Module 02

Fullstack Web Development

React Hooks

Job Connector Program

Outline

- Hooks concept
- Common used hooks (useState, useEffect, etc)
- Rules of hooks
- Custom hooks

What is React Hooks?

React hooks are like special tools that help you build better web applications with React, a popular JavaScript library. These tools were added to React around version 16.8.

What's the Problem Hooks Solve?

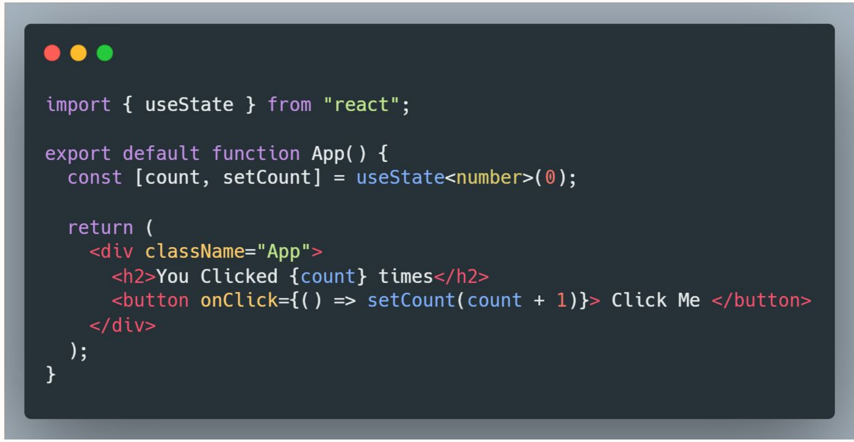
In the past, if you wanted to keep track of data (like how many likes a post has) or do things when your web page changes (like loading new data when you scroll down), you had to write your code in a specific way called "class components." This way was a bit complicated and had a lot of extra rules.

How Hooks Help?

React hooks make this much simpler, especially for something called "functional components." These are a more straightforward way to write your code. With hooks, you can now use state (like counting likes) and do things when your web page changes (like loading new data) in a much neater and easier way.

Why we need Hooks?

- **Easier Reuse**: You can take the code for things like counting likes and use it in many different parts of your website without rewriting it.
- **Cleaner Code**: Your code looks cleaner and is easier to read and understand.
- **No More Class Components**: You can use functional components for everything, which is a simpler way to write your code.



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

export default function App() {
  const [count, setCount] = useState<number>(0);

  return (
    <div className="App">
      <h2>You Clicked {count} times</h2>
      <button onClick={() => setCount(count + 1)}> Click Me </button>
    </div>
  );
}
```

So, in simple terms, React hooks are like special tools that make it easier to work with data and do things on your website. They make your code cleaner and let you use a simpler way of writing your web applications.

React Hooks Benefits and Purposes in React applications

- **Simplified State Management:**

  useState: Simplifies state management in functional components. It allows you to declare and manage component-level state without using class components.

- **Enhanced Code Reusability:**

  Custom Hooks: You can create your own custom hooks to encapsulate and reuse stateful logic across different components. This promotes code reusability and a modular code structure.

- **Improved Readability:**

  Functional components with hooks tend to be more concise and easier to read than class components, leading to cleaner and more maintainable code.

React Hooks Benefits and Purposes in React applications

- **Reduced Boilerplate:**

  Hooks eliminate the need for writing constructor methods, using this references, and the render method, resulting in less boilerplate code.

- **Easy Side Effect Handling:**

  useEffect: Simplifies the management of side effects like data fetching, DOM manipulation, and subscriptions. You can specify dependencies to control when effects run, which helps prevent unnecessary re-renders.

- **Improved Performance:**

  useMemo and **useCallback**: These hooks can help optimize your components by memoizing values and functions, reducing unnecessary re-renders and improving performance.

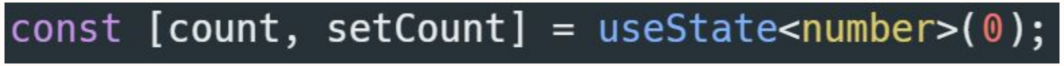
Most Commonly Used React Hooks

- **useState,** allows change of value in variable and re-rendering the entire component after value changes
- **useEffect,** hooks that triggered after variable of useState changed. It can have dependencies or what variables of useState that exactly can trigger the useEffect.
- **useRef**, allows you to persist values between renders. It can be used to store a mutable value that does not cause a re-render when updated. It can be used to access a DOM element directly.
- **useMemo**, returns a memoized value. The useMemo Hook can be used to keep expensive, resource intensive functions from needlessly running.
- **useContext**, React Context is a way to manage state globally. It can be used together with the useState Hook to share state between deeply nested components more easily than with useState alone.
- **useReducer**, It allows for custom state logic. If you find yourself keeping track of multiple pieces of state that rely on complex logic, useReducer may be useful.
- **useCallback**, returns a memoized callback function. This allows us to isolate resource intensive functions so that they will not automatically run on every render.
- Reference : https://react.dev/reference/react/hooks

useState

Starts by importing the useState Hook from React, then when we want to declare a new state variable, make a syntax like example below. Count is a state variable and its value is 0 because we define that in parameter useState(0). setCount is a function for update the count state.

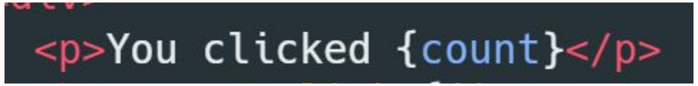
The <number> is a datatype value that we must stated in the useState. Next time , we cant assign other datatypes beside number



