## **Experiment No: 3**

## **Aim:** Manage complex state with redux or context API

## **Theory**

In modern web applications, **state management** plays a vital role in ensuring that the data displayed across different components remains synchronized and consistent. As applications grow in complexity, managing state becomes more challenging because multiple components may need to access and update the same data.

React provides two widely used approaches for managing global state: **Redux** and **Context API**.

* **Redux** is a powerful state management library that follows a predictable state container pattern. It is often used for large-scale applications that involve complex state transitions, middleware for side effects, and debugging tools.
* **Context API**, on the other hand, is a **built-in feature of React** that allows developers to share state across the component tree without relying on external libraries. It eliminates the problem of **prop drilling**, where data must be passed through multiple layers of components unnecessarily.

By choosing the appropriate state management approach, developers can improve the **scalability, maintainability, and performance** of their applications.

### **What is Context API?**

The **Context API** in React is a mechanism that enables developers to share data (state) globally across components, without needing to pass props at every level. This approach is particularly useful in applications where multiple components depend on the same piece of information.

The **React Context API** is a built-in feature that simplifies **state sharing across components** without prop drilling. It mainly consists of three core concepts:

1. **createContext()** – Used to create a new **Context object** that holds shared data or state.
2. **Provider** – Wraps around components to **make the context value available** to all its child components.
3. **useContext()** – Allows any child component to **directly consume the context value** without the need to pass props manually.

This approach promotes **clean, modular, and maintainable code**, where application state is **centralized** and easily accessible from anywhere within the component tree.

**Why Context API for Complex State?**

In applications with multiple components that need to share and update the same data, using props for communication often leads to **prop drilling** — passing props through several intermediate components unnecessarily. The Context API solves this problem efficiently.

For example, in this project:

* The **Feedback Form** updates the list of feedback items.
* The **Feedback List** displays all submitted feedback dynamically.
* The **Theme Toggle** manages the global UI theme (light or dark mode).

Without Context API, these components would need props passed down through many layers, making the app harder to scale and maintain.  
With the Context API, all components can **access shared data directly from a centralized store**, improving both **performance** and **developer productivity**.

**Local Storage in Web Applications (Extra Concept)**

While the Context API handles **state management**, it does not ensure **data persistence**. When the page is refreshed, all in-memory state is lost.  
To preserve data between sessions, web applications can use **Local Storage**.

**Local Storage** is a browser-based mechanism that allows developers to store **key-value pairs persistently** on the client side. Unlike session storage, data in Local Storage **remains available even after the browser is closed and reopened**.

**Key Features of Local Storage**

* Stores data in the browser **with no expiration time**.
* Data is **domain-specific** — only accessible by the domain that created it.
* Provides easy-to-use APIs:
  + setItem() → Save data
  + getItem() → Retrieve data
  + removeItem() → Delete specific data
  + clear() → Remove all stored data
* Can store up to **5–10 MB** of data depending on the browser.

**Use of Local Storage in This Project**

* **User Feedback Persistence:** Feedback items are stored in Local Storage, so the data remains even after the page is refreshed.
* **Theme Preference:** The chosen theme (light or dark) is saved, allowing the same theme to be applied automatically when the user revisits.

