

EXPERIMENT NO. 03 Manage Complex State with Redux or Context API

AIM: To learn and implement state management using Redux and React Context API for handling complex and shared application states, ensuring efficient data flow, scalability, and maintainability in React applications.

THEORY:

Prerequisites

Before starting this experiment, make sure you have the following knowledge and setup:

- **Programming Skills:** Strong basics of JavaScript (ES6+).
- **React Knowledge:** Understanding of components, props, state, and React hooks like `useState` and `useEffect`.
- **State Management Fundamentals:** Difference between local state (inside one component) and global state (shared across multiple components).
- **Environment & Tools:**
 - Node.js installed on your system.
 - A running React project (created using `create-react-app` or Vite).
 - Libraries: `redux` and `react-redux` (only if working with Redux).

State management is one of the key aspects of building modern React applications. As apps grow, it becomes challenging to maintain and synchronize state across multiple components. Two common solutions are **Context API** (built into React) and **Redux** (external library).

When building modern web applications, especially with React, state management plays a crucial role. For small applications, passing props and using local component state is sufficient. However, as applications grow larger, the need for a more structured and centralized state management solution arises. This is where tools like **Redux** and **React Context API** come in.

Redux is a predictable state container for JavaScript applications. It follows the principle of a single source of truth, meaning all the state of the application is stored in one central store. Components interact with this store by dispatching actions, which are plain JavaScript objects describing what happened. These actions are processed by **reducers**, which specify how the application's state changes in response. This makes the application easier to debug, test, and maintain, especially in projects with many developers. Redux also supports middleware, allowing additional features like asynchronous data fetching, logging, and performance monitoring.

On the other hand, the **Context API** is a built-in React feature that allows state to be shared across components without the need for prop drilling. It uses a **Provider-Consumer** pattern, where the **Provider** component makes state available, and any nested component can consume it using the **useContext** hook. Context API is simpler to implement than Redux and is best suited for small to medium applications or when you only need to share limited state, such as theme preferences, user authentication, or language settings.

In summary, **Redux** provides more power and structure for complex applications where state management logic is heavy and spans multiple features, while the **Context API** is a lightweight option for sharing data without external libraries. Many modern projects even use them together: Context API for global, lightweight state and Redux for complex state handling.

♦ React Context API

The **Context API** is a built-in feature of React that makes it easier to pass data across the component tree without manually passing props at each level.

- **How It Works:**

- A context object is created using `React.createContext()`.
- A **Provider** component wraps the app (or part of it) and supplies state + functions.
- Components inside can access the state using **useContext** hook (or **Consumer**).

- **Benefits:**

- No need for third-party libraries.

- Perfect for handling global settings like themes, authentication, or language.
- Simple and lightweight to use.
- **Drawbacks:**
 - Updating context can trigger re-renders in all consuming components.
 - Becomes harder to debug when application logic gets bigger.
 - Doesn't provide advanced features (like async handling or middleware).

◆ Redux

Redux is a standalone state management library often used with React. It provides a predictable way to handle global state with strict patterns.

- **Main Concepts:**
 - **Store:** A single object that contains the entire application state.
 - **Actions:** Plain JS objects describing events/changes in the app.
 - **Reducers:** Pure functions that take current state + action → return new state.
 - **Dispatch:** Method used to send actions to the store.
 - **Selectors:** Functions used to read specific pieces of state.
- **Benefits:**
 - **Predictability:** One-way data flow ensures consistency.
 - **Debugging:** Redux DevTools allow developers to inspect state changes, rewind, and replay actions.
 - **Middleware Support:** Easy handling of async logic (API requests, logging, etc.) using tools like Redux Thunk or Redux Saga.
 - **Scalability:** Works great for complex, enterprise-scale apps.

- **Drawbacks:**

- Extra boilerplate code (though **Redux Toolkit** reduces this).
- Beginners might find it difficult to learn initially.
- Might be over-engineering for very small apps.

30% Extra Work (Extended Section)

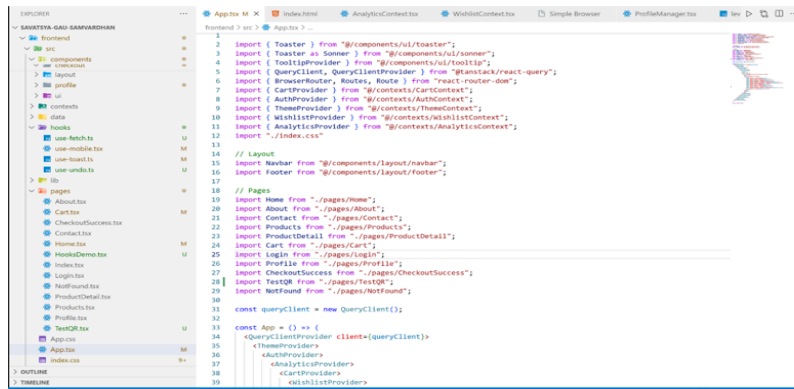
- **Context API Example:**

Suppose you are building a theme-switcher app. Context API can easily manage the "light" or "dark" theme across the app without passing props.

