FRONTEND DEVELOPMENT WITH REACT.JS

**AM²S SHOPPING**

* **1.Introduction :**  
    **- Project Title:** Store Manager – Keep Track of Inventory  
    **Team *Members:***  
   -*T.Affrin Jamela-Frontend Developer*

*-S.Shafrin Rifaya-Frontend Developer*

*-M.Magila-Documentation Creator*

*-B.Mahalakshmi-Documentation Creator*

**2. Project Overview :**

* **Purpose:**  
   The purpose of the Store Manager Inventory Frontend project is to create an intuitive and responsive user interface that enables store managers to efficiently monitor and manage product inventory. This includes displaying product details such as name, quantity, price, and images, while allowing for real-time updates like adding, editing, or removing items. The frontend aims to streamline inventory tracking through features like search, filtering, and low-stock alerts, ensuring quick access and decision-making. Designed with usability and scalability in mind, the project also focuses on seamless integration with backend services and adaptability across devices to support smooth operations in a retail environment.
* **Features:**

The frontend provides a clean and responsive interface that allows store managers to view, manage, and update inventory with ease. Core features include dynamic product listings with images, prices, and stock quantities, along with intuitive controls to add, edit, or delete items. It supports real-time search and filtering to quickly locate products, and includes visual indicators for low-stock alerts. The design ensures smooth navigation across devices, with modular components that integrate seamlessly with backend APIs for live data updates. Altogether, the frontend empowers managers to maintain accurate inventory and streamline store operations efficiently.

**3. Architecture :**

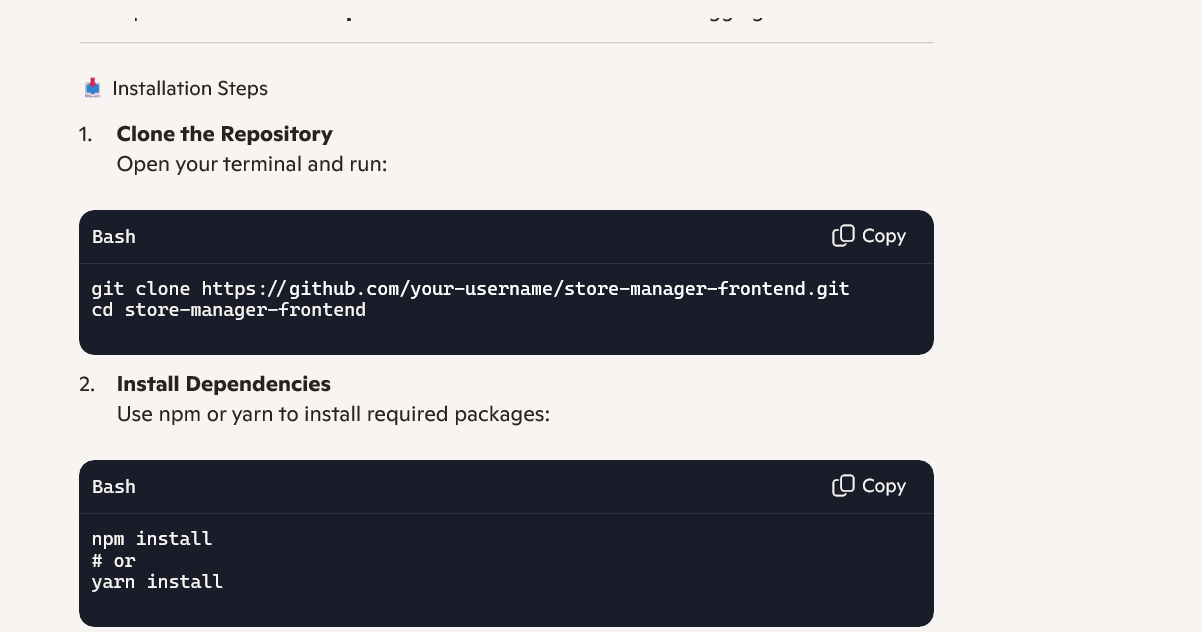
* **Component Structure:**  
   The Store Manager Inventory Frontend is structured around modular React components that work together to deliver a seamless user experience for managing inventory. At the core is the App component, which handles routing and global state management, serving as the entry point for the application. The InventoryList component fetches and displays product data, rendering individual ProductCard components that showcase details like name, price, quantity, and image, along with edit and delete options. For adding or updating products, the ProductForm component provides a controlled form interface, submitting changes back to the backend and triggering UI updates. A SearchBar component enables real-time filtering of products, while the LowStockAlert component highlights items that need restocking. Supporting components like Header and Footer ensure consistent navigation and layout. Together, these components interact through props, callbacks, and shared state to create a dynamic, responsive, and maintainable frontend system.
* **State Management:**  
   The Store Manager Inventory Frontend uses a centralized state management approach to ensure smooth data flow and component interaction. Typically, this is achieved using either the Context API or Redux, depending on the complexity of the application. For smaller to medium-scale projects, the Context API is often sufficient—it allows global state sharing across components, such as product lists, user actions, and inventory updates, without prop drilling. In more complex setups, Redux may be preferred for its predictable state container, middleware support, and debugging tools. It enables structured handling of actions like adding, editing, or deleting products, and ensures consistent synchronization between the UI and backend. Both approaches help maintain a responsive and scalable frontend architecture.
* **Routing:**  
  In a frontend inventory management project, routing is typically handled using **React Router** to enable smooth navigation between different views without reloading the page. The routing structure is defined in the main App component, where each route corresponds to a specific feature or page. For example, the root path (/) may render the InventoryList component to display all products, while /add loads the ProductForm for adding new items, and /edit/:id allows editing a specific product using dynamic route parameters. Additional routes like /alerts can display low-stock warnings, and /search may handle filtered results. Navigation is managed using Link or NavLink components in the header, and hooks like useNavigate and useParams help with programmatic navigation and accessing route data. This structure ensures a modular, scalable, and user-friendly experience across the application.

