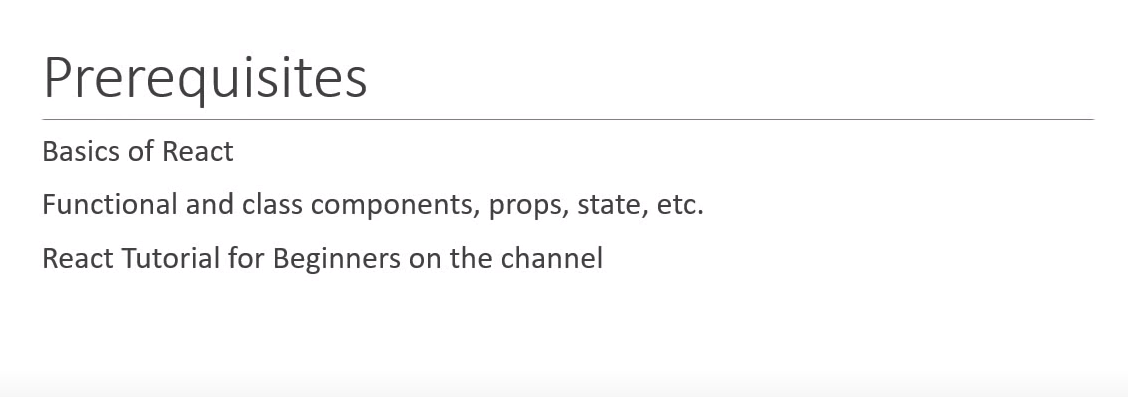
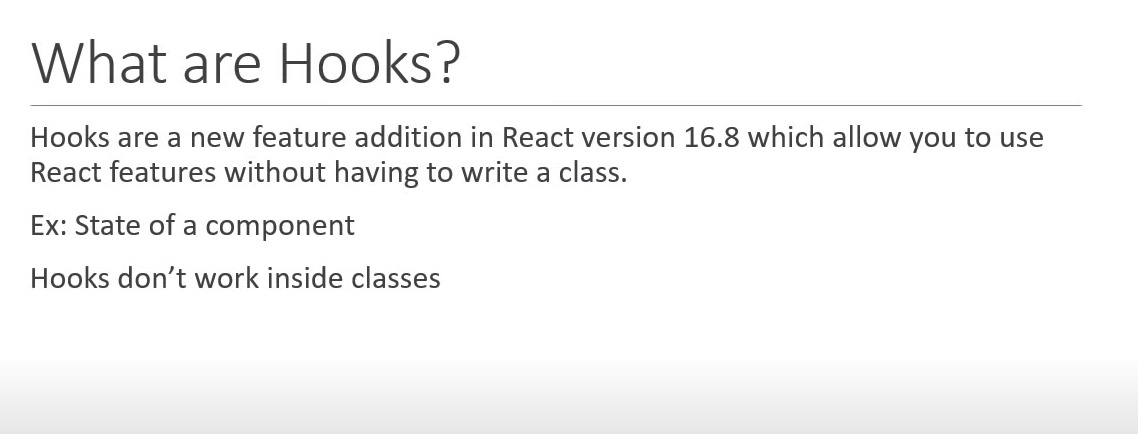
# React Hooks:

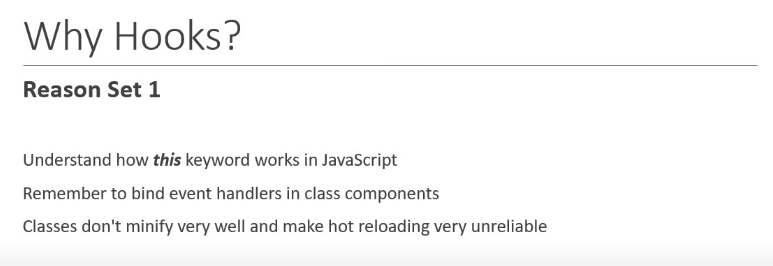


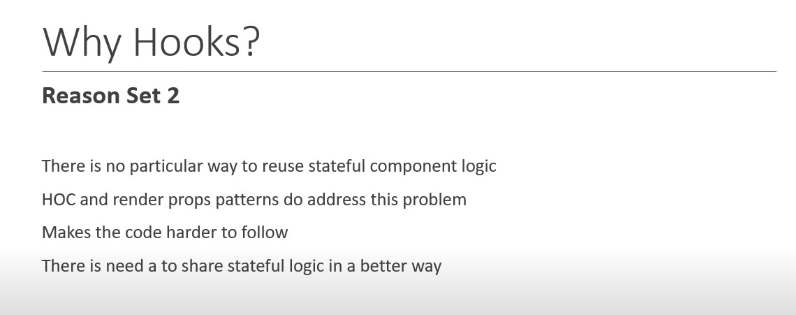
# What are Hooks?

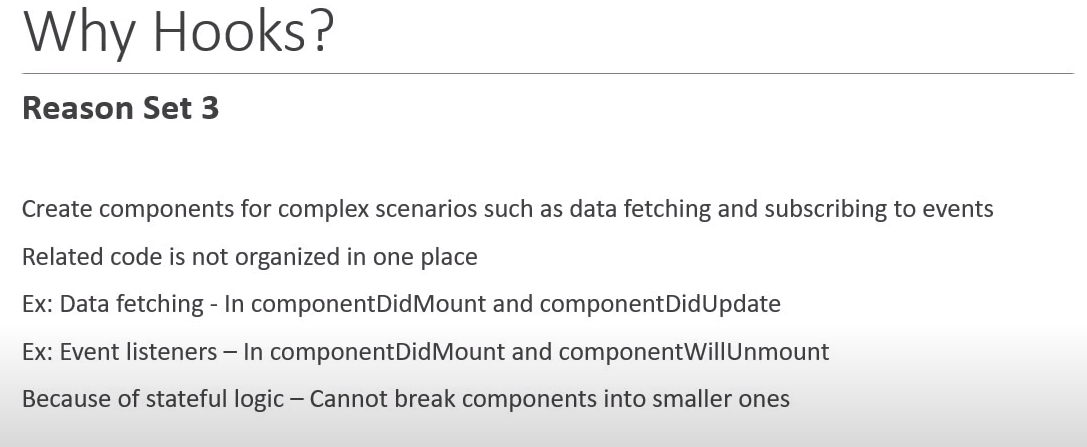
It’s a new feature addition in react. With hooks it is possible to use state and other react features without writing a class i.e., in functional component. Hooks don’t work inside classes.

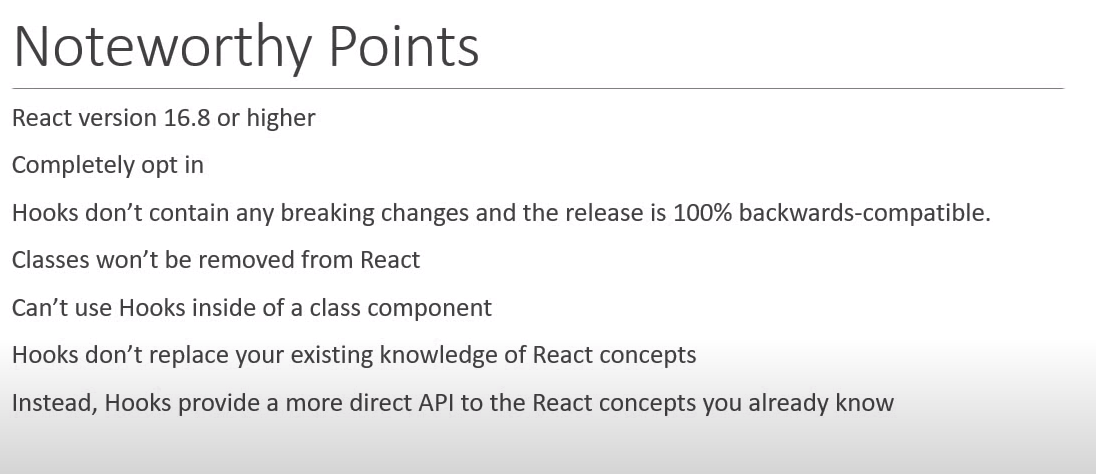


# Why Hooks?











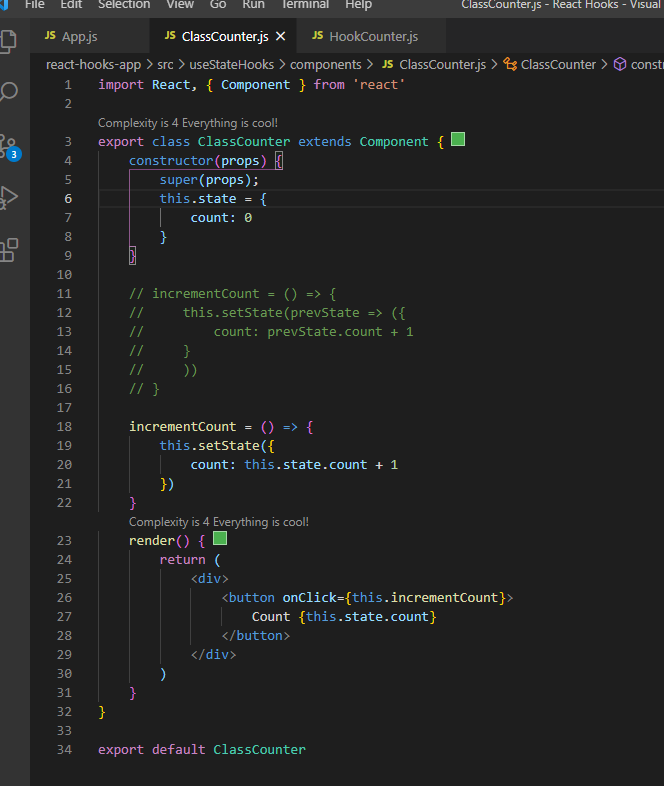
# useState Hook:

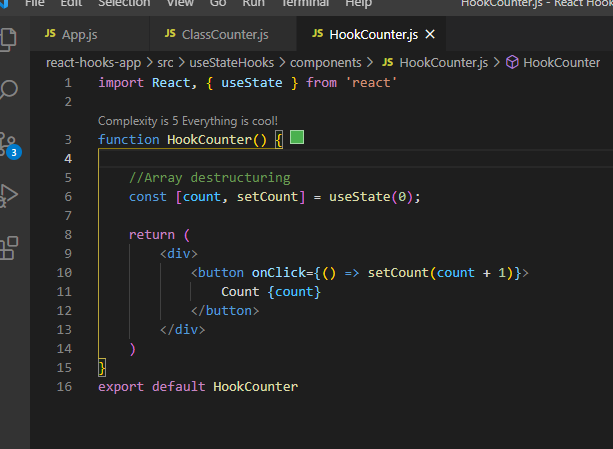
useState hooks allow us to use state within functional component.

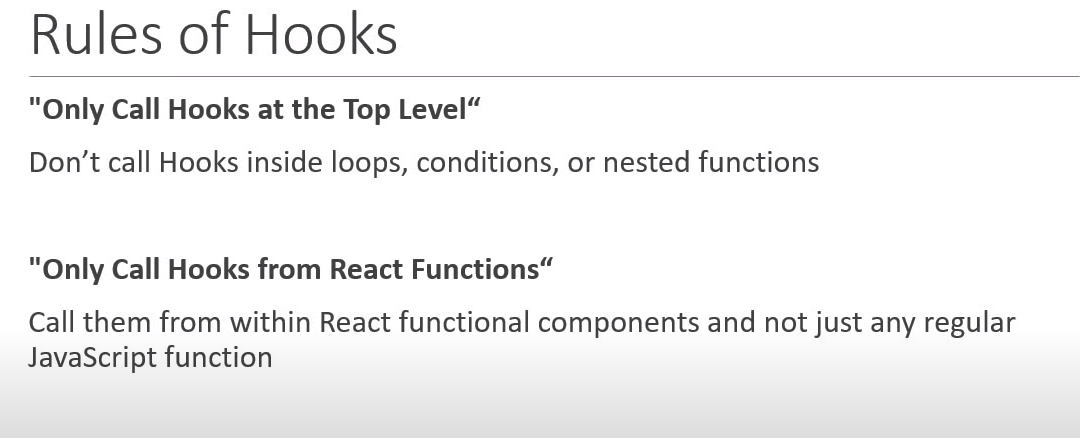
As we know hook is a special function that let us hook into react feature.

useState hook let us add react state into functional component.

useState hook function takes an argument which is the initial value of the state property and returns the current value of the state property and the method which is capable of updating that state property.