**Source Code :**

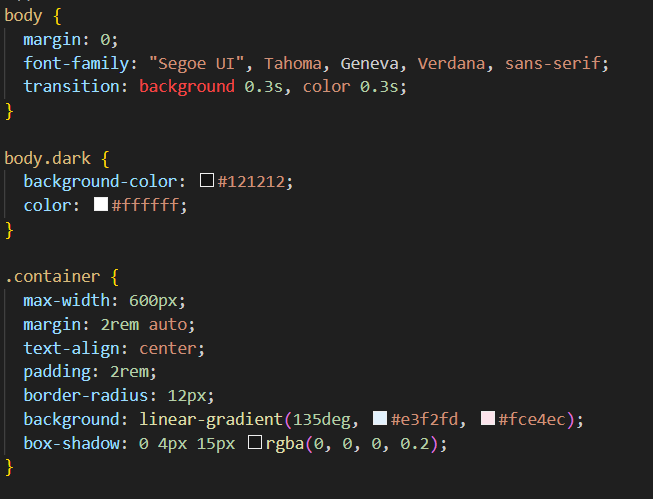
## **1 – HTML / JSX Structure**

## 

## 

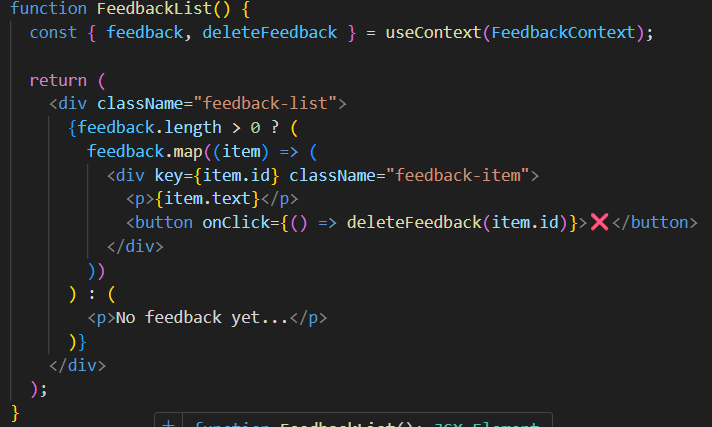
## 

## **2 – CSS (Website Styling)**

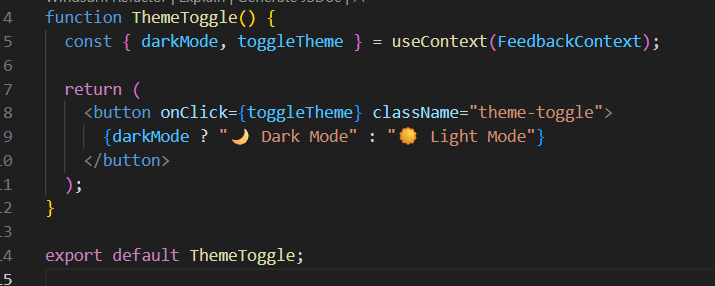
****

## **3 – Feedback Logic (JavaScript)**

## 

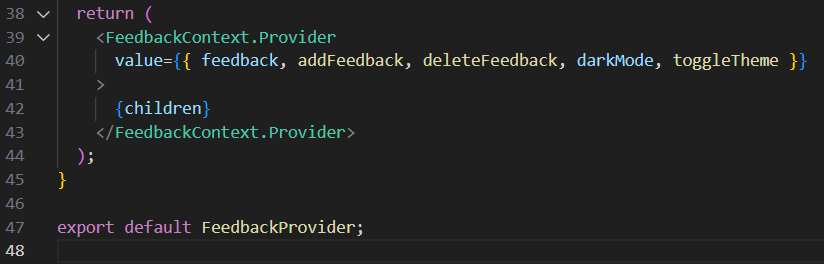
****

## **4 – Theme Toggle (JavaScript)**

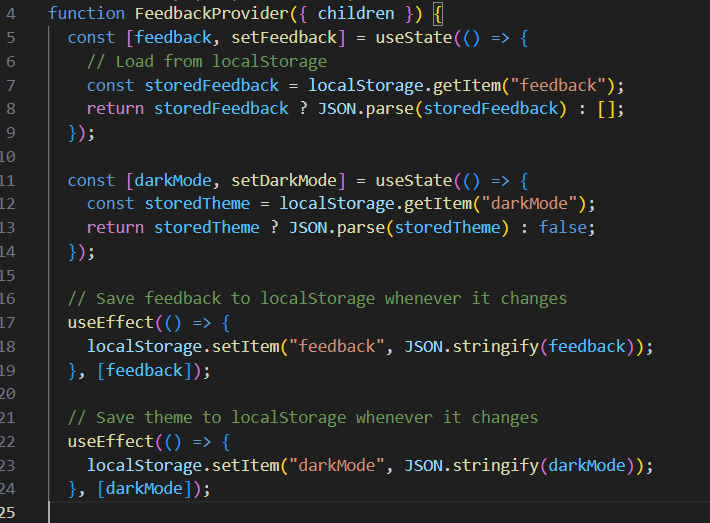
****

## **5 – Context API Implementation**

## 

****

## **6 – Local Storage (JavaScript)**

****

## 

## **Output**

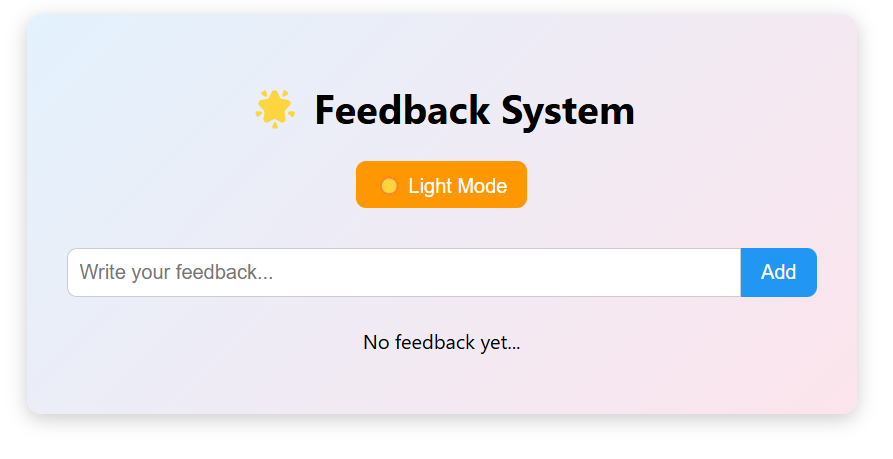


Fig 3.1 Homepage view.

