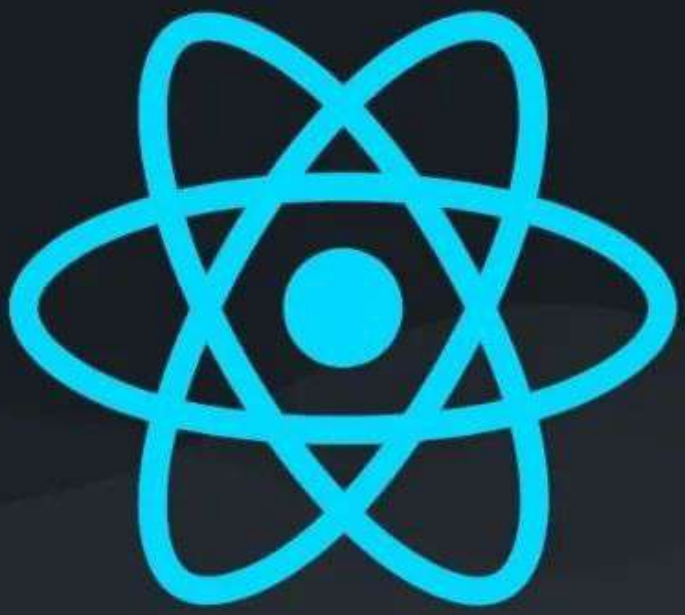
Most Useful React Hooks

useState( )
useEffect( )
useContext( )
useMemo( )
useCallback( )
useReducer( )



Mallikarjun | @CodeBustler

useState( )

'**useState**' is a React hook that lets functional components **manage state**. It takes an **initial state** value and **returns an array** with the current state and a function to update it.

```
import React, { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0);

  const increment = () => {
    /* Using the functional form of setState
      to access previous state */
    setCount(prevCount => prevCount + 1);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
```

@CodeBustler

useEffect( ) | Syntax

```
useEffect(() => {
  /* Code to run when the component
     mounts or when dependencies change */

  return () => {
  /* Cleanup code to run when the component
     unmounts or when dependencies change */
  };

}, [dependency1, dependency2, ...]);

/* Empty Dependency Array: Runs once after
  initial render, like componentDidMount.

  With Dependencies: Runs after initial render
  and when any dependency changes.

  No Dependency Array: Runs after every render;
  use cautiously to avoid performance issues. */
```

useEffect

@CodeBustler

useEffect( )

useEffect is a React hook for performing **side effects** in functional components. It **takes a function** to run the effect and an optional **array of dependencies**.

The **effect executes** after component **renders** and can **return a cleanup function**. If dependencies **change**, the effect re-runs, providing control over when it executes.

Common Use Cases

1. Data Fetching
2. Subscriptions
3. DOM Manipulation
4. State Updates
5. Cleanup

@CodeBustler

useCallback( )

@CodeBustler

useCallback Hook **memoizes callback functions**, ensuring they **only change when dependencies change**.

Ideal for **optimizing performance** by preventing **unnecessary re-renders, especially** when passing callbacks to **child components**

Syntax

```
const memoizedCallback = useCallback(
() => {
    // Callback function logic
  },

  [dependencies]
);
```

useContext( )

@CodeBustler

useContext is a to access the values of a **Context**. Context provides a way to pass data through the component tree **without having to pass props** down manually at every level

```
// 1. Create Context
const MyContext = React.createContext(defaultValue);

// 2. Provide context to parent Componest
const data = "CodeBustler";
<MyContext.Provider value= {data} >
  <App/>  {/* Components can access the context value*/}
</MyContext.Provider>

// 3. Consume context data in a nested components
const contextData = useContext(MyContext);
console.log(contextData) // "CodeBustler"
```

useContext **simplifies sharing state** between components **without prop drilling.**

useReducer( )

@CodeBustler

useReducer is an alternative to **useState** hook & is used for **state management** in functional components.

It's beneficial when state transitions have **specific logic** or when the next **state depends on the previous state.**

Syntax

```
const [state, dispatch] = useReducer(reducer, initialState);
```

- **state** : Current state.
- **dispatch** : Function to change state using actions.
- **reducer** : A function that accepts the current state and an action, and returns a new state.
- **initialState** : Initial state value.

useMemo( )

@CodeBustler

useMemo Hook is **memoizes function** or **computation results**, it preventing unnecessary recalculations when the component re-renders.

It takes a **function** and an **array of dependencies**, recalculating the value only **when dependencies change**, optimizing performance.

Syntax

```
const memoizedValue = useMemo(() => {
  return computeExpensiveValue(a, b);
}, [a, b]);
```

Return Result

useMemo and useCallback **serve similar purposes** in optimizing performance