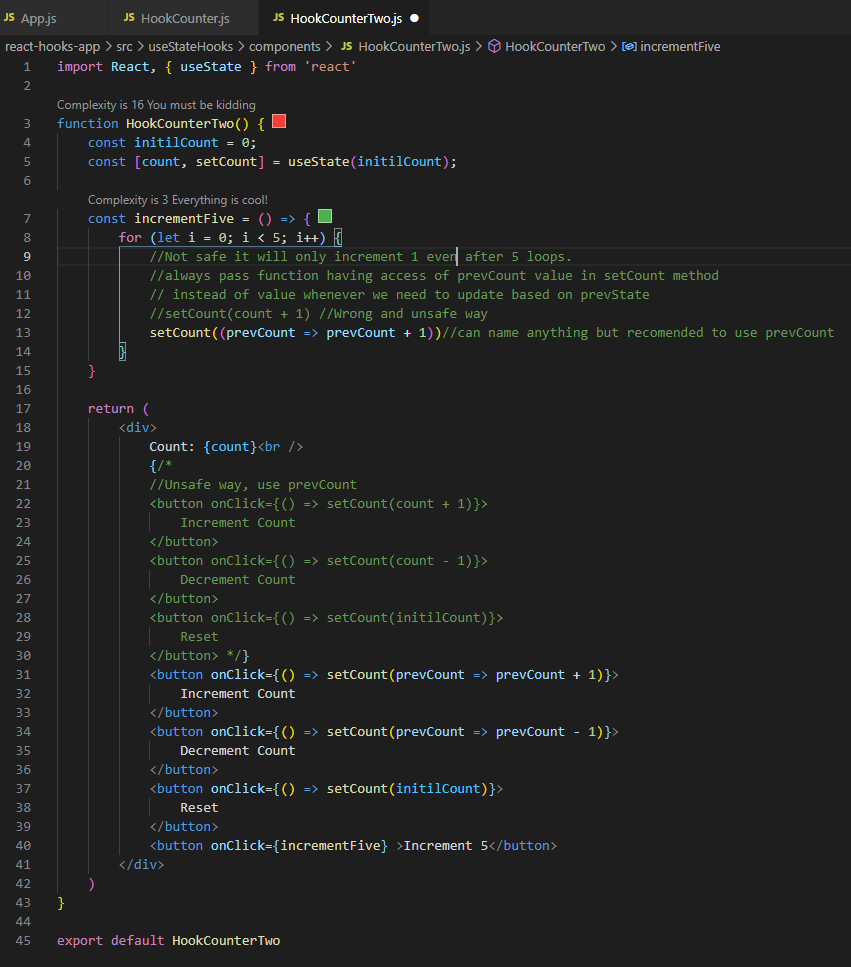






## useState with previous state:

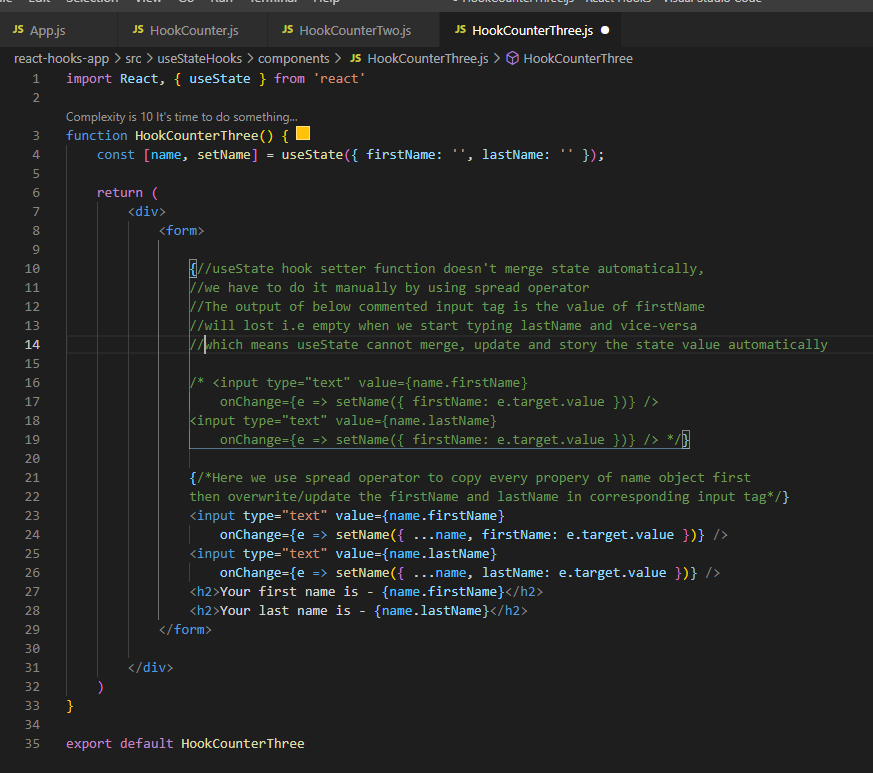
It is not safe to use count state property to update the value of count. Whenever we need to update the state value based on previous state value always pass the function instead of value in setCount which will have the access of prevCount value.



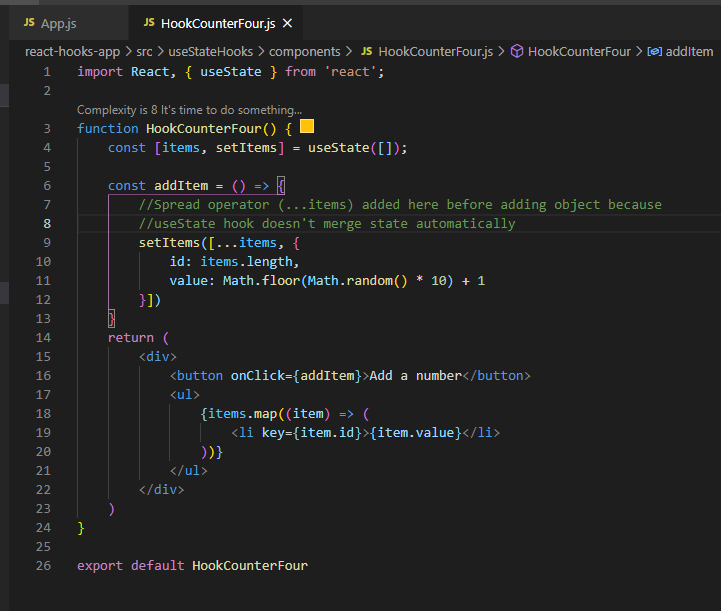
## UseState with Object:

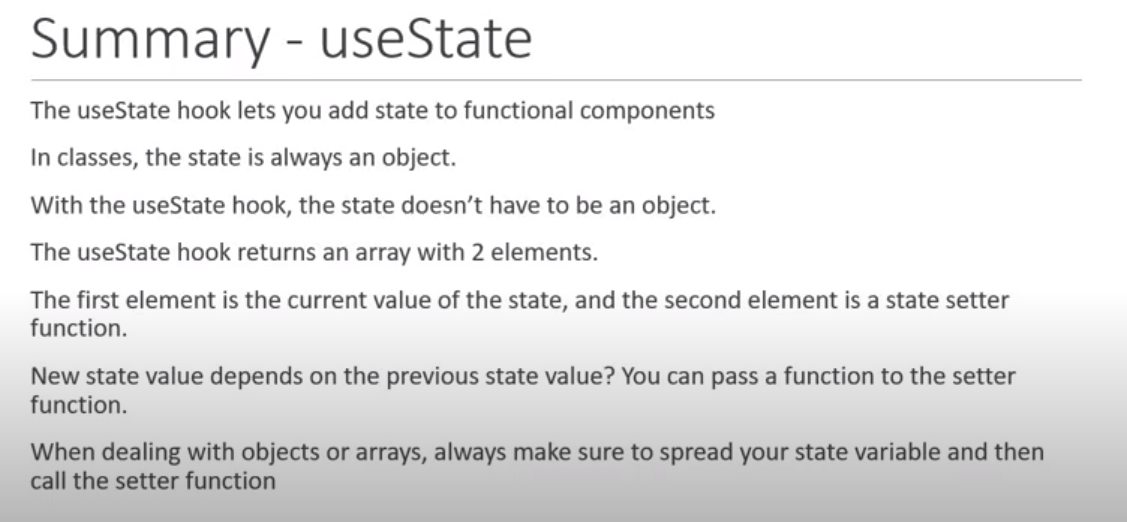
State variable can be anything like string, number, Boolean, object, array etc.

Note: useState does not automatically merge and update the object which is the key difference to setState which we come across to class component. setState in class component merge the state useState hooks setter function will not merge the state, we have to do it manually by using spread operator (… operator).



## useState with Array:





# useEffect Hook:

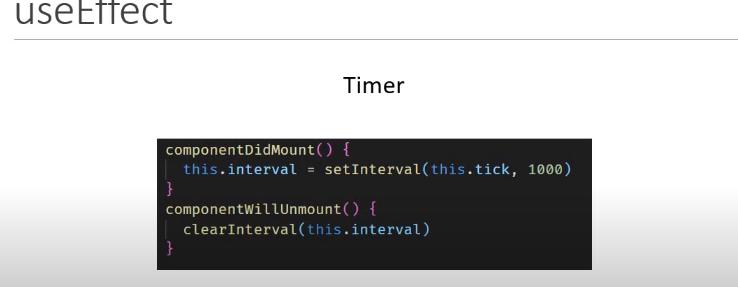
While working with class component we have performed side effects in our components example fetching data from api endpoint, updating DOM, setting up subscriptions or timers.

Render method is too early to perform side effects so we had to make use of lifecycle methods like componentDidMount, componentDidUpdate etc.

In below screenshot we can see that we have to write two lifecycle method, first to render first time and second to render whenever some update performed. Duplicate piece of code we have to write in two lifecycle method.

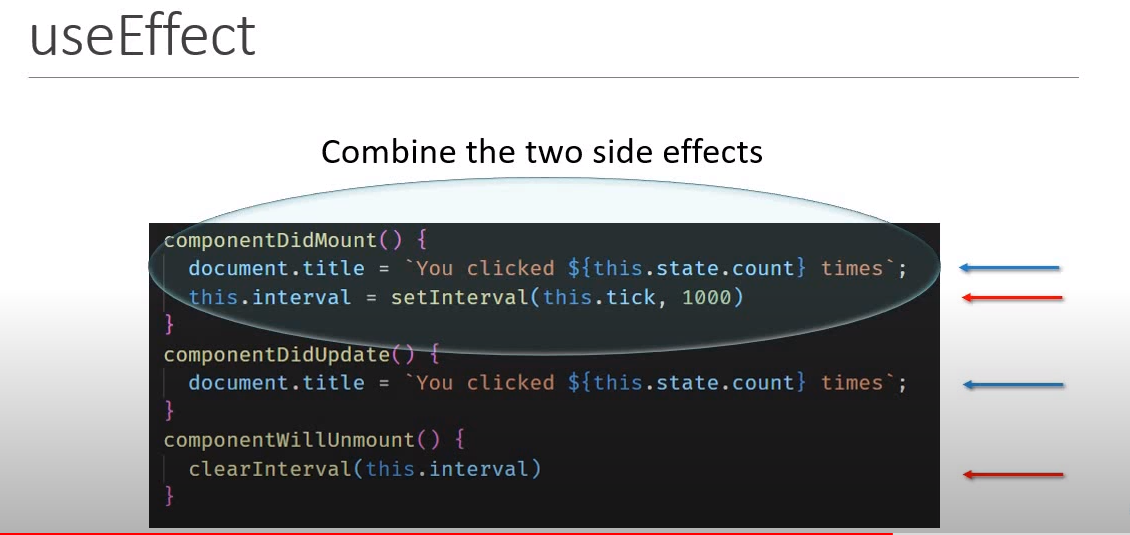


Another example of side-effect is in below screenshot componentDidMount we set the timer and in another lifecycle method we have to unmount it.