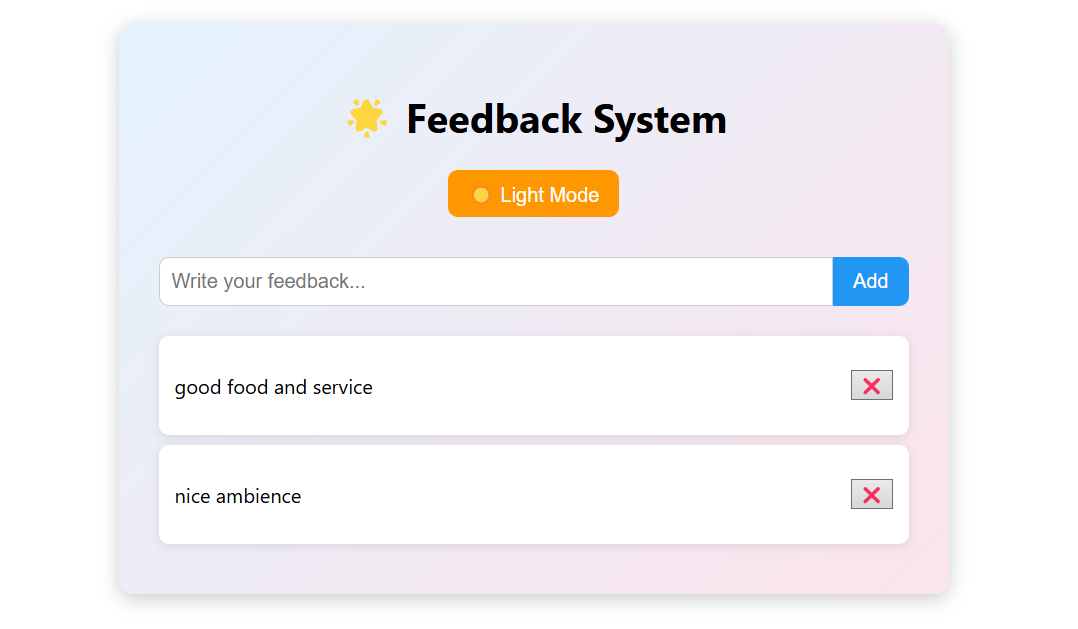


Fig 3.2 Feedback section after submission (showing persistent data from Local Storage).