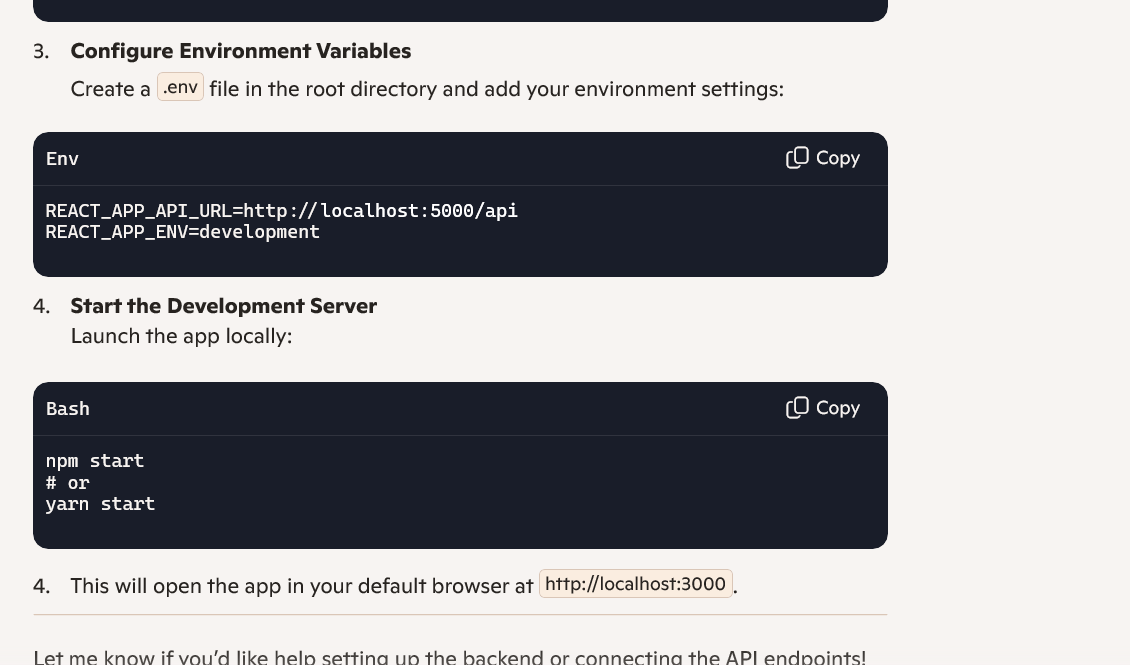
**4. Setup Instructions :**

* **Prerequisites:**

Before you begin, make sure the following software is installed on your system:

* Node.js (v16 or higher recommended)
* npm or yarn (comes with Node.js)
* Git (for cloning the repository)
* A code editor like VS Code
* **Installation:**





**5. Folder Structure**

* **Client:**

The client directory is organized as follows:

/client

├── /components # Reusable UI components

├── /pages # Page-level components

├── /assets # Images, icons, and other static files

├── /context # Context providers for global state

├── /hooks # Custom React hooks

├── App.js # Root component with routing

└── index.js # Entry point for React app

* **Utilities:**

The Store Manager Inventory Frontend leverages a set of utility functions and custom hooks to streamline logic and improve maintainability across key components like Home, Wishlist, Cart, Sales, and Orders. For the Home component, utilities such as getSummaryStats() may be used to fetch and format dashboard metrics like total sales or active orders. The Wishlist component often relies on a custom hook like useWishlist() to manage saved items, handle local storage, and sync with user preferences. Similarly, the Cart component uses useCart() to track selected products, calculate totals, and update quantities dynamically. In the Sales component, utilities like formatCurrency() and groupSalesByDate() help present revenue data clearly, while useSalesData() fetches and filters transaction records. The Orders component benefits from useOrders() to retrieve order history, update statuses, and sort by fulfillment priority. These utilities encapsulate reusable logic, reduce redundancy, and ensure consistent behavior across the application.

**6. Running the Application**

To run the Store Manager Inventory Frontend locally, follow these steps:

1. **Navigate to the client directory** Open your terminal and move into the project folder:  
     
    cd client
2. **Start the frontend development server** Use the following command to launch the app:  
     
    npm start

This will start the React development server and open the application in your default browser at http://localhost:3000.

**7. Component Documentation**

The Store Manager Inventory Frontend includes several key React components tailored to enhance user interaction and streamline store operations. The Home component serves as the landing page, offering quick access to core features like inventory, sales, and orders. The Button component is a reusable element used throughout the app, styled for consistency and configured via props such as label, variant, and onClick handlers. The Wishlist component allows users to save products for future reference, interacting with product data and user preferences. The AddToCart component enables users to select items for purchase, updating cart state and reflecting quantity changes. The Sales component displays transaction summaries, revenue insights, and product performance, while the Orders component tracks customer orders, statuses, and fulfillment details. Together, these components create a cohesive and responsive interface that supports both store management and user engagement.

**8.State Management**

* **Global State:**

Global state management ensures that shared data—like product lists, cart items, wishlist entries, and order details—flows consistently across all components. This is typically handled using either the Context API or Redux, depending on the scale and complexity of the app. A centralized store or context provider is set up at the root level (usually in the App component), allowing child components to access and update state without prop drilling.

For example, when a user adds a product to the cart from the Home or Wishlist component, the cart state updates globally and reflects immediately in the Cart component. Similarly, changes in product availability or order status in the Orders or Sales components are synchronized across the app, ensuring real-time accuracy. Custom hooks like useCart(), useWishlist(), or useOrders() often encapsulate this logic, making it easier to manage side effects, API calls, and UI updates. This approach keeps the application reactive, modular, and easy to maintain as features grow.

* **Local State:**

Local state is managed within individual React components using the useState hook. This allows each component to control its own data and behavior independently, without affecting the rest of the application. For example, the Wishlist component might use local state to track which items have been added or removed during a session, while the Cart component maintains its own state for selected products, quantities, and total price. In the Home component, local state can be used to toggle UI elements like banners or filters. Similarly, the Sales and Orders components may use local state to manage loading indicators, sorting preferences, or temporary form inputs. By isolating state within components, the app remains modular and responsive, with each part reacting quickly to user interactions without unnecessary re-renders across the entire application.

**10. Styling**

* **CSS Frameworks/Libraries:**

Styling is typically handled using a combination of CSS tools to ensure a clean, responsive, and maintainable user interface. A popular choice is **Sass (Syntactically Awesome Stylesheets)**, which allows for nested rules, variables, mixins, and modular stylesheets that keep the codebase organized and scalable. For component-level styling, **Styled-Components** may be used to write CSS directly within JavaScript, enabling dynamic styling based on props and theme values. Additionally, CSS frameworks like **Bootstrap** or **Tailwind CSS** can be integrated to speed up layout design with pre-built utility classes and responsive grid systems. These tools work together to enhance the visual consistency of components such as buttons, wishlist cards, cart summaries, sales dashboards, and order tables, while supporting customization and branding across the application.

