useState

you can not put hooks inside

* If statement
* In loop
* In the function (but not the Component function itself)

They have to be at the top level and all the useState will be called in exact order they are.

useState returns an Array of value. With 2 values.

1st is State. Which is the current value

2nd is current which is going to update the current state.

import { useState } from "react";

function App() {

  const [value,setValue]=useState(4);

  const decrementValue =()=>{

    setValue(value-1);

  }

  return (

    <center>

    <button *onClick*={decrementValue}>-</button>

    <span>{value}</span>

    <button>+</button>

    </center>

  );

}

export default App;

now if we update the state by clicking minus button then the component going to rerender. ☺

NOW, if we see that we are putting default value in our state as 4. And it is going run everytime the state is going to run. So if we click + or – button then everytime its going to run .as we can console long it in the code and can see it. But just imagine having big computation as a default value rather than just 4. Then its very tough and will slow my application. So to run the default value initialization once we can write in function way. Following.

import { useState } from "react";

function App() {

  const [value,setValue]=useState(()=>{return 4});

  const decrementValue =()=>{

    setValue((*prevValue*)=>*prevValue*-1);

  }

    const incrementValue =()=>{

    setValue((*prevValue*)=>*prevValue*+1);

  }

  return (

    <center>

    <button *onClick*={decrementValue}>-</button>

    <span>{value}</span>

    <button *onClick*={incrementValue}>+</button>

    </center>

  );

}

export default App;

so it’s a good way to initialize state in function way style.

But now if we have an object as a default value in usetate with a 2 key:value pair

import { useState } from "react";

function App() {

  const [state,setState]=useState({count:4,theme:'blue'});

  const count =state.count;

  const theme= state.theme

  const decrementValue =()=>{

    setState((*prevState*)=> {return {count: *prevState*.count-1}});

  }

    const incrementValue =()=>{

    setState((*prevState*)=> {return {count: *prevState*.count+1}});

  }

  return (

    <center>

    <button *onClick*={decrementValue}>-</button>

    <span>{count}</span>

    <span>{theme}</span>

    <button *onClick*={incrementValue}>+</button>

    </center>

  );

}

export default App;

if we run this code and + - then the theme completely disappear. Whye ? cause useState overrite everything all over if we do change any state. So if we want ot keep the theme as we change the value alongside. Then we have to spread operate the prevState and then keep the same code.

import { useState } from "react";

function App() {

  const [state,setState]=useState({count:4,theme:'blue'});

  const count =state.count;

  const theme= state.theme

  const decrementValue =()=>{

    setState((*prevState*)=> {return {...*prevState*,count: *prevState*.count-1}});

  }

    const incrementValue =()=>{

    setState((*prevState*)=> {return {...*prevState*,count: *prevState*.count+1}});

  }

  return (

    <center>

    <button *onClick*={decrementValue}>-</button>

    <span>{count}</span>

    <span>{theme}</span>

    <button *onClick*={incrementValue}>+</button>

    </center>

  );

}

export default App;

this is code will run according what we want. This is keep the theme even if we change the value. Why cause we are merging new state with the previous one by the help of spread operator.

But rather than changing sthe state in class component style we just can have 2 hooks for counter and theme and do functionality accordingly

import { useState } from "react";

function App() {

  const [value,setValue]=useState(()=>{return 4});

  const [theme,setTheme]=useState(()=>{return 'red'})

  const decrementValue =()=>{

    setValue((*prevValue*)=>*prevValue*-1);

    setTheme((*prevValue*)=>'black')

  }

    const incrementValue =()=>{

    setValue((*prevValue*)=>*prevValue*+1);

    setTheme((*prevValue*)=>'blue')

  }

  return (

    <center>

    <button *onClick*={decrementValue}>-</button>

    <span>{value}</span>

    <span>{theme}</span>

    <button *onClick*={incrementValue}>+</button>

    </center>

  );

}

export default App;

SIMPLe.

UseEFFECT

Whenever we change any resource of the component there is a effect. In class component we use lifecycle method but in fucntionald component we have useEffect hooks. Which is actually a function which going to work as soon as any resource has been changed. This useEffect hook take some parameter which is dependency as if that particular dependency changed then the function works other than that it does not work.

import React from 'react'

import { useState, useEffect } from 'react'

function MyUseEffectHook() {

  const [resourceType, setResourceType]= useState('post')

  useEffect(()=>{

    console.log('resource has been changed')

  },[resourceType]);

  return (

    <>

    <button *onClick*={()=>{setResourceType('posts')}}>Posts</button>

    <button *onClick*={()=>{setResourceType('User')}}>Users</button>

    <button *onClick*={()=>{setResourceType('comments')}}>comments</button>

    <h1>{resourceType}</h1>

    </>

  )

}

export default MyUseEffectHook

now if we change the component by clicking in buttons useEffect will work but if we change the same resourceType then it will not work, as we have set the dependency of the useEffect hook that if the resourceType changes then you execute the function. That’s it.

