Folder Structure

Explanation of Folders

- All static resources – assets (images, icons, styles)
- Reusable UI elements – components (buttons, modals, charts)
- State management – store (re dux slices, context)
- Handles API calls – services (e.g., fetching products, posting sales)
- Custom logic reusable across components – hooks
- Helper functions – u tils (date formatting, currency conversion)
- Route definitions for navigation – routes
- App-wide contexts – context (theme, settings)

  This setup ensures scalability, reusability, and maintainability, making it easy to add new features without cluttering the project.

```
Store -manager/
│── Public/              # Static files (index.html, favicon, etc.)
│
│── S r c/               # Main source code
│   │── assets/           # Images, icons, fonts, styles
│   │   ├── images/
│   │   ├── icons/
│   │   └── styles/       # Global CSS/SCSS or Tailwind
│   │
│   │── components/       # Reusable UI building blocks
│   │   ├── common/       # Buttons, Inputs, Modals, Loaders
│   │   ├── layout/       # Navigation bar, Sidebar, Footer
│   │   └── charts/        # Reusable chart components (Sales Chart,
Inventory Chart)
│   │
│   │── pages/            # Page-level components (mapped to routes)
│   │   ├── Dashboard/
│   │   ├── Products/
│   │   ├── Sales/
│   │   ├── Inventory/
│   │   ├── Employees/
│   │   └── Settings/
│   │
│   │── store/            # State management (Re dux / Context API)
│   │   ├── slices/        # Re dux slices (productsSlice.js,
salesSlice.js)
│   │   ├── actions/       # A sync actions (API calls)
│   │   └── index.js       # Store setup
│   │
│   │── services/         # API calls
│   │   ├── apiClient.js
│   │   ├── productService.js
│   │   └── salesService.js
│   │
│   │── hooks/            # Custom hooks (e.g., Fetch)
│   │
│   │── u tils/           # Helper functions (format Date, calculate
Totals)
```

```
|   |
|   |── routes/           # Central route definitions
|   |   └── AppRoutes.js
|   |
|   |── context/          # React Context (if used for theme, etc.)
|   |
|   |── App.js            # Root app component
|   |── main.js           # Entry point (React DOM .render / create
Root)
|
|── .e n v               # Environment variables
|── Package .j son
|── tailwind.config.js /    # Configuration (if using Tailwind or other
frameworks)
|── vite.config.js / webpack.config.js
```

Utilities

In store manager project, the utilities are small, reusable pieces of logic that make the code base cleaner, more maintainable, and prevent duplication.

File Handling Utilities

Reads from inventory. Csv and loads data into a python Dictionary

Utility role: abstracts away file reading logic, so main code Does not deal with CSV parsing directly.

Inventory Management Utilities

Add a new item or increases quantity if it already exists

Encapsulates “add” logic, so we don’t repeat the same steps Everywhere

Updates an item’s quantity

Removes an item completely from the inventory
Prints all items and their quantities neatly

6. Running the application

In most store manager projects with a frontend (usually react, Angular, or Next.js)

```
 If using NPM
    NPM start
 If using yarn
     Yarn start
  If using PNPM
       PNPM start
  Angular
      Ng serve
      #or if using NPM scripts
   Next .j s
      NPM run dev
       #or
       Yarn dev
        #or
        PNPM dev
```

To start the frontend server in your store manager project, go into the client directory and run:

```
Cd client
 NPM start
```

7. Component Documentation

- App: Root component sets up routing global layout, and Context providers (e.g. authentication, theme, state)
- Sidebar: Provides navigation across pages (Dashboards, Products, Orders, Users, Settings)
- Dashboard: Display key business metrics (total sales, number Of products, low stock alerts, recent orders)

- Product List: Lists all products with search, sort, and filter Functionality
- Product Form: Form to add or edit a product
- Order List: Shows all customer orders with status and filtering
- User List: Manages store users(admins, staff)
- Authentication Components: Login form (Authenticates users)
- Shared Components: Provides a consistent clickable action across the app (submit forms, trigger modals, navigation actions)

  Reusable Component

  Button: Standard clickable action button with consistent Styling

  Modal: Displays overlay content for confirmation, forms, or Details

  Table: Displays data in tabular format with sorting, filtering, And pagination

  Card: Compact container for displaying summary data

  Input: Consistent input element for forms

  Loader: Indicates ongoing data fetch or action

  Notification: Displays feedback messages to the user

  Pagination: Handles navigation for long lists of data

  Search Bar: Filters or searches through lists

8. State Management

   Global state is shared across multiple parts of the app and Must be accessible from anywhere (e.g. authentication status, Products catalog, shopping cart).

