Experiment No. 2

Aim : Experiment based on React Hooks (useEffect, useContext, custom hooks)

Theory:

React Hooks are special functions introduced in React 16.8 that let you use state and other React features without writing class components. The most common ones are **useState**, which manages local component state, and **useEffect**, which handles side effects like fetching data or updating the DOM. Hooks like **useContext** help in accessing global context values, while **useReducer** is useful for complex state management. **useRef** allows accessing and persisting mutable values without re-rendering, and useMemo or **useCallback** help in optimizing performance by memoizing values and functions. Overall, hooks make React code cleaner, reusable, and easier to manage by giving functional components the same power as class components but with less boilerplate.

1. useState:

- Purpose: Manages state (data that changes over time) in a functional React component.
- How it works: Returns an array with two values — the current state and a function to update it.
- When it runs: State updates cause the component to re-render with the new value.
- Example usage: Tracking form inputs, counters, toggles.
- Key points:
    - State updates are asynchronous.
    - Multiple state variables can be used in one component.
    - The initial value is set only during the first render.

2. useEffect:

- Purpose: Handles side effects in React — tasks that happen outside the normal UI rendering.
- Examples of side effects: Fetching data from an API, setting up subscriptions, updating the DOM manually, timers.
- Syntax: useEffect(callback, dependenciesArray)
    - callback runs after render.
    - dependencies array determines when the effect runs.
        - [] → runs only on mount.

        - [var] → runs when var changes.
        - Omit array → runs on every render.
- Key points:
    - Clean up side effects (like removing event listeners) by returning a function from the callback.
    - Prevents race conditions by managing dependencies carefully.

3. useContext:

- Purpose: Allows data to be shared across components without passing props manually at every level (prop drilling).
- How it works: Uses a Context Provider to supply data and a Context Consumer (or useContext) to read it.
- Best for: Theme settings, authentication state, user preferences, global app data.
- Key points:
    - Avoids unnecessary prop passing.
    - Overuse can lead to re-renders of all consumers when context changes — be mindful of performance.

4. Custom Hooks:

- Purpose: Reusable logic for state and effects in React, extracted into a function that starts with use.
- Why use them: Keeps components clean by separating complex logic.
- Example use cases: Form handling, data fetching, animations, local storage handling.
- Key points:
    - Must follow React's hook rules (only call hooks at the top level and inside React components or other hooks).
    - Can combine multiple hooks inside one custom hook**.**