```
const [count, setCount] = useState<number>(0);
```

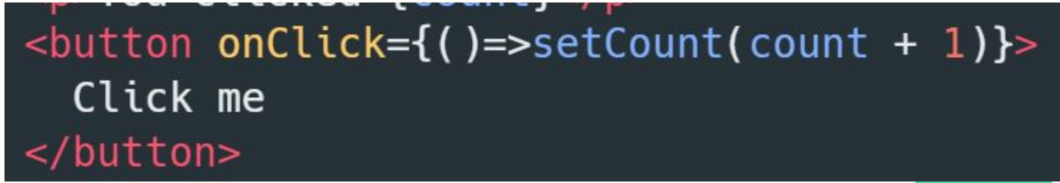
useState

When we want to display the current value of count, we can use count directly:



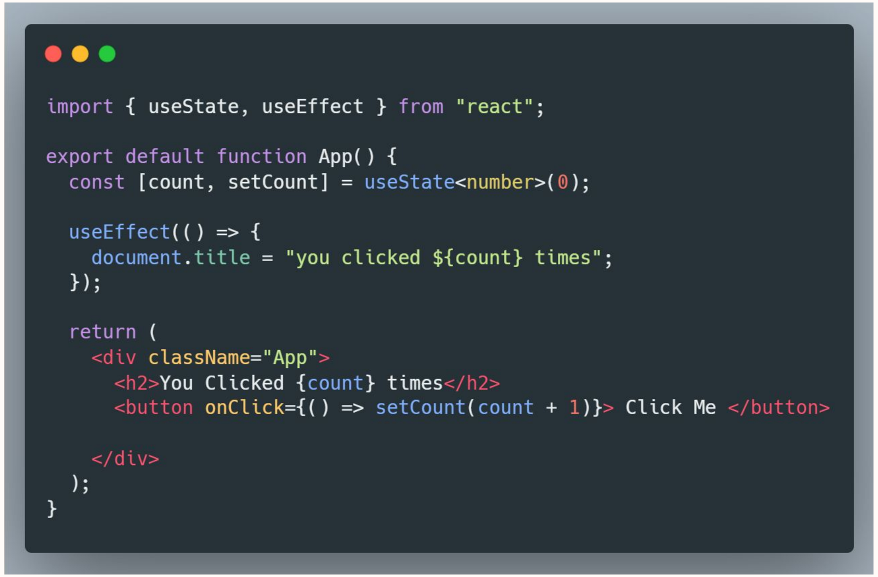
```
<p>You clicked {count}</p>
```

Just call setCount that we defined before, then put new value in parameter's setCount that we want to replace the current value of count.



```
<button onClick={()=>setCount(count + 1)}>
  Click me
</button>
```

useEffect



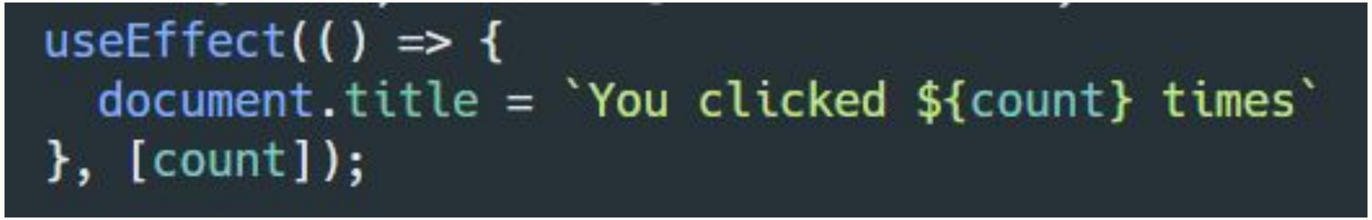
```
import { useState, useEffect } from "react";

export default function App() {
  const [count, setCount] = useState<number>(0);

  useEffect(() => {
    document.title = "you clicked ${count} times";
  });

  return (
    <div className="App">
      <h2>You Clicked {count} times</h2>
      <button onClick={() => setCount(count + 1)}> Click Me </button>

    </div>
  );
}
```

Start with import useEffect from react, then use it like syntax beside. What will happen? useEffect will trigger if state or props is updated. When we click button “Click me”, it will updated count to count + 1.

When count updated, useEffect will trigger and execute code inside it, which is in this case, useEffect will execute to show how much we clicked the button “Click me”.

useEffect

If we feel that trigger every state and props updated is too much, you can tell React to *skip* applying an effect if certain values haven't changed between re-renders. To do so, pass an array as an optional second argument to useEffect:



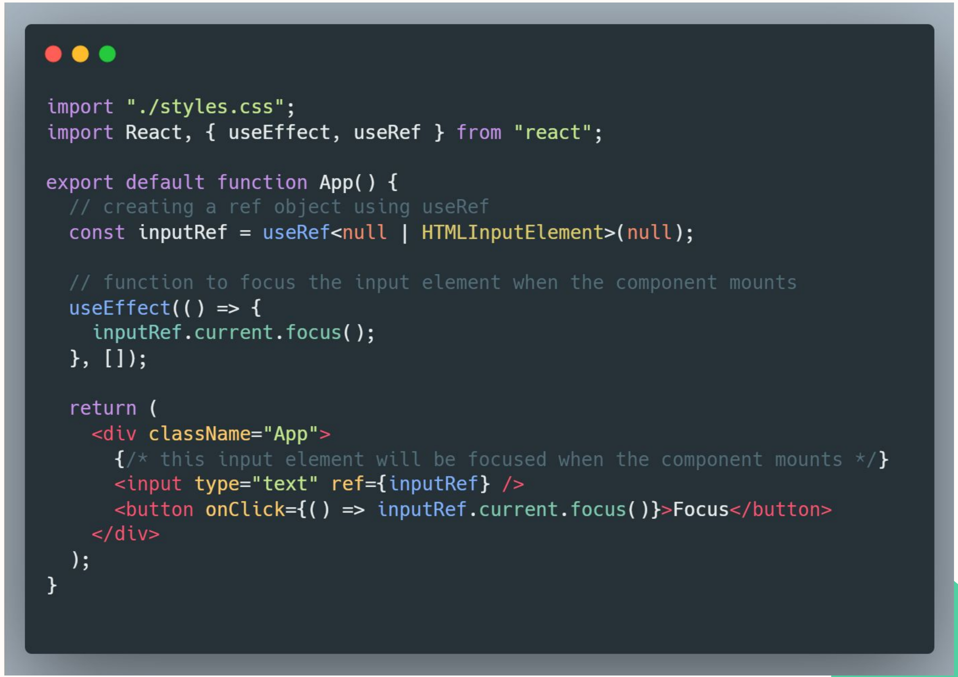
```
useEffect(() => {
  document.title = `You clicked ${count} times`
}, [count]);
```

In the example above, we pass [count] as the second argument. What does this mean? If the count is 5, and then our component re-renders with count still equal to 5, React will compare [5] from the previous render and [5] from the next render. Because all items in the array are the same (5 === 5), React would skip the effect. That's our optimization.

useRef

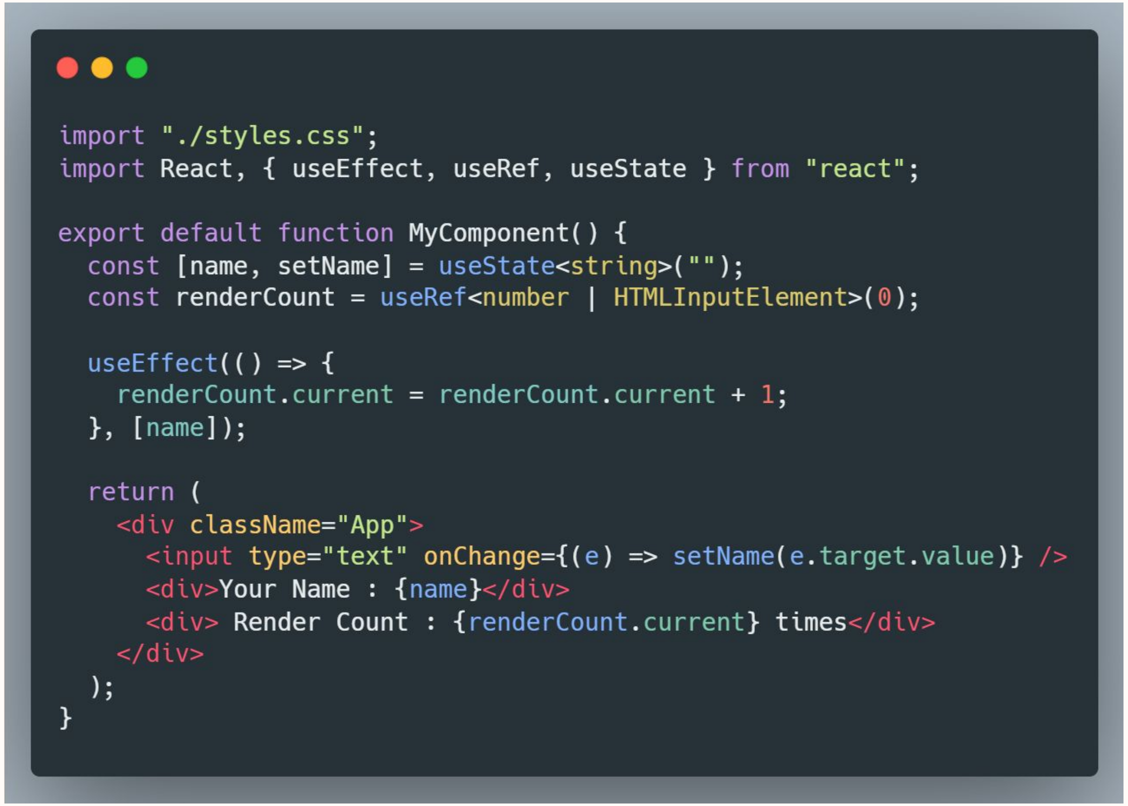
the useRef hook is used to create a mutable object whose current property can hold a reference to a DOM element or to store any mutable value that persists across re-renders without causing the component to re-render when the ref value changes.

It simplified using react state and using useEffect and causing re-render when state is updated. Using useRef not causing component render again



