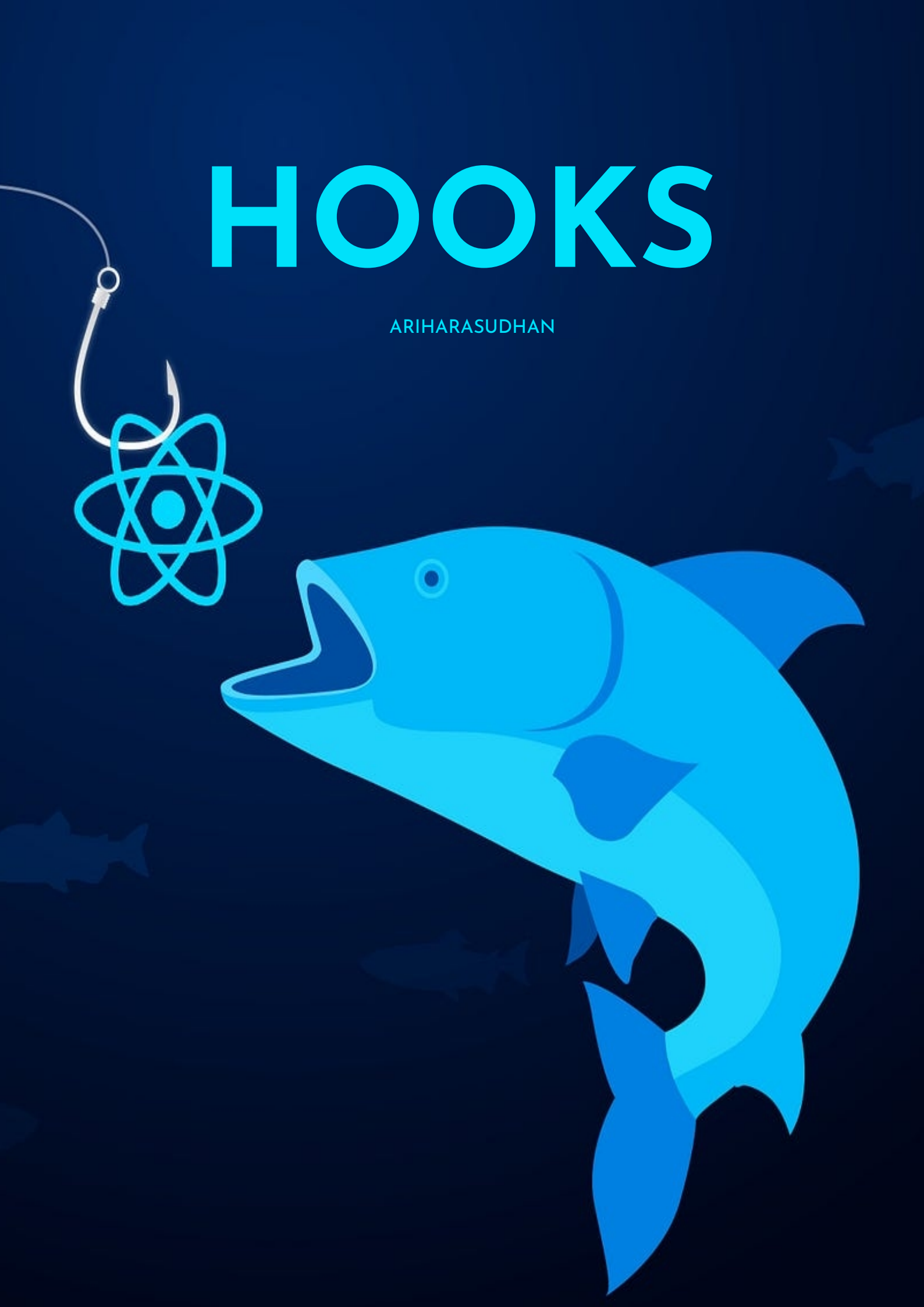
HOOKS

ARIHARASUDHAN



THIS BOOK

This book is written (typed) by Ari, who hails from the South and has keen interests in Computer Science, Biology, and Tamil Literature. Occasionally, he updates his website, where you can reach out to him.

https://arihara-sudhan.github.io

HOOKS'RE FUNCTIONS

Hooks are functions! In React, **hooks** are special functions that let us use state and other React Features in functional components (i.e., Without writing classes). A definition says, A **hook** is a feature in React that allows us to hook into React's state and lifecycle features in functional components. Hooks don't work inside classes. There are some problems in class components which are addressed by using Hooks.

WHY WE NEED HOOKS

Reason ONE: As we all know, Event Handlers must be bound (past of bind) with the current context. In JavaScript, the value of this inside a function is determined by how the function is called. Without binding, the value of this inside an event handler such as handleClick wouldn't refer to the class instance, but rather to the context in which the function was called. So, it must be bound.

BINDING IN CLASS COMPONENTS

```
constructor(props) {
    super(props);
    this.state = {
      count: 0,
    };
    this.handleClick = this.handleClick.bind(this);
}
```

Reason TWO: When using class components, React creates an instance of the class each time the component is rendered. Hot reloading often relies on replacing or updating the component definitions in memory without losing the existing state. Using Class components makes Hot-Reloading unreliable.

Reason THREE: There is no simple way to reuse the stateful class components. Some people use HOC and render props which requires restructuring of the components, making the code awkward looking.

A SIMPLE STATE HOOK

It was possible only with The Class Components to perform state management. But now, State Hook allows us to perform state management inside A Functional Component. If we want to make a counter using Class Component, it will be like:

```
class Component extends React.Component {
    constructor(props){
        super(props);
        this.state = {
            count: 0,
        }
    }

    increment = () => {
        this.setState({count: this.state.count+1})
    }

    decrement = () => {
        this.setState({count: this.state.count-1})
    }

    render() {
        return (
            <>
                <h1>Count: {this.state.count}</h1>
                <button onClick={this.increment}>INCREMENT</button>
                <button onClick={this.decrement}>DECREMENT</button>
            </>
        )
    }
}
```

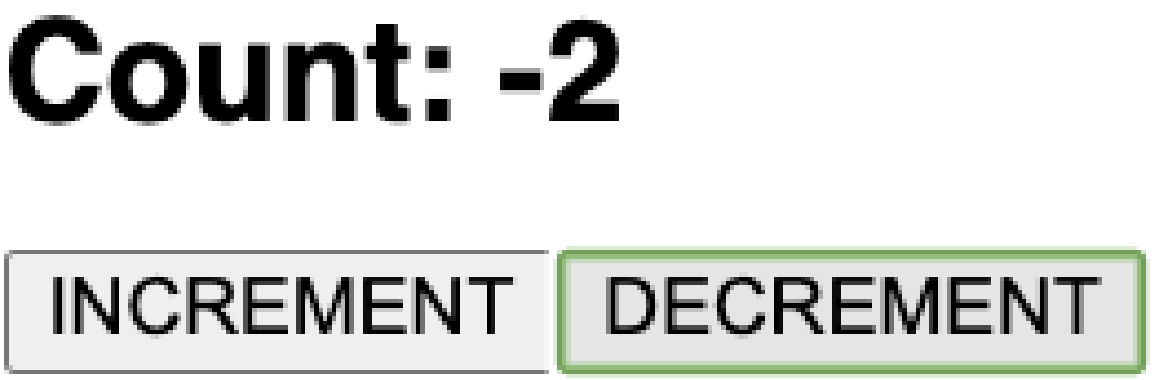
You can observe the complexities in creating a simple class component. It is quite easy in Functional Component with useState() Hook.