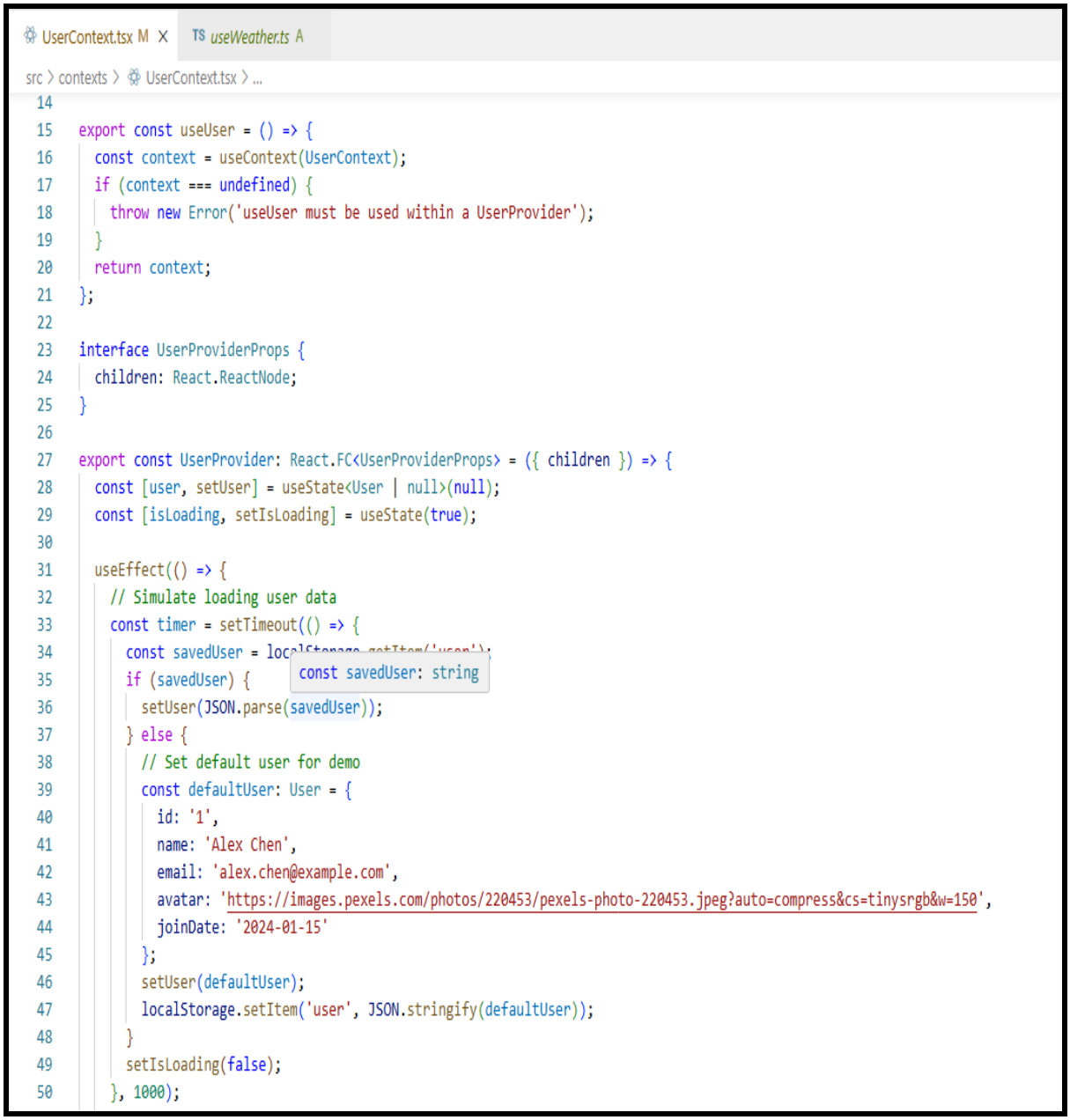
5. Framer Motion (Extra 30%):

- Purpose: A popular animation library for React that makes animations easy and declarative.