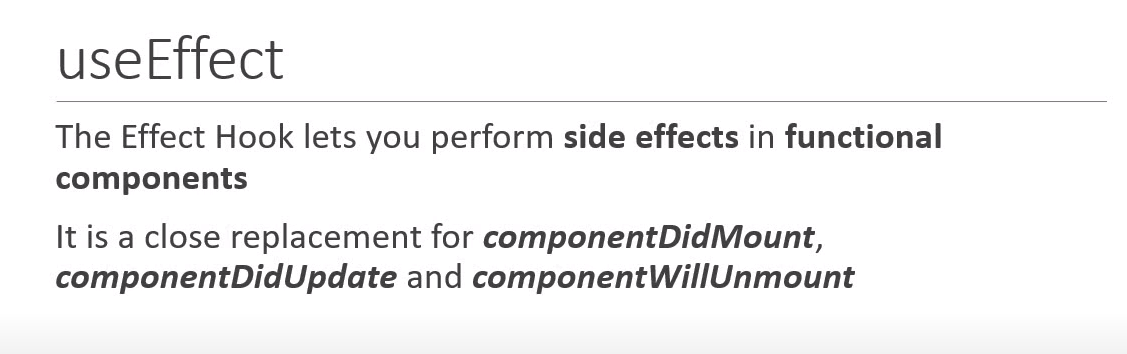


When combined above side-effect. Although the code works fine but need to find if there is any other way to make it simpler.

Target is to don’t repeat the code and don’t put the unrelated code together like in componentDidMount title update and setInterval code are together.

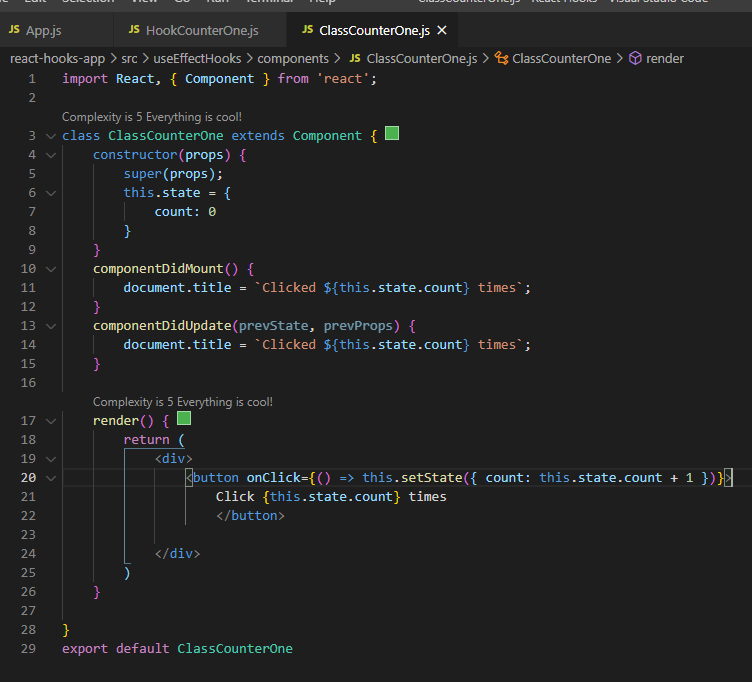


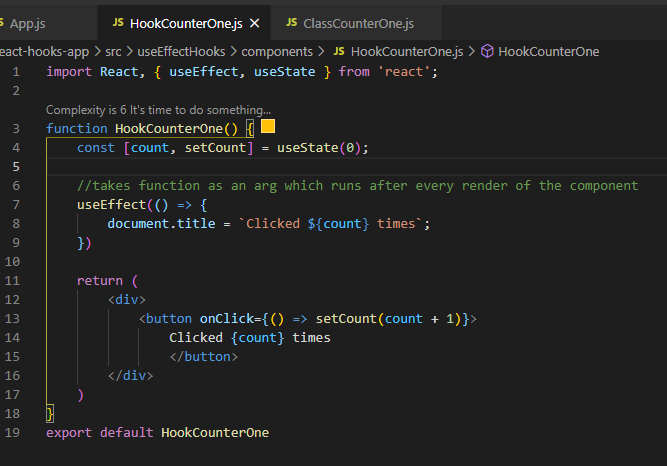
Solution is useEffect hook.



## useEffect after render:

Like useState hook useEffect hook is also a function which takes function as an argument which gets executed after every render of the component.

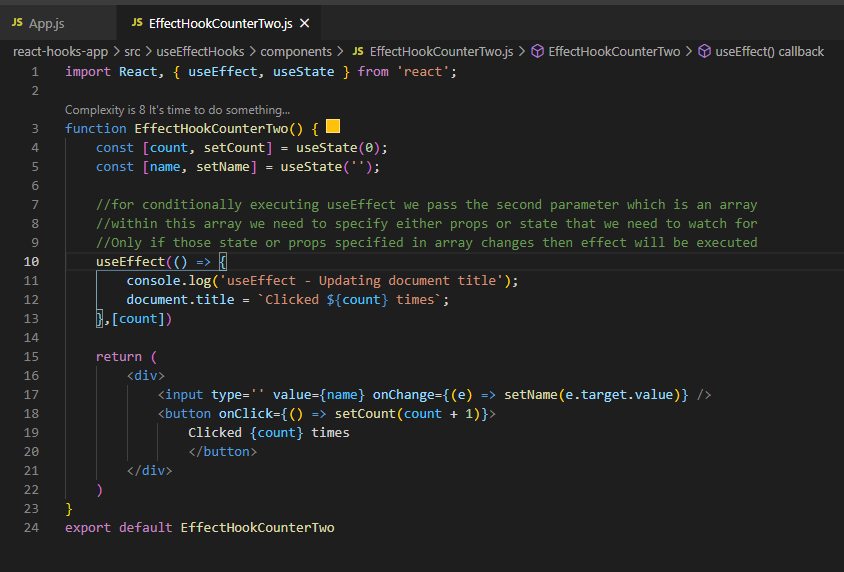




## Conditionally run effects:

As we learnt useEffect runs at every time when components render, which might bring performance issue and we may get some scenario where we don’t want to run useEffect. To do this we have to run useEffect conditionally.

For conditionally executing useEffect we pass the second parameter in useEffect hook which is an array. Within this array we need to specify either props or state that we need to watch for. Only if those state or props specified in array changes then effect will be executed.

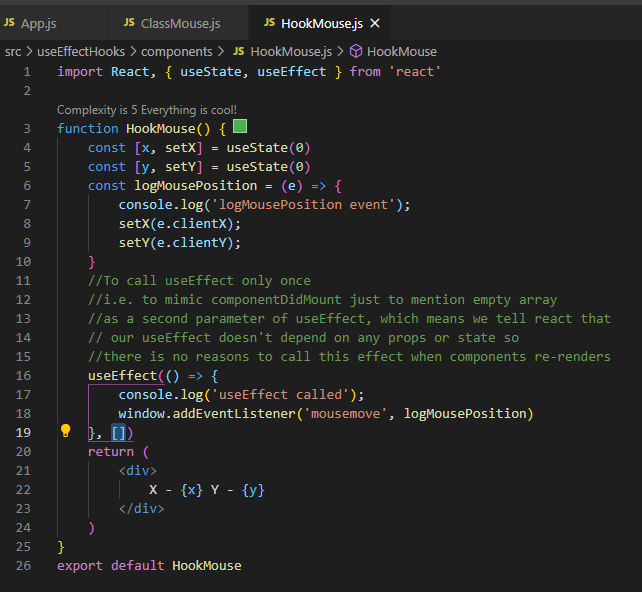


## Run useEffect only once:

To run useEffect only once or can say to run only componentDidMount in functional component.

To call useEffect only once i.e., to mimic componentDidMount, just to mention empty array as a second parameter of useEffect. which means we tell to react that our useEffect doesn't depend on any props or state so there is no reasons to call this effect when components re-renders.



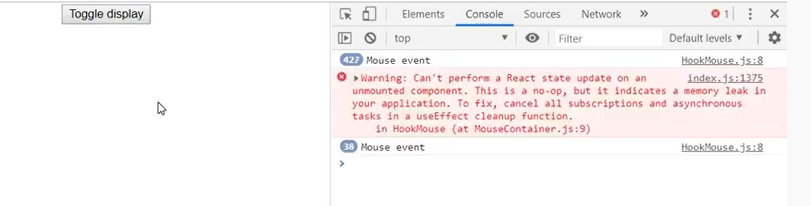


## useEffect with cleanup:

Call componentWillUnmount using useEffect.

As we unmount the component we should clean or remove all the memory taken by that component. When we unmount component but doesn’t clean memory that is called memory leak and react will give the warning of memory leak.

In below example component has been removed but still the event listener belongs to that component is listening an event and log is getting printed which is memory leak.

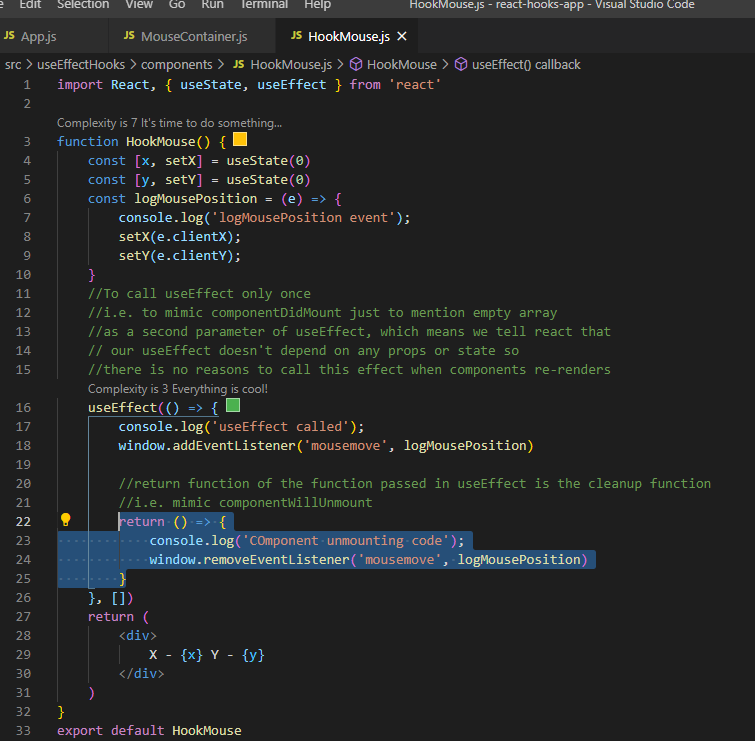


When we un-mount the component make sure we cancel all subscription and listener. In other words cleanup after the component unmount.

In class component we do cleanup listener by using lifecycle method componentWillUnmount and remove listener by removeEventListner method of window class.



To implement the same kind of functionality in function component we use the different version of useEffect hook. **The function passed to useEffect can return a function which will be executed when the component will unmount**.