COUNTER: FUNCTIONAL COMPONENT

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

function Component() {
        const [count, setCount] = useState(0);
        return (
            <>
                <h1>Count: {count}</h1>
                <button onClick={()=>{setCount(count+1)}}>INCREMENT</button>
                <button onClick={()=>{setCount(count-1)}}>DECREMENT</button>
            </>
        )
}
```

As we have said, Hooks are simply functions. So, we just have to call them. The hook, useState is given a state value of 0. It will return the state and a function that can change the state. We de-structure and get those in a list.

Now, we can interact with these buttons.

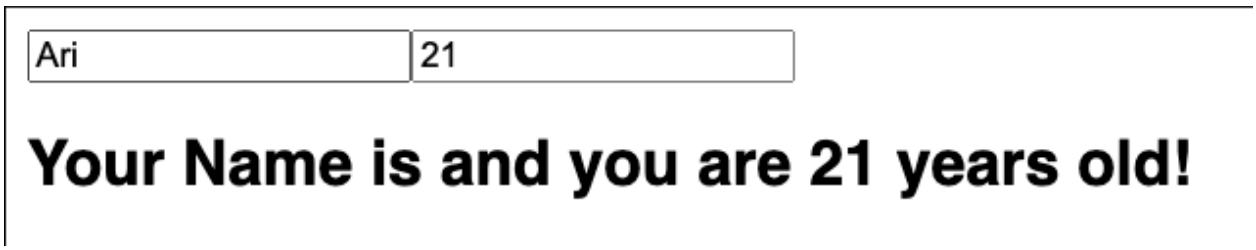


In this book, we expect you to understand that components are exported and rendered inside index.js :)

We can use the useState hook with objects simply.

USE STATE WITH OBJECTS

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

function Component() {
        const [user, setUser] = useState({
            name: "",
            age: ""
        });
        return (
            <>
                <input type="text" onChange={(e)=>{setUser({name: e.target.value})}}/>
                <input type="number" onChange={(e)=>{setUser({age: e.target.value})}}/>
                <h1>Your Name is {user.name} and you are {user.age} years old!</h1>
            </>
        )
}
```



This is pretty cool! Imagine the situation where we have to write the same for class components. Here, we have to notice one interesting stuff happening.

Let us just stringify the object and render to notice one property of the useState hook.

USE STATE WITH OBJECTS

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

function Component() {
        const [user, setUser] = useState({
            name: "",
            age: ""
        });
        return (
            <>
                <input type="text" onChange={(e)=>{setUser({name: e.target.value})}}/>
                <input type="number" onChange={(e)=>{setUser({age: e.target.value})}}/>
                <h1>Your Name is {user.name} and you are {user.age} years old!</h1>
                <h1>{JSON.stringify(user)}</h1>
            </>
        )
}
```

Now, if we look at the output, it is something like:

Your Name is and you are years old!

{"name":"","age":""}

But, when we change the first input box,

Ari

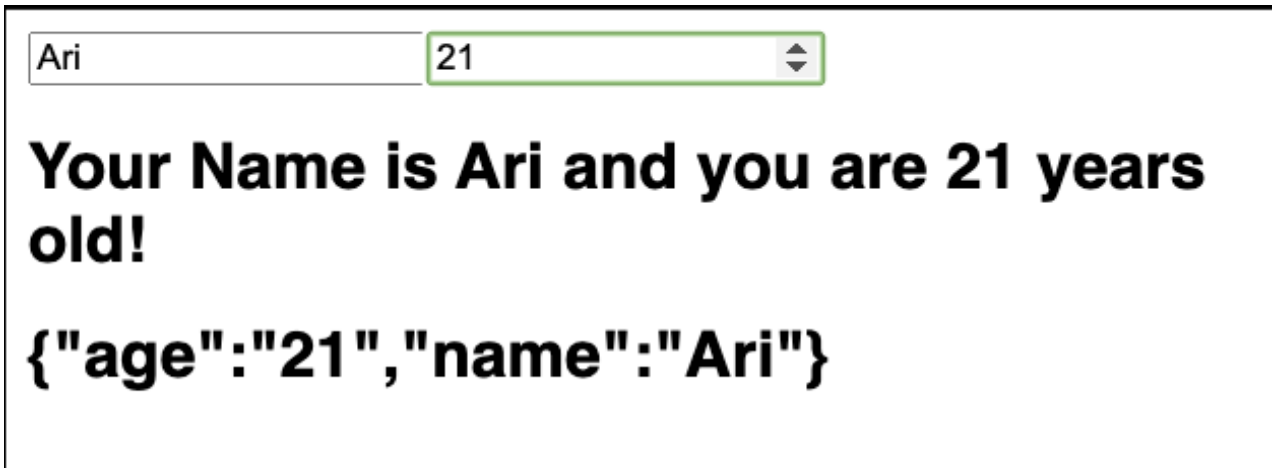
Your Name is Ari and you are years old!

{"name":"Ari"}

You can see that the object doesn't have the untouched state. This is because, the useState doesn't automatically merge and update the object. The this.setState of class components will do merge and update. In useState, we have to do it manually (maybe using a spread).

USE STATE WITH OBJECTS

