**Day -2**

Today’s Goal :

* UseEffect
* Component Of LifeCycle in React
* Prop vs State

Task :

* Creating a ProfileCard that gives the description.

1. UseEffect Hook

The useEffect hook is a fundamental part of React that allows functional components to handle side effects. It serves as a unified API that combines the functionality of several class component lifecycle methods into a single declarative interface. When you call useEffect, you're telling React to execute your effect function after the component renders and after the browser has updated the screen.

The hook accepts two arguments: an effect function containing your side effect logic, and an optional dependency array that controls when the effect runs. The effect function can optionally return a cleanup function that React will call when the component unmounts or before re-running the effect. This cleanup mechanism helps prevent memory leaks and ensures proper resource management.

By default, effects run after every completed render, but you can optimize performance by specifying dependencies. When you include values in the dependency array, React will only re-run the effect if those values change between renders. An empty dependency array tells React the effect doesn't depend on any values and should only run once after the initial render. If you omit the array entirely, the effect runs after every render.

The useEffect hook is particularly useful for data fetching, setting up subscriptions, manually changing the DOM, and any other operations that interact with the outside world or can't be done during rendering. It provides a clean way to keep side effects separate from your component's rendering logic while still having access to the component's props and state.

1. LifeCycle of components

The series of events that happen from the mounting of a React component to its unmounting.

Mounting – Birth of your component

Update – Growth of your component

Unmount – Death of your component

The render() method is responsible for outputting the component's HTML to the DOM in React. This required method for class components executes during both the initial mounting phase and subsequent updates. As a fundamental rule, the render() method must remain pure - it should not modify component state or produce side effects.

Following rendering, the componentDidUpdate() method triggers after React commits updates to the DOM. This method activates whenever the component updates, making it particularly useful for performing DOM operations that depend on recent changes to props or state.

Finally, the componentWillUnmount() method serves as the component's cleanup phase, executing immediately before React removes the component from the DOM. Developers typically use this method to cancel pending network requests, remove event listeners, or clear any intervals that might otherwise persist after the component's destruction.

1. Props Vs State

Props are read-only data passed from parent components to their children. They serve as a communication channel between components, allowing parents to configure and customize child components. Since props are immutable, the receiving component cannot modify them directly - any changes must come from the parent component. This enforced unidirectional flow makes applications more predictable and easier to debug. Props are particularly useful for creating reusable UI components that can display different content based on their input properties. For example, a Button component might accept color and size props to render variations of the same base component.

State represents a component's internal memory - mutable data that persists between renders and triggers updates when changed. Unlike props, state is fully controlled and managed within the component itself using either the useState hook (in function components) or this.state (in class components). State is ideal for tracking interactive elements like form inputs, toggles, or any data that changes over time in response to user actions or network responses. When state updates via setState or the state setter function, React automatically re-renders the component and its children. This reactive nature makes state essential for building dynamic interfaces, though it should be used judiciously as excessive state can make components harder to maintain.