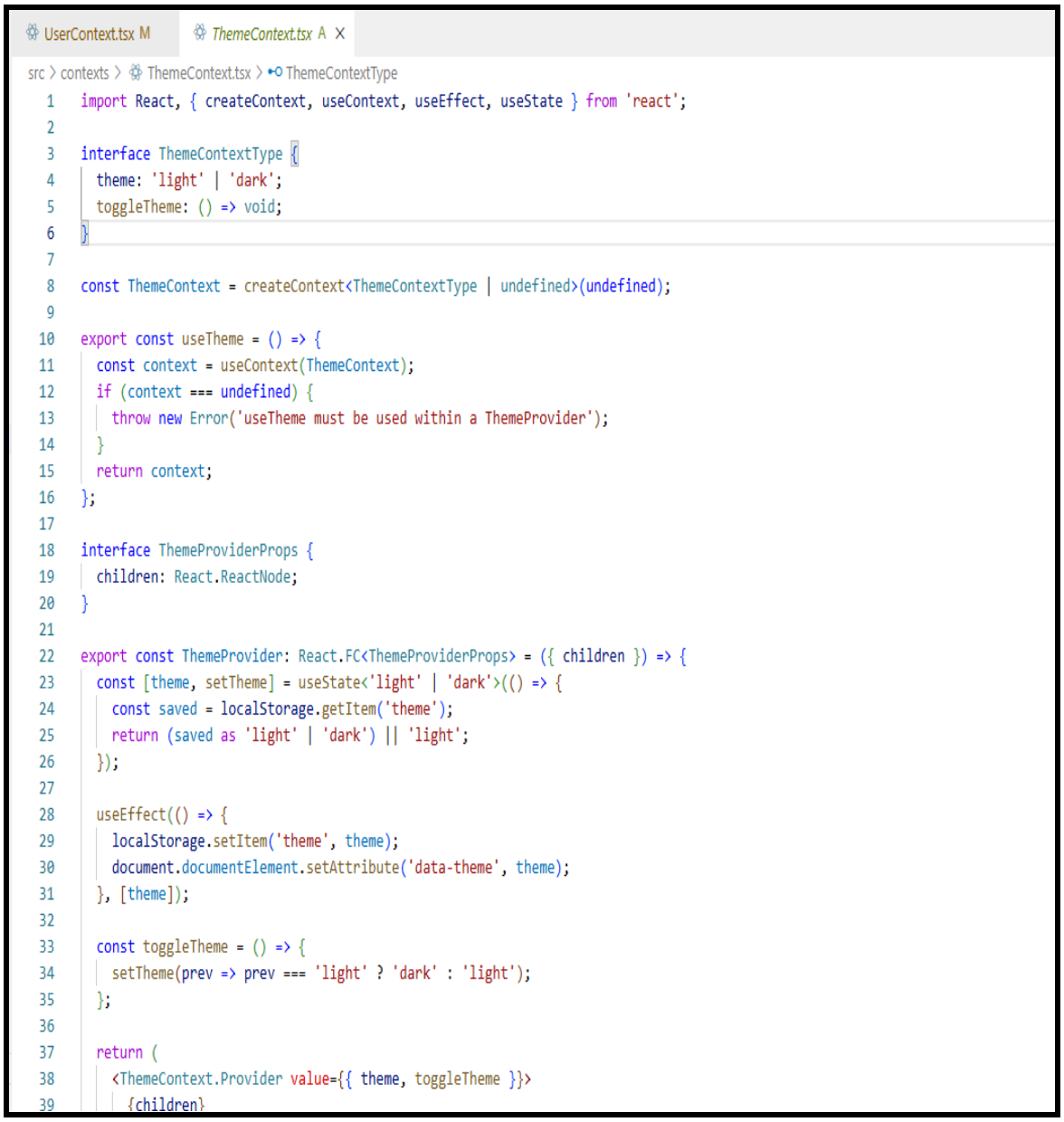
- Features:
    - Smooth transitions for elements (motion.div instead of div).
    - Gestures (drag, tap, hover) with animations.
    - Page transitions and layout animations.
    - Variants for orchestrating multiple animations.
- Key points:
    - Highly customizable and integrates well with React state.
    - Supports spring physics and keyframe animations.
    - Animations can respond dynamically to state changes.

Source Code:

1) ThemeContext.tsx
a) (useEffect):



b) useContext:



2) Custom hooks:

UserContext.tsx M useLocalStorage.ts A

src > hooks > useLocalStorage.ts > ...

```ts
import { useState, useEffect } from 'react';

export function useLocalStorage<T>(key: string, initialValue: T) {
  const [storedValue, setStoredValue] = useState<T>(() => {
    try {
      const item = window.localStorage.getItem(key);
      return item ? JSON.parse(item) : initialValue;
    } catch (error) {
      console.error(`Error reading localStorage key "${key}":`, error);
      return initialValue;
    }
  });

  const setValue = (value: T | ((val: T) => T)) => {
    try {
      const valueToStore = value instanceof Function ? value(storedValue) : value;
      setStoredValue(valueToStore);
      window.localStorage.setItem(key, JSON.stringify(valueToStore));
    } catch (error) {
      console.error(`Error setting localStorage key "${key}":`, error);
    }
  };

  useEffect(() => {
    const handleStorageChange = (e: StorageEvent) => {
      if (e.key === key && e.newValue) {
        try {
          setStoredValue(JSON.parse(e.newValue));
        } catch (error) {
          console.error(`Error parsing localStorage value for key "${key}":`, error);
        }
      }
    };

    window.addEventListener('storage', handleStorageChange);
    return () => window.removeEventListener('storage', handleStorageChange);
  }, [key]);

  return [storedValue, setValue] as const;
}
```

