REACT HOOKS

Hooks are a new feature addition in react version 16.8 which allow you to use react features without having to write a class

Ex:- state of a component

Hooks don’t work inside classes

Reasons :

Understand how this keyword in js

Remember to bind event handlers in class components

Classes don’t minify very well and make hot reloading very unreliable.

There is no particular way to reuse stateful component logic.

HOC and render props patterns do address this problem

Makes the code harder to follow

There is need a to share stateful logic in better way.

Usestate :

The useState hook is used to manage state in functional components. It allows you to declare a state variable and a function to update it. Unlike class components where this.setState is used, useState provides a simpler and more flexible way to handle state.

- **state**: The current state value.
- **setState**: A function to update the state.
- **initialValue**: The initial value of the state, which can be any data type (e.g., number, string, array, object, etc.).

```jsx
import React,{useState} from 'react';
function Usestate(){
    const [count,setCount]=useState(0)
    const increment=()=>setCount(count+1)
    const decrement=()=>setCount(count-1)
    return(
        <div>
        <button onClick={increment}>increment</button>
        <h1>{count}</h1>
        <button onClick={decrement}>decrement</button>
        </div>
        )
}
export default Usestate;
```

Prev state:

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

function Prevstate(){
    const initial=0
    const [count,setCount]=useState(initial)

    return(
        <div>
        <h1>{count}</h1>
        <button onClick={()=>setCount(initial)}>Reset</button>
        <button onClick={()=>setCount(count+1)}>increment</button>
        <button onClick={()=>setCount(count-1)}>decrement</button>
        </div>
        )
}
export default Prevstate;
```

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

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

  const increment = () => {
    setCount(prevCount => prevCount + 1);
  };

  const decrement = () => {
    setCount(prevCount => prevCount - 1);
  };

  return (
    <div>
      <h1>{count}</h1>
      <button onClick={() => setCount(initial)}>Reset</button>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
}

export default Prevstate;
```

objects

```
function Cred() {
  const [name, setName] = useState({ firstname: '', lastname: '' }

  const handleFirstNameChange = (e) => {
    setName((prevState) => ({
      ...prevState,
      firstname: e.target.value,
    }));
  };

  const handleLastNameChange = (e) => {
    setName((prevState) => ({
      ...prevState,
      lastname: e.target.value,
    }));
  };

  return (
    <div>
      <input
        type="text"
        placeholder="firstname"
        value={name.firstname}
        onChange={handleFirstNameChange}
      />
      <input
        type="text"
        placeholder="lastname"
        value={name.lastname}
        onChange={handleLastNameChange}
      />
      <h2>{name.firstname}</h2>
      <h2>{name.lastname}</h2>
    </div>
  );
}

export default Cred;
```

Arrays :

```
import React,{useState} from 'react';
function Array(){
  const [items,setItems]=useState([])
  const additem=()=>{
    setItems([...items,{
      id:items.length,
      value:Math.floor(Math.random()*10)+1}])
  }

  return(
    <div>
    <button onClick={additem}>add newitem</button>
    <ul>
    {items.map(item=>(
      <li key={item.id}>{item.value}</li>
      ))}
    </ul>
    </div>
    )
}
export default Array;
```

Useeffect

The useEffect hook in React is used to handle side effects in functional components. Side effects can include tasks like fetching data, directly manipulating the DOM, or setting up subscriptions.

It is a close replacement for componentDidMount, ComponentDidUpdate and componentWillUnmount

```
import React,{ Component } from 'react';
class CountOne extends Component{
    constructor(props){
        super(props)
        this.state={
            count:0
        }
    }
    componentDidMount(){
        document.title=`clicked ${this.state.count} times`
    }
    componentDidUpdate(prevProps,prevState){
        document.title=`clicked ${this.state.count} times`
    }
    render(){
        return(
            <div>
            <button onClick={()=>this.setState({count:this.state.count+1})}>
            click {this.state.count} times
            </button>
            </div>
            )
    }
}
export default CountOne;
```

```
import React,{useState,useEffect} from 'react';
function UseEffectOne(){
    const [count,setCount]=useState(0)
    useEffect(()=>{
        document.title=`clicked ${count} times`
    })
    return(
        <div>
        <button onClick={()=>setCount(count+1)}>clicked {count} times</button>
        </div>
        )
}
export default UseEffectOne;
```

Conditionally run effect :