* **Theming:**

implement theming or a custom design system to ensure visual consistency, brand alignment, and scalable UI development. Theming typically involves defining a centralized set of style variables—such as colors, fonts, spacing, and component sizes—often stored in a theme.js file or within a CSS preprocessor like Sass. These values are then applied across components using libraries like Styled-Components or CSS Modules, allowing for dynamic styling based on light/dark modes or user preferences. A custom design system might also include reusable UI patterns such as standardized buttons, cards, modals, and form elements, each following consistent spacing, typography, and interaction rules. This approach not only improves maintainability but also enhances the user experience by creating a cohesive and polished interface across pages like Home, Wishlist, Cart, Sales, and Orders.

**11. Testing**

* **Testing Strategy:**

**Unit Testing**

* **Tool:** Jest
* **Purpose:** Test individual components in isolation
* **Examples:**
* Ensure Button renders with correct label and triggers onClick
* Verify WishlistItem displays product info properly
* Confirm CartSummary calculates totals accurately

**Integration Testing**

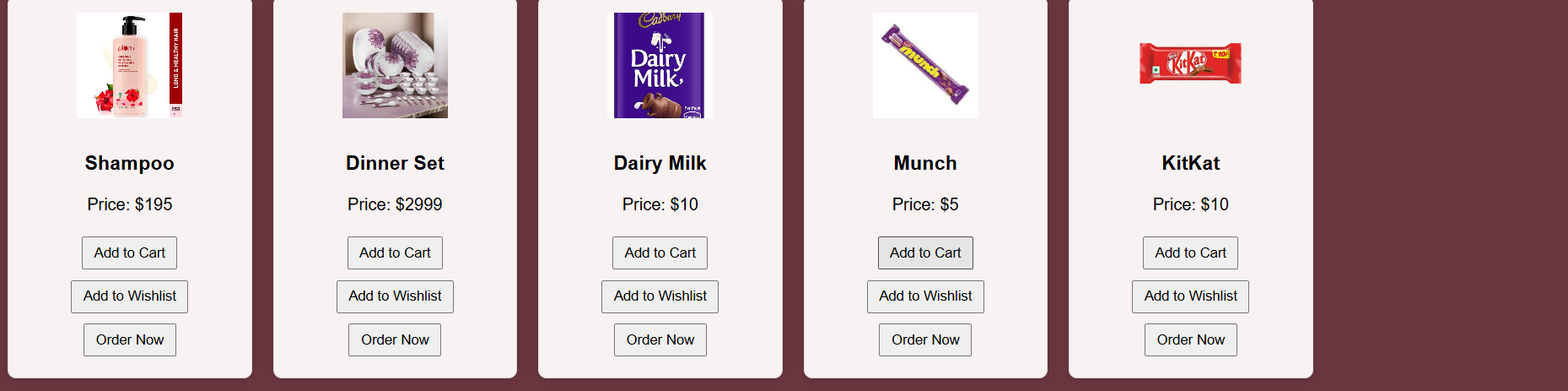
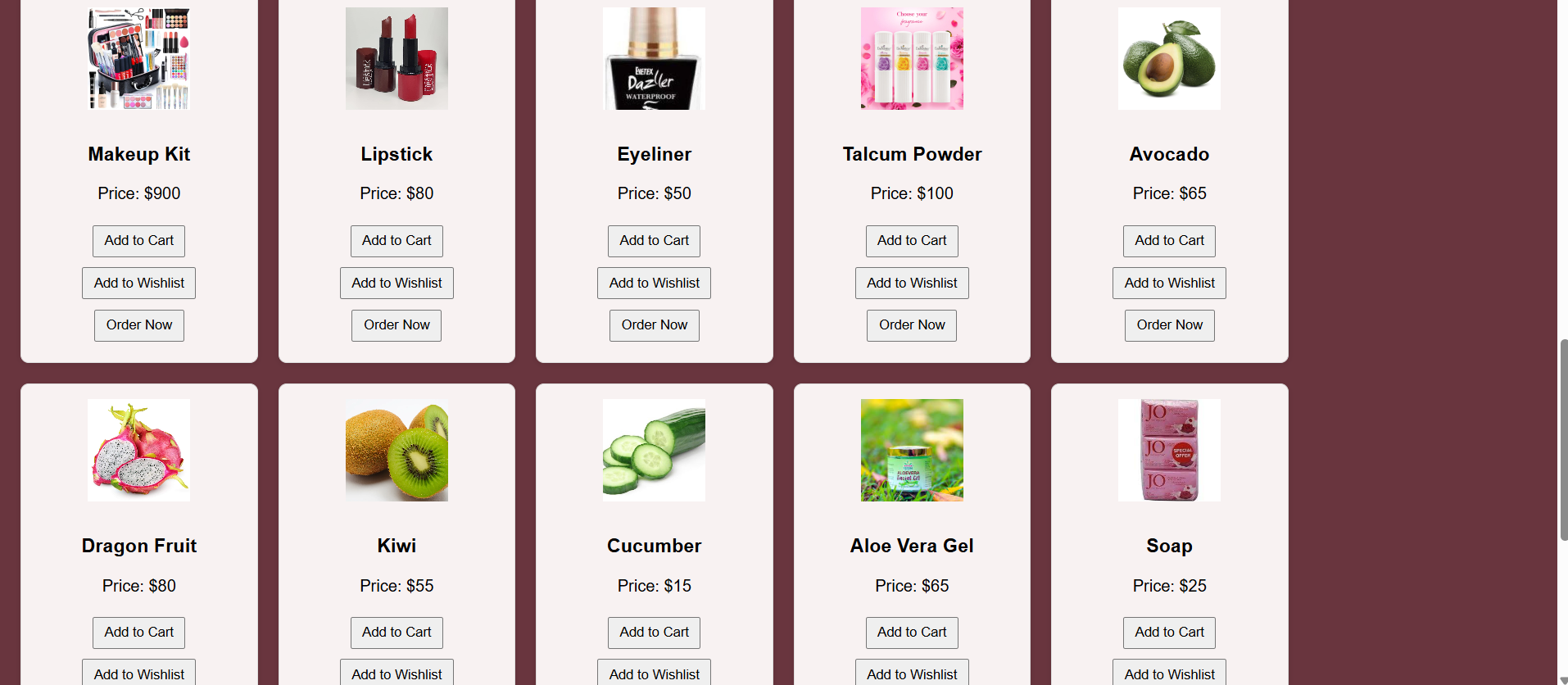
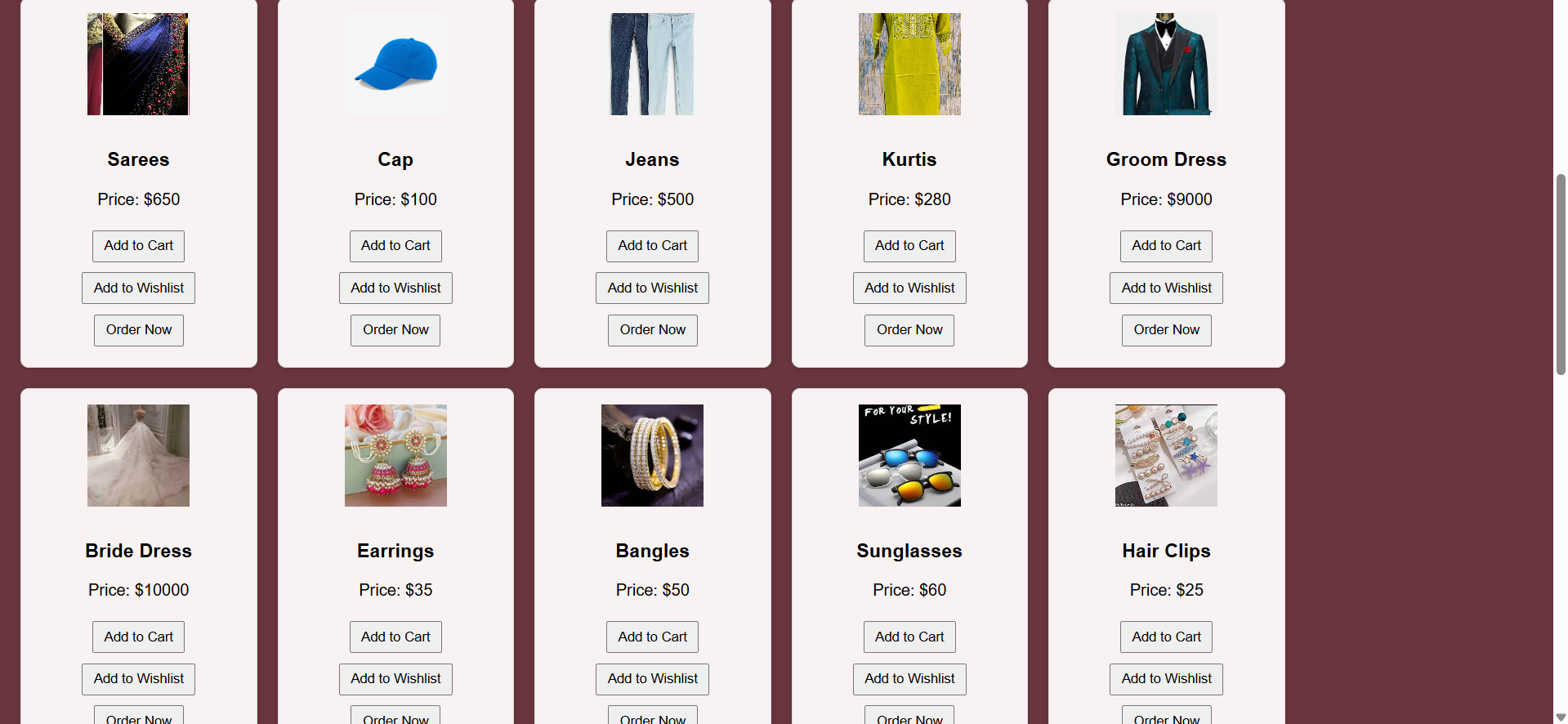
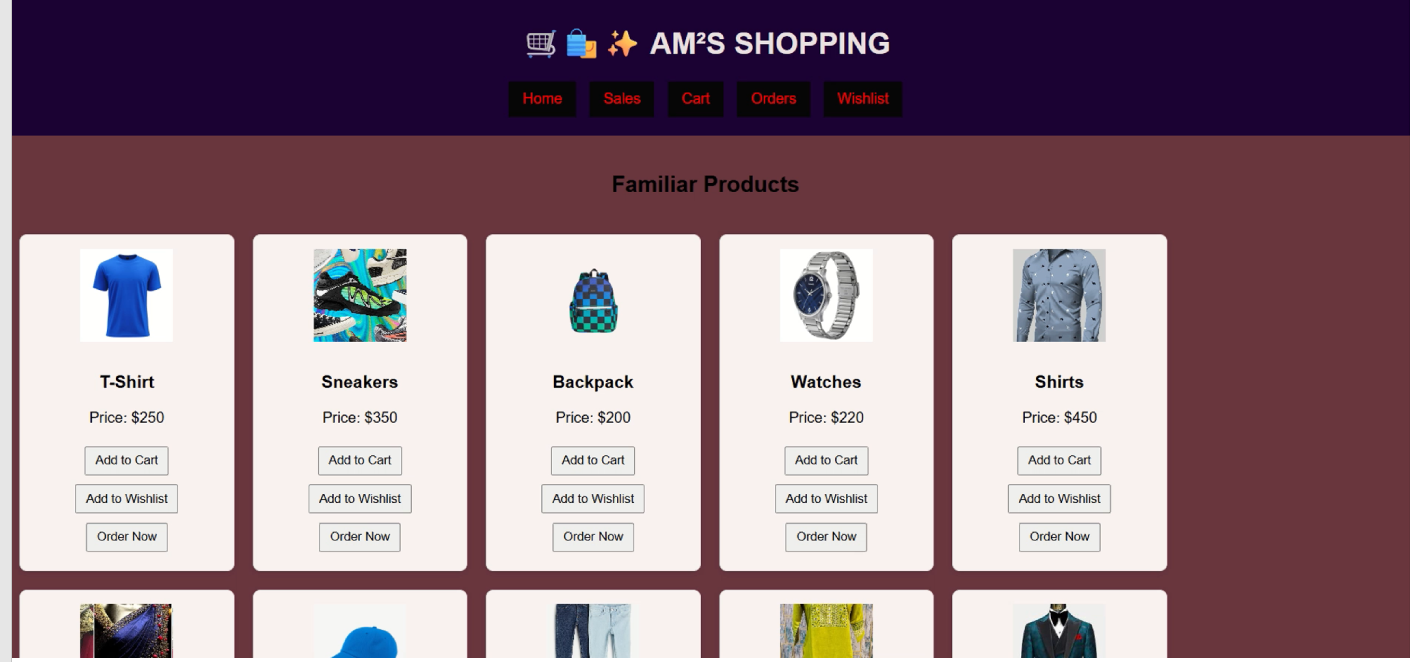
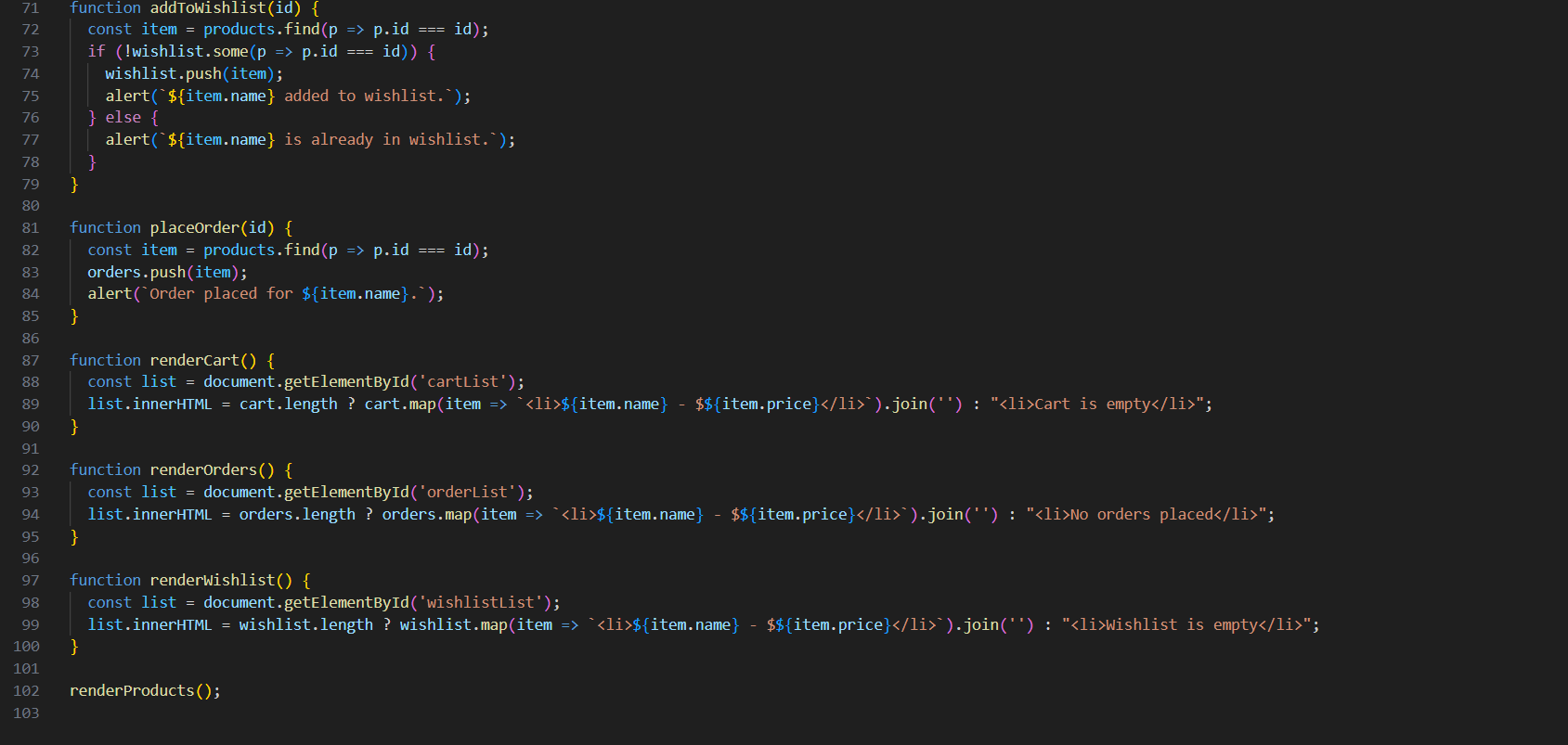
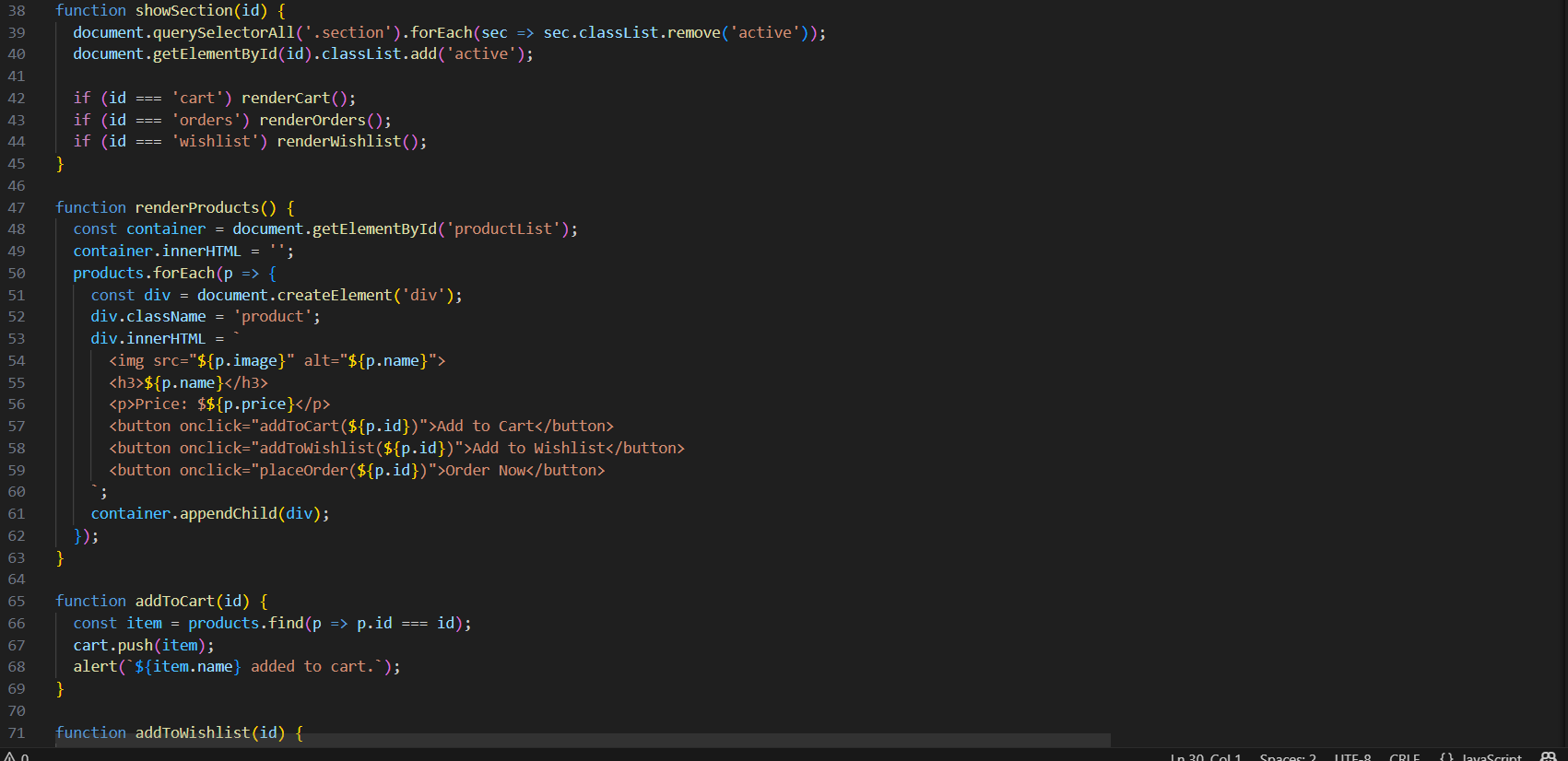
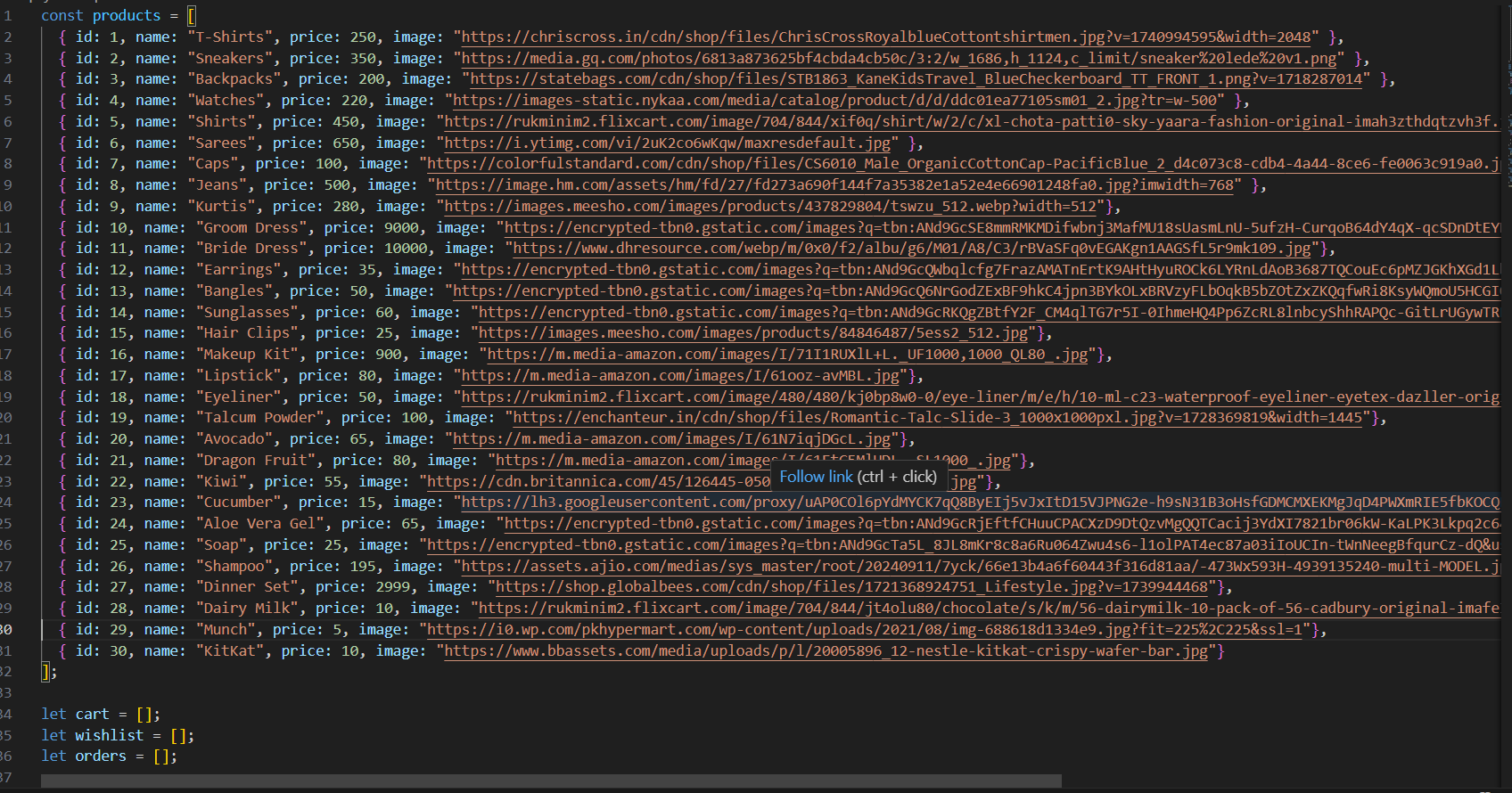
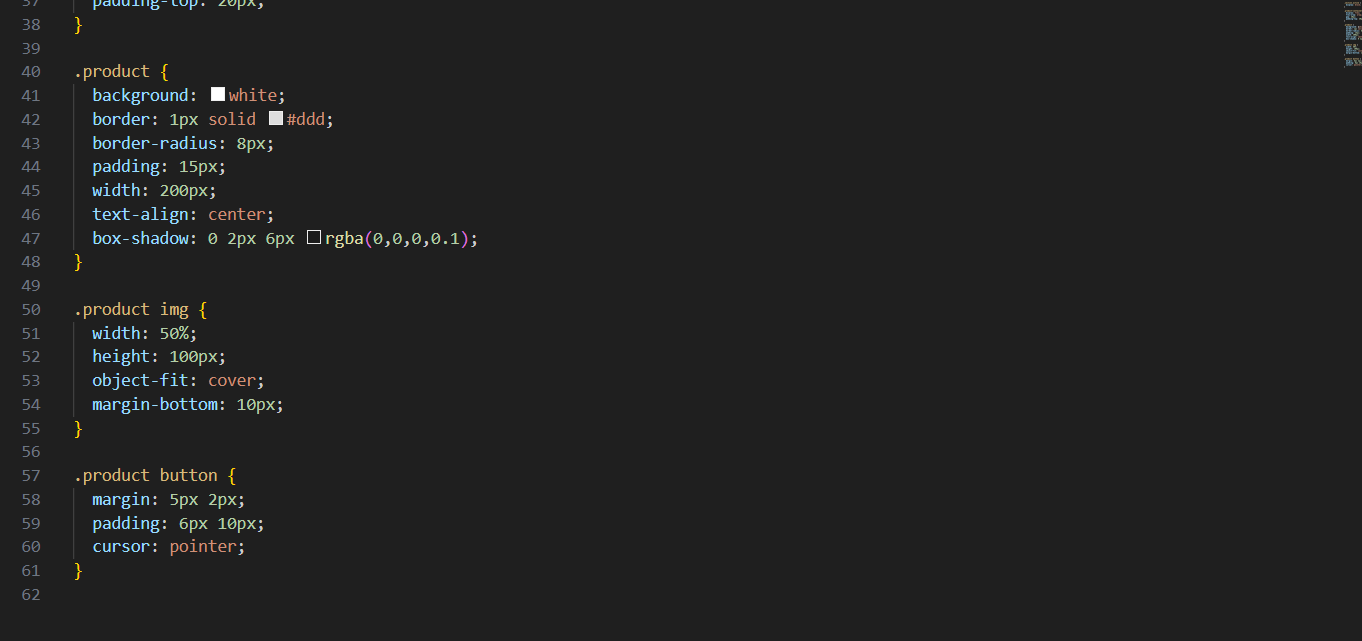
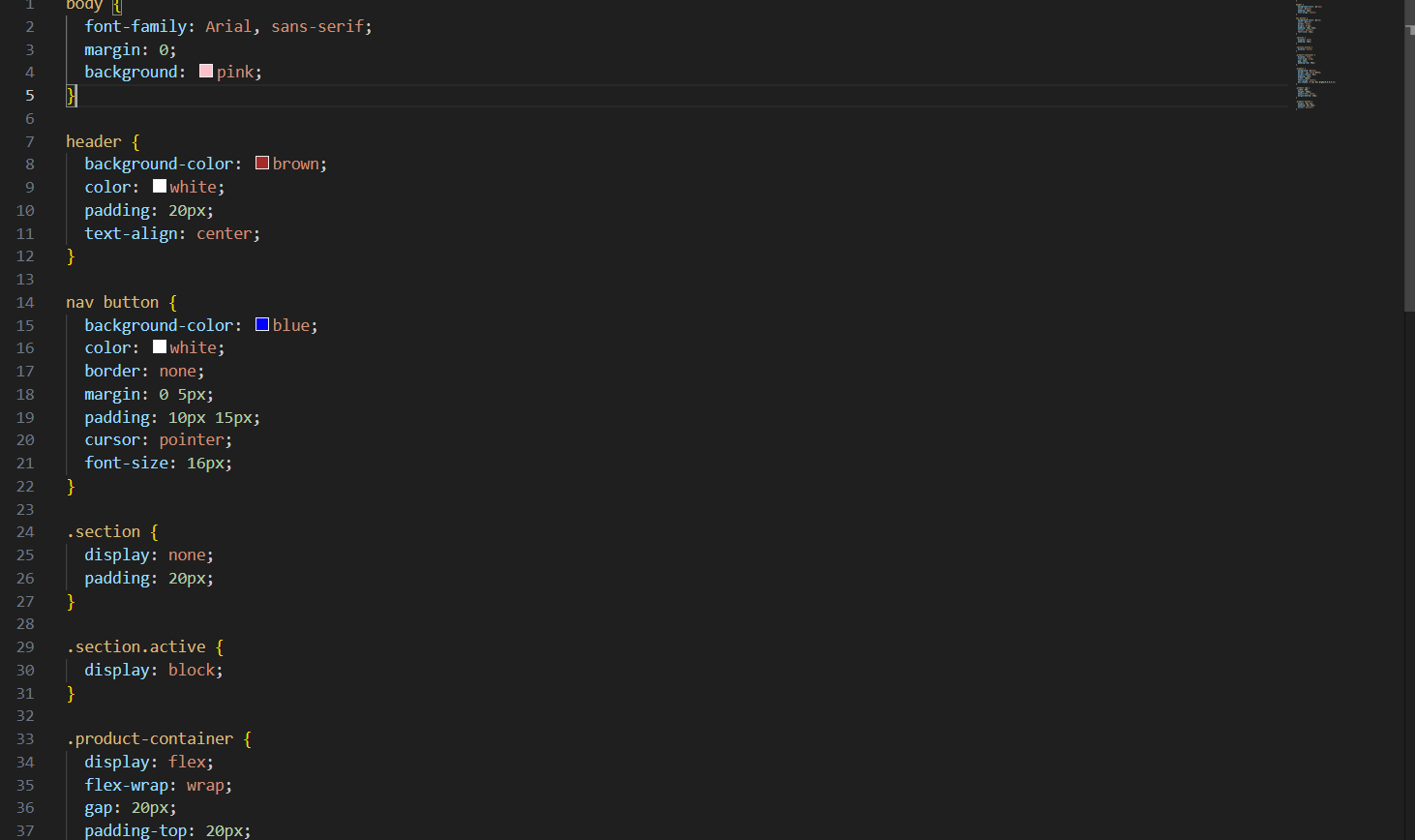
