**React Hooks**

**Use State:**

useState is a built in react-hook and most commonly used hooks in react. It allows to manage and update their own state.

**Example without using Use state hook and Why we use it?**

**export default function App() {**

**let counter=0;**

**const increment=()=>{**

**counter=counter+1;**

**console.log(counter);**

**}**

**return (**

**<div className="App">**

**{counter}**

**<button onClick={increment}>Increment</button>**

**</div>**

**);**

**}**

The above code dosen’t work, because we didn’t define react to re-render the page and show the new values whenever the changes done. But the value gets consoled with correct values only.To fix this we use UseState hook to update & manage the value.

**Example with using Use state hook:**

**import { useState } from "react";**

**export default function App() {**

**const[value,setValue]=useState(0);**

**const increment=()=>{**

**setValue(value+1);**

**}**

**return (**

**<div className="App">**

**{value}<button onClick={increment}>Increment</button>**

**</div>**

**);**

**}**

**Value:**Current value. It’s the data want to store & manage within the functional component.

**setValue:**It updates the state with the new value. setValue is the setter function, we call the setValue function with anew value.We can use the state value in JSX, or any other part of our component.

**Functional Update:**setValue function with a function argument to perform more advanced state updates based on previous state.Especially useful in cases of asynchronous updates.

**Example using useState hook with previous value &asynchronous:**

**1:**

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

**function Counter() {**

**const [count, setCount] = useState(0);**

**const incrementByTwo = () => {**

**// Using a function to update the state based on the previous state**

**setCount((prevCount) => prevCount + 1);**

**setCount((prevCount) => prevCount + 1);**

**}**

**return (**

**<div>**

**<p>Count: {count}</p>**

**<button onClick={incrementByTwo}>Increment by Two</button>**

**</div>**

**);**

**}**

**export default Counter;**

**2:**

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

**function DelayedCounter() {**

**const [count, setCount] = useState(0);**

**const incrementWithDelay = () => {**

**setTimeout(() => {**

**setCount((prevCount) => prevCount + 1);**

**}, 1000); // Increment the counter after a 1-second delay**

**}**

**return (**

**<div>**

**<p>Count: {count}</p>**

**<button onClick={incrementWithDelay}>Increment with Delay</button>**

**</div>**

**);**

**}**

**export default DelayedCounter;**

Alternatives includes Redux for global state management, the context Api for sharing state among components. Third party like Mobx.

**Use Reducer:**

useReducer is another hook to handle state & an alternative to useState hook.useReducer hook is not a replacement of Redux.

**Where to use useState & useReducer:**

useState for primitive types and where state managing is easy.useState can be a little complexity for non-primitive.

useReducer for non-primitive types where state complexity comes & handles better.