```
import "./styles.css";
import React, { useEffect, useRef } from "react";

export default function App() {
  // creating a ref object using useRef
  const inputRef = useRef<null | HTMLInputElement>(null);

  // function to focus the input element when the component mounts
  useEffect(() => {
    inputRef.current.focus();
  }, []);

  return (
    <div className="App">
      {/* this input element will be focused when the component mounts */}
      <input type="text" ref={inputRef} />
      <button onClick={() => inputRef.current.focus()}>Focus</button>
    </div>
  );
}
```

useRef



```
import "./styles.css";
import React, { useEffect, useRef, useState } from "react";

export default function MyComponent() {
  const [name, setName] = useState<string>("");
  const renderCount = useRef<number | HTMLInputElement>(0);

  useEffect(() => {
    renderCount.current = renderCount.current + 1;
  }, [name]);

  return (
    <div className="App">
      <input type="text" onChange={(e) => setName(e.target.value)} />
      <div>Your Name : {name}</div>
      <div> Render Count : {renderCount.current} times</div>
    </div>
  );
}
```

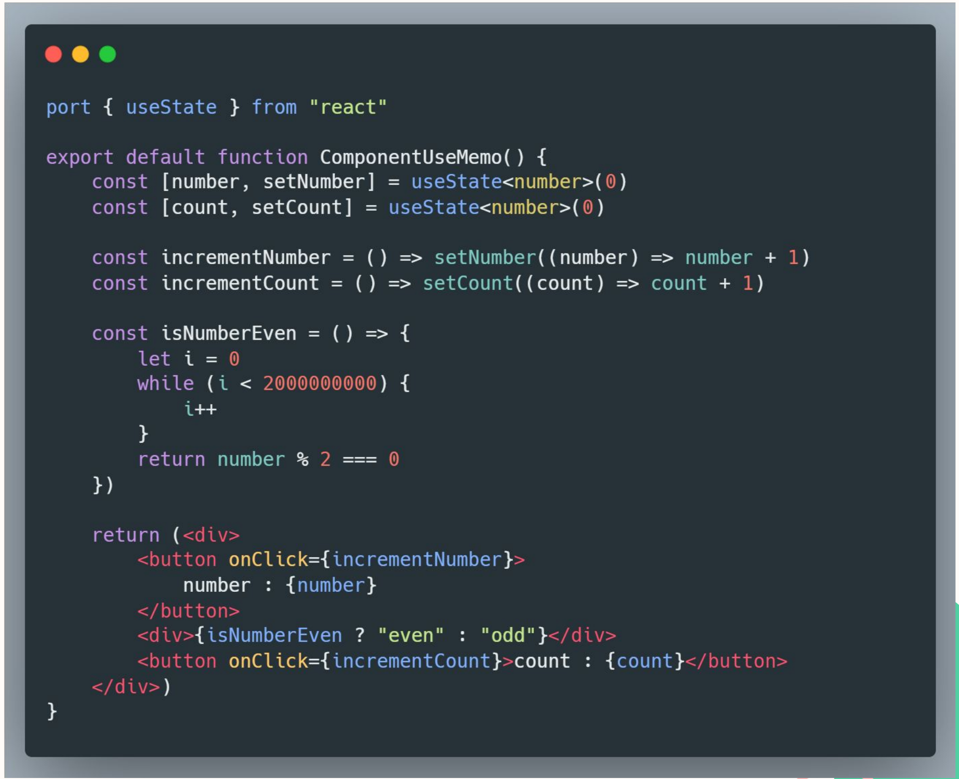
Another example when using useRef. Imagine you want to show the component , how many times the component rendered.

If you using useState and useEffect instead of useRef, the component will rendered forever.

useMemo

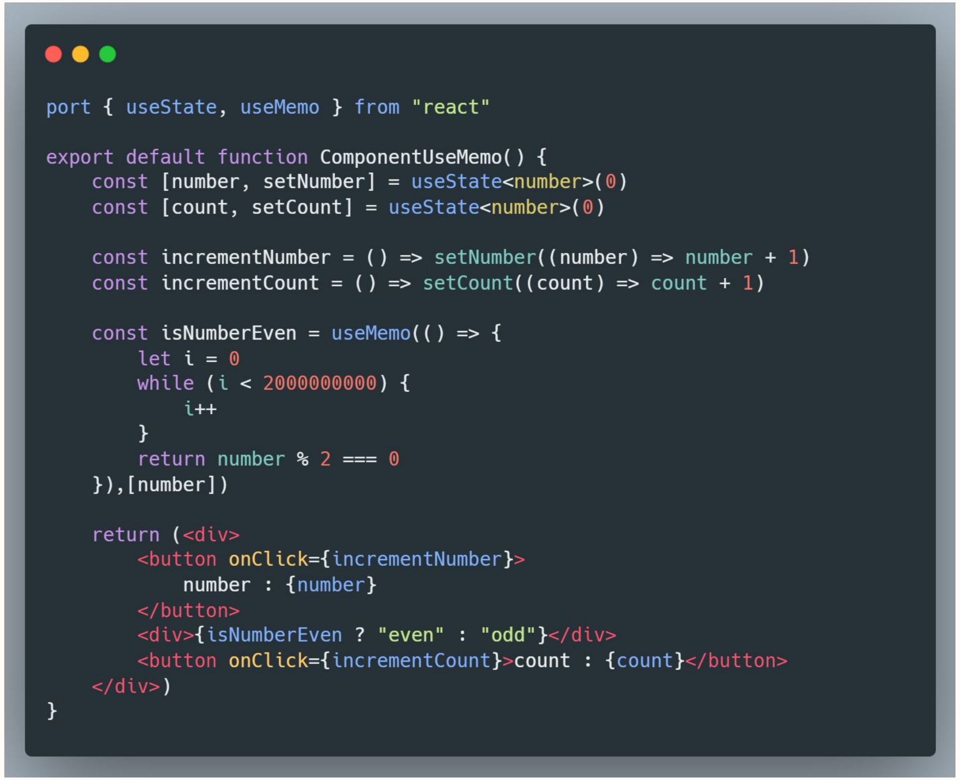
The React useMemo Hook returns a memoized value. Think of memoization as caching a value so that it does not need to be recalculated. The useMemo Hook only runs when one of its dependencies update. This can improve performance.

Take a look at that code, as you can see, this code will run with performance issue. If we click increment button , the render process will slow. Because the expensiveCalculation() process complexity.