3) useState

4) Framer Motion (30% Extra)

```jsx
return (
  <motion.div
    className="bg-white/70 dark:bg-slate-800/70 backdrop-blur-lg border border-slate-200 dark:border-slate-700 rounded-2xl p-6 cursor-poin
    variants={containerVariants}
    initial="hidden"
    animate="visible"
    whileHover="hover"
  >
    <div className="flex items-center justify-between mb-4">
      <motion.div
        className={`p-3 rounded-xl bg-gradient-to-r ${colorClasses[color].split(' ')[0]} ${colorClasses[color].split(' ')[1]}`}
        variants={iconVariants}
        initial="initial"
        whileHover="hover"
      >
        <Icon className="w-6 h-6 text-white" />
      </motion.div>
      {change && (
        <motion.span
          className="text-xs font-medium text-green-600 bg-green-100 dark:bg-green-900/30 px-2 py-1 rounded-full"
          variants={changeVariants}
          initial="hidden"
          animate="visible"
        >
          {change}
        </motion.span>
      )}
    </div>

    <div>
      <h3 className="text-sm font-medium text-slate-600 dark:text-slate-400 mb-1">
        {title}
      </h3>
      <p className="text-2xl font-bold text-slate-900 dark:text-white group-hover:text-transparent group-hover:bg-gradient-to-r group-hove
        {value}
```

App.tsx 1

src > App.tsx > ...

```tsx
import React from 'react';
import { ThemeProvider } from './contexts/ThemeContext';
import { UserProvider } from './contexts/UserContext';
import { Layout } from './components/Layout';
import { Header } from './components/Header';
import { Dashboard } from './components/Dashboard';

Windsurf: Refactor | Explain | Generate JSDoc | ×
function App() {
  return (
    <ThemeProvider>
      <UserProvider>
        <Layout>
          <Header />
          <Dashboard />
        </Layout>
      </UserProvider>
    </ThemeProvider>
  );
}

export default App;
```

Output:

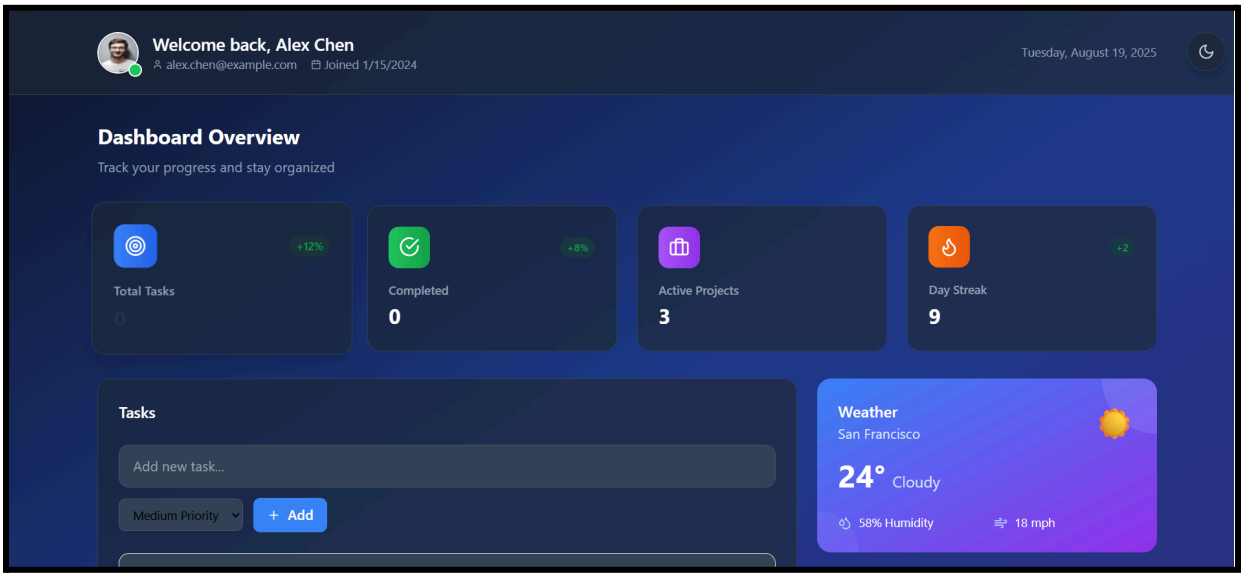


Figure 1.1 Home page

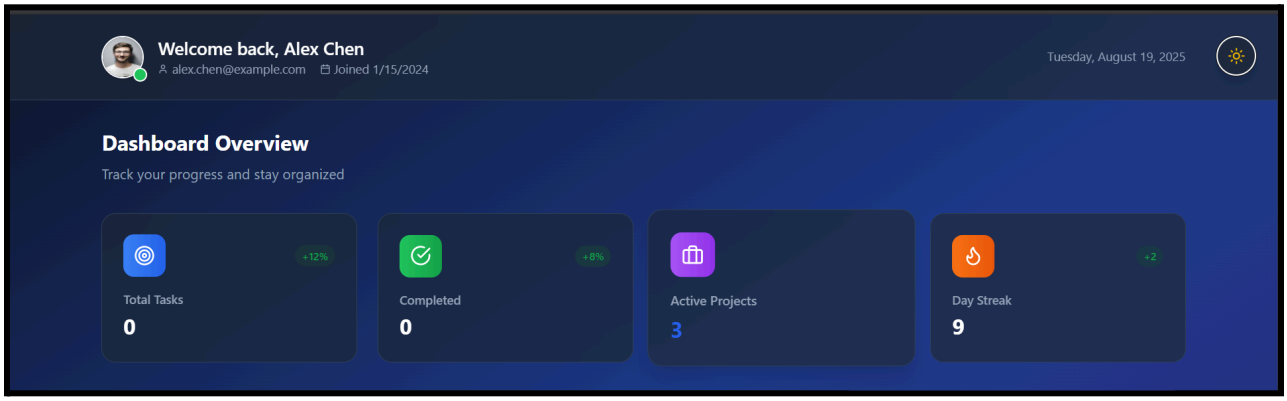


Figure 1.2 Animated Card (On hover)