**Example 1:**

**Simple counter app using useReducer:**

**import React, { useReducer } from 'react';**

**const counterReducer=(state,action)=>{**

**switch(action.type){**

**case 'INCREMENT':**

**return{count:state.count+1}**

**case 'DECREMENT':**

**return{count:state.count-1}**

**default:**

**return state;**

**}**

**}**

**const Counter=()=>{**

**const[state,dispatch]=useReducer(counterReducer,{count:0})**

**return(**

**<div>**

**<p>{state.count}</p>**

**<button onClick={()=>{dispatch({type:'INCREMENT'})}}>increment</button>**

**<button onClick={()=>{dispatch({type:'DECREMENT'})}}>decrement</button>**

**</div>**

**);**

**}**

**export default Counter;**

**Example 2:**

**import React, { useReducer } from "react";**

**const reducer = (state, action) => {**

**switch (action.type) {**

**case "INCREMENT":**

**return { count: state.count + 1, showText: state.showText };**

**case "TOGGLESHOW":**

**return { count: state.count, showText: !state.showText };**

**default:**

**return state;**

**}**

**};**

**const App = () => {**

**const [state, dispatch] = useReducer(reducer, { count: 0, showText: true });**

**return (**

**<div>**

**<p>{state.count}</p>**

**<button**

**onClick={() => {**

**dispatch({ type: "INCREMENT" });**

**dispatch({ type: "TOGGLESHOW" });**

**}}**

**>**

**click here**

**</button>**

**{state.showText && <p>this is a text</p>}**

**</div>**

**);**

**};**

**export default App;**

**Example 3:**

**Todos using ‘useState’ hook:**

**import React, { useState } from "react";**

**const App = () => {**

**const [input, setInput] = useState("");**

**const [todos, setTodos] = useState([]);**

**const addTodoHandler = (e) => {**

**setTodos([...todos, input]);**

**setInput("");**

**e.preventDefault();**

**console.log(todos);**

**};**

**const deleteHandler = (index) => {**

**const newTodos = [...todos];**

**newTodos.splice(index, 1);**

**setTodos(newTodos);**

**};**

**return (**

**<div>**

**<input type="text" onChange={(e) => setInput(e.target.value)} />**

**<button onClick={addTodoHandler}>Add Todo</button>**

**<div>**

**{todos.map((todo, index) => {**

**return (**

**<div key={index}>**

**<p>{todo}</p>**

**<button onClick={() => deleteHandler(index)}>delete</button>**

**</div>**

**);**

**})}**

**</div>**

**</div>**

**);**

**};**

**export default App;**

**///The complexity may arises if we used more useStates. The above code can be written in simple way using useReducer with less complexity.//**

**Example 4 :**

**Todos app using useReducer:**

**import React, { useReducer, useState } from "react";**

**const initialState = [];**

**const reducer = (state, action) => {**

**switch (action.type) {**

**case "ADDTODO":**

**return [**

**...state,**

**{**

**id: state.length + 1,**

**input: action.payload**

**}**

**];**

**case "DELETE":**

**return state.filter((d) => d.id !== action.payload);**

**case "RESET":**

**return action.payload;**

**default:**

**return state;**

**}**

**};**

**const App = () => {**

**const [input, setInput] = useState("");**

**const [todos, dispatch] = useReducer(reducer, initialState);**

**const handleAddTodo = () => {**

**dispatch({ type: "ADDTODO", payload: input });**

**setInput(""); // Reset the input field**

**};**

**return (**

**<div>**

**<input**

**type="text"**

**value={input}**

**onChange={(e) => setInput(e.target.value)}**

**/>**

**<button onClick={handleAddTodo}>ADD todo</button>**

**<button onClick={() => dispatch({ type: "RESET", payload: initialState })}>**

**reset**

**</button>**

**{todos.map((todo, index) => {**

**return (**

**<div key={index}>**

**<p>{todo.input}</p>**

**<button onClick={() => dispatch({ type: "DELETE", payload: todo.id })}>**

**Delete**

**</button>**

**</div>**

**);**

**})}**

**</div>**

**);**

**};**

**export default App;**

useReducer returns state & dispatch. useReducer takes reducer function & intial state as an argument. Reducer is a pure function that takes previous state & action as an argument and return new state.

All the logic of our state update is going to hold inside the reducer.If we need to pass some data from UI to reducer, we use payload.

Dispatch function is used to trigger the corresponding actions and state update accordingly.

\* We have an other optional argument in useReducer hook that is Lazy intilisation

1. When intial state is not available or not ready
2. Sometimes we do operations to get intial state.

**Example for Lazy intilisation:**

**import React, { useReducer } from 'react';**

**// Reducer function to manage state changes**

**const counterReducer = (state, action) => {**

**switch (action.type) {**

**case 'INCREMENT':**

**return { count: state.count + 1 };**

**case 'DECREMENT':**

**return { count: state.count - 1 };**

**default:**

**return state;**

**}**

**};**

**const initialState = { count: 0 };**

**function Counter() {**

**// Use useReducer with lazy initial state**

**const [state, dispatch] = useReducer(counterReducer, initialState, (initialState) => {**

**// Lazy initialization, you can compute the initial state here if needed**

**return { count: 10 };**

**});**

**return (**

**<div>**

**<p>Count: {state.count}</p>**

**<button onClick={() => dispatch({ type: 'INCREMENT' })}>Increment</button>**

**<button onClick={() => dispatch({ type: 'DECREMENT' })}>Decrement</button>**

**</div>**

**);**

**}**

**export default Counter;**

**Use Effect:**

‘useEffect’ is one of the most commonly used React hooks. It allows you to perform side effects in functional components, such as data fetching, DOM manipulation. React components should be pure without any side effects. It is similar to life cycle methods in class components but is designed to work with functional conponents.