```
function Component() {
        const [user, setUser] = useState({
            name: "",
            age: ""
        });
        return (
            <>
                <input type="text" onChange={(e)=>{setUser({ ... user, name:
e.target.value})}}/>
                <input type="number" onChange={(e)=>{setUser({ ... user, age:
e.target.value})}}/>
                <h1>Your Name is {user.name} and you are {user.age} years old!</h1>
                <h1>{JSON.stringify(user)}</h1>
            </>
        )
}
```



Let's write a useState hook that manages an array.

It outputs:

Ari

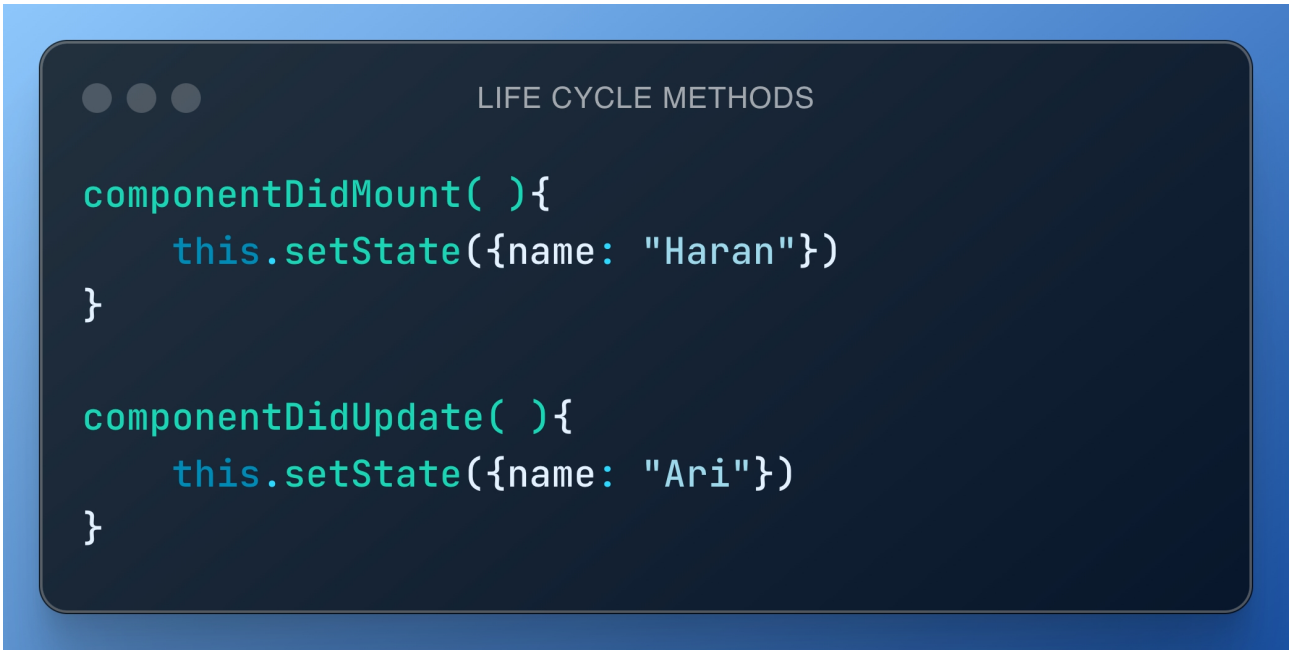
Haran

Sudhan

With useState, we don't need the state to be an object as in class components.

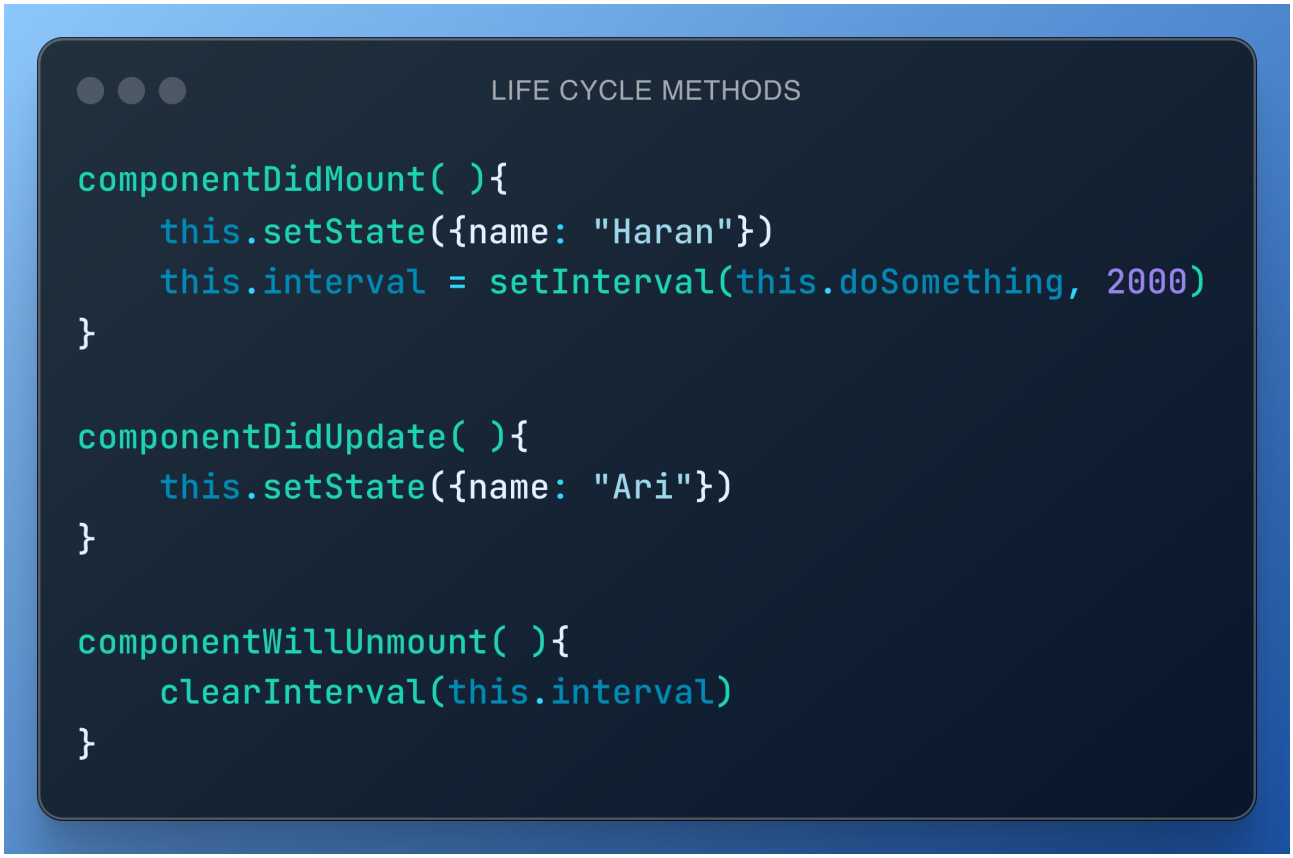
SIDE EFFECT HOOK

Updating the DOM on an event being happened, fetching data, setting timers are side effects. In Class Components, we achieve these all stuffs using Life Cycle Methods.



Here, If we observe one thing closer, we write the same code inside componentDidMount and componentDidUpdate Methods. We have to somehow optimize it!

Now, observe the following code.



Here, we have the code to set a state and to set an interval inside same method. This is not a good grouping. Secondly, the related parts , i.e., setting and clearing the interval, are not together. So, there is no proper grouping! And there is repetition!

The useEffect can be the apt solution for these all. It allows us to perform such side effects in Functional Components. It is a close replacement for

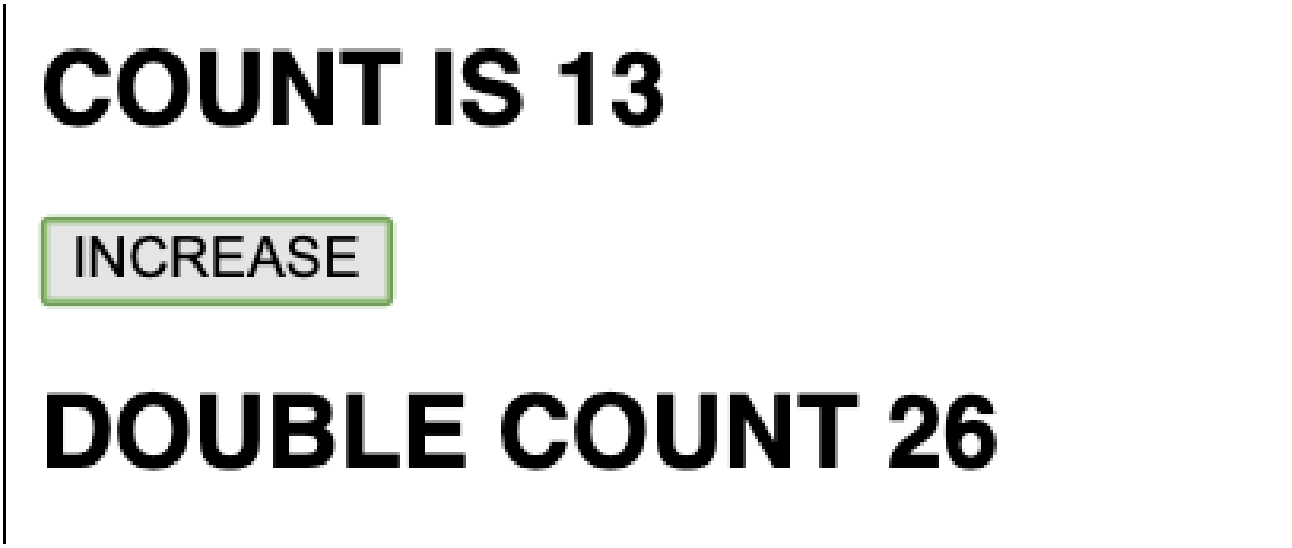
1. componentDidMount
2. componentDidUpdate
3. componentWillUnmount

Inside the useEffect, we pass a function that is executed whenever the component is updated, the component is mounted, the component gets unmounted. Yes! The useEffect hook executes the given function after every renders.

Following is an example where we update the state of the dblCount after component is initially rendered and also changed. The component changes when it's state changes.

USE EFFECT HOOK

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

function Component() {
        const [count, setCount] = useState(0);
        const [dblCount, setDblCount] = useState(0);
        useEffect(()=>{
            setDblCount(count*2)
        })
        return (
            <>
                <h1>COUNT IS {count}</h1>
                <button onClick = {()=>{setCount(count+1)}}>INCREASE</button>
                <h1>DOUBLE COUNT {dblCount}</h1>
            </>
        )
}
```

