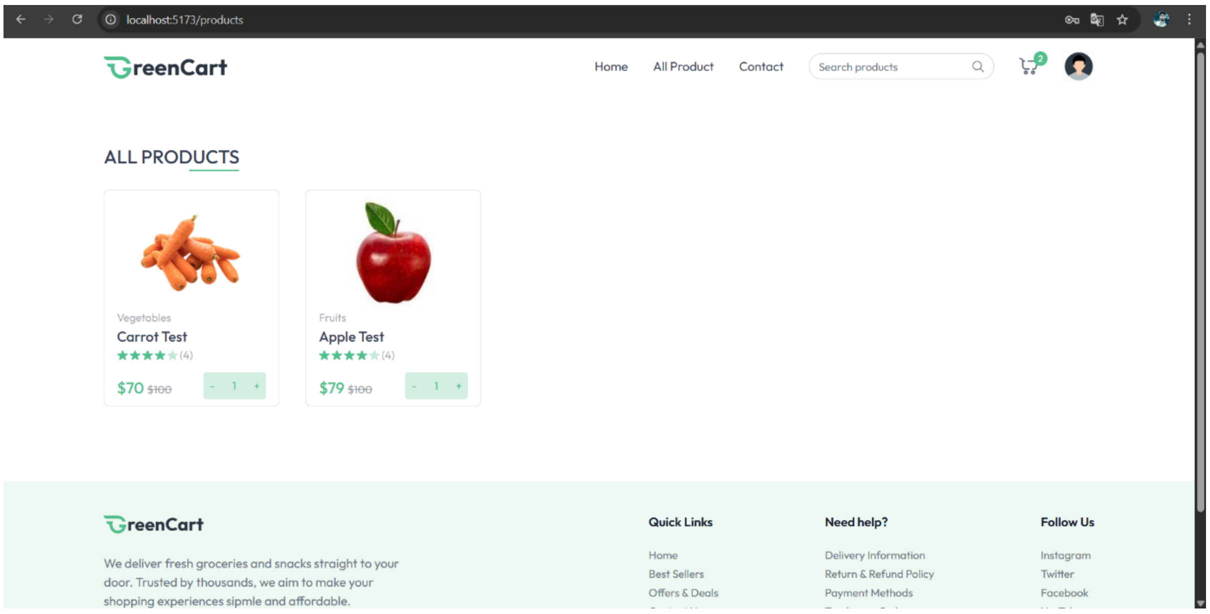
Never Miss a Deal!