**Syntax:** useEffect(effectFunction, dependencies);

Effect Function: This is a function that contains the code for the side effect to perform.

Dependencies: An array of values that the effect depends on. If any of these values change, the effect will be re-run. If this array is not provided or is empty, the effect runs after every render.

By default, the effect runs after every render, including the intial render.

We can control when the effect should run by specifying the dependencies. If the array is provided the effect will only run when the values are changed in the array between renders.To run the effect only once, provide an empty dependency.

Clean up: useEffect can also return a clean up function if needed. This function is called when the dependencies are changing and the effect needs to be re-run. This is useful for cleaning up resources such as unsubscribing from event handlers.

**Example 1:**

**useEffect(() => {**

**const handleClick = (e) => {**

**// Handle the click event**

**};**

**window.addEventListener('click', handleClick);**

**return () => {**

**// Clean up by removing the event listener**

**window.removeEventListener('click', handleClick);**

**};**

**}, []);**

**Example 2:**

**useEffect(() => {**

**const timer = setInterval(() => {**

**// Perform a periodic task**

**}, 1000);**

**return () => {**

**clearInterval(timer); // Cleanup by clearing the timer**

**};**

**}, []);**

**Example 2:**

**import React, { useEffect, useState } from 'react';**

**const App=()=>{**

**const[users,setUsers]=useState([]);**

**const[refresh,setRefresh]=useState(false);**

**useEffect(()=>{**

**fetch('https://fakestoreapi.com/products').then((response)=>response.json())**

**.then((data)=>setUsers(data));**

**},[refresh])**

**console.log(users);**

**return(**

**<div>**

**{users.map((user,index)=>{**

**return(**

**<div key={index}>**

**<ul>**

**<li>{user.title}</li>**

**</ul>**

**</div>**

**)**

**})}**

**<button onClick={()=>setRefresh(true)}>refresh</button>**

**</div>**

**)**

**}**

**export default App;**

**Use Ref:**

The ‘useRef’ hook is one of the built-in-hooks in React and it’s primarily used for manipulating DOM directly, storing mutable values that will not re-render the components.

Syntax:

**import { useRef } from 'react';**

**const myRef = useRef(initialValue);**

**//useRef returns an object ‘current’**

**Example 1:**

**import React, { useEffect,useState } from 'react';**

**const App=()=>{**

**const[val,setVal]=useState('');**

**const[count,setCount]=useState(0);**

**useEffect(()=>{**

**setCount(count+1);**

**})**

**return(**

**<div>**

**<input type='text' value={val} onChange={(e)=>setVal(e.target.value)}/>**

**{count}**

**</div>**

**)**

**}**

**export default App;**

-->>>The above code undergoes infinite executions, if we give dependency it executes only once.To avoid this we use useRef.

Re-written Code:

**import React, { useState, useRef, useEffect } from "react";**

**const App = () => {**

**const [val, setVal] = useState("");**

**// const[count,setCount]=useState(0);**

**const count = useRef(0);**

**useEffect(() => {**

**count.current = count.current + 1;**

**});**

**return (**

**<div>**

**<input type="text" value={val} onChange={(e) => setVal(e.target.value)} />**

**{count.current}**

**</div>**

**);**

**};**

**export default App;**

**Example 2:**

**Dom Manipulation:**

**import React, { useState, useRef} from "react";**

**const App = () => {**

**const [val, setVal] = useState("");**

**const inputEl = useRef('');**

**const changeStyle=()=>{**

**inputEl.current.style.backgroundColor='skyblue'**

**inputEl.current.focus()**

**}**

**return (**

**<div>**

**<input ref={inputEl} type="text" value={val} onChange={(e) => setVal(e.target.value)} />**

**<button onClick={changeStyle}>submit</button>**

**</div>**

**);**

**};**

**export default App;**

We can easily manipulate Dom elements just by attaching a ‘ref’ to a DOM element and then access it using the ‘.current’ property of the ref.

‘useref’ does not trigger re-renders when the value changes. If you want to trigger re-renders, consider using ‘useState’ or ‘useReducer’.

**Use LayoutEffect:**

It is similar to ‘useEffect’ hook. They both share the same functionalities but the key difference is that useLayoutEffect runs synchronously after a render but before the screen is updated, useEffect runs asynchronously

After the render has been committed to the screen. It dosent block the painting of the screen.useLayoutEffect forces react to stop rendering until the code inside finishes.