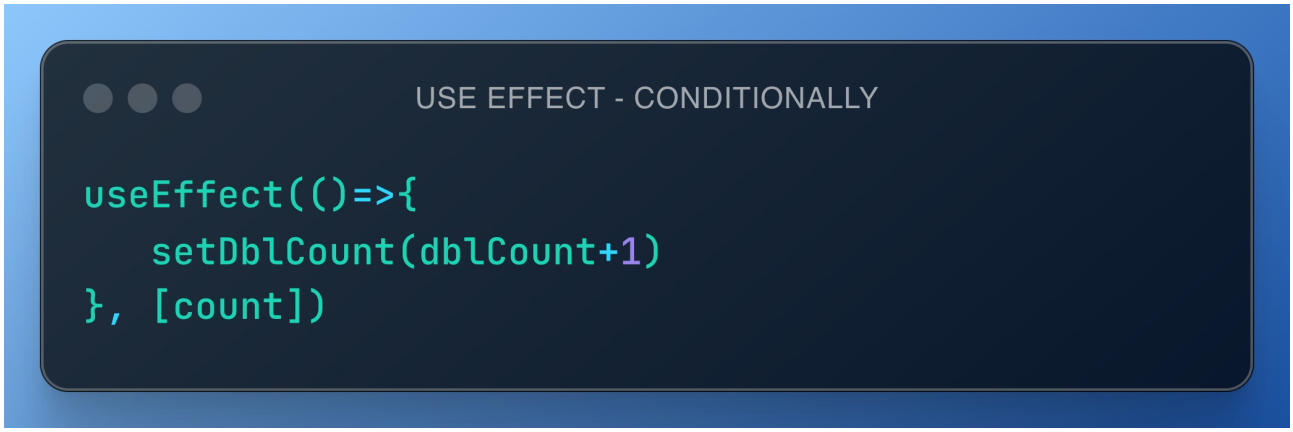


If we change our hook to do something like the following,

```
    useEffect(()=>{
        setCount(count+1)
     })
```

We can see the count being updated for every renders.

If we want to call the useEffect only when a particular state (count in our case) changes, we can tell it using an array. If we have a snippet like the following:



The dblCount will be incremented only when the count is changed.

If we don't specify the count in an array, it will update the dblCount forever for each renders. We can also have more states on changing which the useEffect will be called.

USE EFFECT - CONDITIONALLY

```
useEffect(()=>{
   setDblCount(dblCount+1)
}, [count, anotherCount])
```

If we want to call the useEffect only once, we can specify an empty array.

USE EFFECT - ONLY ONCE

```
useEffect(()=>{
   setDblCount(dblCount+1)
}, [])
```

The dblCount will be rendered only for the initial render.

As we have already seen, useEffect can be used even in component unmounting. We have to write something new inside the useEffect hook to perform cleanups!