- User Interaction – A store clicks "Add Product" and "Add to Cart".
- Dispatch Action – The UI dispatches an action:

  Manager dispatch (add Product (product Data))

  Customer dispatch (add To Cart (product Id))

- Mutation Updates Global State: The products reducer updates the product list
- Store Read via Selectors: Components use selectors to grab only what they need:
  Product page select Product By id (state, product id)
  Cart summary select Cart Total (state)

Local State

Local state refers to data that is owned and managed by a single Component

Form Handling: product name, price, stock, cate gory

Search & Filters: search query, selected category, sort by

UI Controls & Toggles: Modal open/close (is Modal Open)

Dropdowns (is Drop down visible) Tabs (active Tab)

Temporary Feedback: Error messages and Success messages

9. User Interface

Show you how to generate GIFs of your UI

Use tools like: LICE cap (light weight, free, perfect for Recording UI GIFs), Screen to GIF (Windows) and KAP (Mac/Linux)

```
                  ________________________________________________
____________
| Search Bar        | Filters (Category, Stock)   |
---------------------------------------------------
| Product Image | Name | SKU | Price | Stock | Actions |
---------------------------------------------------
| img1         | A B C | 001 | ₹500 | 10    | Edit/Delete |
| img2        | X Y Z | 002 | ₹300 | 5    | Edit/Delete |
| ...         | ...  | ... | ...  | ...  | ...         |
-------------------------------------------------
| Pagination: 1 2 3 ... | Showing 1-10 of 50 |
```

10. Styling

CSS frameworks/Libraries

The Store Manager Project leverages modern CSS tools and Frameworks to ensure a consistent, responsive, and visually appealing user interface.

- CSS frameworks/Libraries
  Tailwind CSS is used as the primary styling framework. It provides utility-first classes that make it easy to build responsive layouts quickly without writing custom CSS from scratch. Tailwind also ensures design consistency across all components. Faster development, mobile-first responsiveness, built-in theming
- CSS Preprocessors/Styling Approaches
  For complex styles, SCSS was used to organize variables, and reusable styling patterns, making the stylesheet more maintainable. To scope styles to individual components, CSS modules (or styled-components, if implemented) were used. This prevents style conflicts and ensures encapsulation of component-specific-styles,
- Custom Styling
  Some custom CSS was written for fine-grained control over specific UI elements that were not covered by frameworks or libraries. Variables for color palette, typography, and spacing were defined to maintain design consistency.
  The store manager project primarily used Tailwind CSS (with optional libraries like Shad CN UI), along with SCSS/Styled-components (if applicable) for customization. This combination

allowed for fast development, responsive layouts, and clean, maintainable styling.

11. Testing

Testing strategy in store manager project follows a comprehensive testing approach to ensure that all components, features, and user workflows function correctly and reliably. Testing was carried out at multiple levels:

- Unit Testing
  To validate individual components, functions, and utilities in isolation. Tools are Jest – Provides a fast and reliable test runner for executing unit tests React Testing Library (RTL) – Focuses on testing components based on user interactions rather than implementation details. Examples: Testing product card components to verify correct rendering of product details and Checking that form validation logic works as expected before submission
- Integration Testing
  To verify that different components and modules work together correctly Tools are Jest + RTL for rendering multiple components and modules work together correctly. Examples: Testing the interaction between the product list and product detail components and Verifying that adding an item updates both the shopping cart state and UI display.
- End to End Testing
  To test real user workflows across the application, simulating how a customer or store manager would use the system. Tools used Cypress (or Playwright, if chosen) – Provides automated browser testing with real user interactions. Examples: Logging in as a store manager and adding a new product and completing a sales transaction from selecting products to checkout to generating receipt.
  Code Coverage
  To ensure that the Store Manager project maintains a high level of reliability and robustness, code coverage tools and techniques

were applied as part of the testing strategy. Code coverage helps measure the extent to which the applications measure the extent to which the applications source code is executed during testing, highlighting untested areas that may contain hidden bugs.

