## **EXPERIMENT**

## **Aim- Manage Complex State with Redux or Context API**

The aim of this project/module is to demonstrate effective management of complex application state in a React-based application using **Redux** or **Context API**. By implementing these state management tools, we seek to create scalable, maintainable, and efficient front-end applications that handle state changes predictably and improve developer productivity.

## **Introduction**

In modern web applications, especially those built with React, managing the state (data that changes over time) is crucial but can become challenging as applications grow in size and complexity. React’s built-in state management using useState and useReducer hooks works well for local component states but can become cumbersome when the state needs to be shared or updated across multiple components.

To address these challenges, state management libraries like **Redux** and React’s **Context API** have been developed:

* **Redux** is a predictable state container for JavaScript applications, enforcing a unidirectional data flow and a strict structure to manage state globally.
* **Context API** is a React built-in feature that allows you to pass data through the component tree without having to pass props down manually at every level.

Both tools help in managing complex, shared states by centralizing state logic and improving code maintainability.

## **Theory**

* **State Management:** The process of handling the data that controls the UI, user input, and server responses. Efficient state management ensures that components render correctly and stay in sync.
* **Redux Concepts:**
  + **Store:** The centralized place that holds the application state.
  + **Actions:** Plain objects that describe “what happened” and carry data.
  + **Reducers:** Pure functions that take the previous state and an action, then return the new state.
  + **Dispatch:** The method used to send actions to the store.
  + **Middleware:** Functions that intercept actions for logging, asynchronous operations, etc.
* **Context API Concepts:**
  + **Provider:** A component that provides the state to its descendants.
  + **Consumer:** Components that consume or use the shared state.
  + **useContext Hook:** Allows functional components to subscribe to React context changes easily.

Both Redux and Context API reduce prop drilling (passing props through many nested components) and make state management more structured and manageable.

## **What We Have Done**

* Created a React application that requires sharing and updating complex state across multiple components.
* Implemented **Redux** for global state management:  
  + Set up the Redux store with reducers and actions.
  + Connected React components to the Redux store using react-redux library (Provider, useSelector, useDispatch).
  + Handled asynchronous data fetching using middleware (e.g., Redux Thunk).
* Alternatively, implemented **Context API** in some modules:  
  + Created context providers and consumers.
  + Used the useContext hook for easier access to the shared state.
* Demonstrated the pros and cons of both approaches in terms of scalability, complexity, and ease of use.
* Managed actions like updating user data, toggling UI elements, and handling form inputs via the chosen state management method.
* Ensured components re-render only when necessary by optimizing selectors or context value updates.

**Technology Used**

* **React.js:** Frontend JavaScript library for building user interfaces.
* **Redux:** State management library for managing complex global state.
* **React-Redux:** Official React bindings for Redux.
* **Redux Thunk:** Middleware to handle asynchronous actions in Redux.
* **Context API:** React's built-in state sharing feature.
* **JavaScript (ES6+):** Core programming language.
* **Node.js & npm:** For package management and running development servers.
* **Webpack / Create React App:** For project bundling and development environment.
* **VS Code / Any Code Editor:** For coding and debugging.

**Tools Used**

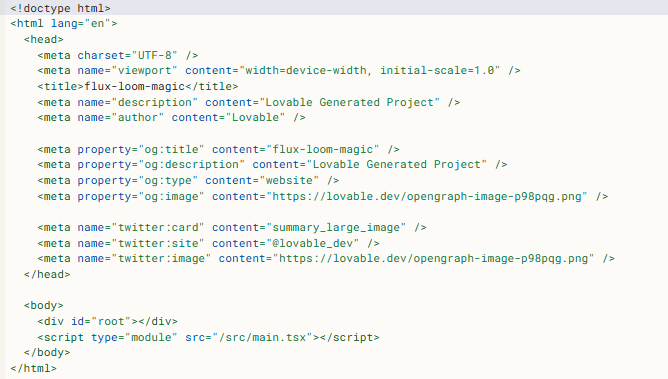
* **Redux DevTools:** Browser extension to debug and inspect Redux state changes.
* **React Developer Tools:** Browser extension to inspect React component tree and state.
* **Postman:** For testing backend APIs (if applicable).
* **Git:** Version control system to manage project code.
* **Chrome / Firefox:** Browsers used for testing and debugging.