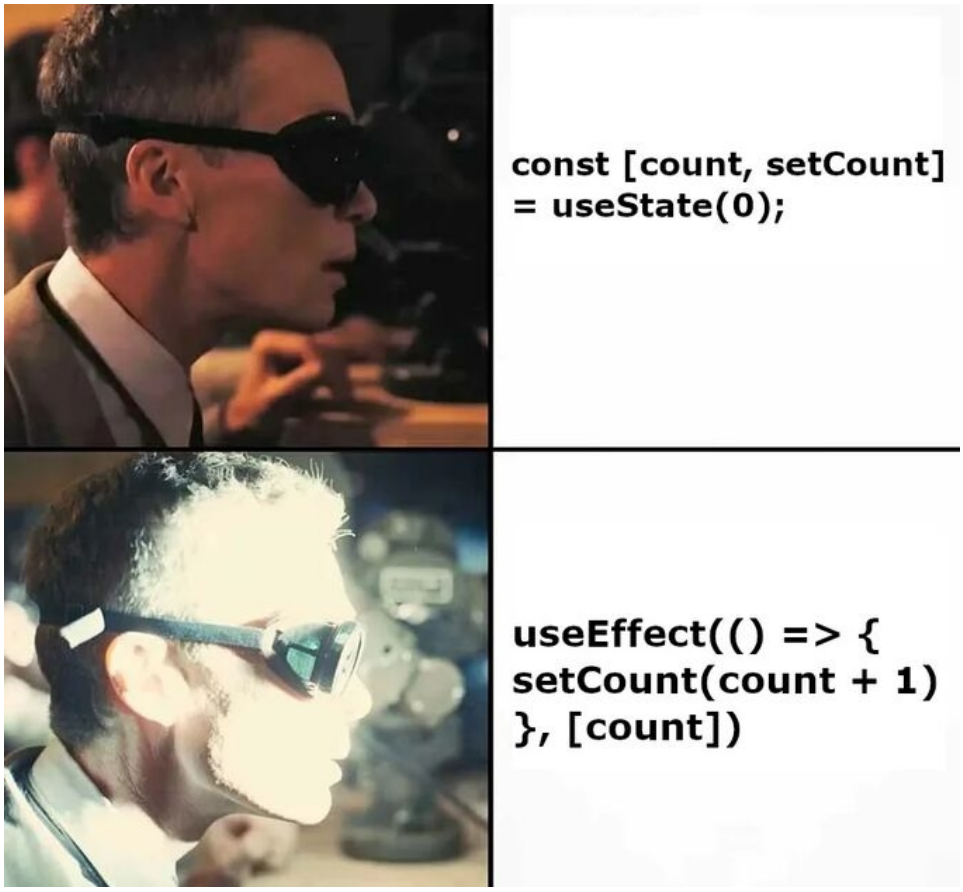
USE EFFECT - CLEAN UP

```
function Component() {
        const [pos, setPos] = useState([0,0])
        const [mX , setMX] = useState(0)
        const [mY, setMY] = useState(0)

        const logMouseButton = e => {
            setMX(e.clientX)
            setMY(e.clientY)
        }

    useEffect(()=>{
            window.addEventListener("mousemove", logMouseButton)
            return ()=>{
                window.removeEventListener('mousemove', logMouseButton)
            }
        }, [])

    useEffect(()=>{
        setPos([mX, mY])
    }, [mX, mY])

    return (
            <>
                <h1>MX: {pos[0]} MY: {pos[1]}</h1>
            </>
        )
}
```

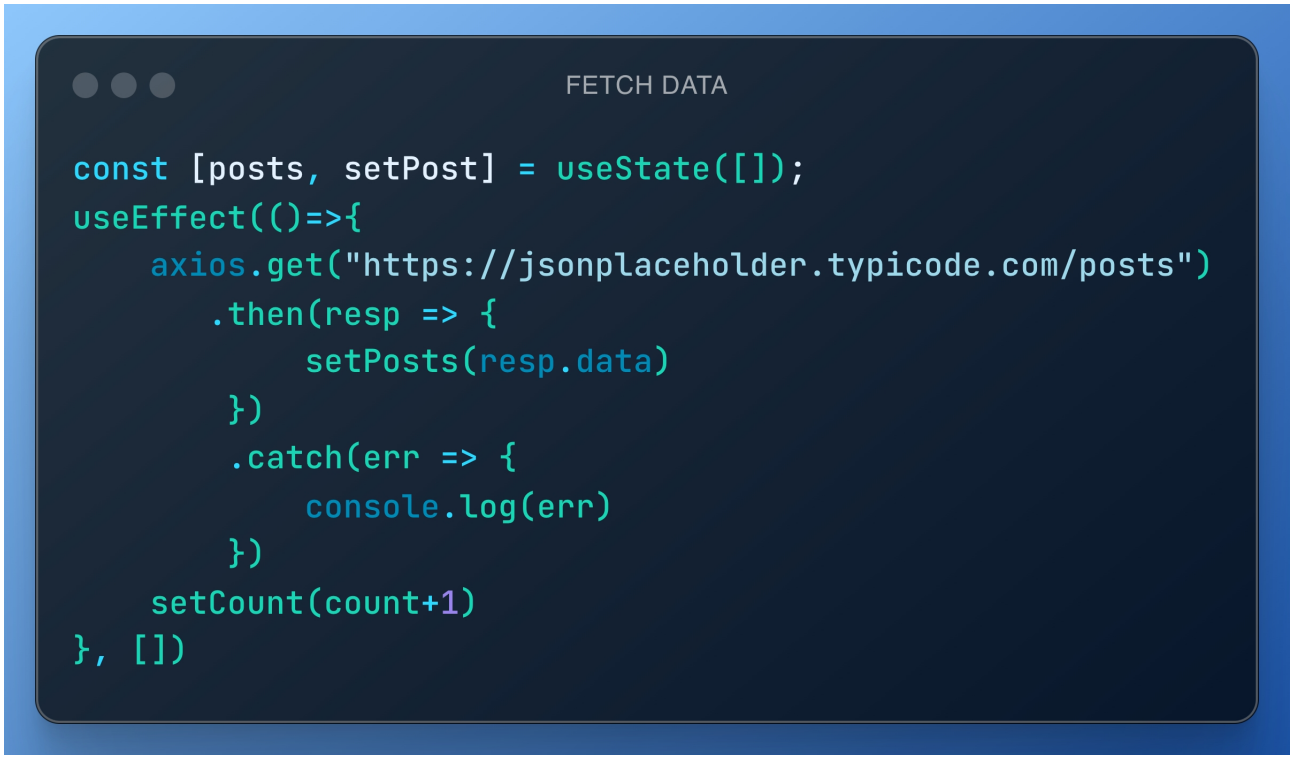
It will perform the cleanup when the component is removed.

MX: 174 MY: 138

We return the cleanup function inside the function!