Tools used

Jest coverage reports: Jest was configured to generate code coverage reports for unit and integration tests.

Reports included metrics such as:

Statement Coverage – Percentage of executed statements

Branch Coverage – checks all possible paths (if/else, switch).

Function Coverage – ensures each function is tested

Line Coverage – tracks which lines of code are executed

Istanbul (built into jest)

Highlighted untested blocks in files so developers could Improve test completeness

Provides detailed instrumentation and visualization of code Coverage

Techniques Applied:

Threshold Enforcement: Minimum coverage threshold were set (e.g. 80-90%) to ensure critical parts of the code base were tested before merging changes.

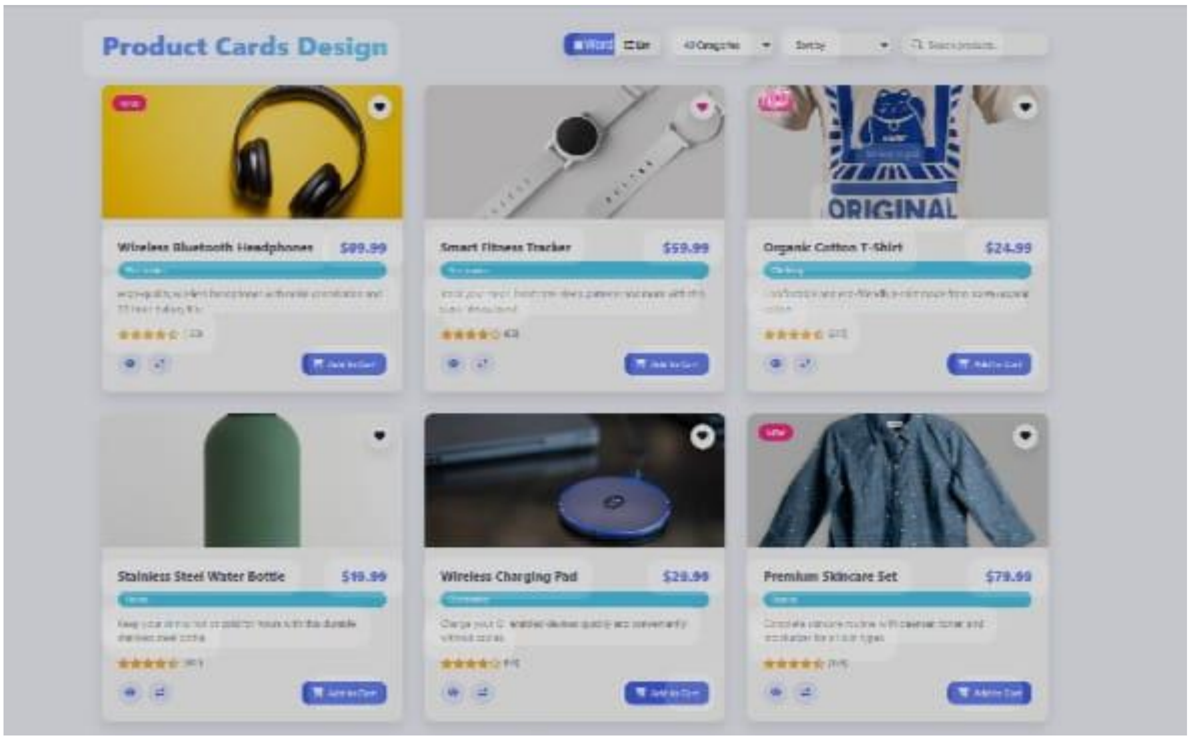
Targeted Testing: Extra tests were written for complex modules such as product management, sales tracking, and authentication flows.

Continuous Integration: Code coverage checks were automated in the CI pipeline to block pull requests that reduced coverage below the threshold.