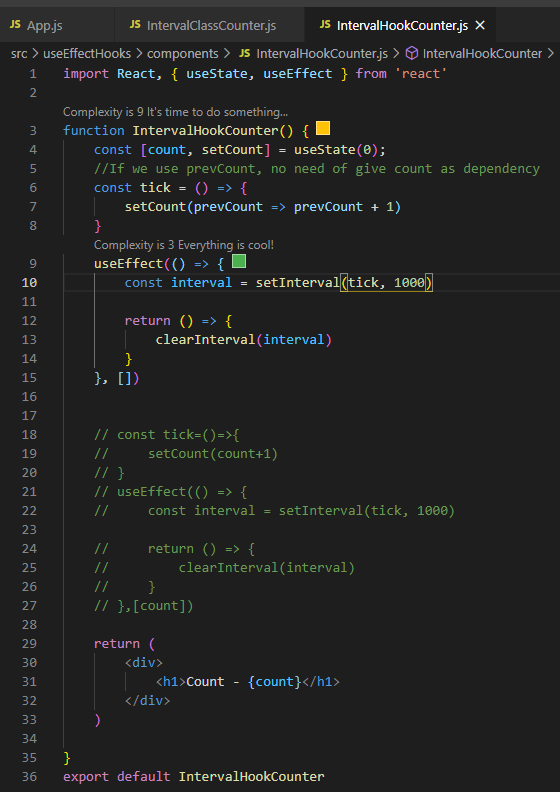


## useEffect with incorrect dependency:

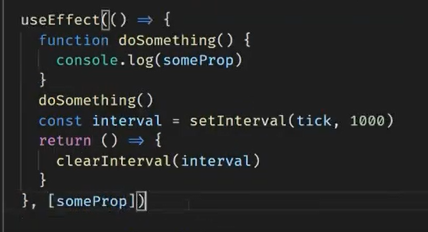
If we think dependency array is a way to specify when we want to re-render the effect then we will be going to run into problem, instead dependency array should be thought of as a way to let react know about the everything that the effect must watch for changes.

It’s a common mistake to leave out the dependency of useEffect.

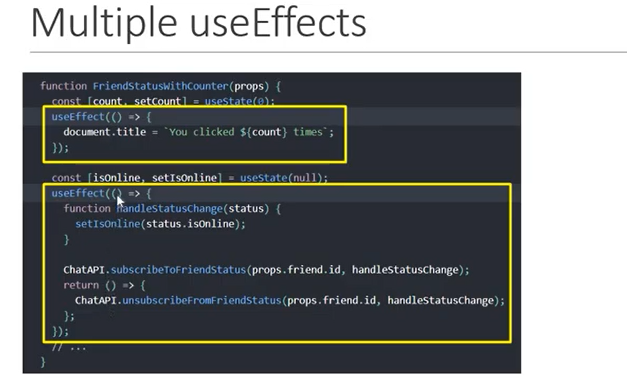
****



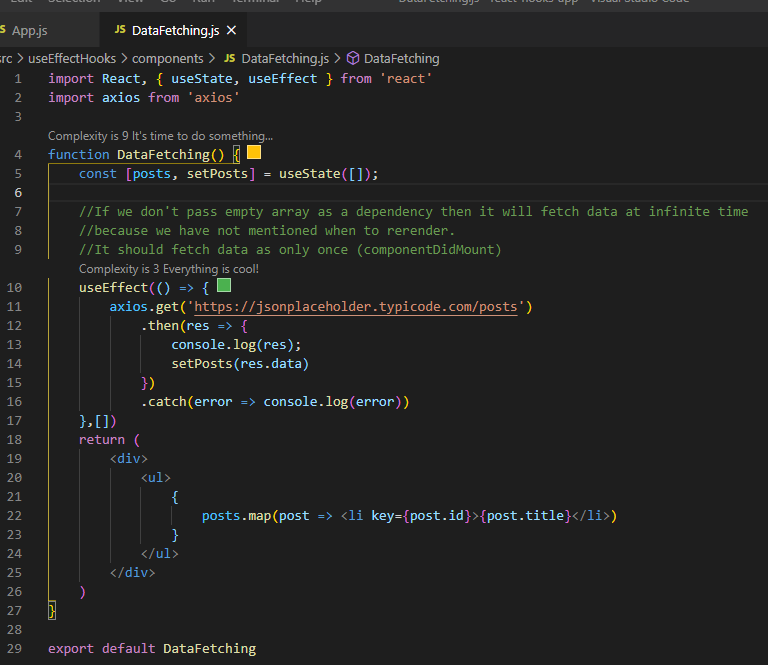
Recommendation to not miss dependency: If we a scenario to call a function from useEffect it’s common to miss the dependency which is necessary by function so it’s better to define the function inside useEffect instead of calling it.



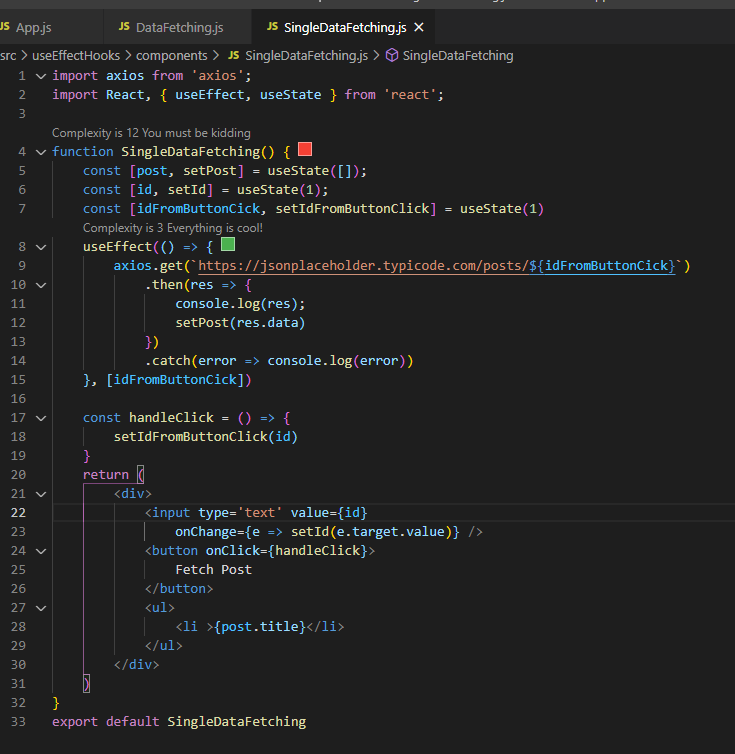
Use multiple useEffect if we have unrelated code rather than keeping everything in a single useEffect.



## Fetching data with useEffect:



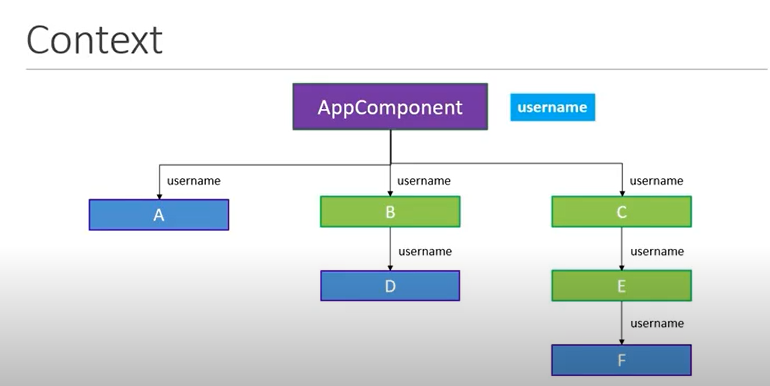
## Fetch single post:

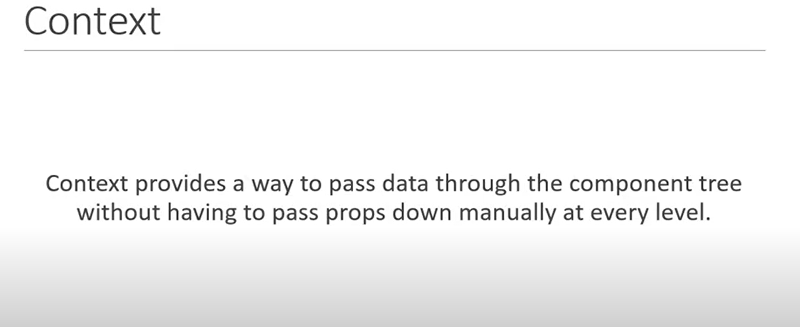


# Context:

Consider a scenario to log the username by component A, D and F but the username information is maintained by App Component. To log the username by component A, D and F we have to pass it to component B, C and E as a prop which doesn’t need this. This is the place where context comes into picture.

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





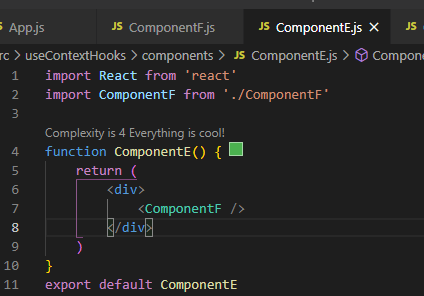
## Steps to create context in class component:

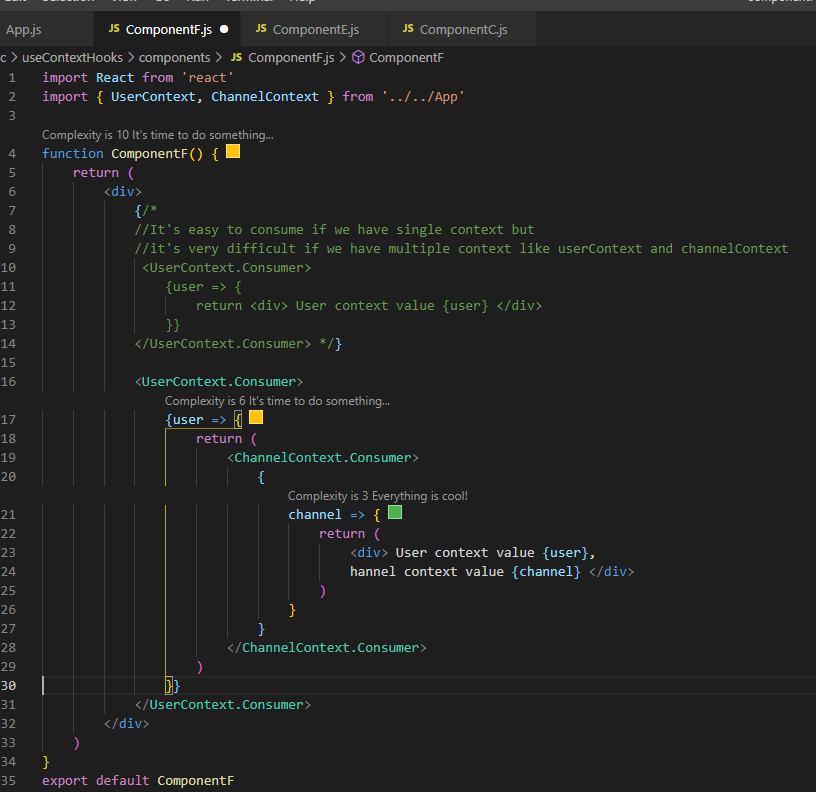
1. create a context using createContext method of React.

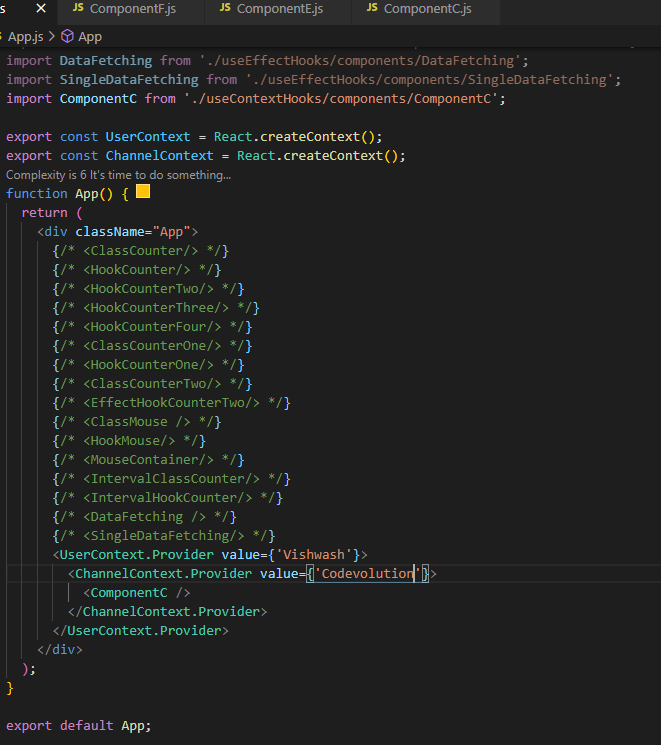
2. wrap the children component with provider so that context value will be available.

3. consume the context value.









# useContext Hook:

Creating a context and providing the context will be same but useContext hook only provide the easiest and the simpler way to consume it.