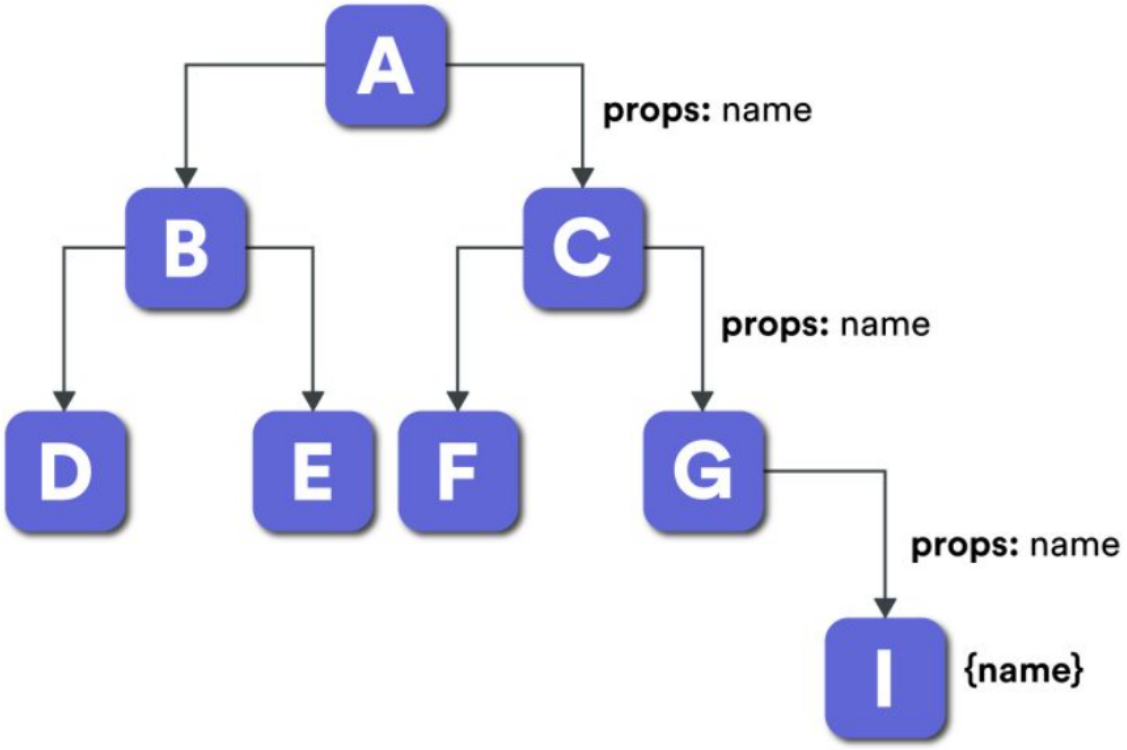
We can fetch data using useEffect.



No exception for conditionally fetching the data by specifying some dependencies.

CONTEXT HOOK

There will be some situations in which we have to pass the data to a component that is not a direct child component. If we want to pass data down through the component tree without having to pass data down at every level, I mean props drilling, Context Hook can be used.



```jsx
import React, { createContext } from "react";

const userContext = createContext();

function A() {
    return (
        <userContext.Provider value={"Ari"}>
            <B />
        </userContext.Provider>
    );
}

function B() {
    return (
        <>
            <C />
        </>
    );
}

function C() {
    return (
        <>
            <D />
        </>
    );
}

function D() {
    return (
        <userContext.Consumer>
            {name => {
                return <h1>HELLO {name}</h1>;
            }}
        </userContext.Consumer>
    );
}

export default function App() {
    return <A />;
}
```

Using this context hook, we don't need to pass the props to the deepest level. We can simply use the createContext and set the parent Provider, and set the child Consumer.

Inside the child consumer, it is a function that takes the value and manipulates. The way we consume seems to be a bit verbose! It becomes even more verbose with multiple contexts. Look at the following :)

CONTEXT HOOK

```
import React, { createContext } from "react";

const userContext = createContext();
const animalContext = createContext();

function A() {
    return (
        <animalContext.Provider value={"Tiger"}>
            <userContext.Provider value={"Ari"}>
                <D />
            </userContext.Provider>
        </animalContext.Provider>
    );
}

function D() {
    return (
        <userContext.Consumer>
            {userName => {
                return <animalContext.Consumer>
                    { animalName => {
                            return <h1>{userName} is a {animalName}</h1>
                        }
                    }
                </animalContext.Consumer>
            }}
        </userContext.Consumer>
    );
}

export default function App() {
    return <A />;
}
```

Ari is a Tiger

Just to consume, we have to write so much of nesting! How can we make it simple? There is a hook for this! The hook is useContext.

CONTEXT HOOK

```
import React, { createContext, useContext } from "react";
const userContext = createContext();

function A() {
    return (
        <userContext.Provider value={"Ari"}>
            <B/>
        </userContext.Provider>
    );
}

function B() {
    const name = useContext(userContext);
    return <h1>{name} is Here!</h1>
}

export default function App() {
    return <A/>;
}
```

Ari is Here!

A COMPLEX STATE HOOK

We have a hook, useState for state management. Meanwhile, useReducer is another hook for state management. The hook, useState is built upon useReducer. The hook, useReducer is related to reducers. We all know the reduce() method in arrays.

reduce in JavaScript	useReducer in React
array.**reduce**(***reducer***, initialValue)	**useReducer**(***reducer***, initialState)
singleValue = ***reducer***(accumulator, itemValue)	newState = ***reducer***(currentState, action)
reduce method returns a single value	**useReducer** returns a pair of values. [newState, dispatch]

The hook, useReducer takes the reducer function and an initial state.

The reducer function takes the current state and performs the specified action on the current state. Once the action is performed, a new state will be returned. The hook, useReducer returns the new state and a dispatch method. Well! It may be perplexing now! Let's see an example for a better grasping...

REDUCER

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

const initialState = 23;
const reducer = (state, action) => {
    switch (action) {
        case "increment":
            return state + 1;
        case "decrement":
            return state - 1;
        case "reset":
            return initialState;
        default:
            return state;
    }
};

function A() {
    const [newState, dispatch] = useReducer(reducer, initialState);
    return (
        <>
            <h1>STATE: {newState}</h1>
            <button onClick={() => dispatch("increment")}>INCREMENT</button>
            <button onClick={() => dispatch("reset")}>RESET</button>
            <button onClick={() => dispatch("decrement")}>DECREMENT</button>
        </>
    );
}