**Example1:**

**import React, { useEffect, useLayoutEffect } from 'react';**

**const App=()=>{**

**useEffect(()=>{**

**console.log('useEffect')**

**})**

**useLayoutEffect(()=>{**

**console.log('useeffectLayout')**

**})**

**return(**

**<div>**

**<h1>hello</h1>**

**</div>**

**)**

**}**

**export default App;**

**//If we execute above code, useLayoutEffect executes first.If both useEffect & useLayoutEffect are present in the component, React will schedule ‘useLayoutEffect’ to runs first, followed by ‘useEffect’.**

**The above code reflects the synchronous nature of ‘useLayoutEffect’ and the asynchronous nature of ‘useEffect.’**

**Example2:**

**function MeasureElement() {**

**// Create a ref to the DOM element you want to measure**

**const ref = useRef(null);**

**// Use useLayoutEffect to perform actions after rendering but before painting**

**useLayoutEffect(() => {**

**// Get the bounding client rect of the referenced DOM element**

**const rect = ref.current.getBoundingClientRect();**

**// Log the dimensions to the console**

**console.log("Element dimensions:", rect);**

**}, []); // The empty dependency array means this effect runs only once after the initial render**

**// Return a JSX element with the ref attribute pointing to the DOM element**

**return (**

**<div>**

**<h1 ref={ref}>hello</h1>**

**<p ref={ref}> hefhiew</p>**

**</div>**

**);**

**}**

**export default MeasureElement;**

Choosing b/w useEffect & useLayoutEffect:

‘UseLayoutEffect’ has a higher perfomance cost because it runs synchronously and can potentially block the browser’s rendering.It should be used with caution to avoid Jank(Interface Unresponsive) in UI.Defalut choice should be ‘useEffect’ only. Specific use cases related tasks often use useLayoutEffect.

‘useEffect’ is suitable for most side effects.

**useImperativeHandle:**

‘useImperativeHandle’ is a react hook that allows a child component to customize the information or functions it provides to its parent component when using ‘forwardref’ feature.

‘forwardref’ is used to forward ref from a child component to a parent component.when using ‘forwardref’, the child component can use ‘useImperativeHandle’ to customize what values or functions it exposes through the forwarded ‘ref’.

**Example1:**

**import React, { useImperativeHandle, forwardRef } from 'react';**

**const ChildComponent = forwardRef((props, ref) => {**

**const internalFunction = () => {**

**console.log('Internal function called');**

**};**

**// Use useImperativeHandle to expose only the necessary parts**

**useImperativeHandle(ref, () => ({**

**internalFunction,**

**}), []);**

**return (**

**<div>**

**Child Component**

**</div>**

**);**

**});**

**// Parent component**

**const ParentComponent = () => {**

**const childRef = useRef();**

**useEffect(() => {**

**// Now, you can access the internalFunction from the child component**

**childRef.current.internalFunction();**

**}, []);**

**return (**

**<div>**

**<ChildComponent ref={childRef} />**

**</div>**

**);**

**};**

In the above example ‘ChildComponent’ exposes an ‘internalFunction’ through the ‘ref’ and the ‘ParentComponent’ can use this function when needed.

In short useImperativeHandle used to customize the methods exposed on the ref object. This ensures that only specific methods or values are accessible externally.

**Example2:**

App.js

**import React, { useRef } from 'react';**

**import Parent from './Parent';**

**export default function Form() {**

**const ref = useRef(null);**

**function handleClick() {**

**ref.current.focus();**

**}**

**return (**

**<form>**

**<Parent placeholder="Enter your name" ref={ref} />**

**<button type="button" onClick={handleClick}>**

**Edit**

**</button>**

**</form>**

**);**

**}**

Parent.js

**import React, { useRef, forwardRef, useImperativeHandle } from 'react';**

**const Parent = ({ placeholder }, ref) => {**

**const inputRef = useRef(null);**

**useImperativeHandle(ref, () => ({**

**focus() {**

**inputRef.current.focus();**

**},**

**scrollIntoView() {**

**inputRef.current.scrollIntoView();**

**},**

**}), []); // Empty dependency array ensures this runs once after the initial render**

**return <input placeholder={placeholder} ref={inputRef} />;**

**};**

**export default forwardRef(Parent);**

Suppose we don’t want expose the entire <input> DOM node,but you want to expose two of the methods: focus & scrollIntoView. useImperativeHandle to expose a handle with only the methods that you want the parent component to call.