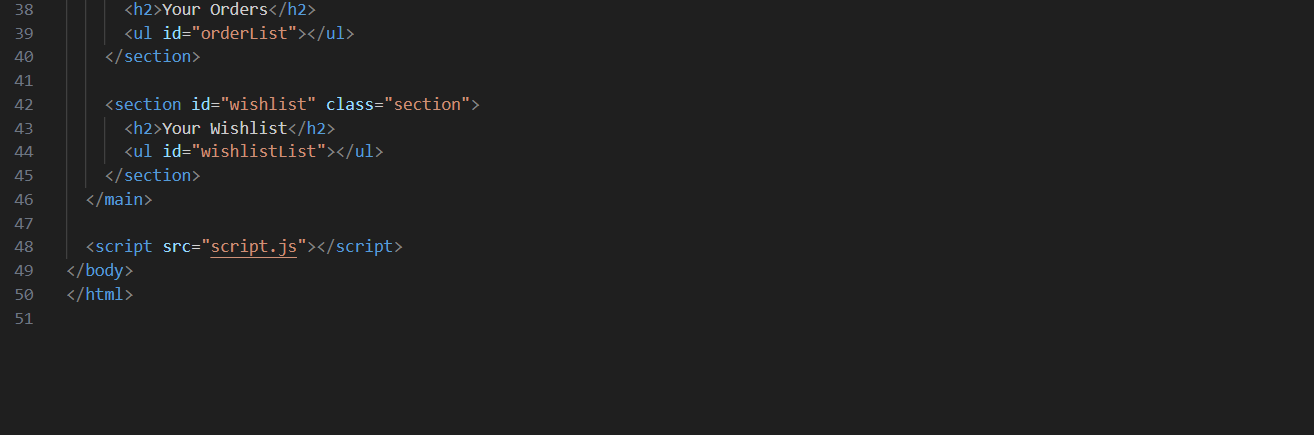
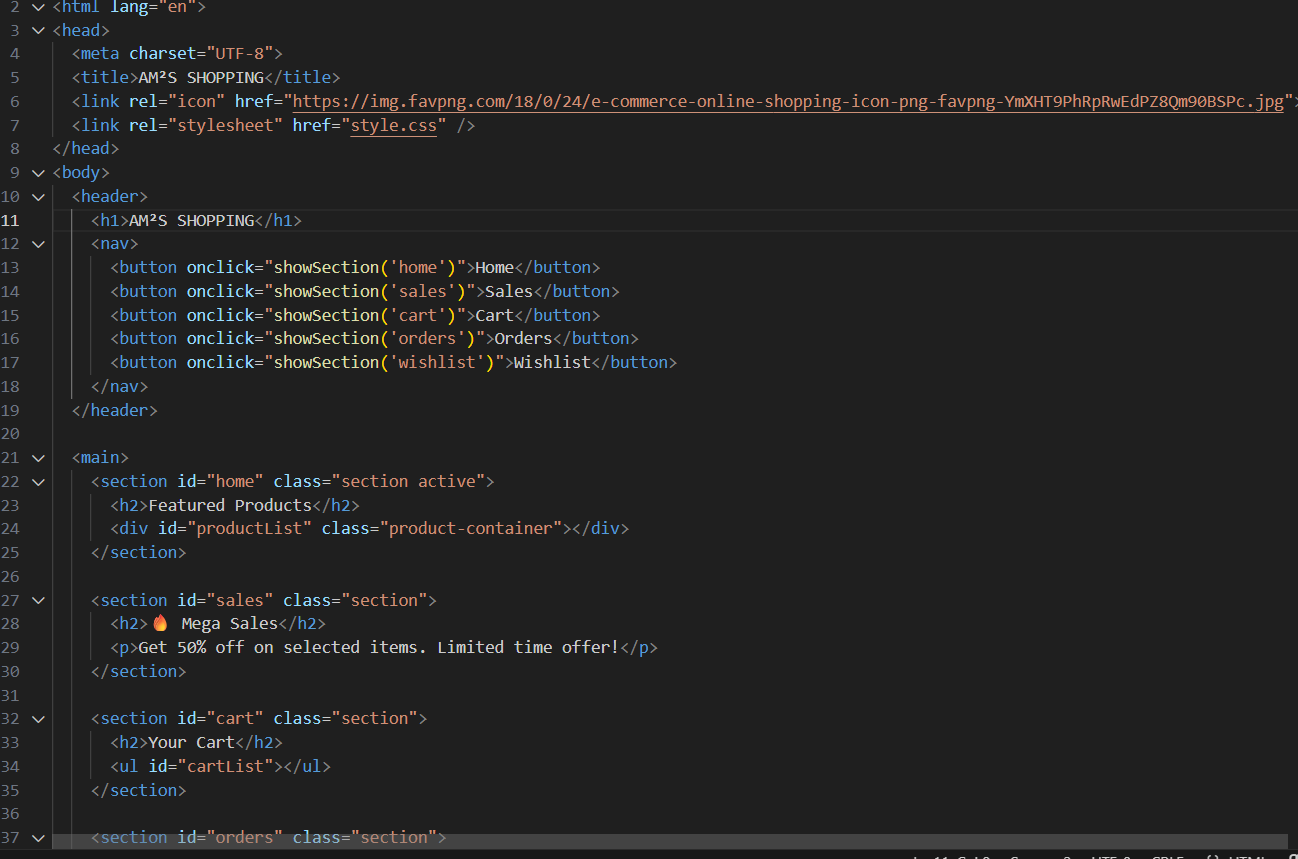
* **Tool:** React Testing Library
* **Purpose:** Test how components interact with each other and shared state
* **Examples:**
* Add product to cart from Home and check cart updates
* Verify Orders component fetches and displays correct data
* Test Sales dashboard filters and renders transaction data

