Folder Structure:

/src

/components	# Reusable UI components
├ Navbar.jsx	# Navigation bar
├ Counter.jsx	# Reusable counter component
/pages	# Individual application pages
├ Home.jsx	#Landing page
├ Dashboard.jsx	# User dashboard with counter and user profile visuals(charts)
├RichTextEditor.jsx	# Standalone Rich text editor
├UserDataForm.jsx	# Standalone Page, Manages Form State
theme.js	# Theme configuration (Chakra UI)
App.jsx	# Main application component
main.jsx	# Entry point for React application

Component Types:

- **Reusable Components (/components/):** UI elements like Navbar and Counter that are shared across pages.
- **Page Components (/pages/):** Route-specific views such as Home.jsx, Dashboard.jsx, UserDataForm.jsx, and RichTextEditor.jsx.

My State Management Approach:

I'm using React's built-in state management **(useState)** along with local storage instead of more complex solutions like Redux or other state management. Here's why I made this choice.

useState:

using useState to manage state within individual components, such as:

- **Counter.jsx** → To handle the counter value.
- **UserDataForm.jsx** → To manage form inputs.
- **RichTextEditor.jsx** → To store the editor's content.

Local Storage (Persistent State):

Since I want user data to persist, I'm using **localStorage** in:

- **UserDataForm.jsx** → To store user info in local storage retrieve from it.
- **Dashboard.jsx** → To retrieve and display stored user data.
- **RichTextEditor.jsx** → To pre-fill user data content from local storage.

Conclusion:

By using useState for UI updates and local storage for persistence, I've kept my project lightweight, efficient, and scalable. If I later add authentication or API integration, I might reconsider Redux Toolkit or other state management, but for now, this setup is the best fit.