```
port { useState } from "react"

export default function ComponentUseMemo() {
    const [number, setNumber] = useState<number>(0)
    const [count, setCount] = useState<number>(0)

    const incrementNumber = () => setNumber((number) => number + 1)
    const incrementCount = () => setCount((count) => count + 1)

    const isNumberEven = () => {
        let i = 0
        while (i < 2000000000) {
            i++
        }
        return number % 2 === 0
    })

    return (<div>
        <button onClick={incrementNumber}>
            number : {number}
        </button>
        <div>{isNumberEven ? "even" : "odd"}</div>
        <button onClick={incrementCount}>count : {count}</button>
    </div>)
}
```

useMemo



```
port { useState, useMemo } from "react"

export default function ComponentUseMemo() {
    const [number, setNumber] = useState<number>(0)
    const [count, setCount] = useState<number>(0)

    const incrementNumber = () => setNumber((number) => number + 1)
    const incrementCount = () => setCount((count) => count + 1)

    const isNumberEven = useMemo(() => {
        let i = 0
        while (i < 2000000000) {
            i++
        }
        return number % 2 === 0
    }),[number])

    return (<div>
        <button onClick={incrementNumber}>
            number : {number}
        </button>
        <div>{isNumberEven ? "even" : "odd"}</div>
        <button onClick={incrementCount}>count : {count}</button>
    </div>)
}
```

Now , using useMemo, we can fix this performance issue. This will cause the function to only run when needed.

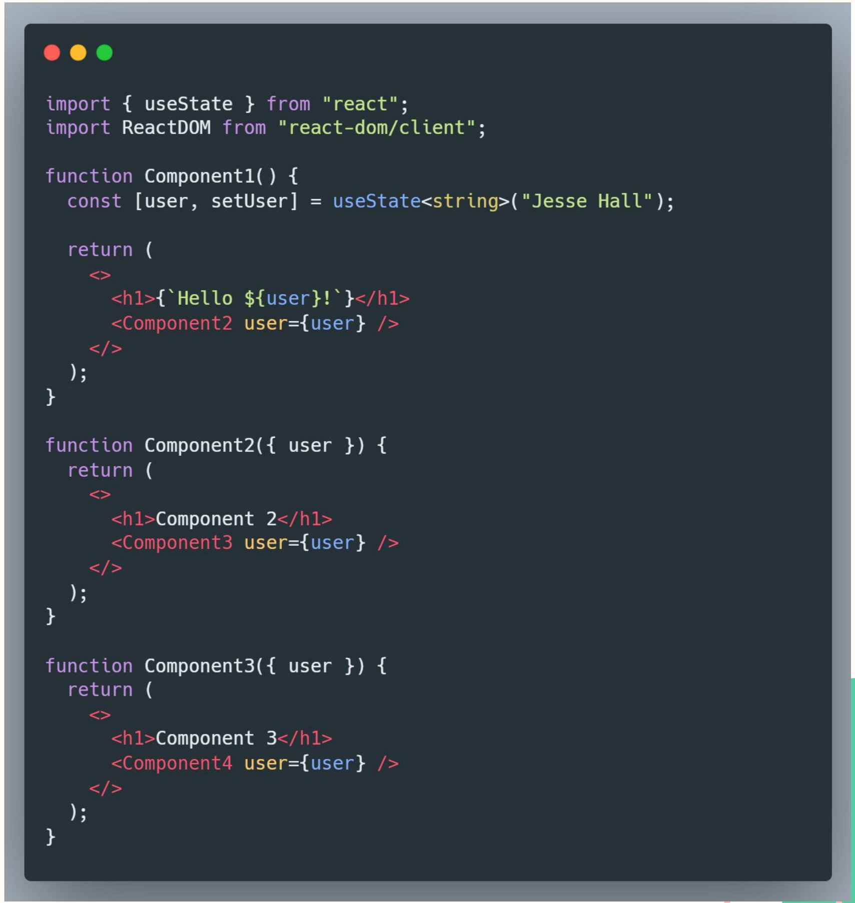
The useMemo Hook accepts a second parameter to declare dependencies. The expensive function will only run when its dependencies have changed.

useContext

useContext is using for manage state globally. The problem we faced often when managing state is when the component is inside a component is inside another component . lets say , we want to passing props from component level 1 to component level 3 deep .

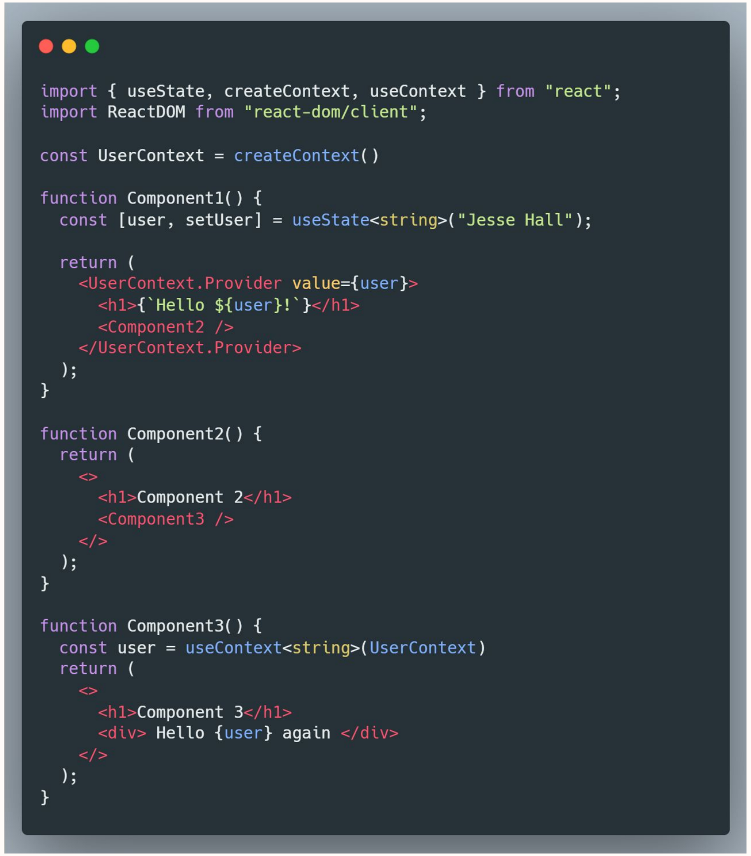
Lets take a look at this code :

You can see we must passing the props by props per component before the props value received in component3 (component level 3 deep)