In React, you can conditionally run a side effect using the useEffect hook by managing its dependency array or adding conditional logic within the effect

```
            this.state={
                count:0,
                name:''
            }
        }
        componentDidMount(){
            document.title=`clicked ${this.state.count} times`
        }
        componentDidUpdate(prevProps,prevState){
            if(prevState.count!== this.state.count){
                console.log("updating document title")
            }
            document.title=`clicked ${this.state.count} times`
        }
        render(){
            return(
                <div>
                <input type="text" value={this.state.name}
                onChange={e=>{this.setState({nmae:e.target.value})}} />
                <button onClick={()=>this.setState({count:this.state.count+1})}>
                click {this.state.count} times
                </button>
                </div>
                )
        }
    }
    export default CountOne;
```

```
    import React,{useState,useEffect} from 'react';
    function UseEffectOne(){
        const [count,setCount]=useState(0)
        const [name,setName]=useState('')
        useEffect(()=>{
            console.log("useeffect updating")
            document.title=`clicked ${count} times`
        },[count])
        return(
            <div>
            <input type="text" value={name}
            onChange={e=>setName(e.target.value)} />
            <button onClick={()=>setCount(count+1)}>clicked {count} times</button>
            </div>
            )
    }
    export default UseEffectOne;
```

Runonlyonce :

the effect to run **only once** (on the initial render), you can achieve this by providing an **empty dependency array ([])** to the useEffect hook

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

const FetchDataOnLoad = () => {
  const [data, setData] = useState(null);

  useEffect(() => {
    console.log('Fetching data...');
    fetch('https://jsonplaceholder.typicode.com/posts')
      .then((response) => response.json())
      .then((json) => setData(json));

    return () => {
      console.log('Cleanup logic, if needed.');
    };
  }, []); // Runs only once on component mount

  return (
    <div>
      <h1>Data:</h1>
      <pre>{JSON.stringify(data, null, 2)}</pre>
    </div>
  );
};

export default FetchDataOnLoad;

```

Empty Dependency Array ([]):

- The effect will only run on the initial render.
- No updates to the state or props will trigger the effect again.

Cleanup :

A **cleanup function** in useEffect is used to clean up resources when a component unmounts or before the effect re-runs. This is particularly useful for tasks like:

- Removing event listeners.
- Clearing timers or intervals.
- Canceling network requests.
- Cleaning up subscriptions.

 Runs When the Component Unmounts: The cleanup function executes when the component is removed from the DOM.

 Runs Before the Effect Re-runs: If dependencies change, the cleanup function runs before the effect is re-invoked.

```
const SubscriptionCleanup = () => {
  const [data, setData] = useState(null);

  useEffect(() => {
    const fakeSubscription = {
      subscribe: (callback) => {
        console.log('Subscribed');
        const intervalId = setInterval(() => callback(new Date().toLocaleTimeString()), 1000)
        return () => {
          console.log('Unsubscribed');
          clearInterval(intervalId);
        };
      },
    };

    const unsubscribe = fakeSubscription.subscribe(setData);

    // Cleanup: Unsubscribe from subscription
    return () => {
      unsubscribe();
    };
  }, []);

  return <div>Time: {data}</div>;
};

export default SubscriptionCleanup;
```

 The cleanup function prevents memory leaks and ensures resources are released.

 Cleanup is called:

- When the component unmounts.
- Before the effect runs again (if dependencies are updated).

Fetching data :

```
import React,{useState,useEffect} from 'react';
import axios from 'axios';
function Fetching(){
    const [posts,setPost]=useState([])
    useEffect(()=>{
        axios.get('https://jsonplaceholder.typicode.com/posts').then(
            (res)=>{
                console.log(res)
                setPost(res.data)
            }).catch((error)=>{
                console.log(error)
            })
    })
    return(
        <div>
        {
            posts.map(post=><li key={post.id}>{post.id} {post.title}</li>)
        }
        </div>
        )
}
export default Fetching;
```

Individual post :

```
import React,{useState,useEffect} from 'react';
import axios from 'axios';
function Fetching(){
    const [post,setPost]=useState([])
    const [id,setId]=useState(1)
    useEffect(()=>{
        axios.get(`https://jsonplaceholder.typicode.com/posts/${id}`).then(
            (res)=>{
                console.log(res)
                setPost(res.data)
            }).catch((error)=>{
                console.log(error)
            })
    },[id])
    return(
        <div>
        <input type="text" value={id} onChange={(e)=>setId(e.target.value)} />
        <h1>{post.title}</h1>
        {/*<ul>
        {
            posts.map(post=><li key={post.id}>{post.id} {post.title}</li>)
        }
    </ul>*/}
        </div>
        )
}
export default Fetching;
```

Button click :

```
import React,{useState,useEffect} from 'react';
import axios from 'axios';
function Fetching(){
    const [post,setPost]=useState([])
    const [id,setId]=useState(1)
    const [idfrombutton,setbutton]=useState(1)
    useEffect(()=>{
        axios.get(`https://jsonplaceholder.typicode.com/posts/${id}`).then(
            (res)=>{
                console.log(res)
                setPost(res.data)
            }).catch((error)=>{
                console.log(error)
            })
    },[idfrombutton])
    const handle=()=>{
        setbutton(id)
    }
    return(
        <div>
        <input type="text" value={id} onChange={(e)=>setId(e.target.value)} />
        <button type="button" onClick={handle}>Fetch</button>
        <h1>{post.title}</h1>

        </div>
        )
}
export default Fetching;
```

Context :

useReducer :

The useReducer hook in React is used to manage more complex state logic in functional components. It's an alternative to useState and is especially helpful when:

- The state depends on previous state.
- State logic involves multiple actions.
- You want to centralize the logic in a reducer function.