**useContext:**

userContext provides a way to pass data through the component through the component tree without having to pass props down manually at every level.

It’s often used when some data needs to accessible by many components that are part of this context.

**Example1:**

**App.js**

**import React from 'react'**

**import { ComponentC } from './ComponentC'**

**export const userContext=React.createContext();**

**const App = () => {**

**return (**

**<div>**

**<userContext.Provider value={'hello from App.js'}>**

**<ComponentC />**

**</userContext.Provider>**

**</div>**

**)**

**}**

**export default App**

**ComponentC.js**

**import React, { useContext} from 'react'**

**import {userContext} from './App';**

**export const ComponentC = () => {**

**const contextValue=useContext(userContext);**

**return (**

**<div>**

**<userContext.Consumer>**

**{value=><div>{value}</div>}**

**</userContext.Consumer>**

**{/\* {contextValue} \*/}**

**</div>**

**)**

**}**

**Export default ComponentC;**

userContext is created using ‘createContext’ from the react library.(userContext.Provider) is used to provide a value to its desecndants.The userContext object is passed as an argument to useContext, and it returns the current context value, which is stored in the contextValue variable.(userContext.Consumer) is used to consume the context value.

**Example2:**

**App.js**

**import React,{useState} from 'react'**

**import { ComponentC } from './ComponentC'**

**export const userContext=React.createContext();**

**const App = () => {**

**return (**

**<div>**

**<userContext.Provider value={'blue'}>**

**<ComponentC/>**

**</userContext.Provider>**

**</div>**

**)**

**}**

**export default App**

**ComponentC.js**

**import React, { useContext,useState} from 'react'**

**import {userContext} from './App';**

**export const ComponentC = () => {**

**const contextValue=useContext(userContext)**

**const[value,setValue]=useState('red');**

**const changeHandler=()=>{**

**setValue(contextValue);**

**}**

**return (**

**<div>**

**<button onClick={changeHandler} style={{backgroundColor:value}}>click to change background</button>**

**</div>**

**)**

**}**

**Export default ComponentC;**

userContext is created using ‘createContext’ from the react library.(userContext.Provider) is used to provide a value to its desecndants.The userContext object is passed as an argument to useContext, and it returns the current context value, which is stored in the contextValue variable.(userContext.Consumer) is used to consume the context value.The component uses the useState hook to manage a piece of local state named value with an initial value of 'red'

The changeHandler function is a click handler for the button. It sets the local state value to the value obtained from the context (contextValue).

**Example3:**

**App.js**

**import React,{useState} from 'react'**

**import { ComponentC } from './ComponentC'**

**export const userContext=React.createContext({username:'guest',updatedUser:()=>{}});**

**const App = () => {**

**const[user,setUser]=useState({username:'john',age:'34'})**

**const updatedUser=(newName)=>{**

**setUser((prevUser)=>({...prevUser,username:newName}))**

**}**

**return (**

**<div>**

**<userContext.Provider value={{...user,updatedUser}}>**

**<ComponentC/>**

**</userContext.Provider>**

**</div>**

**)**

**}**

**export default App**

**ComponentC.js**

**import React, { useContext,useState} from 'react'**

**import {userContext} from './App';**

**export const ComponentC = () => {**

**const contextValue=useContext(userContext);**

**const{username,age,updatedUser}=contextValue;**

**return (**

**<div>**

**<p>username:{username}</p>**

**<p>Age:{age}</p>**

**<button onClick={()=>updatedUser('Bindu')}>updatedUer</button>**

**</div>**

**)**

**}**

**Export default ComponentC;**

userContext is created using ‘createContext’ from the react library.(userContext.Provider) is used to provide a value to its desecndants.The userContext object is passed as an argument to useContext, and it returns the current context value, which is stored in the contextValue variable.(userContext.Consumer) is used to consume the context value.The initial/default context value includes a username of 'guest' and an updatedUser function that does nothing initially

The component uses the useState hook to manage a piece of local state named user with an initial value of { username: 'john', age: '34' }.