Subscribe to get the latest offers, new arrivals, and exclusive discounts

2. Login Page

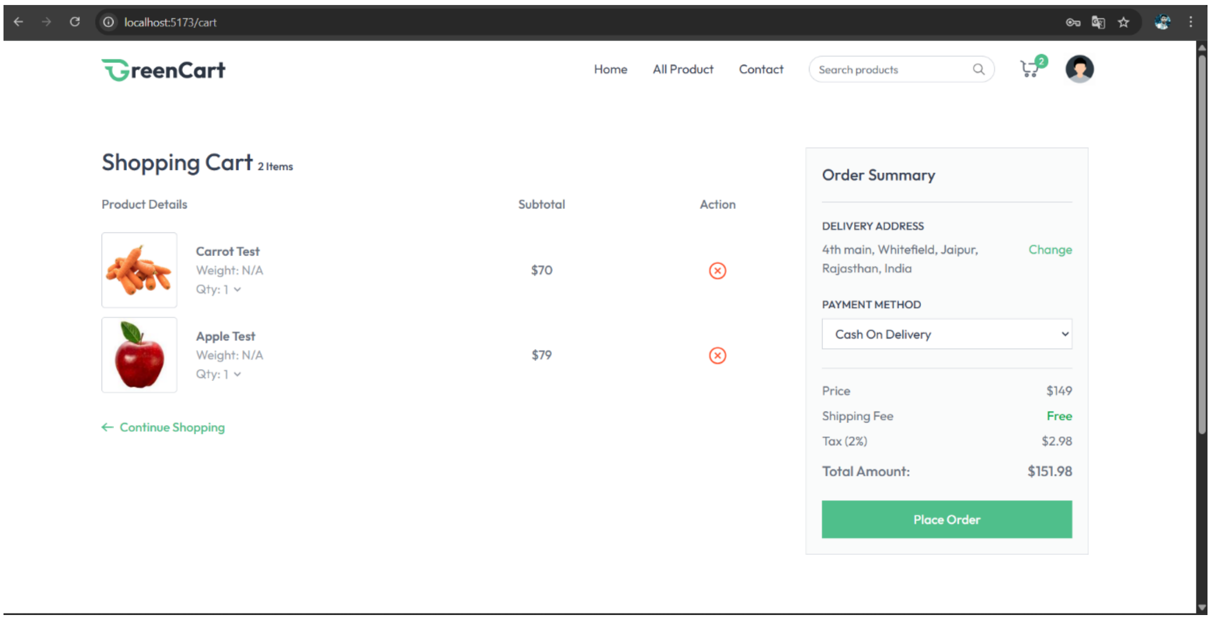


3. Product Listing Page



We deliver fresh groceries and snacks straight to your door. Trusted by thousands, we aim to make your shopping experiences sipmle and affordable.

4. Shopping Cart


localhost:5173/cart

GreenCart   Home   All Product   Contact   Search products

Shopping Cart 2 Items

Product Details	Subtotal	Action
Carrot Test Weight: N/A Qty: 1	$70	⊗
Apple Test Weight: N/A Qty: 1	$79	⊗