**Code:**

**App.css**



**Index.html**

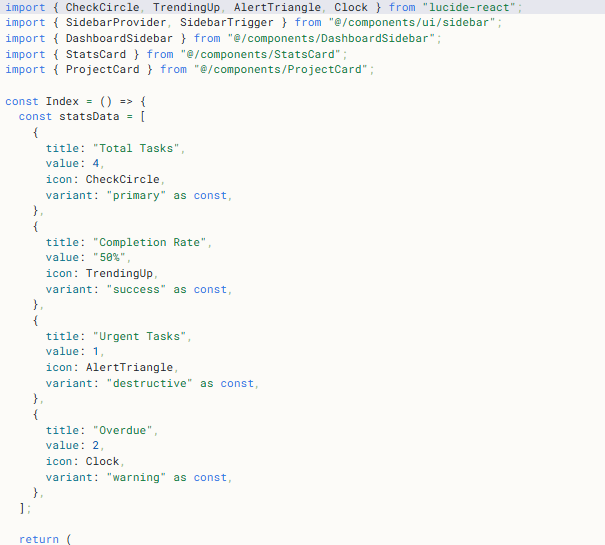


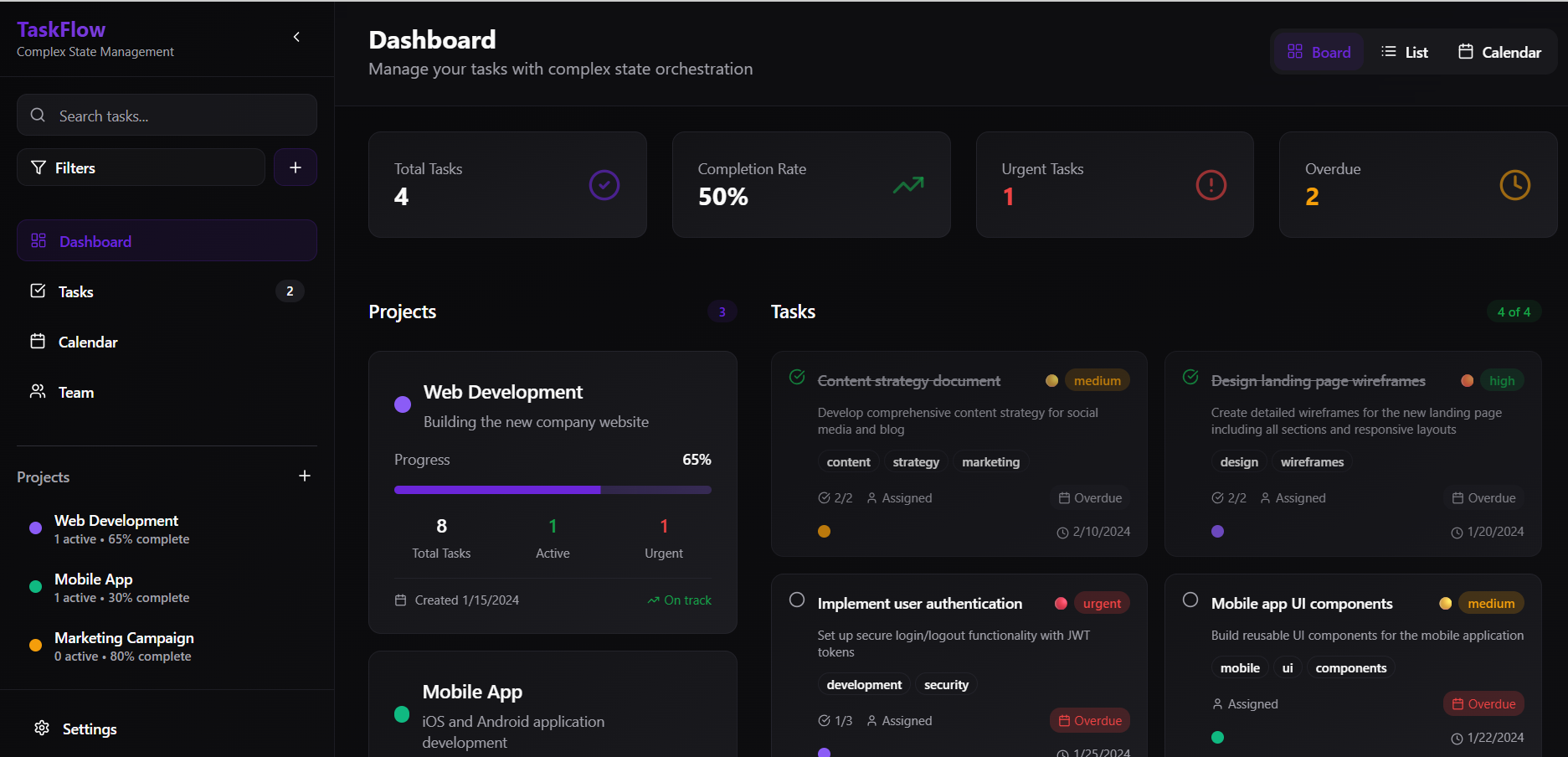
**Main.tsx**

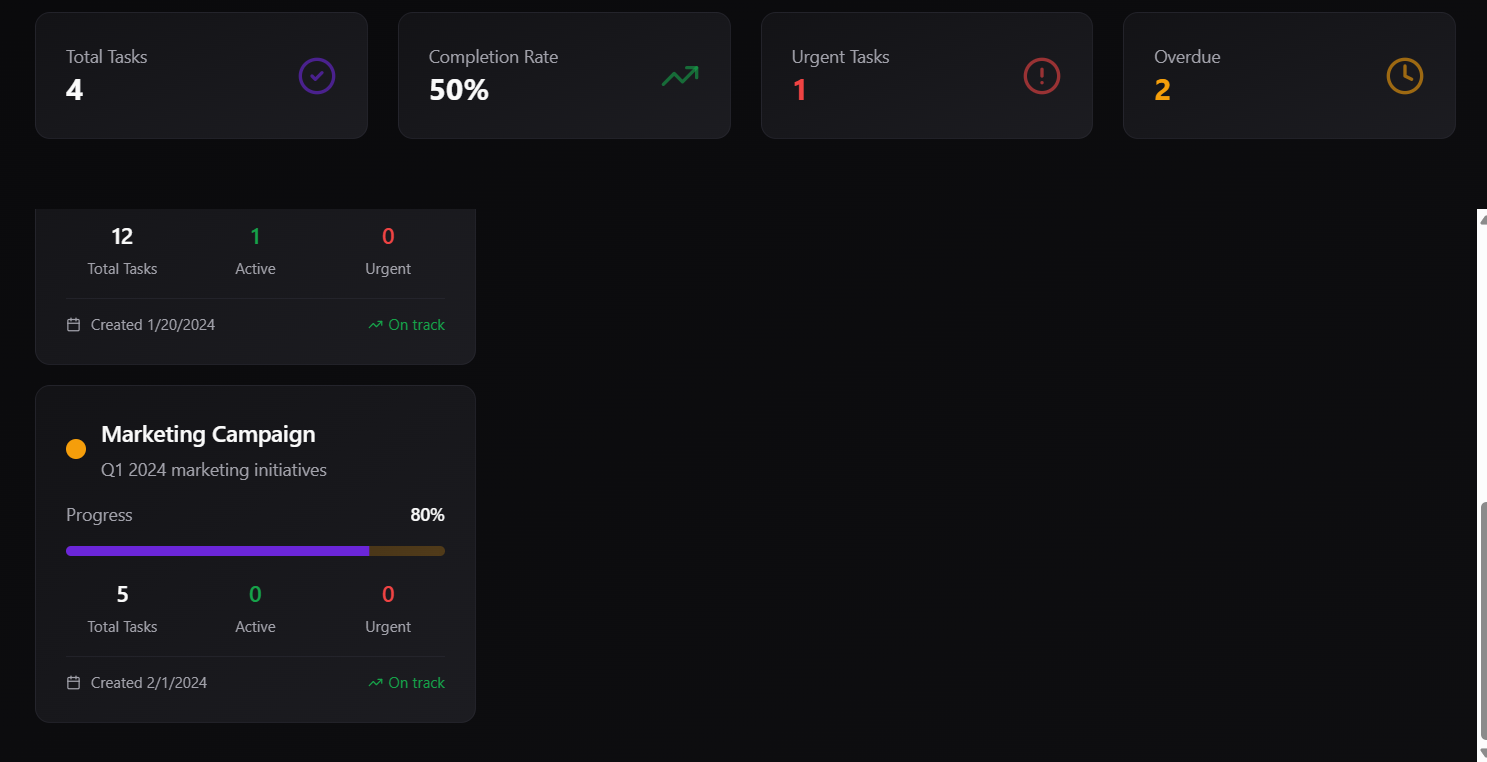
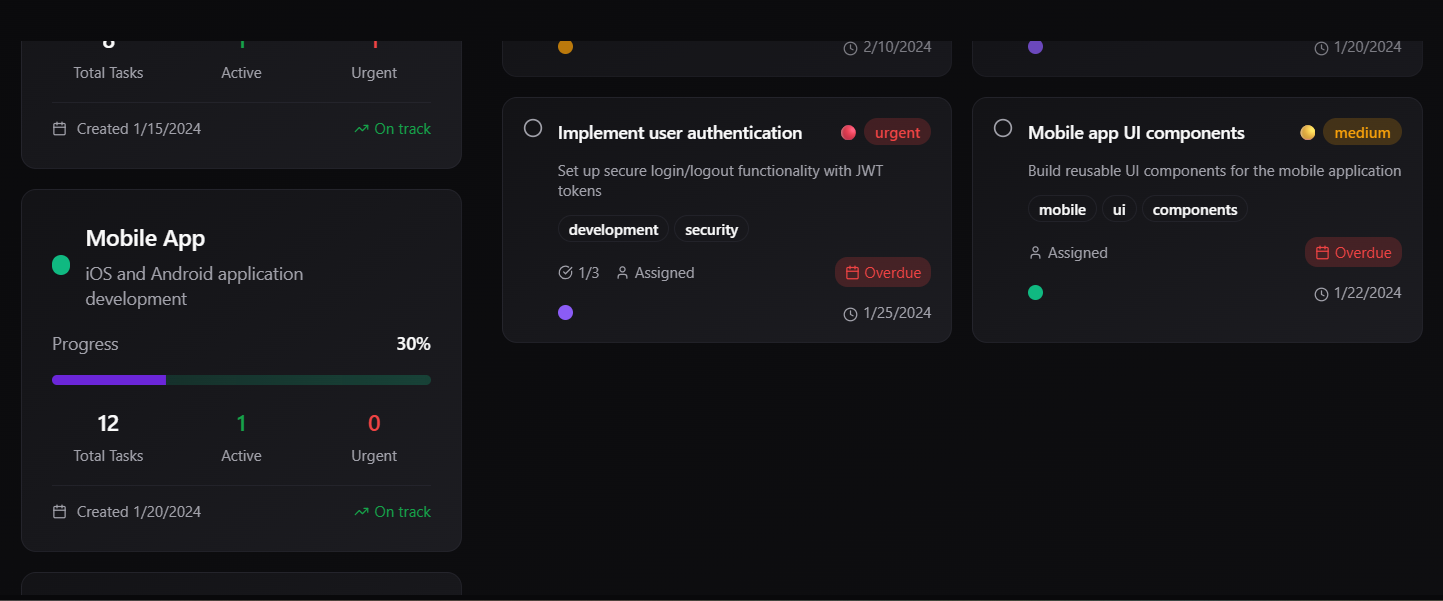
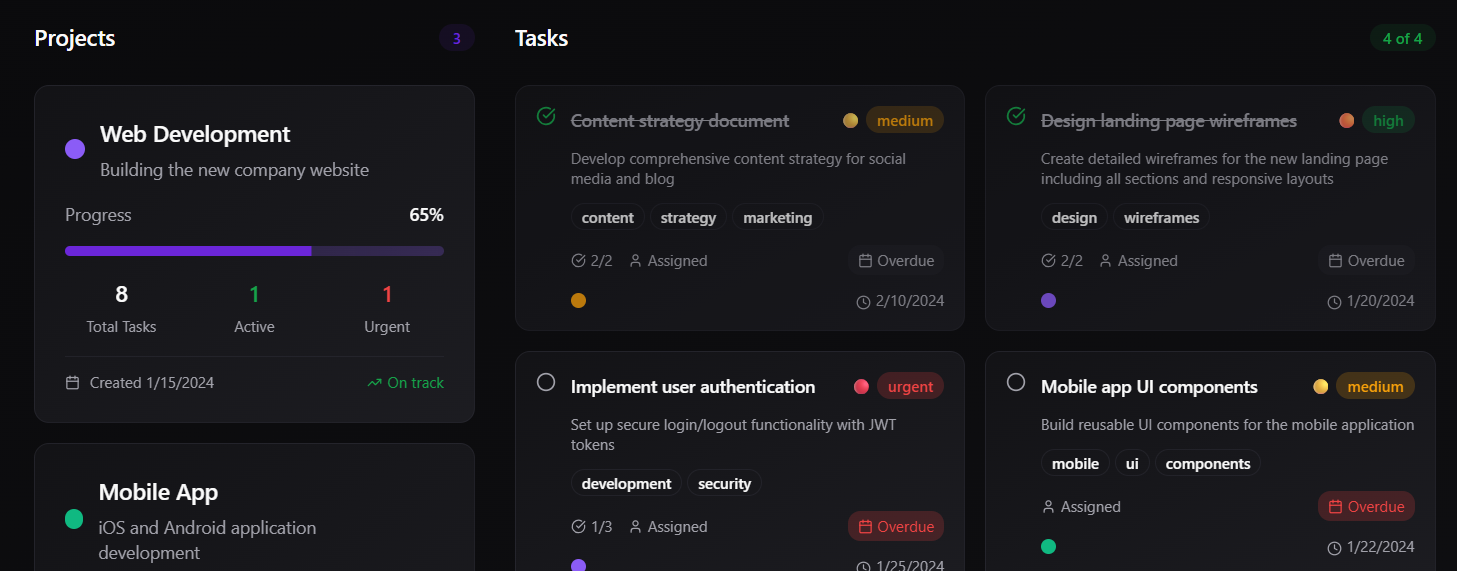


**App.tsx**





**Output**



## **Conclusion**

Managing complex state in React applications is essential for building scalable and maintainable user interfaces. This project demonstrated two popular state management techniques: **Redux** and **Context API**. Redux provides a more structured and scalable approach ideal for large applications with complex state logic, while Context API offers a simpler, built-in solution suitable for medium to small-scale apps.