**useMemo:**

useMemo hook the memorize the result of a function,so that it is only re-computed when the dependencies change. It avoids expensive calculations on every render.Helps to improve the perfomance of application while performing most expensive function.useMemo will not run for every re-render happens.

**Example1:**

**import React, { useMemo, useState } from "react";**

**const App = () => {**

**const[a,setA]=useState(0);**

**const[b,setB]=useState(0);**

**const sum=(c,d)=>{**

**for(i=0;i<10001;i++){}**

**return parseInt(c)+parseInt(d);**

**}**

**const value=useMemo(()=>{**

**return sum(a,b)**

**},[a,b])**

**return (**

**<div>**

**<input type='number' value={a} onChange={(e)=>setA(e.target.value)}/>**

**<input type='number' value={b} onChange={(e)=>setB(e.target.value)}/>**

**{value}**

**</div>**

**);**

**};**

**export default App;**

The sum function is defined to simulate a time-consuming calculation. It adds two numbers after a loop that runs 10,000 times.The useMemo hook is used to memoize the result of the sum function. It takes a function as the first argument (the function to memoize) and an array of dependencies as the second argument.

The memoized value (value) will only be recomputed if either a or b changes.

**Example2:**

**import React, { useEffect, useMemo, useState } from "react";**

**const App = () => {**

**const [a, setA] = useState(0);**

**const [b, setB] = useState(0);**

**const [dark, setDark] = useState(false);**

**const theme =useMemo(()=>{**

**return{**

**backgroundColor: dark ? "black" : "blue",**

**color: dark ? "red" : "pink",**

**}**

**},[dark])**

**const sum = (c, d) => {**

**for (i = 0; i < 10001; i++) {}**

**return parseInt(c) + parseInt(d);**

**};**

**const value = useMemo(() => {**

**return sum(a, b);**

**}, [a, b]);**

**useEffect(()=>{**

**console.log('theme updated')**

**},[theme]);**

**return (**

**<div>**

**<input type="number" value={a} onChange={(e) => setA(e.target.value)} />**

**<input type="number" value={b} onChange={(e) => setB(e.target.value)} />**

**<div style={theme}>{value}</div>**

**<button onClick={() => setDark(!dark)}>changeColor</button>**

**</div>**

**);**

**};**

**export default App;**

Three state variables are created using the useState hook: a, b, and dark. a and b represent the values of two input fields, and dark represents whether the theme is dark or not.

The theme object is memoized using the useMemo hook. It will only be recomputed when the dark state changes. This helps prevent unnecessary recalculations.

The sum function is defined to simulate a time-consuming calculation. It adds two numbers after a loop that runs 10,001 times.

The result of the sum function is memoized using the useMemo hook. It will only be recomputed when either a or b changes.

The useEffect hook is used to log a message when the theme is updated. It has a dependency array [theme], so it will run whenever the theme changes.

The use of useMemo and useEffect in this example helps optimize the rendering and logging behavior based on specific dependencies.

**useCallback:**

useCallback hook memoize the provided function itself, so that it is only recreated when the dependencies change. Returns a memoized callback instead of a single value like memo. It is used to memoize functions in functional components preventing unnecessary re-renders when functions are passed as props. It also increases perfomance application.

useCallback takes two arguments the callback function & array of depedencies.It is particularly useful when passing functions to child components to avoid un necessary re-renders.

**Example1:**

**import React, { useCallback, useState } from 'react';**

**const App=()=>{**

**const HandleButton=({onClick,label})=>{**

**return <button onClick={onClick}>{label}</button>**

**}**

**const[count,setCount]=useState(0);**

**const clickHandler=useCallback(()=>{**

**setCount(count+1);**

**},[count])**

**return(**

**<div>**

**<p>{count}</p>**

**<HandleButton onClick={clickHandler} label='increment'/>**

**</div>**

**)**

**}**

**export default App;**

useState: Initializes a state variable count with an initial value of 0 and its updater function setCount. count holds the current count value.

HandleButton Component: A functional component that renders a button. It takes onClick and label as props and renders a button with the specified label.

useCallback Hook: Memoizes the clickHandler function. This is important to avoid unnecessary recreation of the function when the component re-renders. The dependency array [count] ensures that the function is recreated only if count changes.

clickHandler Function: Increments the count when called. It is memoized using useCallback to prevent unnecessary re-creation.