```
import { useState } from "react";
import ReactDOM from "react-dom/client";

function Component1() {
  const [user, setUser] = useState<string>("Jesse Hall");

  return (
    <>
      <h1>{`Hello ${user}!`}</h1>
      <Component2 user={user} />
    </>
  );
}

function Component2({ user }) {
  return (
    <>
      <h1>Component 2</h1>
      <Component3 user={user} />
    </>
  );
}

function Component3({ user }) {
  return (
    <>
      <h1>Component 3</h1>
      <Component4 user={user} />
    </>
  );
}
```

useContext



```
import { useState, createContext, useContext } from "react";
import ReactDOM from "react-dom/client";

const UserContext = createContext()

function Component1() {
  const [user, setUser] = useState<string>("Jesse Hall");

  return (
    <UserContext.Provider value={user}>
      <h1>{`Hello ${user}!`}</h1>
      <Component2 />
    </UserContext.Provider>
  );
}

function Component2() {
  return (
    <>
      <h1>Component 2</h1>
      <Component3 />
    </>
  );
}

function Component3() {
  const user = useContext<string>(UserContext)
  return (
    <>
      <h1>Component 3</h1>
      <div> Hello {user} again </div>
    </>
  );
}
```

The Solution to this problem is ,createContext. The createContext using as high order component to wrap all over component who needs the props.

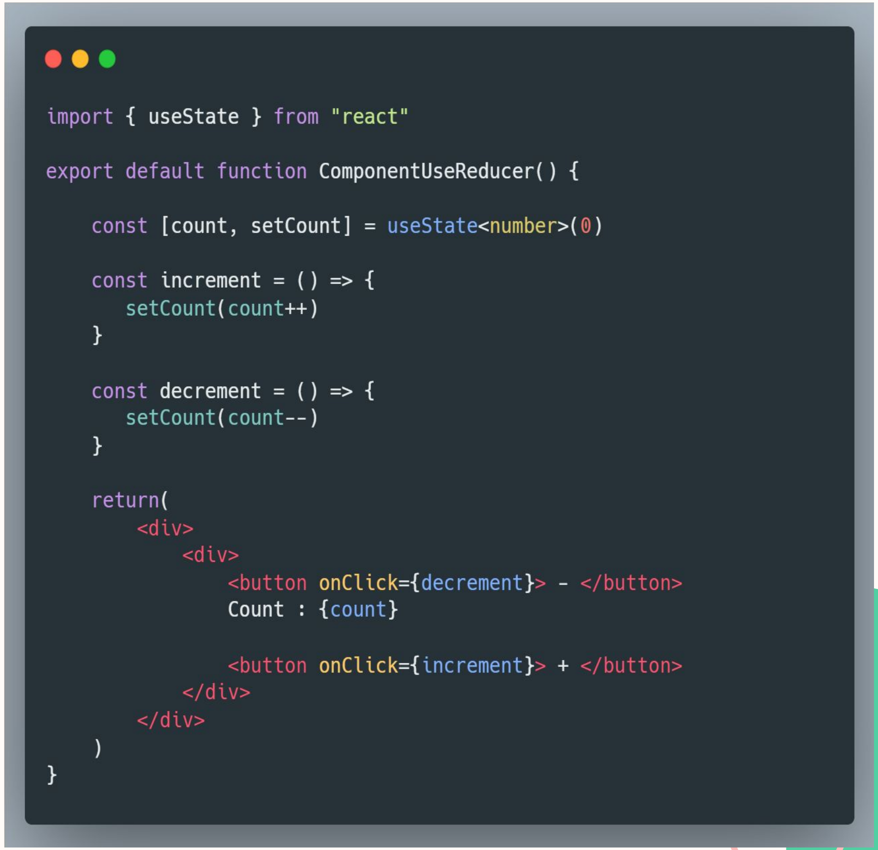
After that, use useContext to access the props from highorder component without adding props inside the component again

useReducer

useReducer is like useState. But , it will useful if you had complex logic to manage your state.

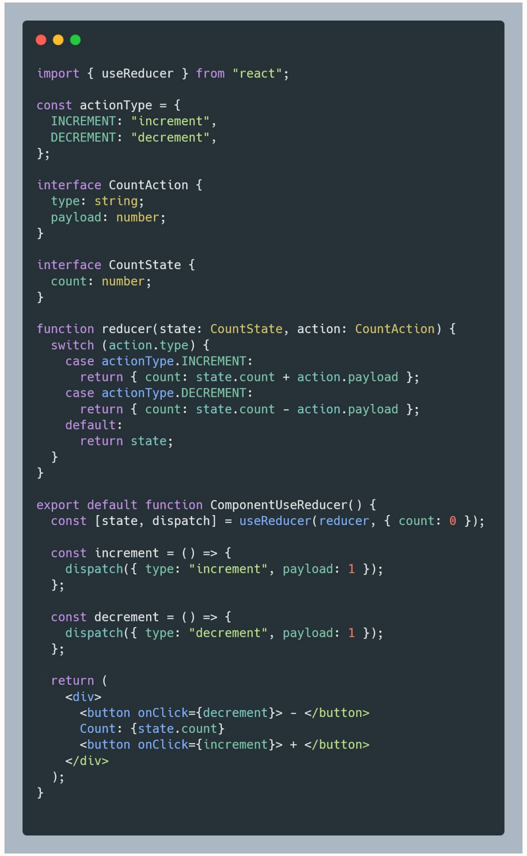
useReducer work with useDispatch to triggering some logic function inside reducer.

Take a look at this simple code. Looks like we can handle it only with useState.