```
import React,{useReducer} from 'react';
const initialstate=0
const reducer=(state,action)=>{
switch(action){
    case `increment`:
        return state+1
    case `decrement`:
         return state-1
    case 'reset':
         return initialstate
    default:
        return state
}
}
function ReducerOne(){
    const [count,dispatch]=useReducer(reducer,initialstate)
    return(
        <div>
        <div>{count}</div>
        <button onClick={()=>dispatch('increment')}>increment</button>
        <button onClick={()=>dispatch('decrement')}>decrement</button>
        <button onClick={()=>dispatch('reset')}>reset</button>
        </div>
        )
}
export default ReducerOne;
```

 reducer: A function that takes the current state and an action, and returns the new state.

 initialState: The initial value of the state.

 state: The current state managed by the reducer.

 dispatch: A function to send actions to the reducer.

When to Use `useReducer` vs. `useState`

Scenario	Recommended Hook
Simple state with few updates	`useState`
Complex state with multiple actions	`useReducer`
State transitions depend on previous state	`useReducer`

```jsx
import React, { useReducer } from 'react';

const initialState = {
  count: 0,
};

const reducer = (state, action) => {
  switch (action.type) {
    case 'increment':
      return { count: state.count + action.value };
    case 'decrement':
      return { count: state.count - action.value };
    case 'reset':
      return initialState;
    default:
      return state;
  }
};

function ReducerOne() {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <div>
      <div>Count: {state.count}</div>
      <button onClick={() => dispatch({ type: 'increment', value: 1 })}>
        Increment
      </button>
      <button onClick={() => dispatch({ type: 'decrement', value: 1 })}>
```

```jsx
  return (
    <div>
      <div>Count: {state.count}</div>
      <button onClick={() => dispatch({ type: 'increment', value: 1 })}>
        Increment
      </button>
      <button onClick={() => dispatch({ type: 'decrement', value: 1 })}>
        Decrement
      </button>
      <button onClick={() => dispatch({ type: 'reset' })}>Reset</button>
      <button onClick={() => dispatch({ type: 'increment', value: 5 })}>
        Increment 5
      </button>
      <button onClick={() => dispatch({ type: 'decrement', value: 5 })}>
        Decrement 5
      </button>
    </div>
  );
}

export default ReducerOne;
```

Two counters :

```jsx
import React, { useReducer } from 'react';

// Initial state
const initialState = {
  counter1: 0,
  counter2: 0,
};

// Reducer function
const reducer = (state, action) => {
  switch (action.type) {
    case 'increment1':
      return { ...state, counter1: state.counter1 + action.value };
    case 'decrement1':
      return { ...state, counter1: state.counter1 - action.value };
    case 'increment2':
      return { ...state, counter2: state.counter2 + action.value };
    case 'decrement2':
      return { ...state, counter2: state.counter2 - action.value };
    case 'reset':
      return initialState;
    default:
      return state;
  }
};

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

```jsx
  return (
    <div>
      <h1>Counter 1: {state.counter1}</h1>
      <button onClick={() => dispatch({ type: 'increment1', value: 1 })}>
        Increment Counter 1
      </button>
      <button onClick={() => dispatch({ type: 'decrement1', value: 1 })}>
        Decrement Counter 1
      </button>

      <h1>Counter 2: {state.counter2}</h1>
      <button onClick={() => dispatch({ type: 'increment2', value: 1 })}>
        Increment Counter 2
      </button>
      <button onClick={() => dispatch({ type: 'decrement2', value: 1 })}>
        Decrement Counter 2
      </button>

      <button onClick={() => dispatch({ type: 'reset' })}>Reset Both Counters</button>
    </div>
  );
}

export default ReducerTwoCounters;
```

useReducer+useContext :

Data fetching using reduce :

```
import React,{useState,useEffect} from 'react'
import axios from 'axios'

function Datafetch(){
    const [loading,setloading]=useState(true)
    const [error,seterror]=useState('')
    const [post,setpost]=useState({})
    useEffect(()=>{
        axios.get('https://jsonplaceholder.typicode.com/posts/1').then(
            (res)=>{
                setloading(false)
                setpost(res.data)
                seterror('')
            }).catch((error)=>{
                setloading(false)
                setpost({})
                seterror('something wrong')
            })
    },[])
    return(
        <div>
        {loading ? 'loading' :post.title}
        {error ? error : null}
        </div>
        )
}
export default Datafetch;
```