Rendering: The main component renders the current value of count and the HandleButton component. The clickHandler function is passed as a prop to HandleButton for handling the button click.

This code demonstrates the usage of useState and useCallback hooks in a simple React component.

**Example2:**

**App.js**

**import React,{useState,useCallback} from 'react'**

**import ComponentB from './ComponentB';**

**const App = () => {**

**const [count,setCount]=useState(0);**

**const [todos,setTodos]=useState([]);**

**const[todo,setTodo]=useState('')**

**const addHandler=()=>{**

**setCount(count+1);**

**}**

**const addTodos = useCallback(() => {**

**if (todo !== '') {**

**setTodos([...todos, todo]);**

**setTodo('');**

**}**

**}, [todos, todo]);**

**return (**

**<div>**

**<center>**

**{count}**

**<button onClick={addHandler}>Increment</button><br/>**

**<ComponentB todos={todos}/>**

**<input type='text' value={todo} name='todo' onChange={(e)=>setTodo(e.target.value)}/>**

**<button onClick={addTodos}>Add todo</button>**

**</center>**

**</div>**

**)**

**}**

**export default App**

**ComponentB.js**

**import React from 'react'**

**const ComponentB =React.memo( ({todos}) => {**

**console.log('rendering')**

**return (**

**<div>**

**{**

**todos.map((todo,index)=>{**

**return(**

**<div key={index}>**

**<ul>**

**<li>{todo}</li>**

**</ul>**

**</div>**

**)**

**})**

**}**

**</div>**

**)**

**})**

**export default ComponentB**

State Variables: count, todos, and todo are state variables with their respective updater functions. count is incremented on button click, and todos is an array of todo items.

addHandler Function: Increments the count state when the "Increment" button is clicked.

addTodos Callback Function: Adds a new todo to the todos state array. It is memoized using useCallback to avoid unnecessary re-creation.

Rendering:

The current value of count is displayed.

The ComponentB component is rendered, passing the todos state as a prop.

An input field allows entering new todos, and a button triggers the addTodos function.

ComponentB.js:

React.memo: Memoizes the ComponentB component to prevent unnecessary re-renders if its props (todos) haven't changed.

Rendering: Maps through the todos array and renders each todo in a list.

Console Log: Logs "rendering" to the console whenever ComponentB is re-rendered.

This setup optimizes performance by memoizing ComponentB and using useCallback for the callback function to prevent unnecessary renders.

**Custom Hooks:**

Custom hooks are more of a convention than a feature. If a functions name starts with ‘use’ and it calls other hooks, we say it is a custom hook. Aim for high reusablity by creating custom hooks that encapsulate a specific piece of logic.custom hooks provide a clean and organised way to share logic between components in react.

**Example1:**

**App.js**

**import React, { useState } from "react";**

**import useCounter from "./ComponentB";**

**const App = () => {**

**const { count, increment, decrement } = useCounter(0);**

**return (**

**<div>**

**<p>Count: {count}</p>**

**<button onClick={increment}>Increment</button>**

**<button onClick={decrement}>Decrement</button>**

**</div>**

**);**

**};**

**export default App;**

**componentB:**

**import { useState } from 'react';**

**const useCounter = (initialValue = 0) => {**

**const [count, setCount] = useState(initialValue);**

**const increment = () => {**

**setCount(count + 1);**

**};**

**const decrement = () => {**

**setCount(count - 1);**

**};**

**return { count, increment, decrement };**

**};**

**export default useCounter;**

Import Statements: Import React and the custom hook useCounter from the "./ComponentB" file.

Functional Component App: Define a functional component named App.

Destructuring Values: Destructure the values returned by the useCounter hook, which are count, increment, and decrement.

JSX Rendering: Render a simple JSX structure displaying the current count and two buttons for incrementing and decrementing the count.

Import Statement: Import useState from React.

Custom Hook: Define a custom hook named useCounter that takes an optional initialValue parameter with a default value of 0.

State: Use the useState hook to create a state variable count initialized with the provided or default value.

Functions: Create two functions, increment and decrement, which update the count state accordingly.

Return Object: Return an object with the count, increment, and decrement functions.

Export: Export the useCounter hook.