- **Redux Example:**

For an e-commerce store with cart management, Redux is ideal. It ensures cart data is consistent across product listing, checkout, and user account components, and also allows asynchronous operations like fetching product prices or stock availability.

SOURCE CODE:



```
1 import { Toaster } from "@components/ui/toaster";
2 import { Toaster as Sonner } from "@components/ui/sonner";
3 import { TooltipProvider } from "@components/ui/tooltip";
4 import { QueryClient, QueryClientProvider } from "@tanstack/react-query";
5 import { BrowserRouter, Routes, Route } from "react-router-dom";
6 import { CartProvider } from "@contexts/CartContext";
7 import { AuthProvider } from "@contexts/AuthContext";
8 import { ThemeProvider } from "@contexts/ThemeContext";
9 import { WishlistProvider } from "@contexts/WishlistContext";
10 import { AnalyticsProvider } from "@contexts/AnalyticsContext";
11 import "./index.css";
12
13 // Layout
14 import Navbar from "@components/layout/navbar";
15 import Footer from "@components/layout/footer";
16
17 // Pages
18 import Home from "@pages/Home";
19 import About from "@pages/About";
20 import Contact from "@pages/Contact";
21 import Products from "@pages/Products";
22 import ProductDetail from "@pages/ProductDetail";
23 import Cart from "@pages/Cart";
24 import Login from "@pages/Login";
25 import Profile from "@pages/Profile";
26 import CheckoutSuccess from "@pages/CheckoutSuccess";
27 import TestQR from "@pages/TestQR";
28 import NotFound from "@pages/NotFound";
29
30 const queryClient = new QueryClient();
31
32 const App = () => {
33   <QueryClientProvider client={queryClient}>
34     <ThemeProvider>
35       <AuthProvider>
36         <CartProvider>
37           <AnalyticsProvider>
38             <WishlistProvider>
```

FIG 1

output:

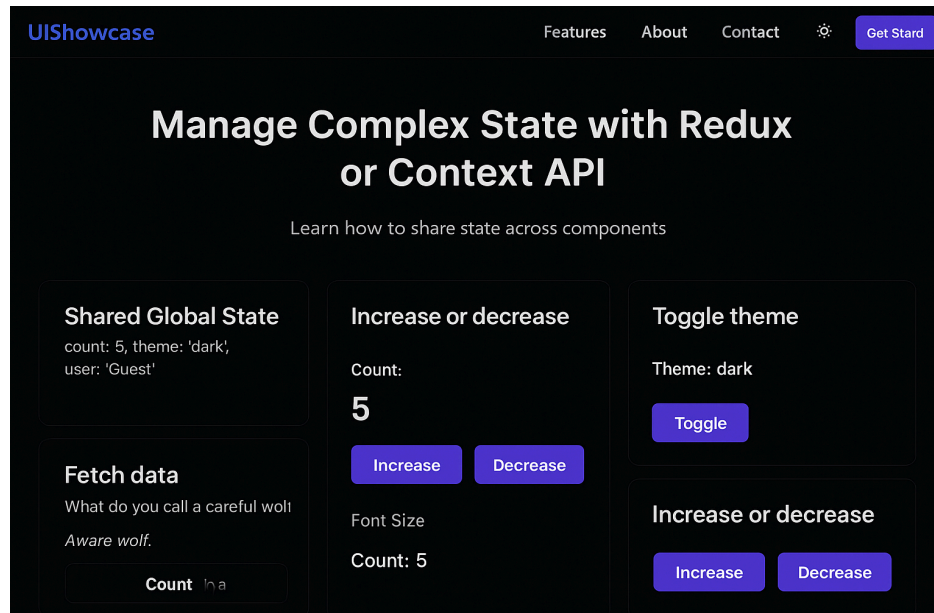


FIG 2

Conclusion:

Managing state effectively is one of the most important aspects of building reliable and scalable React applications. For small to medium projects, the **Context API** is a simple and efficient choice, as it avoids prop-drilling and provides direct access to shared state. However, as applications grow in size and complexity, **Redux** offers better structure, predictability, and powerful debugging features. Choosing between the two depends on the **scope and complexity** of the project: use Context API for lightweight global state management (themes, authentication, preferences) and Redux for enterprise-level applications requiring advanced features like middleware, async actions, and detailed debugging tools.