**End-to-End (E2E) Testing**

* **Tools:** Cypress or Playwright
* **Purpose:** Simulate real user behavior across the full app
* **Examples:**
* Navigate from Home to Wishlist, add item to cart, place order
* Log in, view sales report, and check order status
* Validate full checkout flow and UI responsiveness
* **Code Coverage:**

**Tools for Test Coverage**

* **Jest**: Provides built-in support for tracking test coverage. Running jest --coverage generates detailed reports showing which lines, functions, and branches are covered.
* **React Testing Library**: Encourages testing from the user’s perspective, which improves meaningful coverage of UI interactions.
* **Coverage Reports:** Tools like Istanbul (integrated with Jest) produce HTML reports that highlight untested code visually, helping developers identify gaps.
* **Cypress (for E2E):** While not focused on line-by-line coverage, it ensures critical user flows are tested, contributing to functional coverage.

**12. Screenshots or Demo**

**13. Known Issues**

* **Navigation Errors**
  + Some routes (e.g., /wishlist or /orders) may trigger 404 errors if the directory structure or route imports are misconfigured.
  + Tab names occasionally fail to update dynamically when switching views.
* **Image Sizing Inconsistencies**
  + Product images may appear stretched or misaligned due to missing aspect ratio constraints or inconsistent styling across components.
* **Wishlist Sync Issues**
  + Items added to the wishlist may not persist across sessions if local storage or backend sync is not properly implemented.
* **Cart Quantity Bugs**
  + Rapidly adding or removing items from the cart can cause quantity mismatches or delayed UI updates due to race conditions in state updates.
* **Sales Data Rendering**
  + Sales charts or summaries may fail to load if the backend response format changes or if null values are returned unexpectedly.
* **Order Status Updates**
  + Order status changes may not reflect immediately in the UI due to missing re-fetch logic or stale cache in context/state.

**14. Future Enhancements**

The Store Manager Inventory Frontendproject offers exciting opportunities for enhancement through new features, refined components, and improved visual design. Future development could introduce advanced components such as a real-time notifications panel for order updates and stock alerts, or a user dashboard that personalizes sales insights and wishlist activity. Animations and transitions—powered by libraries like Framer Motion or CSS keyframes—could be added to elevate user experience, making interactions like adding to cart or toggling wishlist items more dynamic and engaging. Styling improvements may include implementing a full design system with consistent typography, spacing, and color themes, along with support for dark mode and accessibility enhancements. Additionally, integrating drag-and-drop functionality for managing inventory or customizing product layouts could offer a more intuitive interface. These upgrades would not only modernize the application but also deepen user engagement and streamline store management workflows.