This example demonstrates how to create a custom hook (useCounter) for managing a counter and then use it in a component (App) to handle state and logic related to counting.

**Example2:**

**App.js**

**import React, { useState } from "react";**

**import useFetch from "./ComponentB";**

**const App = () => {**

**const { data, loading, error } = useFetch('https://api.example.com/data');**

**if (loading) {**

**return <p>Loading...</p>;**

**}**

**if (error) {**

**return <p>Error: {error.message}</p>;**

**}**

**return (**

**<div>**

**<h2>Data:</h2>**

**<pre>{JSON.stringify(data, null, 2)}</pre>**

**</div>**

**);**

**};**

**export default App;**

**component B;**

**import { useState, useEffect } from 'react';**

**const useFetch = (url) => {**

**const [data, setData] = useState(null);**

**const [loading, setLoading] = useState(true);**

**const [error, setError] = useState(null);**

**useEffect(() => {**

**const fetchData = async () => {**

**try {**

**const response = await fetch(url);**

**const result = await response.json();**

**setData(result);**

**} catch (error) {**

**setError(error);**

**} finally {**

**setLoading(false);**

**}**

**};**

**fetchData();**

**}, [url]);**

**return { data, loading, error };**

**};**

**export default useFetch;**

Functional Component App: Define a functional component named App.

Destructuring Values: Destructure the values returned by the useFetch hook, which are data, loading, and error.

Loading Check: If loading is true, display a loading message.

Error Check: If error exists, display an error message.

Data Render: If no loading or error, render the fetched data using the pre tag for better formatting.

Custom Hook: Define a custom hook named useFetch that takes a URL parameter.

State Variables: Create state variables data (to hold fetched data), loading (to track loading status), and error (to hold potential errors).

Fetch Data Effect: Use the useEffect hook to initiate the data fetching process when the URL changes.

Fetch Data Function: Create an asynchronous function fetchData to handle the actual data fetching using the provided URL. It updates state variables accordingly.

Return Statement: Return an object with data, loading, and error to be used in the component using this hook.

Dependency Array: The effect has a dependency on the url, ensuring it runs whenever the URL changes.

Export Statement: Export the useFetch hook.

This example demonstrates the use of a custom hook (useFetch) for making API requests in a component (App). The hook manages the fetching process and provides the component with data, loading status, and potential errors. The component then renders different content based on the loading and error states.

**Limitations**

**React Hooks have been a powerful addition to React, simplifying state management and side effects in functional components. However, like any technology, React Hooks have some limitations. Here are some of the notable ones:**

**Only Call Hooks at the Top Level:**

**Hooks must be called at the top level of the functional component or in custom hooks. They should not be called inside loops, conditions, or nested functions. This rule ensures that Hooks are called in the same order every time a component renders.**

**Hooks in Functional Components Only:**

**Hooks can only be used in functional components and custom hooks. They cannot be used in class components. This is because hooks rely on the order of function calls and component rendering, which is different in functional components compared to class components.**

**No Conditionals in useEffect:**

**In the useEffect hook, you should not use conditions to determine whether to run an effect or not. This can lead to unexpected behavior. Instead, place the condition inside the useEffect callback function.**

**Custom Hooks Naming Convention:**

**Custom hooks should always start with "use" to follow the convention and help tools and developers recognize them as hooks. This is not a strict limitation but a convention.**

**Performance Concerns with React.memo and useMemo:**

**Memoization techniques like React.memo and useMemo can have a performance cost, especially if not used judiciously. Memoization should be used when necessary, and the performance benefits should be weighed against the complexity it introduces.**

**Learning Curve:**

**For developers transitioning from class components to functional components with hooks, there might be a learning curve. Understanding the nuances of each hook and their use cases requires some time and practice.**

**Large-Scale Refactoring:**

**Transitioning an existing codebase from class components to functional components with hooks can be a significant undertaking. This might not be practical or feasible for every project.**

**Limited Lifecycle Methods:**

**Hooks provide alternatives to lifecycle methods in class components, but they are different. If you are migrating from class components, you need to adapt to the new way of handling component lifecycle events.**

**Despite these limitations, React Hooks have proven to be a valuable tool for building React applications, offering a more concise and readable way to manage state and side effects in functional components. It's essential to understand these limitations and use hooks appropriately in your projects.**