Review process: Developers regularly reviewed coverage reports to identify gaps and strengthen test cases in under-tested areas.

The store manager project used jests built-in coverage tool (Istanbul) with enforced threshold and CI integration to maintain adequate test coverage. This ensured that critical business logic was thoroughly tested, improving code quality, reliability, and maintainability.

12. Screenshots or Demo



This picture shows UI design concept for product cards in an online store or e-commerce platform. It is meant to help a store manager visualize how products are displayed to customers.

Product Cards (Example)

- Product Image – a big clear photo of the product
- Labels – Some have a “New” tag or a heart icon (wish list)
- Product Name & Price – (bold and easy to read)
- Category Tag – (e.g. “Electronics,” “Clothing,” “Home”).
- Short Description of the product
- Rating & Reviews: Star ratings + number of reviews
- Add to Cart Button – for direct purchase

Examples Products Shown

- Wireless Bluetooth Headphone - $89.99
- Smart Fitness Tracker - $59.99

- Organic Cotton T-shirt - $24.99
- Stainless Steel Water Bottle - $19.99
- Wireless Charging Pad - $29.99
- Premium Skincare Set - $79.99

Provides a visual inventory overview (images + details).

Helps manage pricing, categories, and stock efficiently.

Shows customer-facing details (ratings, reviews, like "new").

Allows quick updates to product (e.g. changing prices, Descriptions, Categories).

13. Known issues

This section outlines currently identified bugs, limitations, or issues that developers and users should be aware of.

Product Cards Display: Images sometimes fail to load or appear stretched when uploaded in unsupported formats. Product descriptions may cut off if too long, without a proper "read more" option. "New or "Sale" tags are not removed automatically after the set duration.

Search & Filters: The search bar is case-sensitive, which may return incomplete results. Filter options (category, price, etc.) occasionally don't reset after clearing. Sorting by popularity doesn't always reflect accurate sales or rating data.

Add to Cart & Wish list: Some users report duplicate entries when quickly clicking "Add to Cart." Wish list heart icon doesn't update instantly without refreshing the page.

Ratings & Reviews: Start ratings may round incorrectly (e.g. 4.2 displays as 5). Reviews counter does not synchronize in real time after new submissions.

Performance Issues: Large product catalogs (>500 items) cause slow page loading in grid view. Mobile version has layout inconsistencies, especially with 2-column display.

Admin/Manager Slide: Price or stock updates sometimes take a few minutes to reflect on the customer side. No bulk edit option for categories, requiring manual updates per product.

These issues are being tracked and prioritized. Developers should test across multiple devices and browsers to reproduce consistently. Users should be aware of temporary workarounds, like refreshing the page or limiting upload file sizes.

14. Future Enhancements

This section highlights potential improvements and new features planned for upcoming versions of the store manager.

UI/UX improvements:

Smooth transitions when adding items to the cart or switching filters. Dark mode support toggle between light and dark themes. Improved Responsiveness optimize layouts for tablets and small-screen devices. Customizable Product Cards allow store managers to change card size, layout, or color schemes.

Product & Inventory feature:

Update multiple products price, stock, category at once. Low-stock alerts notifications when inventory is running low. Automated tagging system generated “New”, “Trading”, or “Sale”, labels based on rules. Enhanced image handling auto cropping and format support for cleaner product visuals.

Search & Filtering:

AI- powered suggestions and typo corrections. Allow filtering by price, category, rating, and tags simultaneously. Users can save frequently used search/filter settings.

Card & Checkout Enhancements:

Faster interactions with animations. Quick view of added items without navigating away. Sync wish list across devices for logged-in users.

Ratings & Reviews:

Show which reviews come from real buyers. Allow customers to upload product photos with reviews. Instantly update ratings and reviews without page refresh.

Manager Dashboard:

Sales charts, top products, and customer trends. Different permissions for admins, staff, and managers. Easier product reordering in collections.

Styling & Branding:

Store owners can select from pre-designed themes. Hover effects on buttons and cards for a modern feel. Expanded icon library with category-specific icons.

These enhancements are designed to improve usability, performance, and visual appeal for both store managers and end users.