← Continue Shopping

Order Summary

DELIVERY ADDRESS

4th main, Whitefield, Jaipur, Rajasthan, India   Change

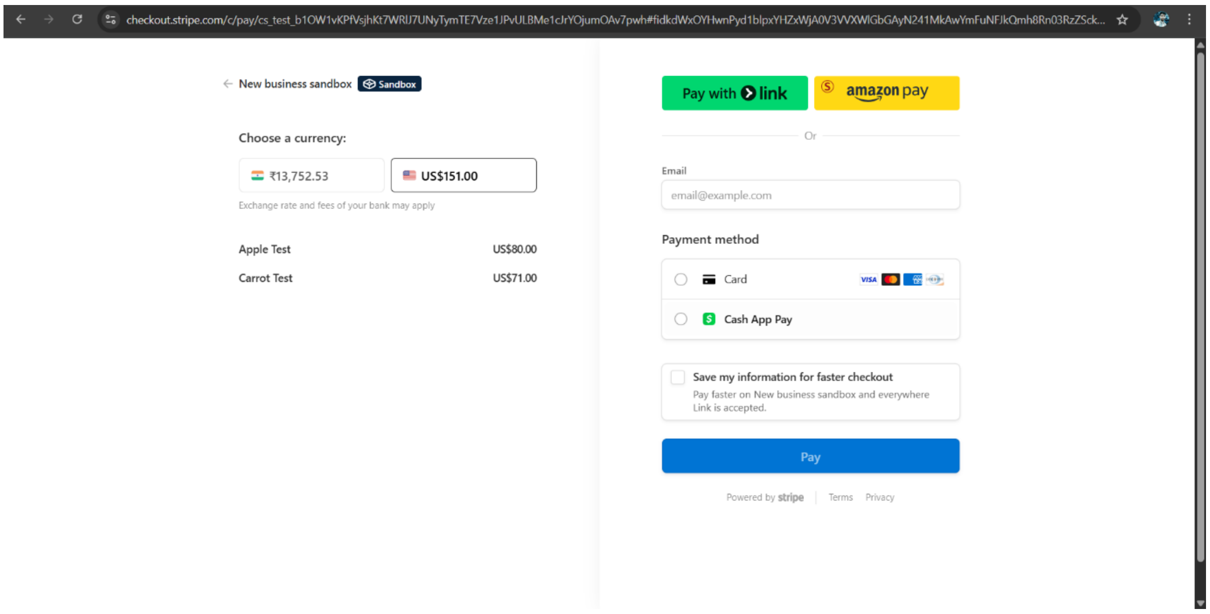
PAYMENT METHOD

Cash On Delivery

Price	$149
Shipping Fee	Free
Tax (2%)	$2.98
Total Amount:	$151.98

Place Order

5. Checkout Page


checkout.stripe.com/c/pay/cs_test_b1OW1vKPfVsjhKt7WRlJ7UNyTymTE7Vze1JPvULBMe1cJrYOjumOAv7pwh#fidkdWxOYHwnPyd1blpxYHZxWjA0V3VVXWlGbGAyN241MkAwYmFuNFJkQmh8Rn03RzZSck...

← New business sandbox Sandbox

Choose a currency:

₹13,752.53   US$151.00

Exchange rate and fees of your bank may apply

Apple Test	US$80.00
Carrot Test	US$71.00

Pay with link   amazon pay

Or

Email

email@example.com

Payment method

○ Card

○ Cash App Pay

☐ Save my information for faster checkout
Pay faster on New business sandbox and everywhere Link is accepted.

Pay

Powered by stripe | Terms Privacy

Styling:-

1. CSS Frameworks / Libraries

Description:

The project's visual styling is built using a combination of modern, developer-friendly frameworks and libraries to ensure **scalability, responsiveness, and maintainability**. This approach reduces redundant CSS, speeds up development, and keeps the UI consistent across all pages.

A. Tailwind CSS

- **Purpose:** A utility-first CSS framework that allows styling directly in the markup using pre-defined class names.
- **Advantages in this project:**
  - **Rapid Prototyping:** Developers can build responsive layouts faster without switching between HTML and CSS files.
  - **Consistent Design:** All styling follows a standardized spacing, color, and typography scale.
  - **Responsive Breakpoints:** Built-in breakpoints like sm, md, lg, xl ensure the UI adapts to different screen sizes without writing custom media queries.
  - **Dark Mode Support:** Native configuration for dark/light themes with minimal code changes.
  - **Hover and Focus States:** Easy state management using classes like hover:, focus:, active:.
- **Example Usage in the Project:**

```
<button className="bg-green-500 hover:bg-green-600 text-white font-semibold py-2 px-4 rounded-
  Add to Cart
</button>
```

B. React Icons *(If Implemented)*

- **Purpose:** Provides scalable and customizable icons directly in React components.
- **Benefits in the Project:**
  - **Lightweight:** No need to import entire icon libraries; import only required icons.
  - **Customizable with Tailwind:** Easily adjust icon size and color using Tailwind classes.
  - **Consistency:** Icons follow a unified visual style across the application.
- **Example Usage:**

```
import { FaShoppingCart } from 'react-icons/fa';

<FaShoppingCart className="text-xl text-green-500" />
```

C. Custom CSS Modules *(Optional but Recommended)*

- **Purpose:** Encapsulates CSS rules so they apply only to a specific component, preventing style leakage.
- **Why It's Used:**
  - Tailwind handles most of the styling, but unique or complex UI elements (animations, special effects) benefit from custom CSS.
  - Allows fine-grained control over specific UI parts without bloating the global stylesheet.
- **Example File Structure:**

```
import styles from './ProductCard.module.css';

<div className={styles.cardWrapper}>
  {/* Product details */}
</div>
```

2. Theming

Description:
The application follows a custom theme to maintain brand consistency:

- Primary Colour Palette:
  - Primary: #3B82F6 (Blue) – used for buttons, highlights, and links.
  - Secondary: #F59E0B (Amber) – used for accents and alerts.
  - Neutral Shades: Gray and White for backgrounds, text, and cards.
- Typography:
  - Font Family: "Inter", sans-serif for modern readability.
  - Font weights:
    - Bold for headings
    - Medium/Regular for body text
- Spacing & Layout:
  - Follows Tailwind’s spacing scale for padding and margins to ensure uniform spacing across components.
- Dark Mode Support:
  - Implemented using Tailwind’s dark: variant.
  - Allows users to switch between light and dark themes for better accessibility.
- Reusable Theme Classes:
  - Pre-defined utility classes for buttons, form elements, and cards to ensure visual consistency.

