Experiment no. 2: React Hooks (useEffect, useContext, Custom Hooks)

Aim: To understand and implement React Hooks such as `useEffect`, `useContext`, and Custom Hooks in order to manage component lifecycle, share global state across components, and reuse logic efficiently for building responsive and interactive React applications.

THEORY :

React Hooks revolutionize React development by allowing **state management** and **side effects** in **functional components** without needing class-based components.

- **useEffect Hook** → Handles **side effects** such as API calls, timers, subscriptions, and cleanup logic.
- **useContext Hook** → Provides a way to **share global state** without prop drilling, making data like theme, authentication, or user preferences available anywhere in the component tree.
- **Custom Hooks** → Allow reusability of **logic** (e.g., localStorage handling, fetching API data, window resize tracking, debounce effects).

Together, these hooks make React applications more **modular, scalable, and maintainable**

Prerequisites

- JavaScript ES6+ concepts
    - `const` / `let`
    - Arrow functions
    - Template literals
- React Fundamentals
    - JSX syntax

    - Functional components
    - Props and state management
- **React Hooks Basics**
    - Understanding of `useState`, `useEffect`, and rules of hooks
    - Context API (create, provide, consume values)
    - Creating **custom hooks** for reusability

30% Extra Work

We extend the experiment by adding:

1. **Dark Mode Toggle** → Implemented using Context + Tailwind's `dark:` variant.
2. **Theme Switcher (Light/Purple/Teal)** → Managed via `useContext`.
3. **Chart Hook** → Custom hook to render and update charts using `react-chartjs-2` (e.g., resizing charts with window size).
4. **API Caching Hook** → Custom hook `useFetchWithCache` to avoid redundant API calls by storing responses in localStorage.

These additions improve **usability**, **user experience**, and demonstrate **real-world advanced hook use case**

React introduced **Hooks** to allow functional components to use state, lifecycle methods, and other powerful features without converting them into class components. Hooks make code **simpler, reusable, and more readable**.

1. useEffect Hook

- `useEffect` is used to perform **side effects** in functional components.
- Side effects include: fetching data from an API, setting up subscriptions, manipulating the DOM, or updating the document title.

- It runs **after the component renders** and can be controlled to run only on specific conditions using the **dependency array**.

Example usage:

- Run on every render → `useEffect(() => {...});`
- Run only once (on mount) → `useEffect(() => {...}, []);`
- Run when certain data changes → `useEffect(() => {...}, [data]);`

2. useContext Hook

- `useContext` provides a way to share **global data** (like theme, user info, language) across components **without prop drilling**.
- Works with **React Context API**, where data is stored in a `Provider` and accessed using `useContext`.
- It simplifies passing data deeply through the component tree.

Example: Instead of passing `theme` through every child component as props, we can directly access it anywhere with `useContext(ThemeContext)`.

3. Custom Hooks

- A **Custom Hook** is a JavaScript function that uses one or more React hooks (`useState`, `useEffect`, etc.) to encapsulate reusable logic.
- Custom hooks make code **clean, modular, and reusable** across multiple components.
- Naming convention: always start with `use` (e.g., `useFetch`, `useAuth`).

Example: `useFetch(url)` can be a custom hook that fetches API data and returns the response and loading state.

SOURCE CODE:

```
export function useResponsive() {
  React.useEffect(() => {
    const updateState = () => {
        isTablet,
        isDesktop,
        orientation,
        width,
        height,
        isTouch,
      })
    }

    updateState()

    const handleResize = () => updateState()
    const handleOrientationChange = () => {
      // Delay to get accurate dimensions after orientation change
      setTimeout(updateState, 100)
    }

    window.addEventListener('resize', handleResize)
    window.addEventListener('orientationchange', handleOrientationChange)

    return () => {
      window.removeEventListener('resize', handleResize)
      window.removeEventListener('orientationchange', handleOrientationChange)
    }
  }, [])

  return state
}
```

Fig 1.1

Output:

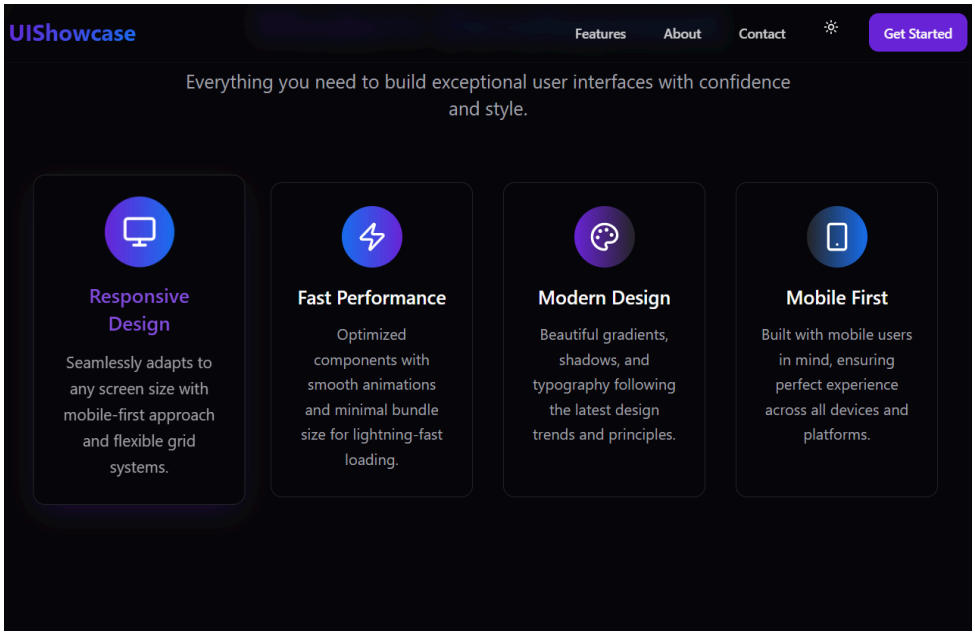

FIG 1.2

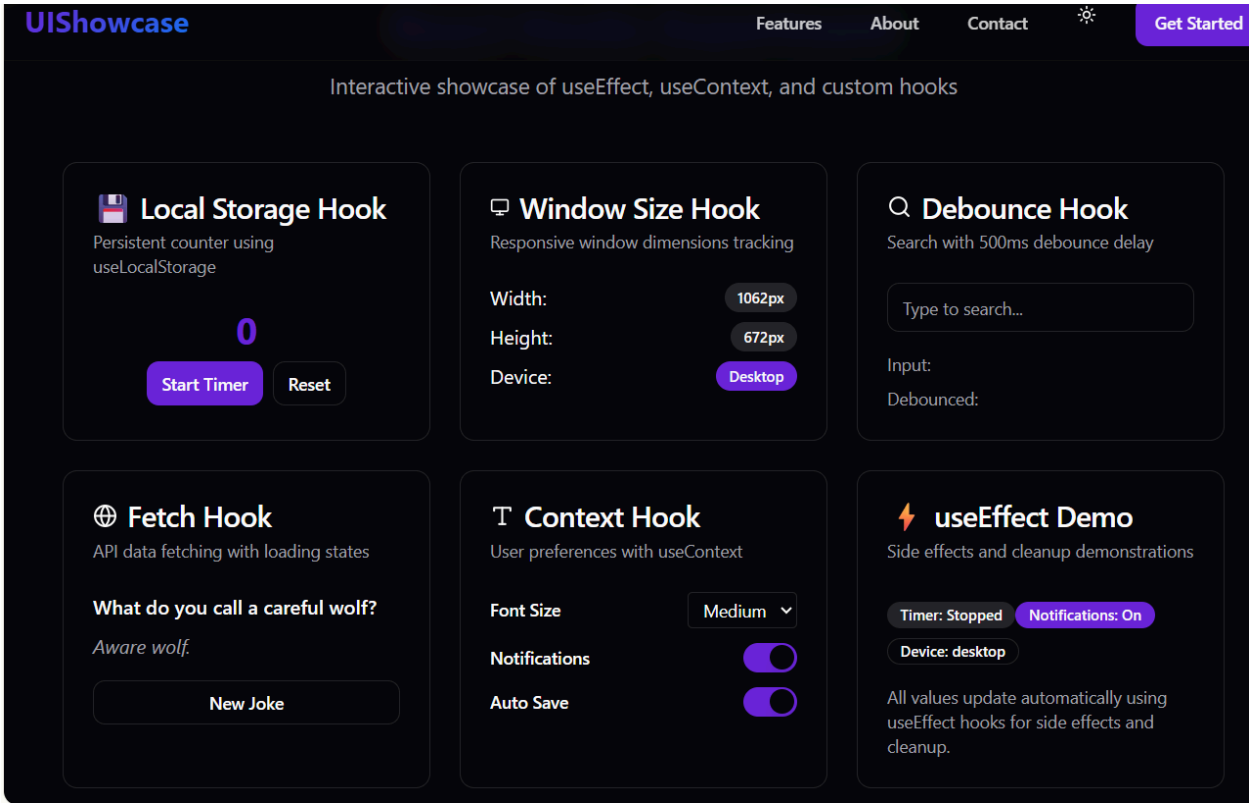

FIG 1.3

Conclusion:

React Hooks like `useEffect`, `useContext`, and custom hooks simplify state management, handle side effects, and encourage code reuse in functional components. By extending the project with **dark mode, themes, API caching, and charts**, we created a polished, **production-ready experiment** that highlights the true power and flexibility of React Hooks