Now if we keep an empty array rather than putting resourcetype there. Then the useEffect going to render once and then DONE that means onMount, means when the component first renders.. If we even change any resource it will not rerender again.

import React from 'react'

import { useState, useEffect } from 'react'

function MyUseEffectHook() {

  const [resourceType, setResourceType]= useState('post');

  const [windowWidth, setWindowWidth] = useState(window.innerWidth)

  const handleResize=()=>{

    setWindowWidth(window.innerWidth)

  }

  useEffect(()=>{

    window.addEventListener('resize', handleResize)

  },[])

  useEffect(()=>{

    console.log('resource has been changed')

  },[resourceType]);

  return (

    <>

    <button *onClick*={()=>{setResourceType('posts')}}>Posts</button>

    <button *onClick*={()=>{setResourceType('User')}}>Users</button>

    <button *onClick*={()=>{setResourceType('comments')}}>comments</button>

    <h1>{resourceType}</h1>

    <h1>{windowWidth}</h1>

    </>

  )

}

export default MyUseEffectHook

we are taking another useState which we setting default value as current window width. And useeffect is there which will mount only once the component render the first time. As we are putting empty array. This useEffect will add event listener of resize and call a function handleWidth. Which is going to be called everytime user change the size of the widnow that function will change the resource and will dyncamilcally show in the component. REMEMBER useEffect is working everytime we changing the width but the function of useEffect is going to render once as we put an empty array. SIMPLE.

BUT,

As we can see that we are only mounting the addeventlisner, its not been unmounting. Which is going to cost some speed in big application. So to do unmount everytime we mount , useEffect has a return function inside useEffect which we can use for our purpose. This works as a cleanup code.

So remember, that return function of the useEffect runs first as it return and remove whatever has been mounted and then call the function that we want to be excute

UseMEMO

import { useState } from "react";

export default function MyUseMemo(){

  const [number,setNumber]=useState(0);

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

  const doubleNumber = slowFunction(number);

  const themeStyles ={

    backgroundColor: dark ? 'black':'white',

    color: dark? 'white':'black'

  }

  return(

    <>

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

    <button *onClick*={()=>setDark(*prevDark*=>!*prevDark*)}>Change Theme</button>

    <div *style*={themeStyles}>{doubleNumber}</div>

    </>

  )

}

function slowFunction(*value*){

  console.log('calling your number wait')

  for(let i=0;i<1000000000;i++){}

  return *value*\*2;

}

This code is self explanatory and we can see that by changing the number it calling slowFucntion which is going to make this code execution process slow and then changing the theme if clicking on change theme button. So there are 2 fucntionality in this component changing number clicking button. But when we change the number that means slowfucntion is going to call and make the process slow but why changing theme takes time to change. ???? it should not be like that.

Cause,

Changing anything will re render the whole component. That means component will start from the beginning and that means again the slow function will get called and then it will do our change theme function.

This problem can be solve by useMemo. Memo is just memoization which is caching memory single time. Following code is just the upper part of the code. Return part is not here as not neccesarry.

export default function MyUseMemo(){

  const [number,setNumber]=useState(0);

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

  const doubleNumber = useMemo(()=>{

    return slowFunction(number)

  },[number]);

  const themeStyles ={

    backgroundColor: dark ? 'black':'white',

    color: dark? 'white':'black'

  }

Now we can see that useMemo is there which is going to be with an arrow function which return the slowFucntion number at the same time it has a dependency of our ‘number’ that means that whenever number changes, then rerender the component, but if only theme is being changed you don’t need to reRUN the slowfucntion, as number has not change and we are caching the number by use memo now.

So its useful, but don’t use it everywhere cause its going to be taking a memory itself to save the previous value. ;)

Even if we use useEffect which has kinda similar activity that if the dependency changes function get executed. But if we do the above code and useEffect for themestyle changes, if we change the theme then the useEffect gets called. That’s okay. But if we change the number that time also useEffect gets called. As the refferece value has been changes even though they have the exact same value as previous. So to stop that, we can wrap the theme changing by useMEMO then change the number. UseEffec will not get called. As now we are saving the reference with the help of useMemo. ☺

UseREF:

import { useState, useRef, useEffect} from "react"

export default function MyUseRef(){

  const [name,setName]=useState('')

  const renderCount = useRef(0);

  useEffect(()=>{

    renderCount.current = renderCount.current+1

  })

  return(

    <>

    <input *value*={name} *onChange*={(*e*)=>setName(*e*.target.value)}></input>

    <div>My Name is {name}</div>

    <div>Component rendered {renderCount.current} times</div>

    </>

  )

}

Useref give us the reference of the object. As above code we can see that renderCount is a variable. But deep donw it’s an object with 1 property which is current. So if we want to keep the record of the render count we could have just used useState. But why we did not ?

Cause if we change the counter in useState, remember whenever we change the state value the whole component render and the start from the beginning, so if the state has been changed that measn the value has been changed and it will again rerender … and again that process. So it’s an INFINITE LOOP.

That’s we we are using useEffect which we can use as many time we want but it will never cause the component to re render cause we are saving our reference in useREF.

So work of useState and useRef kinda similar but slight different as it will never cause u to rerender.

Lets take another example.

import { useState, useRef, useEffect} from "react"

export default function MyUseRef(){

  const [name,setName]=useState('')

  const prevName = useRef('');

  useEffect(()=>{

    prevName.current=name

  },[name])

  return(

    <>

    <input *value*={name} *onChange*={(*e*)=>setName(*e*.target.value)}></input>

    <div>Now My Name is {name} and it was {prevName.current}</div>

    </>

  )

}

Above code we can see that we are saving our current name as a prev name, and everytime we change the name it just save the previous name as prevName.current. so if we change form ‘jan’ the prevName would be ‘ja’.

This is what useRef does. If we had used useState, then we would cause the component re render again. But by using useRef, we are not cause additional rerender which we actually don’t need.