Using reduce :

```
import React,{useReducer,useEffect} from 'react'
import axios from 'axios'
const initialstate={
    loading:true,
    error:"",
    post:{}
}
const reducer=(state,action)=>{
    switch(action.type){
        case 'FETCH_SUCCESS':
        return {
        loading:false,
        post:action.payload,
        error:''
    }
    case 'FETCH_ERROR':
    return {
        loading:false,
        post:{},
        error:"something went wrong"
    }
    default :
    return state
    }
}

function Datafetch(){
    const [state,dispatch]=useReducer(reducer,initialstate)
```

```
function Datafetch(){
    const [state,dispatch]=useReducer(reducer,initialstate)
    useEffect(()=>{
        axios.get('https://jsonplaceholder.typicode.com/posts/1').then(
            (res)=>{
                dispatch({type:'FETCH_SUCCESS',payload:res.data})
            }).catch((error)=>{
                dispatch({type:'FETCH_ERROR'})
            })
    },[])
    return(
        <div>
        {state.loading ? 'loading' :state.post.title}
        {state.error ? state.error : null}
        </div>
        )
}
export default Datafetch;
```

useState vs useReducer

Scenario	useState	useReducer
Type of state	Number, String, Boolean	Object or Array
Number of state transitions	One or two	Too many
Related state transitions?	No	Yes
Business logic	No business logic	Complex business logic
Local vs global	Local	Global

Callback hook :

The useCallback hook in React is used to memoize functions, preventing unnecessary re-creation of functions during re-renders. It is particularly useful when passing callbacks to child components to avoid triggering re-renders unless dependencies change.

- useCallback returns a memoized version of the callback function.
- The memoized function only changes if one of its dependencies changes.
- It optimizes performance by preventing unnecessary re-creation of functions passed as props.

```jsx
import React,{useState,useCallback} from 'react'
import Button from './button'
import Count from './count'
import Title from './title'

function ParentCallback(){
    const [age,setage]=useState(25)
    const [salary,setsalary]=useState(50000)
    const incrementage=useCallback(()=>{
        setage(age+1)
    },[age])
    const incrementsalary=useCallback(()=>{
        setsalary(salary+1000)
    },[salary])
    return(
        <div>
        <Title />
        <Count text='age' count={age} />
        <Button handleClick={incrementage}>Incrementage</Button>
        <Count text='salary' count={salary} />
        <Button handleClick={incrementsalary}>Incrementsalary</Button>
        </div>
        )
}
export default ParentCallback;
```

```jsx
import React from 'react';
function Count({text,count}){
    console.log(`${text} rendering`)
    return(
        <div>{text}-{count}</div>
        )
}
export default Count;
```

```jsx
import React from 'react';
function Title(){
    console.log("title rendering")
    return(
        <h2>
        use callback hook
        </h2>
        )
}
export default React.memo(Title);
```

```jsx
import React from 'react'

function Button({handleClick,children}){
    console.log('Rendering button -',children)
    return(
        <button onClick={handleClick}>{children}</button>
        )
}
export default Button;
```

Use memo :

The React.memo hook is a higher-order component used to optimize React functional components. It prevents unnecessary re-renders of a component by memoizing its rendered output and re-rendering it only if its props change.

In React, when a parent component re-renders, its child components also re-render by default. This can lead to performance issues if the child components perform heavy calculations or render large DOM trees unnecessarily.

React.memo solves this problem by memoizing the rendered output of a functional component. It compares the previous props with the new props and re-renders the component only if the props have changed.

```
function Memo(){
    const [countone,setcountone]=useState(0)
    const [counttwo,setcounttwo]=useState(0)
    const incrementone=()=>{
        setcountone(countone+1)
    }
    const incrementtwo=()=>{
        setcounttwo(counttwo+1)
    }
    const isEven=useMemo(()=>{
        let i=0
        while(i!=10000000){
            i+=1
        }
        return countone%2==0
    },[countone])
    return(
        <div>
        <h1>{countone}</h1>
        <button onClick={incrementone}>incrementone</button>
        <span>{isEven ? 'even' :'odd'}</span>
        <h2>{counttwo}</h2>
        <button onClick={incrementtwo}>incrementtwo</button>
        </div>
        )
}
export default React.memo(Memo);
```

Use Ref :

The useRef hook in React is used to persist a value across renders without causing a re-render when the value changes. It is often used for:

1. **Accessing DOM Elements**: To directly manipulate or reference a DOM element.
2. **Storing Mutable Values**: To store values that don't need to trigger a re-render when updated.
3. **Keeping a Reference**: For timers, event handlers, or any mutable object.