```
import { useState } from "react"

export default function ComponentUseReducer() {

    const [count, setCount] = useState<number>(0)

    const increment = () => {
       setCount(count++)
    }

    const decrement = () => {
       setCount(count--)
    }

    return(
        <div>
            <div>
                <button onClick={decrement}> - </button>
                Count : {count}

                <button onClick={increment}> + </button>
            </div>
        </div>
    )
}
```

useReducer



```
import { useReducer } from "react";

const actionType = {
  INCREMENT: "increment",
  DECREMENT: "decrement",
};

interface CountAction {
  type: string;
  payload: number;
}

interface CountState {
  count: number;
}

function reducer(state: CountState, action: CountAction) {
  switch (action.type) {
    case actionType.INCREMENT:
      return { count: state.count + action.payload };
    case actionType.DECREMENT:
      return { count: state.count - action.payload };
    default:
      return state;
  }
}

export default function ComponentUseReducer() {
  const [state, dispatch] = useReducer(reducer, { count: 0 });

  const increment = () => {
    dispatch({ type: "increment", payload: 1 });
  };

  const decrement = () => {
    dispatch({ type: "decrement", payload: 1 });
  };

  return (
    <div>
      <button onClick={decrement}> - </button>
      Count: {state.count}
      <button onClick={increment}> + </button>
    </div>
  );
}
```

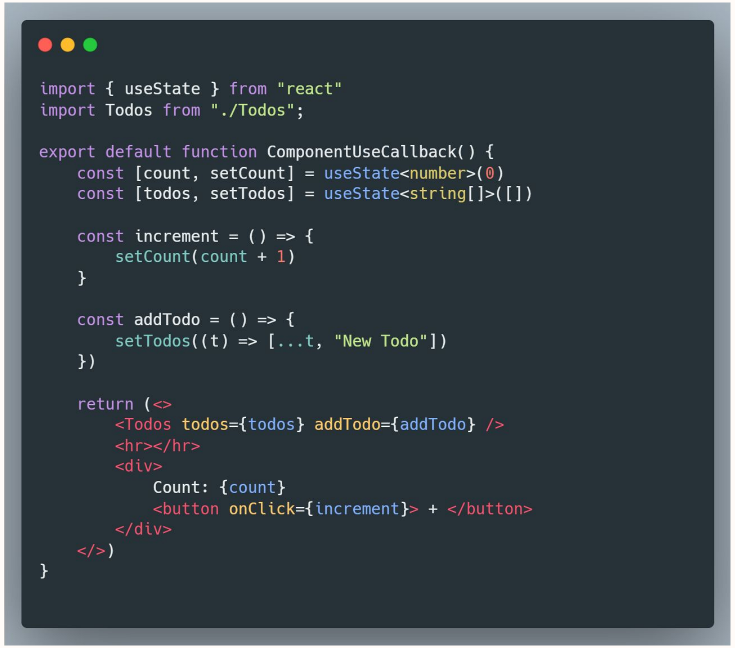
Now , look at when we use useReducer. With this, we can classification our state and method in one place. And triggering the action with useDispatch

This hooks will our deep dive learning when we use state management library like redux toolkit

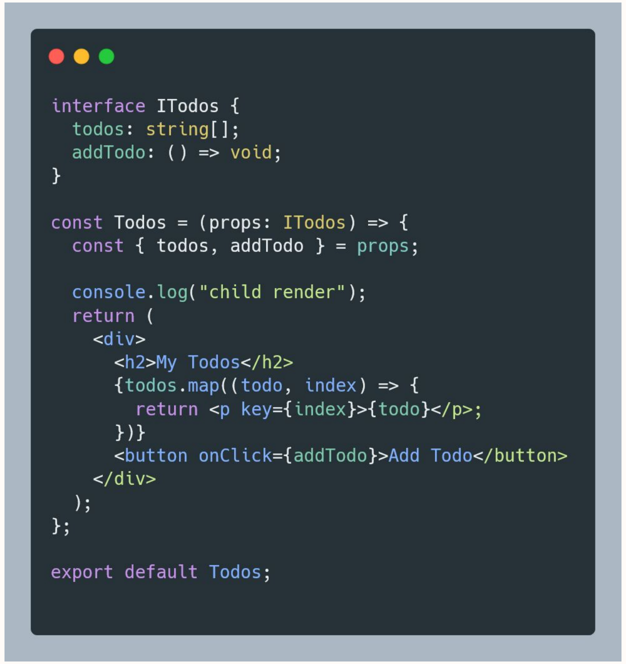
useCallback

The React useCallback Hook returns a memoized callback function. One reason to use useCallback is to prevent a component from re-rendering unless its props have changed.

In this example, you might think that the Todos component will not re-render unless the todos change:



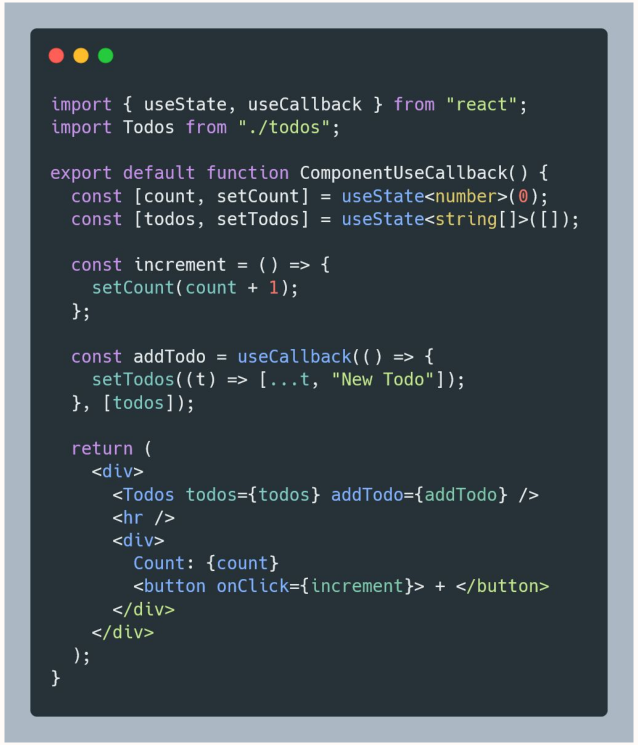
```
import { useState } from "react"
import Todos from "./Todos";