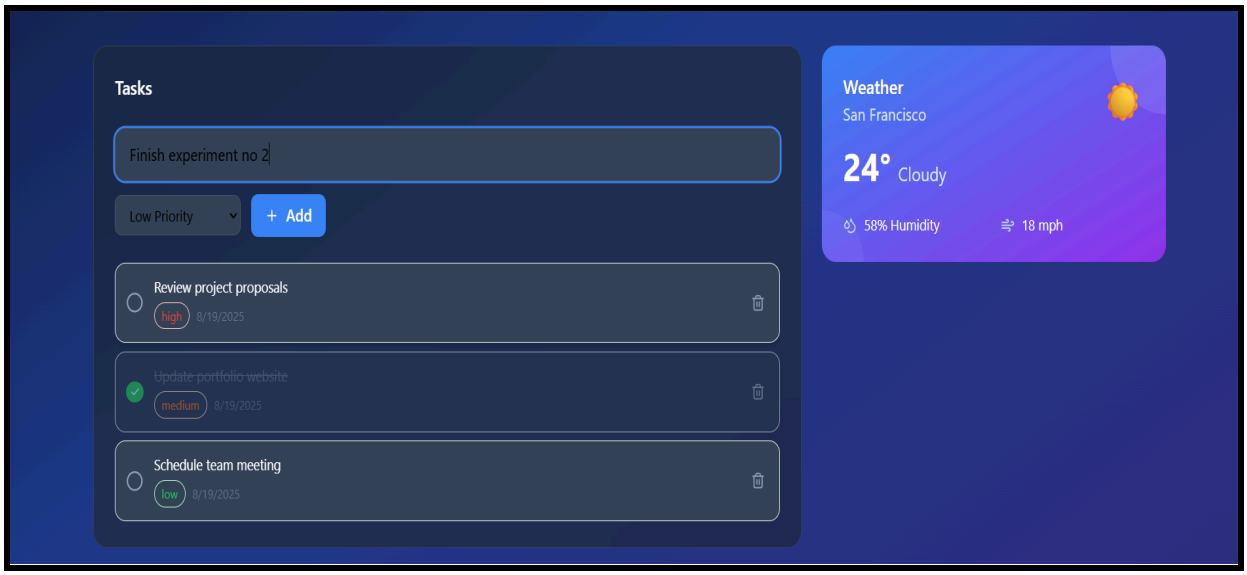


Figure 2.1 Use of useState

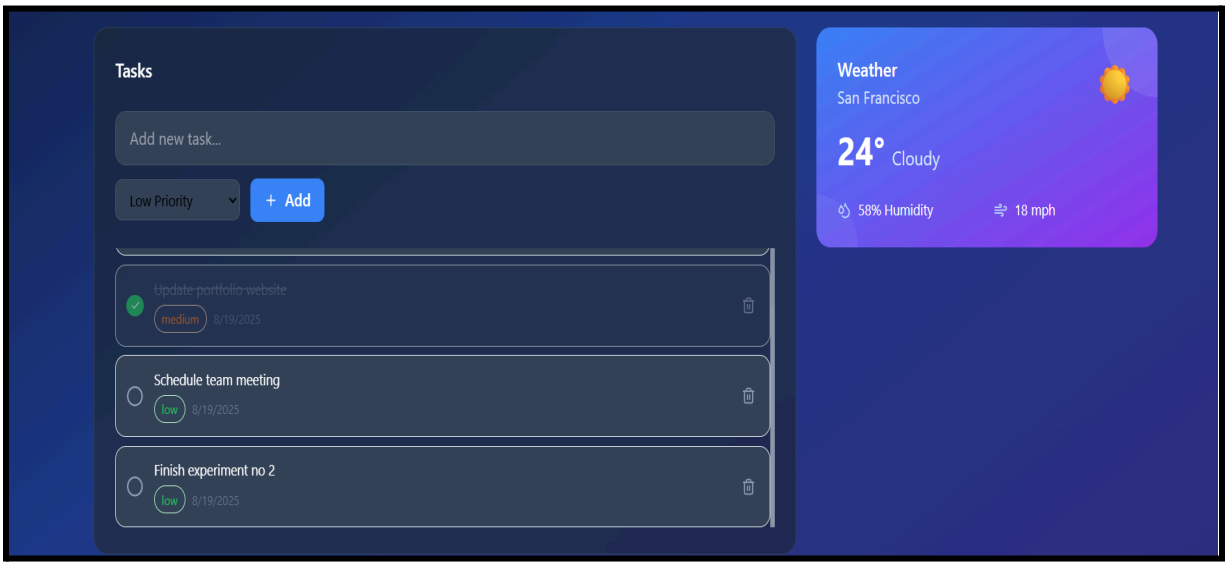


Figure 2.2 Example of useEffect (Whether data Fetching from API)