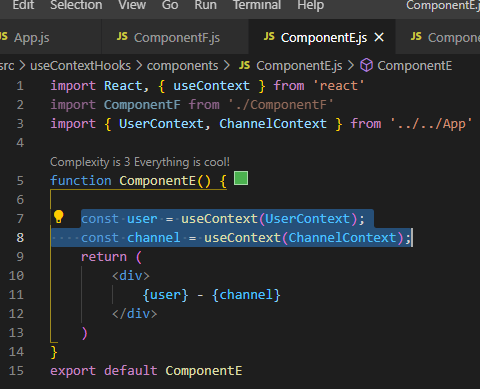
Steps to consume:

1. import useContext from react.

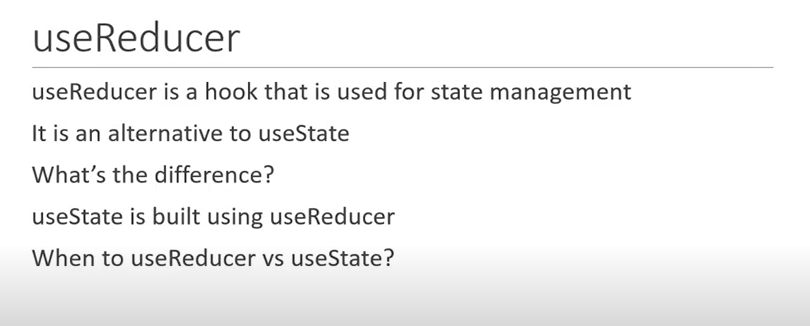
2. import the necessary context.

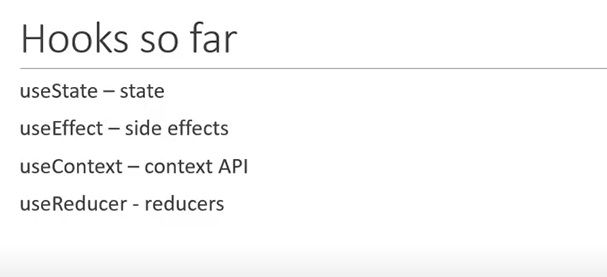
3. call useContext function passing the context as an argument(which have imported in 2nd step).

4. useContext returns a context value passing in context provider.



# useReducer:-





## What is reduce?

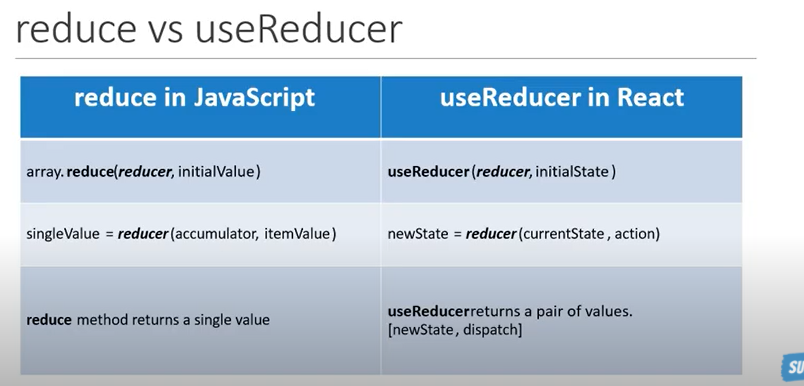
Reduce is a build-in function of Array in Vanilla JavaScript(plain JavaScript).

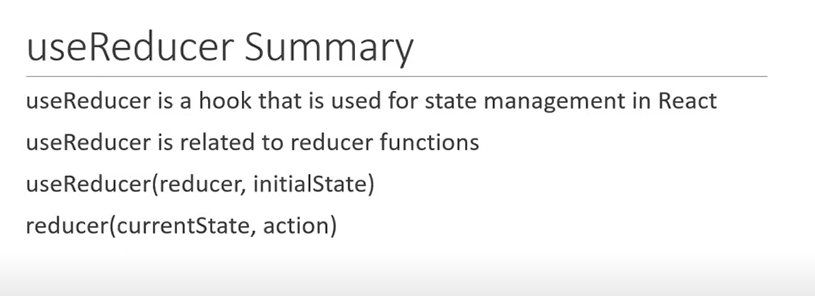
Reduce method of Array takes two parameter 1st is the reducer function and 2nd is the initial value which is optional and returns a single output.

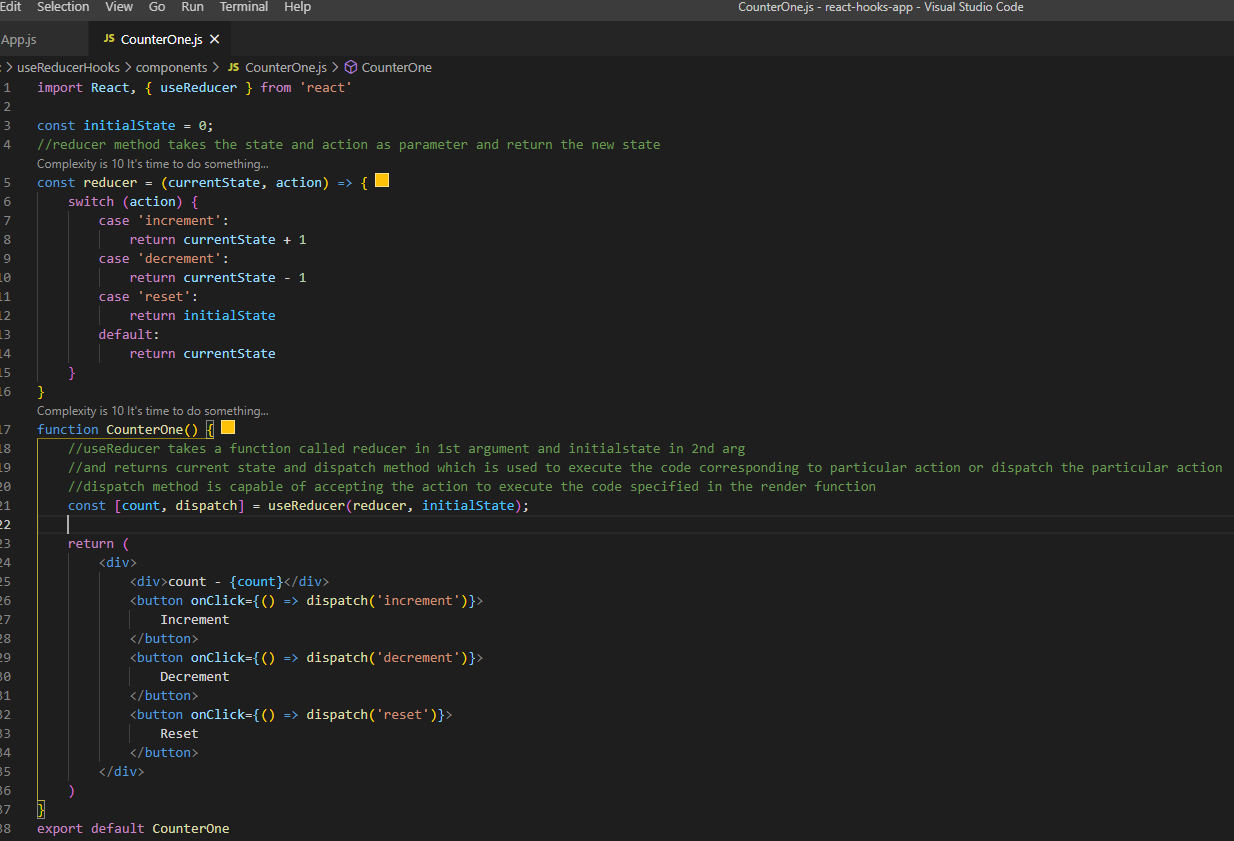
Reducer function also takes 4 parameter:

1. Accumulator
2. Current Value
3. Current Index(optional)
4. Source Array(optional)

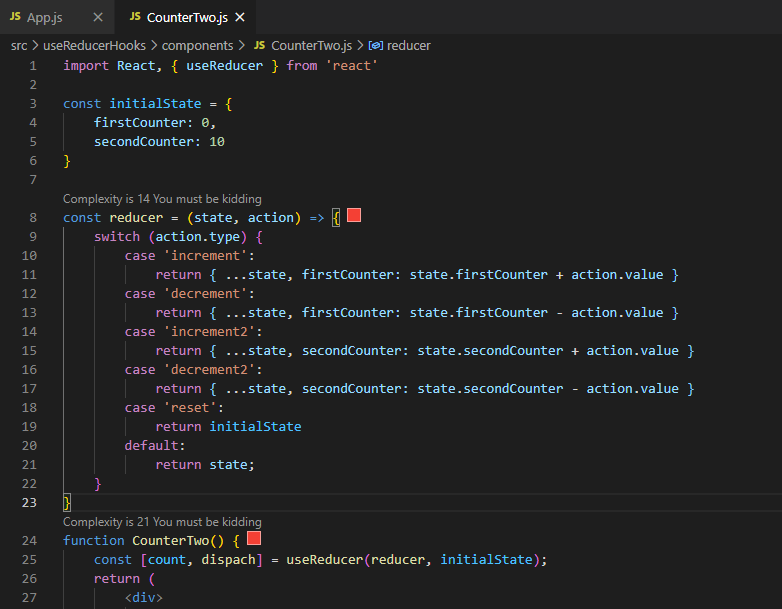
<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce>

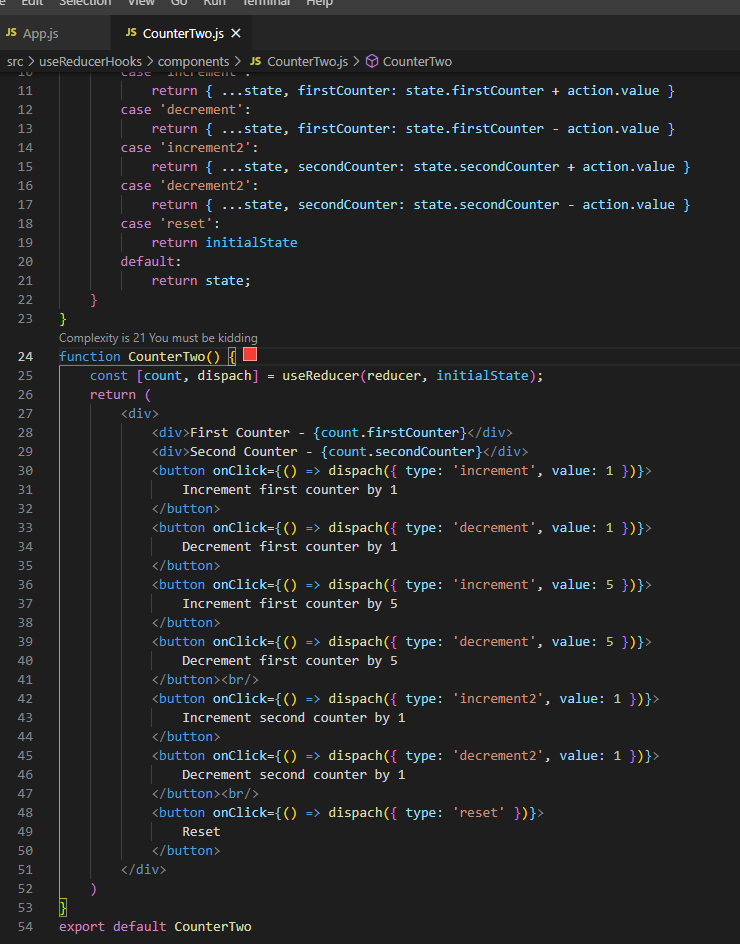


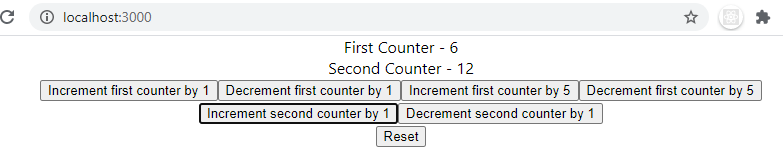




We can make state and action as an object.