export default function ComponentUseCallback() {
    const [count, setCount] = useState<number>(0)
    const [todos, setTodos] = useState<string[]>([])

    const increment = () => {
        setCount(count + 1)
    }

    const addTodo = () => {
        setTodos((t) => [...t, "New Todo"])
    })

    return (<>
        <Todos todos={todos} addTodo={addTodo} />
        <hr></hr>
        <div>
            Count: {count}
            <button onClick={increment}> + </button>
        </div>
    </>)
}
```



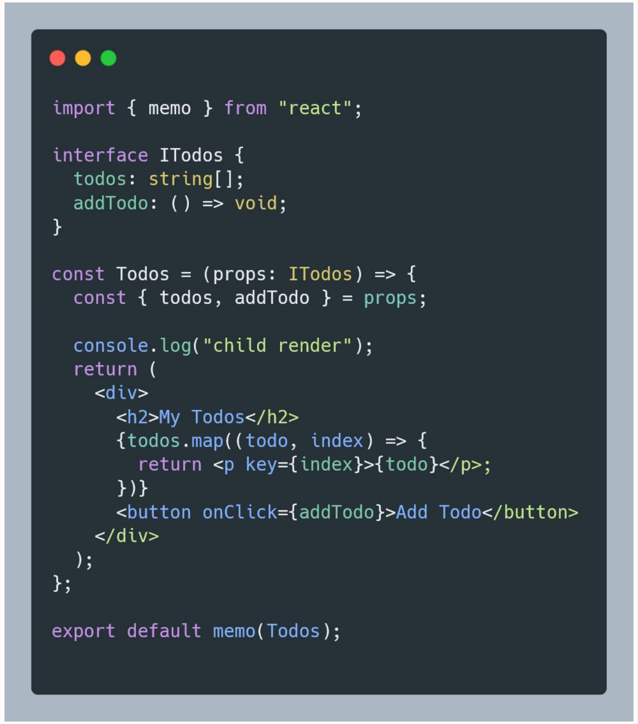
```
interface ITodos {
  todos: string[];
  addTodo: () => void;
}

const Todos = (props: ITodos) => {
  const { todos, addTodo } = props;

  console.log("child render");
  return (
    <div>
      <h2>My Todos</h2>
      {todos.map((todo, index) => {
        return <p key={index}>{todo}</p>;
      })}
      <button onClick={addTodo}>Add Todo</button>
    </div>
  );
};

export default Todos;
```

useCallback

But in fact , the Todos component will re render. The solution for this is useCallback in addTodo and memo in Todos component to prevent the Todos component re-render



```
import { useState, useCallback } from "react";
import Todos from "./todos";

export default function ComponentUseCallback() {
  const [count, setCount] = useState<number>(0);
  const [todos, setTodos] = useState<string[]>([]);

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

  const addTodo = useCallback(() => {
    setTodos((t) => [...t, "New Todo"]);
  }, [todos]);

  return (
    <div>
      <Todos todos={todos} addTodo={addTodo} />
      <hr />
      <div>
        Count: {count}
        <button onClick={increment}> + </button>
      </div>
    </div>
  );
}
```



```
import { memo } from "react";

interface ITodos {
  todos: string[];
  addTodo: () => void;
}

const Todos = (props: ITodos) => {
  const { todos, addTodo } = props;

  console.log("child render");
  return (
    <div>
      <h2>My Todos</h2>
      {todos.map((todo, index) => {
        return <p key={index}>{todo}</p>;
      })}
      <button onClick={addTodo}>Add Todo</button>
    </div>
  );
};

export default memo(Todos);
```

Rules of Hooks

Hooks are JavaScript functions, but you need to follow two rules when using them.

Only Call Hooks at the Top Level

Don't call Hooks inside loops, conditions, or nested functions. Instead, always use Hooks at the top level of your React function, before any early returns.

By following this rule, you ensure that Hooks are called in the same order each time a component renders. That's what allows React to correctly preserve the state of Hooks between multiple useState and useEffect calls.

Only Call Hooks from React Functions

Don't call Hooks from regular JavaScript functions. Instead, you can:

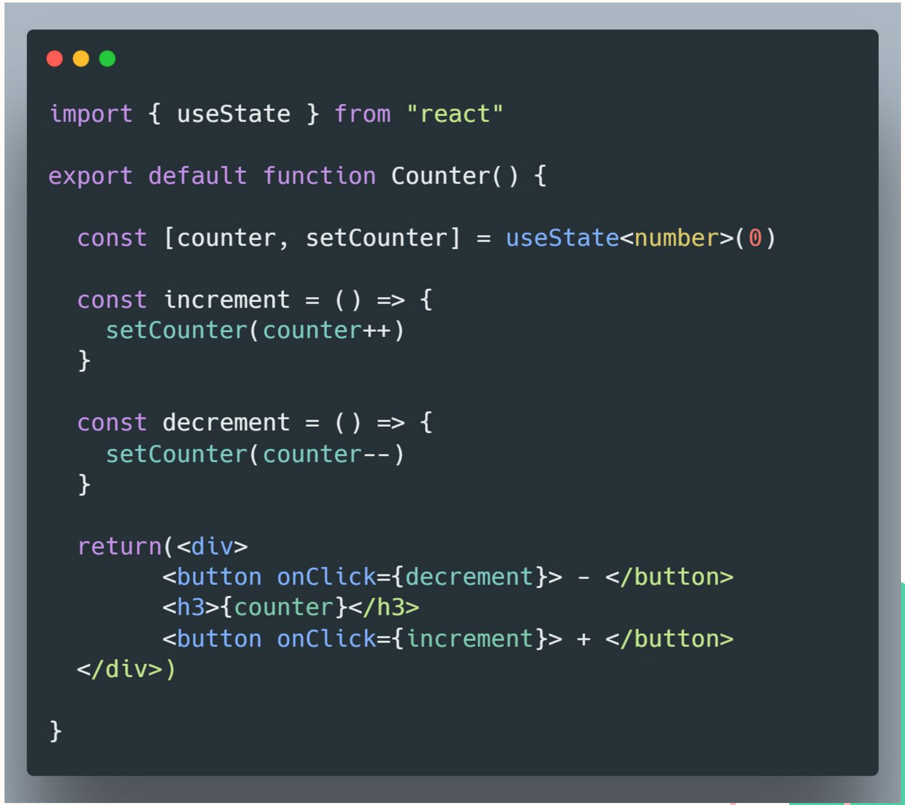
- ✅ Call Hooks from React function components.
- ✅ Call Hooks from custom Hooks.

By following this rule, you ensure that all stateful logic in a component is clearly visible from its source code.

Custom Hook

Building your own Hooks lets you extract component logic into reusable functions.

Now we will see an example of a counter use useState hook. In the beside example, we created a counter in App.js which is increments by 1 when we click on a increment button and decrements by 1 when we click on a decrement button.