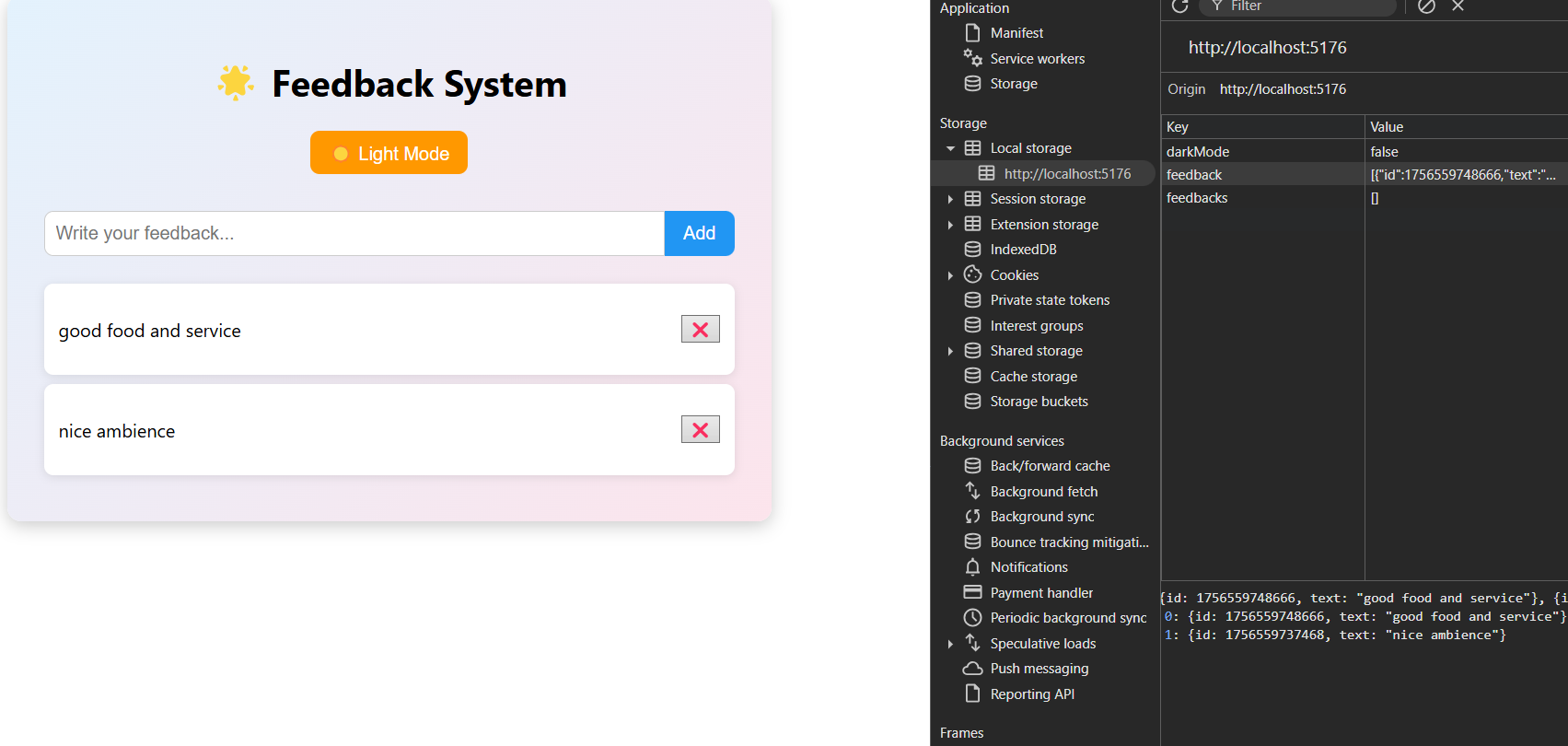


Fig 3.3 showing persistent data from Local Storage

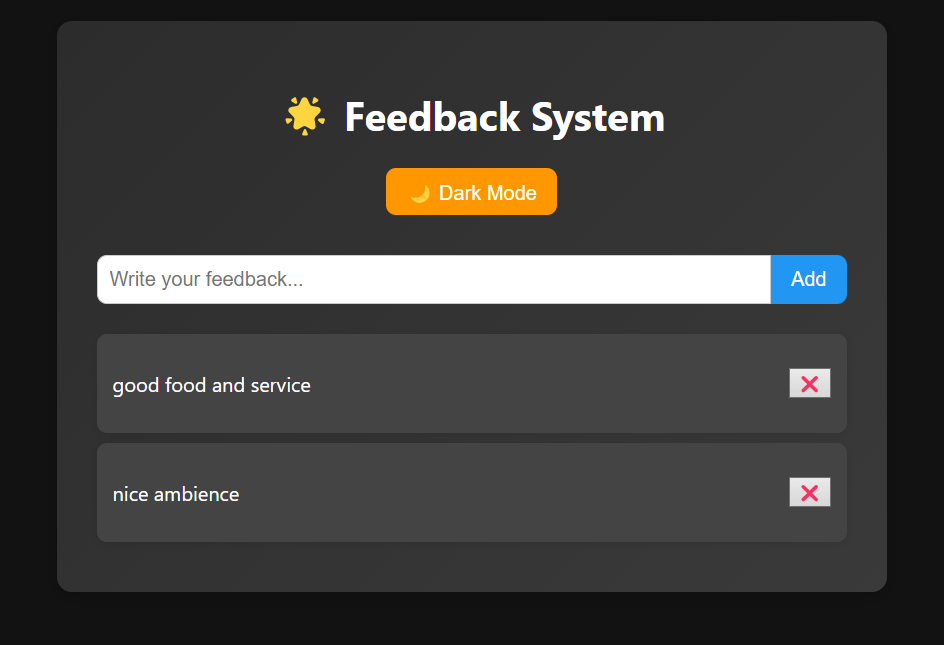


Fig 3.4 Theme toggle in action

**Conclusion**

The experiment successfully demonstrated the use of Local Storage in a website to store user feedback persistently. The addition of a theme toggle system improved usability. This approach eliminates backend dependency for small projects, showing how client-side storage can manage lightweight data efficiently.