Testing:-

1. Testing Strategy

Purpose:

The testing strategy ensures that all application components behave as expected under different scenarios. It helps detect bugs early, maintain code reliability, and ensures that new updates do not break existing functionality.

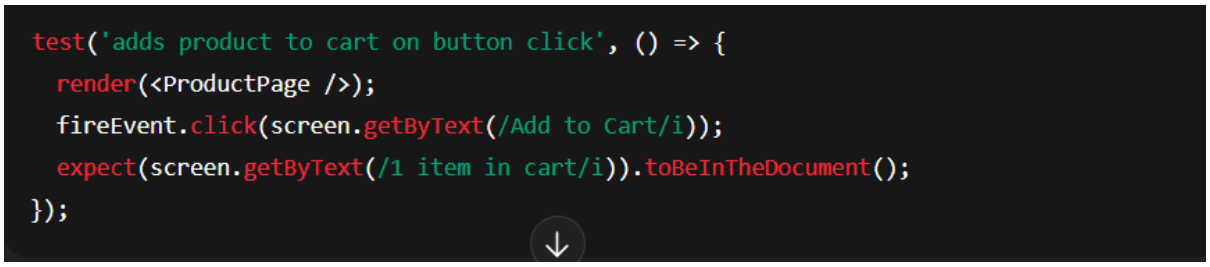
A. Unit Testing

- Objective: Validate that individual components, functions, or modules work correctly in isolation.
- Tool Used: Jest (JavaScript Testing Framework)
- Approach in Project:
  - Each component has a dedicated test file with the naming convention ComponentName.test.js.
  - Tests verify:
    - Rendering: Ensuring components render without errors.
    - Logic Execution: Confirming functions return expected outputs.
    - Props Handling: Verifying that components behave correctly with different prop values.
  - Example:

```
import { render, screen } from '@testing-library/react';
import ProductCard from './ProductCard';

test('renders product title', () => {
  render(<ProductCard title="Laptop" />);
  expect(screen.getByText(/Laptop/i)).toBeInTheDocument();
});
```

B. Integration Testing

- Objective: Test how multiple components work together and ensure correct data flow between them.
- Tool Used: React Testing Library (RTL) with Jest
- Approach in Project:
    - Simulate user interactions like clicks, form submissions, and navigation.
    - Test integration between:
        - UI components and APIs.
        - State management systems (Redux/Context) and UI rendering.
    - Example:



```
test('adds product to cart on button click', () => {
  render(<ProductPage />);
  fireEvent.click(screen.getByText(/Add to Cart/i));
  expect(screen.getByText(/1 item in cart/i)).toBeInTheDocument();
});
```

C. End-to-End (E2E) Testing

- **Objective:** Ensure that the **entire application works as intended** from a real user's perspective.
- **Tool Used: Cypress** (or Playwright if preferred)
- **Approach in Project:**
    - Tests mimic real-world user actions like **logging in, browsing products, adding to cart, and checking out**.
    - E2E tests run in a real browser environment to verify the app's behavior under realistic conditions.

- **Example:**

```
describe('E-commerce Flow', () => {
  it('should allow user to purchase a product', () => {
    cy.visit('/');
    cy.get('button').contains('Add to Cart').click();
    cy.get('button').contains('Checkout').click();
    cy.url().should('include', '/order-confirmation');
  });
});
```

2. Code Coverage

Purpose:

Code coverage ensures that a sufficient percentage of the application's code is tested. It helps identify **untested or under-tested areas** to improve reliability.

A. Tools Used

- **Jest with Istanbul** (Built-in in Jest)
  - Generates a detailed HTML report of coverage.
  - Tracks:
    - **Statements Coverage**: Percentage of executed code statements.
    - **Branch Coverage**: Percentage of executed decision branches (if/else).
    - **Function Coverage**: Percentage of functions invoked during tests.
    - **Line Coverage**: Percentage of lines executed.