```
import { useState } from "react"

export default function Counter() {

  const [counter, setCounter] = useState<number>(0)

  const increment = () => {
    setCounter(counter++)
  }

  const decrement = () => {
    setCounter(counter--)
  }

  return(<div>
        <button onClick={decrement}> - </button>
        <h3>{counter}</h3>
        <button onClick={increment}> + </button>
  </div>)

}
```

Create your own hook

Suppose we need this counter in different places of our app in such cases we can build our custom react hook instead of creating same counter logic again and again.

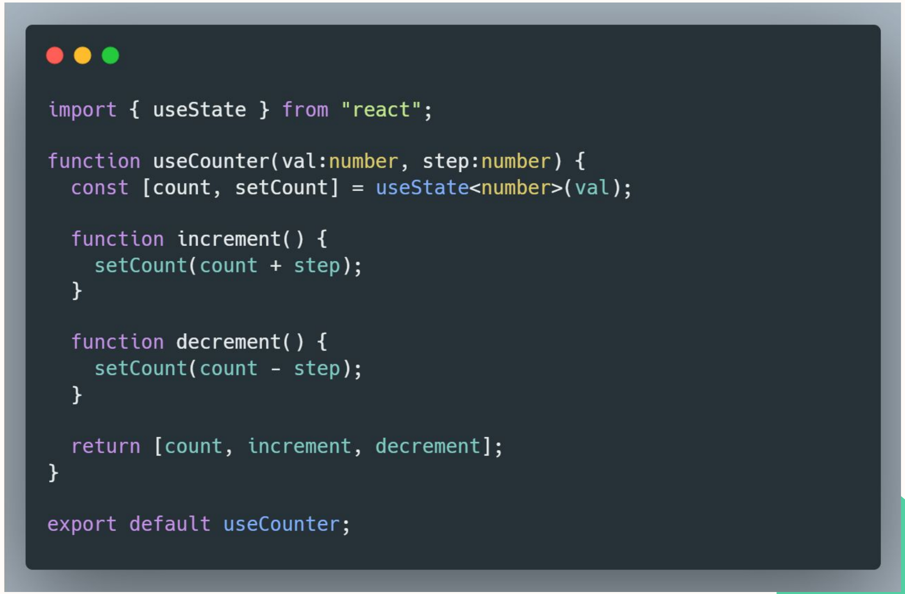
Custom hooks are JavaScript functions, whose name starts with use and they can call other react hooks.

Create your own hook

Now, we are removing the counter logic from the example before and creating our own custom hook called useCounter in counter-hook.js.

In the beside code, we created our own custom hook called useCounter with two parameters val and step and returning an array with count value , Increment and Decrement functions.

- **val**: Initial value of the counter.
- **step**: How many steps counter needs to increment or decrement.

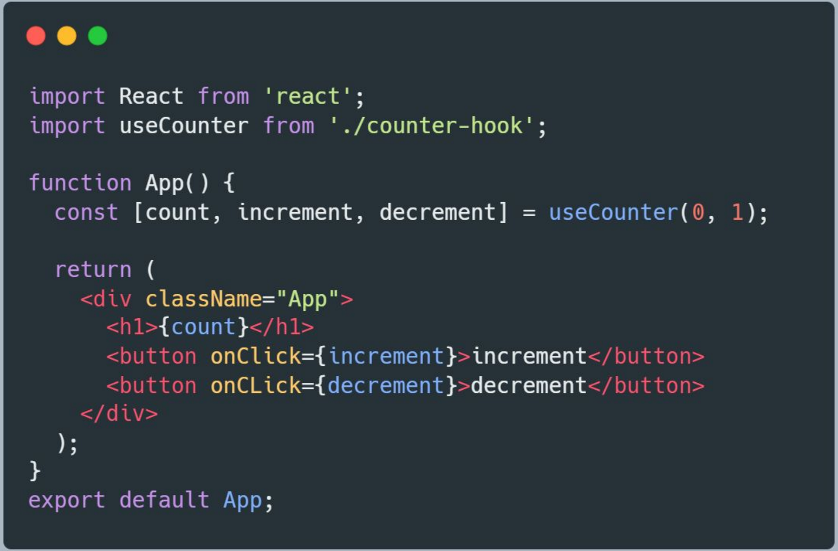


```
import { useState } from "react";

function useCounter(val:number, step:number) {
  const [count, setCount] = useState<number>(val);

  function increment() {
    setCount(count + step);
  }

  function decrement() {
    setCount(count - step);
  }

  return [count, increment, decrement];
}

export default useCounter;
```

Create your own hook

Let’s use the useCounter custom hook inside App.js component by importing it from the counter-hook.js file.

Now, we have a reusable counter logic we can use it whenever we need a counter in our react app.



```
import React from 'react';
import useCounter from './counter-hook';

function App() {
  const [count, increment, decrement] = useCounter(0, 1);

  return (
    <div className="App">
      <h1>{count}</h1>
      <button onClick={increment}>increment</button>
      <button onCLick={decrement}>decrement</button>
    </div>
  );
}
export default App;
```

Thank You!