export default function App() {
    return <A />;
}
```

Now, we can perform the operations. State is reduced based on the action specified.

We can also use objects in action and state to make it look like redux.

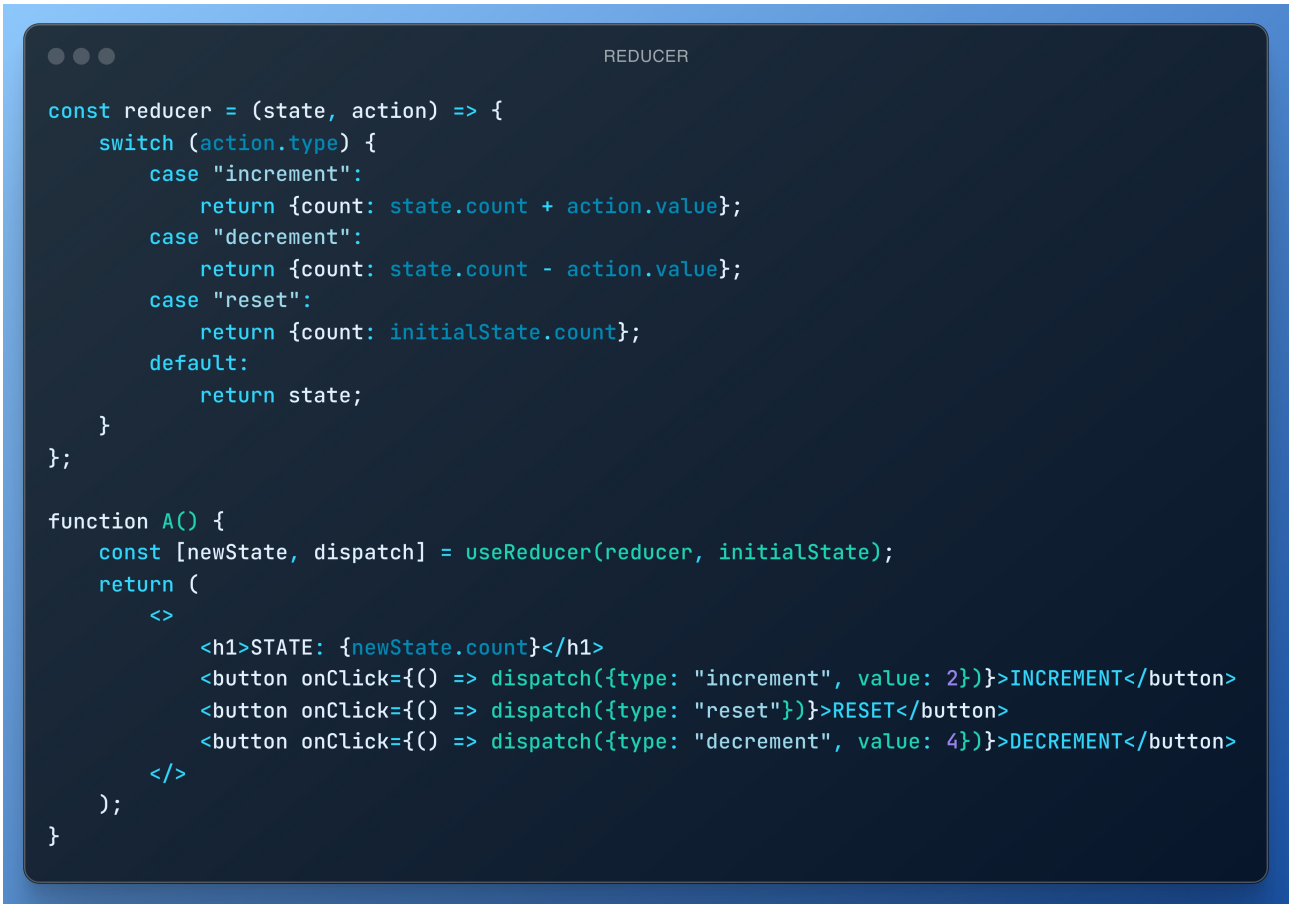
```
const initialState = {
    count: 0
};
const reducer = (state, action) => {
    switch (action.type) {
        case "increment":
            return {count: state.count + 1};
        case "decrement":
            return {count: state.count - 1};
        case "reset":
            return {count: initialState.count};
        default:
            return state;
    }
};

function A() {
    const [newState, dispatch] = useReducer(reducer, initialState);
    return (
        <>
            <h1>STATE: {newState.count}</h1>
            <button onClick={() => dispatch({type: "increment"})}>INCREMENT</button>
            <button onClick={() => dispatch({type: "reset"})}>RESET</button>
            <button onClick={() => dispatch({type: "decrement"})}>DECREMENT</button>
        </>
    );
}
```

Now also, the output is same.

STATE: -6

INCREMENT | RESET | DECREMENT

We can also pass a value in action. This is called, payload.



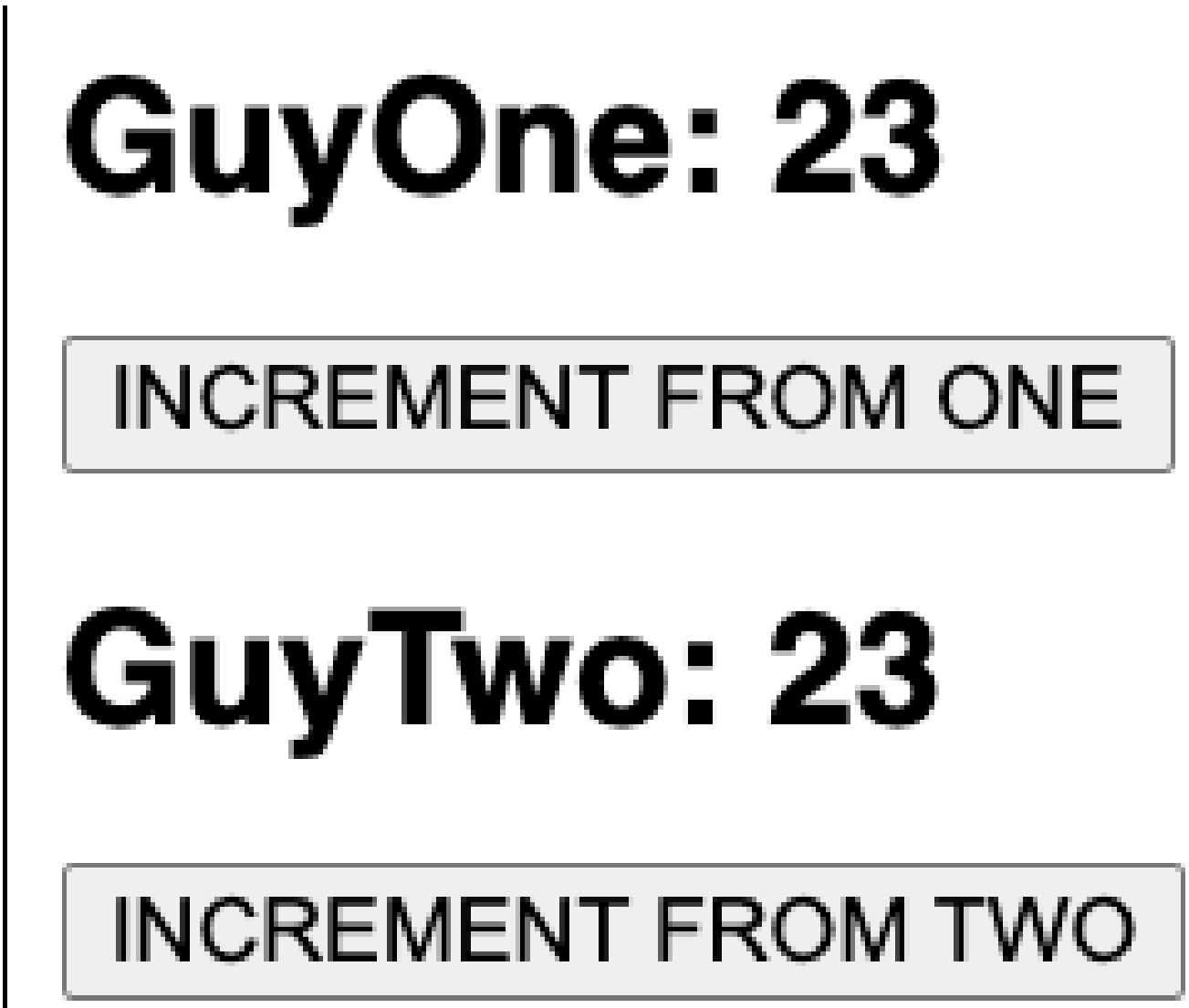
Now, increment and decrement happens with the payload given.

It reduces the risk of duplication. We just need to pass the data and the state is changed based on the value of the action. Without using reducers, we have to write the code for each logic separately.

We can use useReducer and useContext together if we want to keep the state global to components.

CONTEXT HOOK