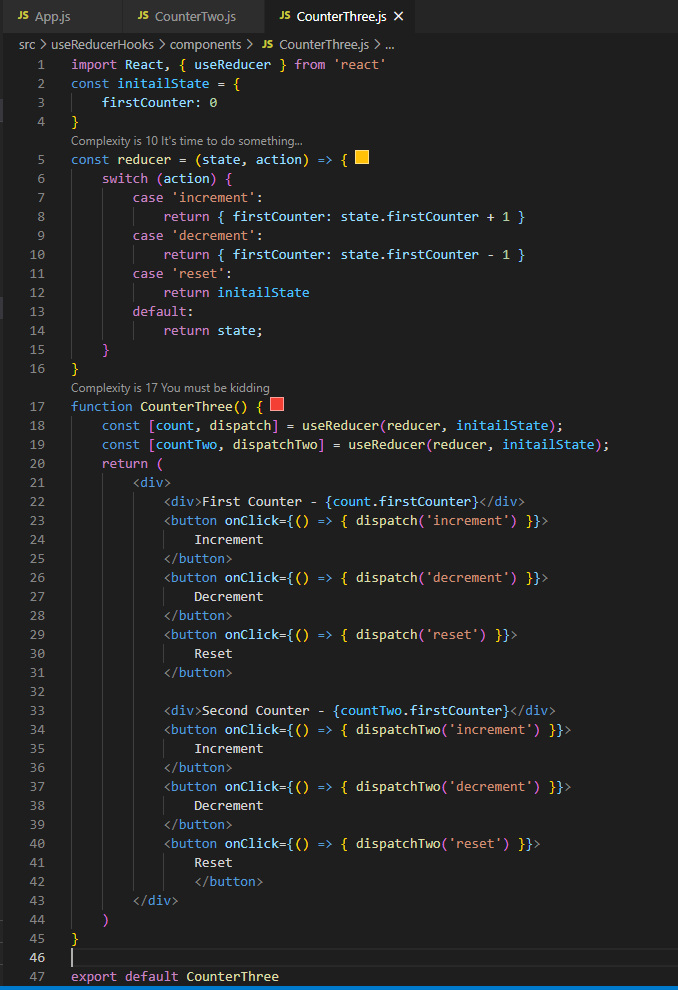




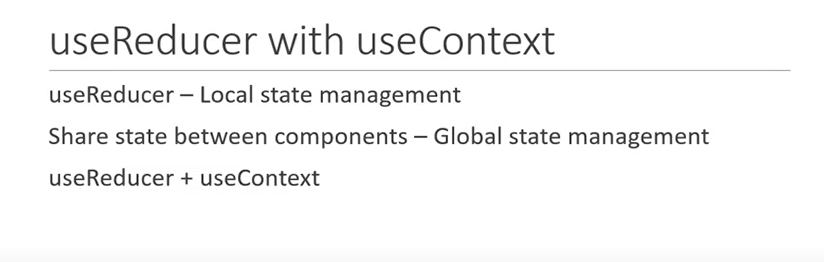


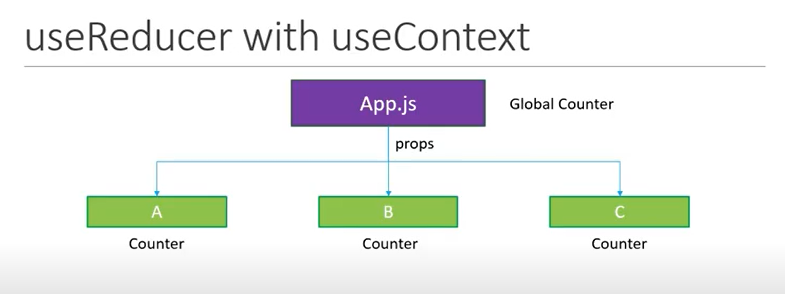
## Simplify previous example by using multiple useReducer.

And each reducer will have its own state and will not change the other state. While dealing with multiple state variable have the same state transition then it is a good idea to have multiple useReducer making use of same reducer function.



# useReducer with useContext:

****

****

Share count value among components.



## Steps to implement:

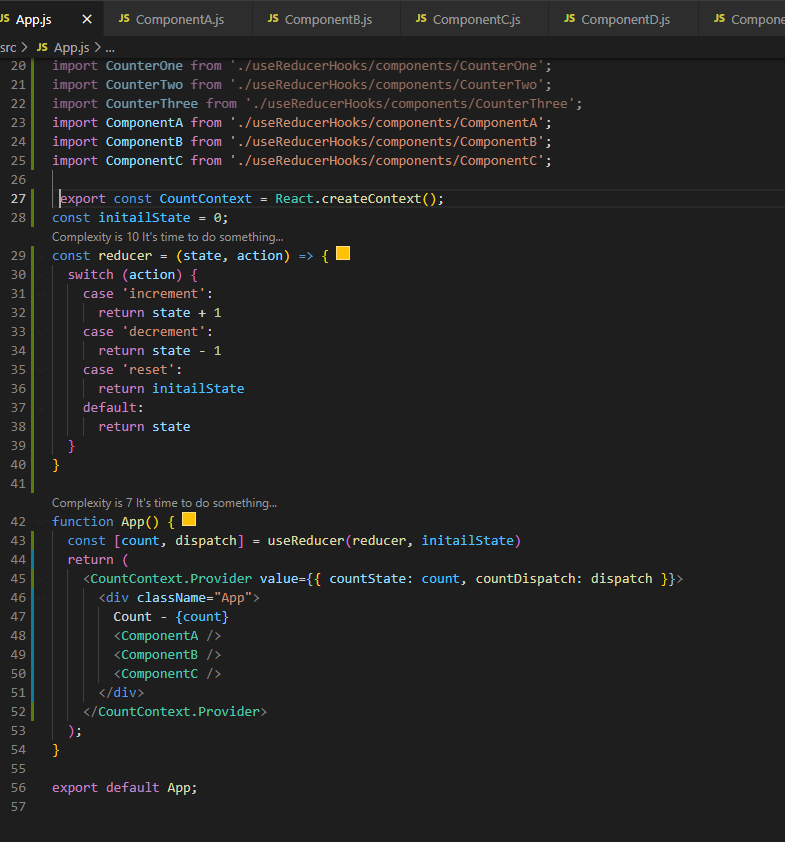
1. Create a reducer and initialstate in App.js.

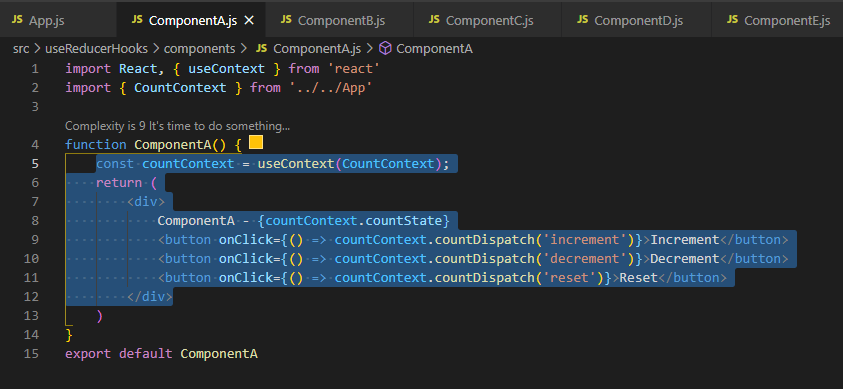
2. call useReducer hooks.

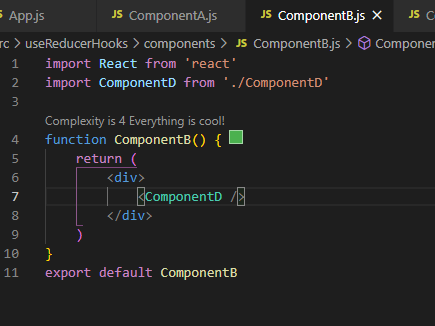
3. Create useContext hooks.

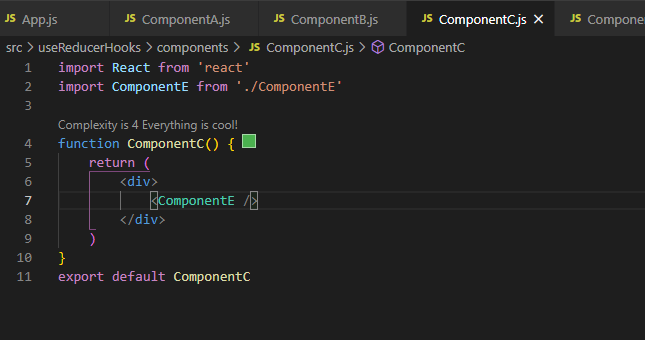
4. Wrap the component with useContext.Provider and pass the value(state and dispatcher).

5. Consume the context into component.

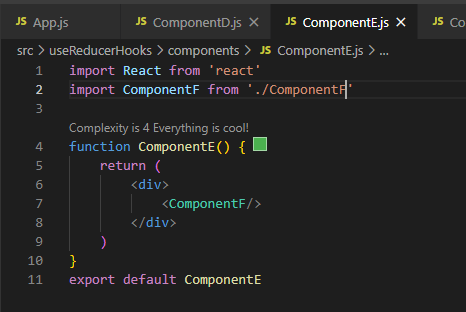


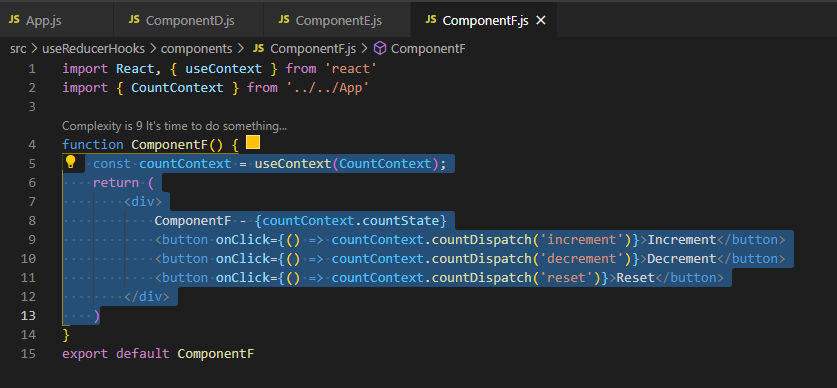


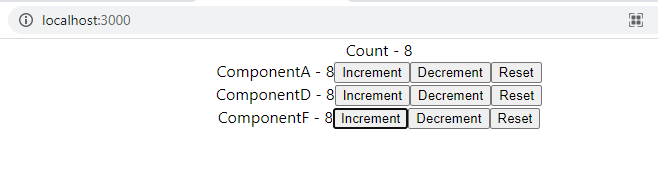












# Fetching data with useReducer:

Show ‘Loading’ when data is getting loaded. Once data loaded successfully hide ‘Loading’ and show data. If any error comes show ‘Something went wrong’.

To implement this, we have created 3 state variable using useState.



To implement above example through useReducer, follow the below steps:

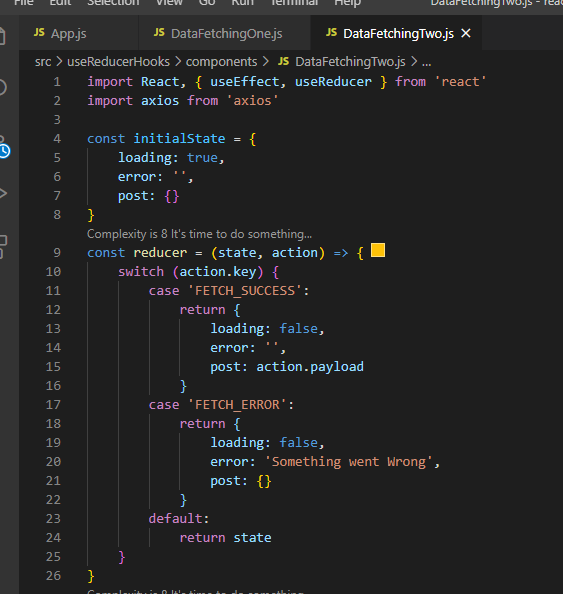
1. Import useReducer, useEffect and axios.