B. Configuration & Execution

- **Command to Run Tests with Coverage:**

```
npm test -- --coverage
```

- Example Coverage Report Output:

```
----------------|---------|----------|---------|---------
File            | % Stmts | % Branch | % Funcs | % Lines
----------------|---------|----------|---------|---------
All files       |   92.45 |    85.71 |   90.00 |   92.45
ProductCard.js  |   100.0 |      100 |   100.0 |   100.0
----------------|---------|----------|---------|---------
```

C. Coverage Targets

- Minimum targets set in package.json to enforce quality:

```
"jest": {
  "coverageThreshold": {
    "global": {
      "branches": 80,
      "functions": 85,
      "lines": 85,
      "statements": 85
    }
  }
}
```

<u>Demo Video Link :-</u>

Known Issues :-

1. Product Search Delay
   - Description: When searching for products using the search bar, there is a noticeable delay before results appear.
   - Cause: The search function is making API requests on every keystroke instead of using a debounce mechanism.
   - Impact: Slightly reduced user experience due to slow response in search results.
   - Suggested Fix: Implement a debounce function to limit API calls until the user pauses typing.
2. Cart Quantity Update Bug
   - Description: Sometimes, updating the quantity of an item in the cart doesn't immediately reflect in the total price.
   - Cause: State update timing issue due to asynchronous operations in React.
   - Impact: Users might get incorrect total price until they refresh the page or perform another cart action.
   - Suggested Fix: Ensure the state is updated synchronously or trigger a re-render after quantity change.
3. Responsive Layout Issues
   - Description: On smaller mobile devices, certain UI elements like the navigation menu and product grid overflow outside the viewport.
   - Cause: Missing CSS breakpoints or incorrect flex/grid configuration.
   - Impact: Poor viewing experience on smaller screens.
   - Suggested Fix: Add responsive breakpoints and test the application on various device sizes.
4. Slow Initial Page Load
   - Description: The homepage takes slightly longer to load on slower networks.
   - Cause: Large image sizes and unoptimized assets.
   - Impact: Longer wait times for users on mobile data or slower internet.
   - Suggested Fix: Compress images, implement lazy loading, and enable caching strategies.
5. Logout Redirection
   - Description: After logging out, the user is sometimes not redirected to the login page.

- Cause: Inconsistent session handling in the logout API response.
- Impact: Users may still see restricted content until they manually navigate away.
- Suggested Fix: Ensure proper session invalidation and routing logic on logout.

Future Enhancements :-

1. **Advanced Search & Filtering**
   - **Description:** Enhance the search functionality with filters for categories, price range, ratings, and availability.
   - **Benefit:** Improves user experience by allowing more precise product discovery.
2. **Dark Mode / Theming**
   - **Description:** Implement a toggle for light and dark themes using CSS variables or a theming library like styled-components or MUI.
   - **Benefit:** Provides personalization options and improves usability in low-light environments.
3. **Product Recommendations Engine**
   - **Description:** Integrate AI/ML-based recommendation systems to suggest products based on browsing and purchase history.
   - **Benefit:** Increases engagement and potential sales by showing relevant products.
4. **Wishlist Feature**
   - **Description:** Allow users to save products to a wishlist for later purchase.
   - **Benefit:** Encourages return visits and helps users track products of interest.
5. **Enhanced Animations & Transitions**
   - **Description:** Use libraries like Framer Motion or GSAP to add smooth animations for page transitions, hover effects, and modal dialogs.
   - **Benefit:** Improves visual appeal and creates a more engaging interface.
6. **Progressive Web App (PWA) Support**
   - **Description:** Enable offline capabilities, app-like behavior, and push notifications by converting the application into a PWA.
   - **Benefit:** Improves performance on slow networks and allows installation on devices.
7. **Multi-Language Support**
   - **Description:** Add localization using libraries like react-intl or i18next to support multiple languages.
   - **Benefit:** Expands accessibility to a global audience.
8. **Performance Optimization**
   - **Description:** Implement code-splitting, image optimization, and lazy loading for better speed.
   - **Benefit:** Reduces initial load time and improves performance on low-end devices.