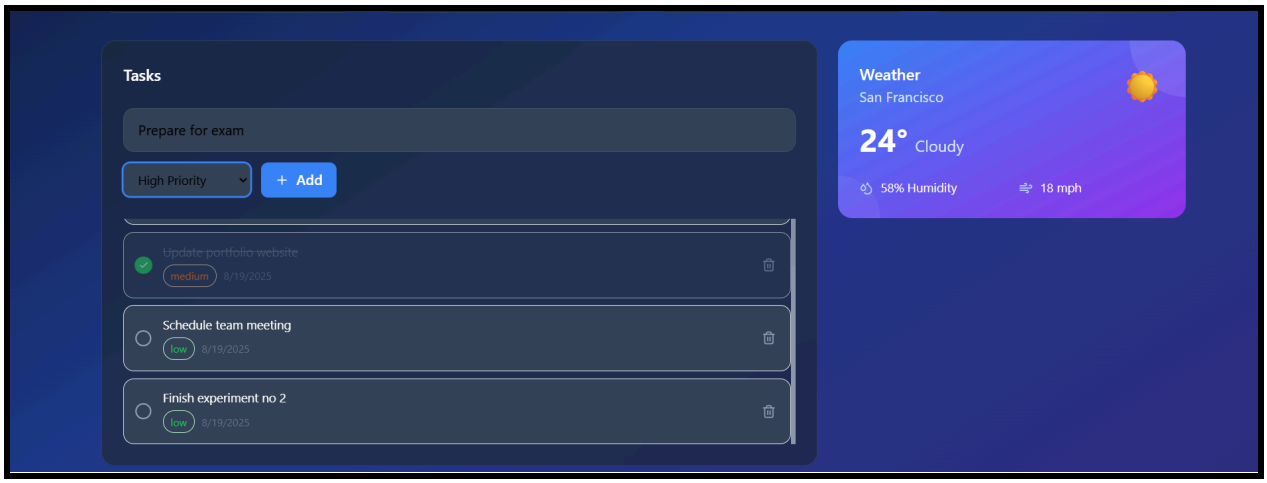


Figure 3.1 Priority high (Giving Input)

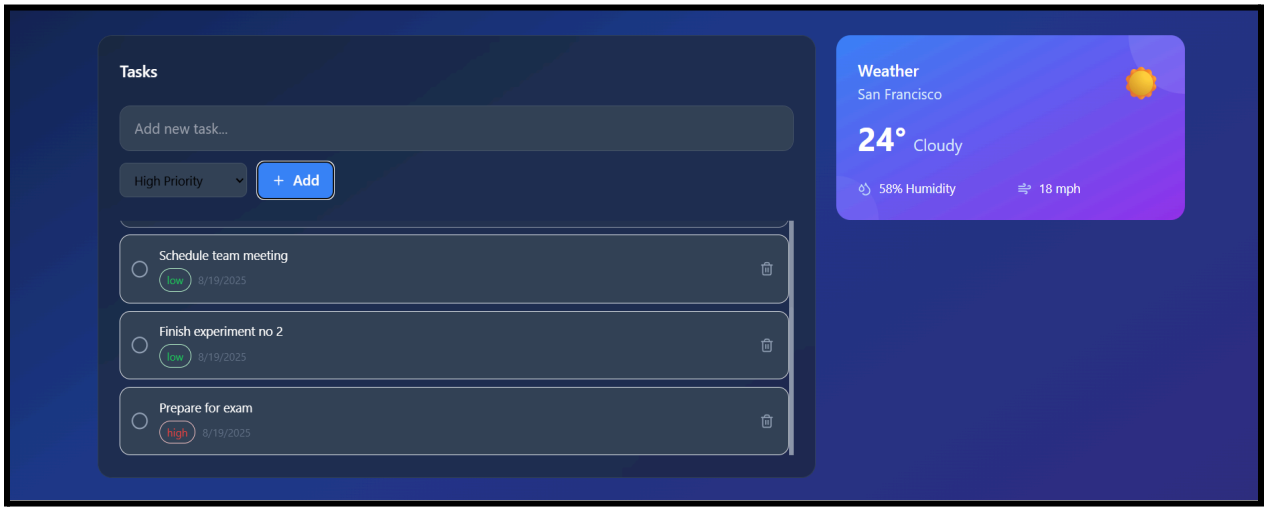


Figure 3.2 Priority high (After Input)

Conclusion :

The experiment on React Hooks demonstrates how hooks simplify state management and side-effect handling in functional components, eliminating the need for class-based components. With **useState**, we can easily manage and update local state, while **useEffect** efficiently handles lifecycle-related operations such as data fetching, subscriptions, and cleanup. The **useContext** hook enables sharing of data across multiple components without the hassle of prop drilling, thereby improving code readability and maintainability. Additionally, the implementation of **Custom Hooks** shows how common logic can be abstracted and reused across different components, reducing code duplication and enhancing modularity.

Moreover, the integration of **Framer Motion** adds smooth, declarative animations to React components, enhancing user experience with visually appealing transitions and interactions. By using Framer Motion's powerful yet intuitive API, developers can create animated elements, gestures, and dynamic UI effects without complex CSS or JavaScript animations.