2. Declare initial state and define reducer function.

3. Call useReducer method.

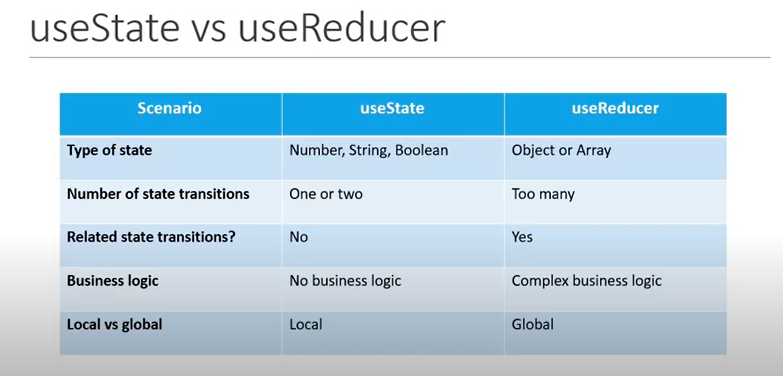
4. Do http call using useEffect and inside useEffect call dispatch method passing appropriate object.

5. print the state in JSX.





# When to use useState and useReducer:

****

# useCallback hooks:

useCallback hook is used for optimization.

Scenario: Create Title, Button and Count component to print title, button and count respectively and add all component into ParentComponent. Add log in all child component.

Button component is used to increase age and salary passing as props. Count component is used to show age and salary passing as props.

Problem: All components’ logs are printed as component loads at first time which is fine but when we click single button (either Increment Age or Increment Salary) then at that time also all components are getting re-rendered and all logs are getting printed. Rather than it should only render the changed component.

It may bring performance issue because all components are re-rendered even though it is not updated. We have to restrict re-renders to only component that need to re-render. Like when we click Increment Age button then only Count and Button Component related to age should re-render. Other three component don’t have to re-render. Similar to the case with Increment Salary button.

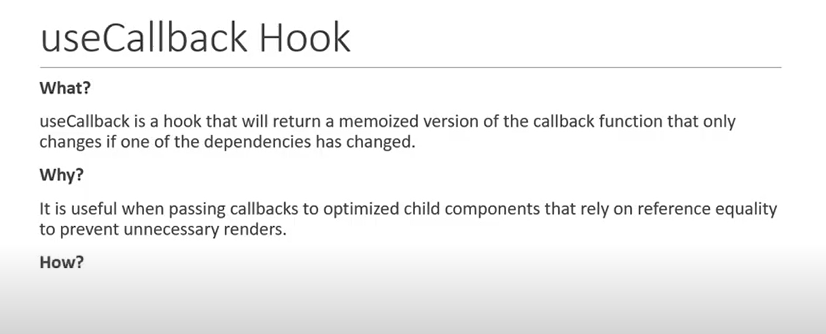
To optimize this, we use **React.memo.** It is a higher order component that will prevent functional component from being re-rendered if it’s props or state do not change. It is a feature of React not hooks.

Now our component will only re-render when there is a change in their props or state. But still when we click Increment Age button, Increment Salary button also re-renders(verified in log) and vice-versa. This is because both button has a method(incrementAge and incrementSalary) as a prop and when ParentComponent re-render both the functions are created newly each time.

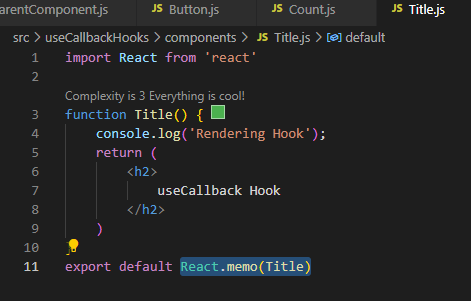
While dealing with function we always must consider reference equality. Even though two function have exact same behavior does not mean they are equal to each other. So, the function before the re-render is different than after the re-render. And since function is a prop to button, React.memo sees that prop has changed to component re-renders.

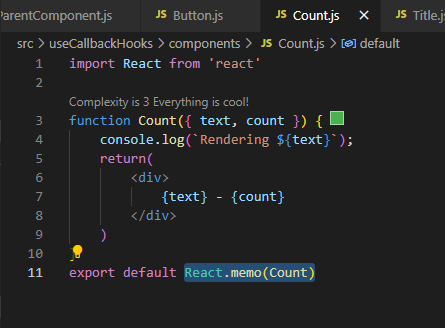


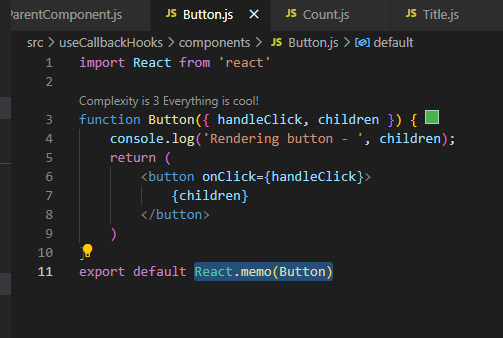
To fix this use useCallback hook. useCallback hook cache the function and return the same function if dependent props/state is not updated/changed.

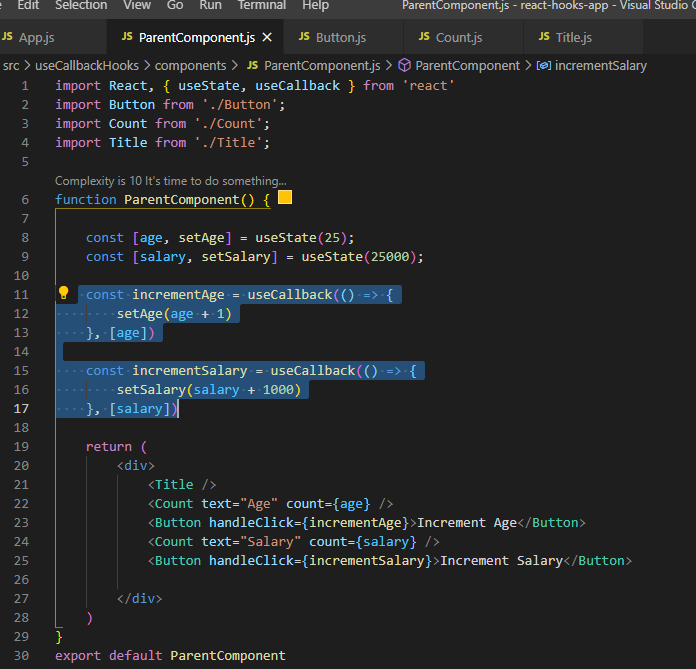


useCallback hook except two argument first the function and second is the dependency array.









**Why to not use useCallback Hook all the time?**

<https://kentcdodds.com/blog/usememo-and-usecallback>

# useMemo hooks:

useMemo is also used for performance optimization and works similar to useCallback but the difference between them is useCallback cache the provided function instance itself whereas useMemo invokes the provided function and cache its result.

So, if we need to cache function use useCallback and if we need to cache result of invoke function use useMemo.

useMemo hook cache the result of function and return the same result if dependent props/state is not updated/changed and stop re-rendering.



# useRef hooks:

useRef hook is used to access the DOM node directly.

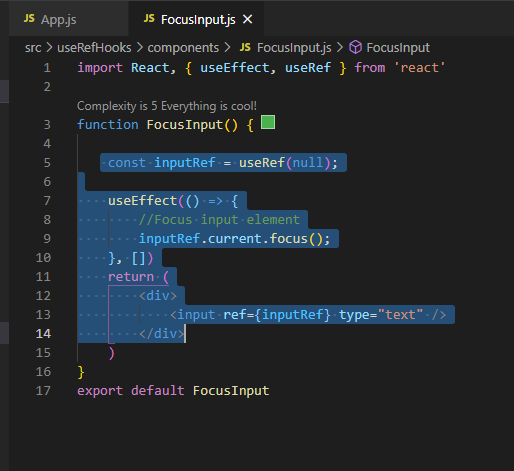
Steps to implement:

1. Import useRef from react.

2. call useRef and pass initialvalue (here initially ref hold null value).

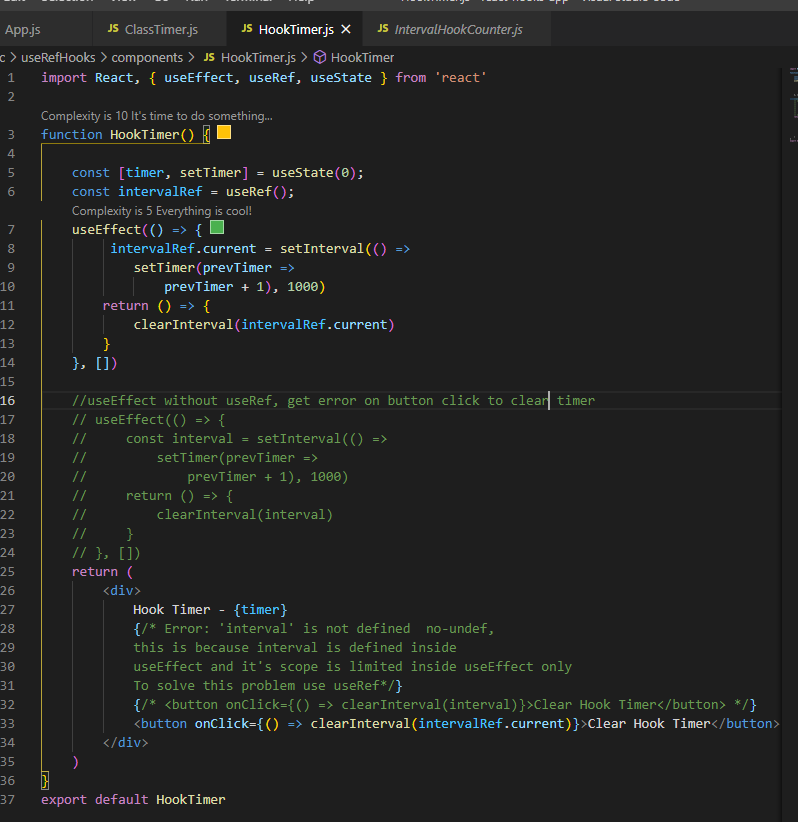
3. Attach ref to input element, to do so use reserve ref attribute.

4. call focus method using inputRef.current.



Although useRef is used to hold the reference of DOM node using ref attribute, it can also be used to store any mutable value, the great thing about useRef is the value will persist through the re-renders while also not causing any additional renders when it’s value changes.



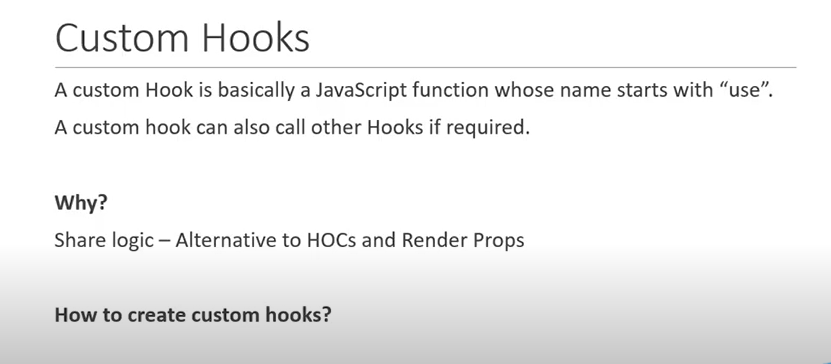


useRef hook can be used to create generic container which can hold a mutable value similar to instance properties on a class component. This generic container doesn’t cause re-renders when the data it stores changes at the same time it also remembers the stored data even after other state variables caused a re-render of this component. Simple use-case is clearing interval time from the event handler(above).

useImperativeHandle, useLayoutEffect, useDebugValue are other rarely used hooks.

# Custom hook: -

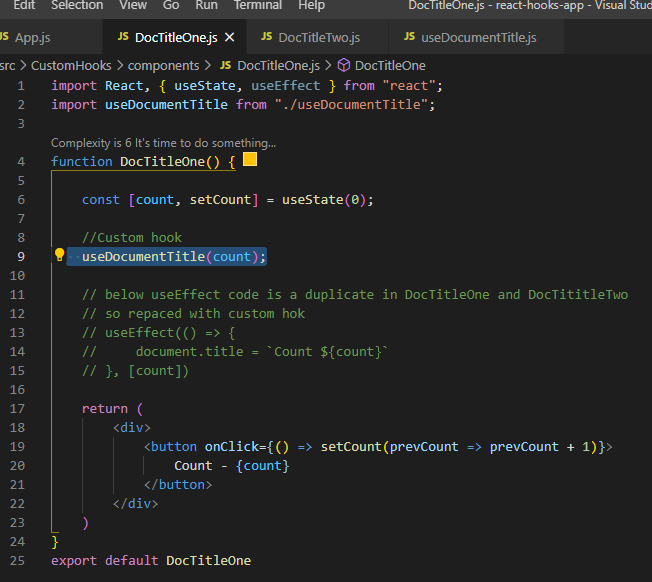
****

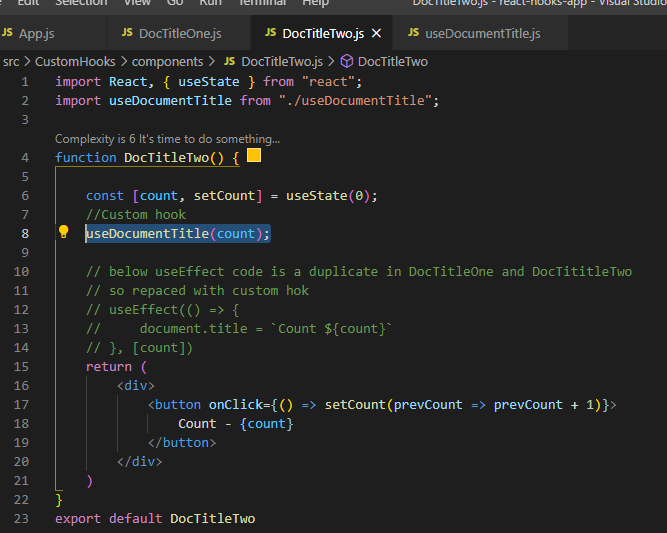
****

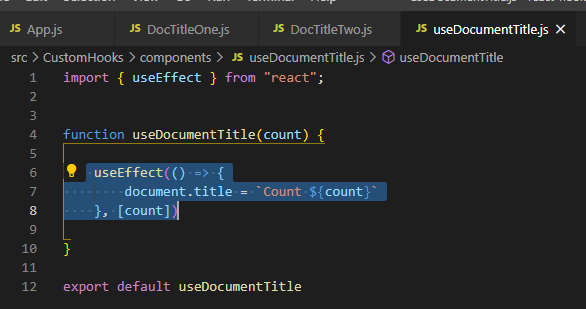
It is recommended to start the name of the custom hooks should be in small case with use.

First use case of custom hooks it to replace duplicate piece of logic among multiple Components. Put that login in custom hook and then use the custom hook.

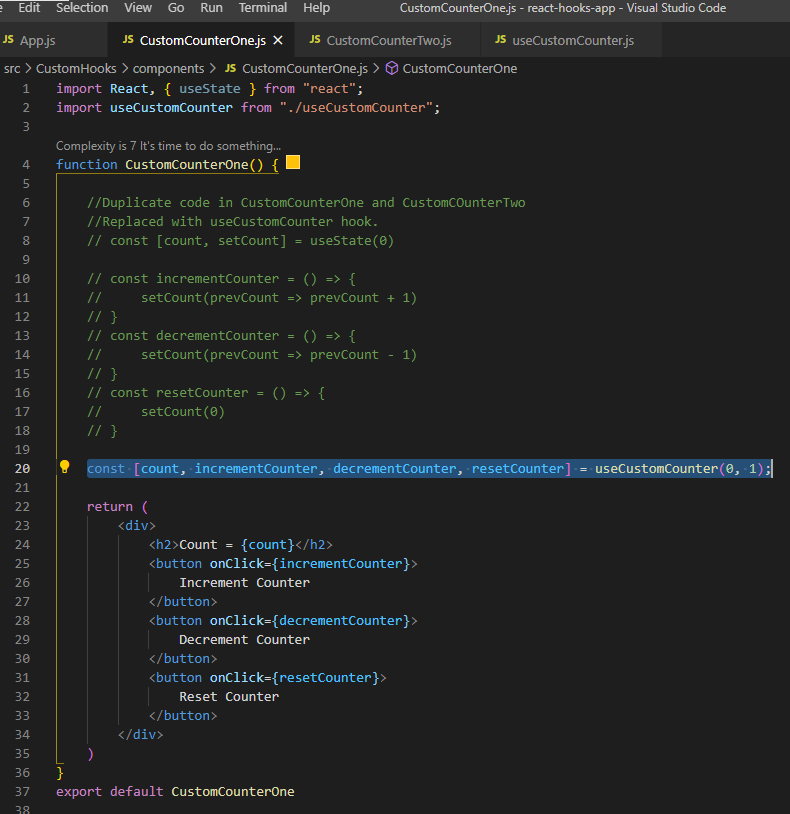
Example-1

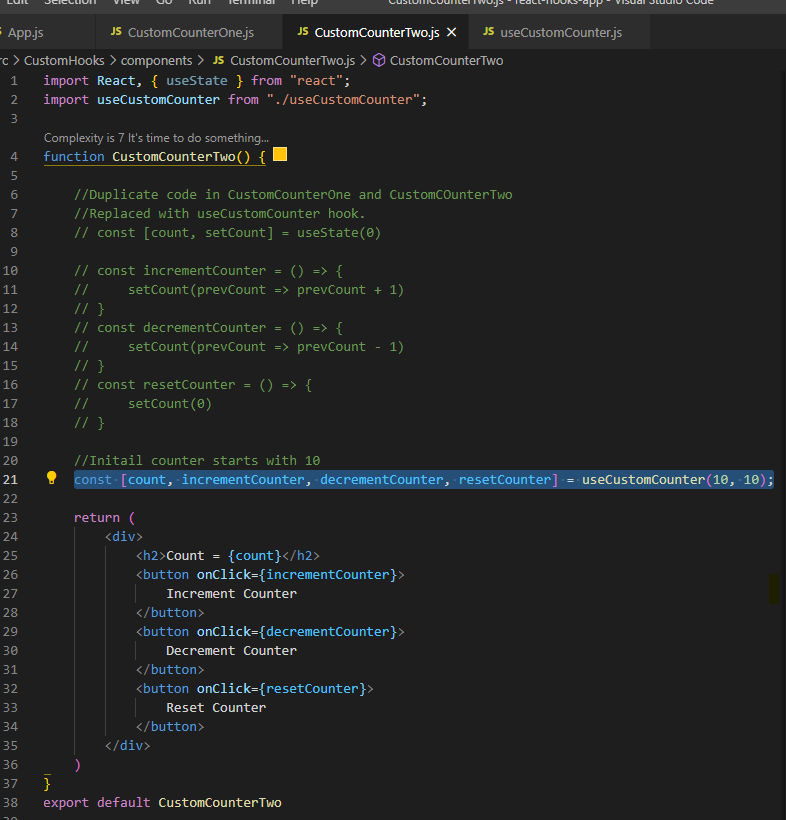


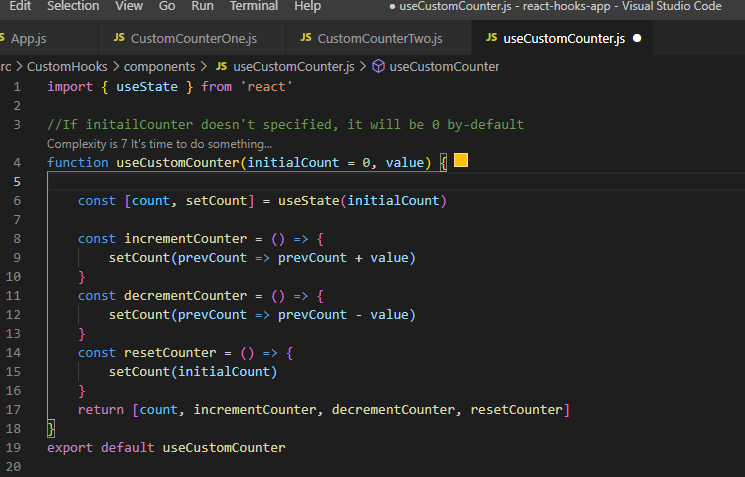




Example-2

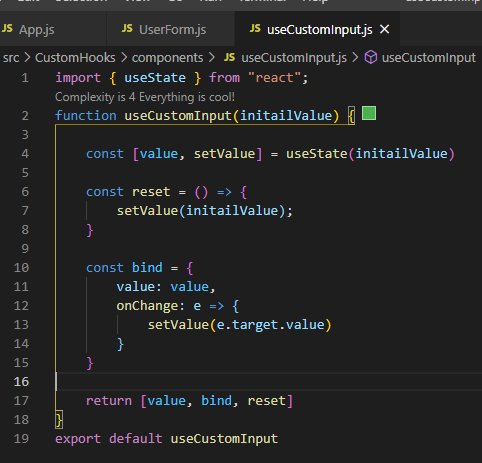




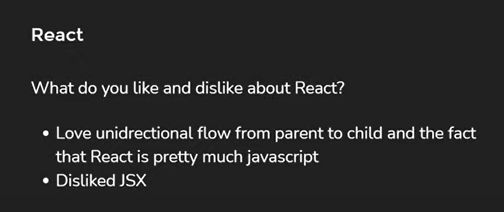


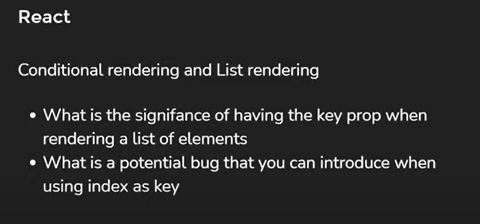
Example-3

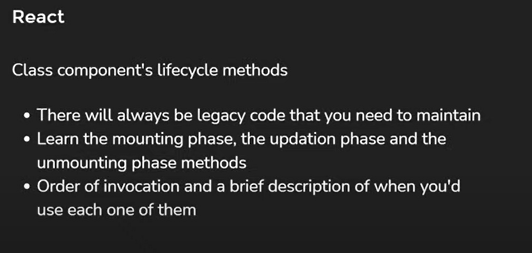


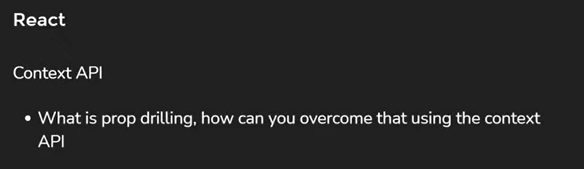


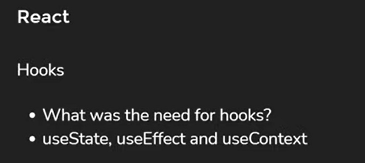
# Interview Questions:

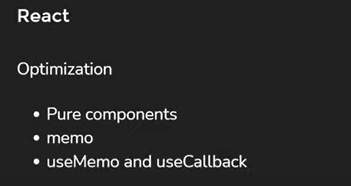
****

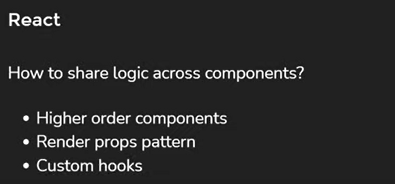
****

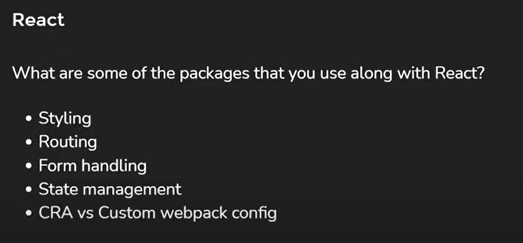
****

****

****

****

****

****