```
import React, { createContext } from "react";

const userContext = createContext();
const animalContext = createContext();

function A() {
    return (
        <animalContext.Provider value={"Tiger"}>
            <userContext.Provider value={"Ari"}>
                <D />
            </userContext.Provider>
        </animalContext.Provider>
    );
}

function D() {
    return (
        <userContext.Consumer>
            {userName => {
                return <animalContext.Consumer>
                    { animalName => {
                            return <h1>{userName} is a {animalName}</h1>
                        }
                    }
                </animalContext.Consumer>
            }}
        </userContext.Consumer>
    );
}

export default function App() {
    return <A />;
}
```



This is how we can achieve this!

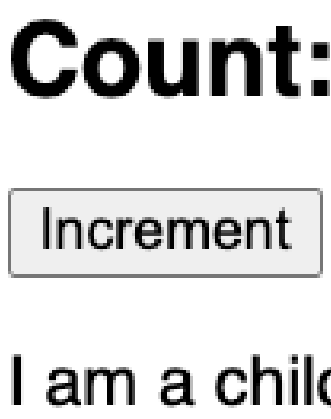
OPTIMIZATION HOOKS

We don't have to re-render the components unnecessarily. Changing the state or props of one component shouldn't cause all other components to re-render once again. Consider the following example, where changing one component results all components to be re-rendered.

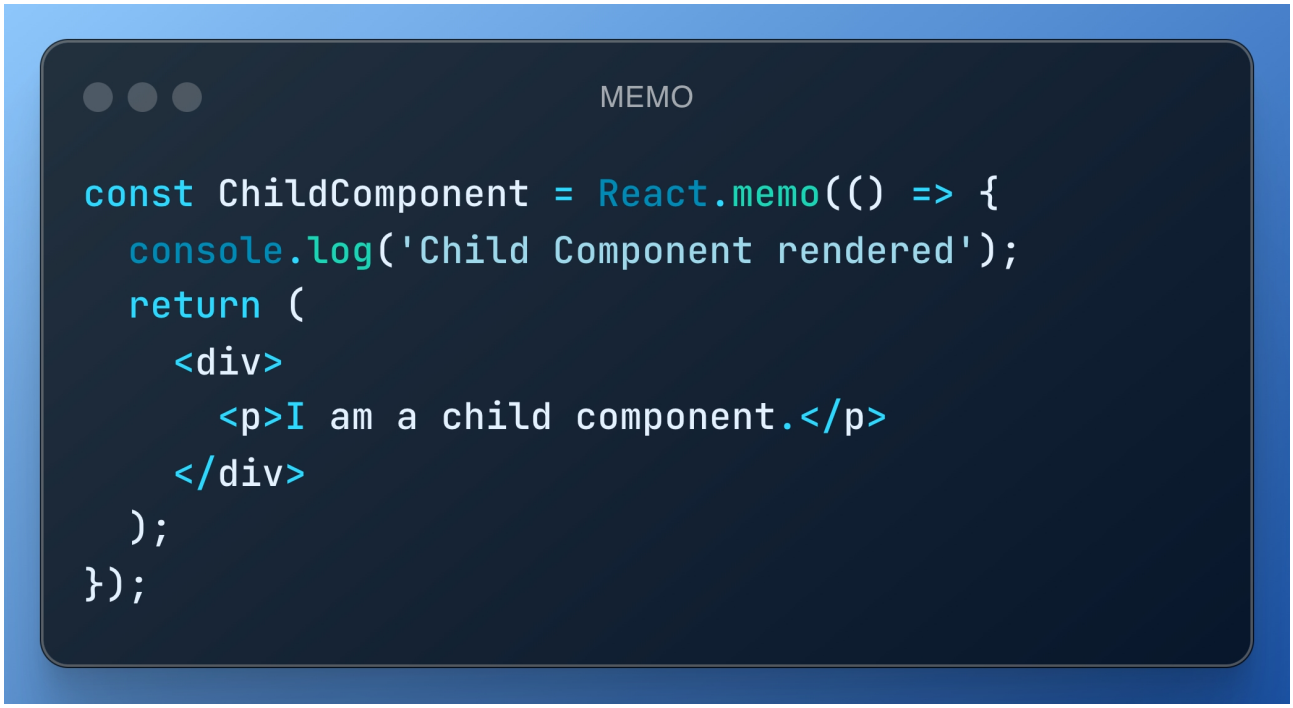
```
import React, { useState } from 'react';

const ChildComponent = () => {
  console.log('Child Component rendered');
  return (
    <div>
      <p>I am a child component.</p>
    </div>
  );
};
```

```jsx
const App = () => {
  const [count, setCount] = useState(0);

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

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={increment}>Increment</button>
      <ChildComponent />
      <ChildComponent />
      <ChildComponent />
    </div>
  );
};

export default App;
```

Here, when we update the count in App, the ChildComponents will be re-rendered. (Look at dev-tools log)



When we click this button, it re-renders the child components once again. To avoid this unnecessary renders, we can use a HOC, React.memo. We have to wrap the components in React.memo to avoid re-rendering. It caches the component and prevents re-rendering. If we wrap our component with React.memo, it won't re-render unless it's props are states get updated.



A brand new Function reference will be created for each render. To memoize a callback function, there is a hook called useCallback. This can be useful to avoid unnecessary re-renders of child components that depend on the callback.

useCallback Hook

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

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

  // Memoize the increment function so it only gets recreated when `count` changes
  const increment = useCallback(() => {
    setCount((prevCount) => prevCount + 1);
  }, [count]);

  return (
    <div>
      <h1>Count: {count}</h1>
      <ChildComponent onClick={increment} />
    </div>
  );
}

function ChildComponent({ onClick }) {
  console.log('ChildComponent re-rendered');

  return <button onClick={onClick}>Increment</button>;
}

export default ParentComponent;
```

The dependency array in useCallback (and other hooks like useEffect and useMemo) is crucial for controlling when the hook should re-run or re-create a function, effect, or memoized value.

Without dependency array, the useCallback behaves just the same as the normal function (which is created for each renders).

Another optimization hook is, useMemo. The useMemo hook in React is used to memoize the result of a computation, preventing expensive calculations from being re-executed unless necessary.

Without useMemo Hook

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

function App() {
  const [counter1, setCounter1] = useState(0);
  const [counter2, setCounter2] = useState(0);

  const heavyComputation = (count) => {
    console.log("Running heavy computation...");
    let i = 0;
    while (i < 10000) i++; // Simulates delay
    return count * 2;
  };

  // Computed value without memoization
  const computedValue = heavyComputation(counter1);

  return (
    <div>
      <h1>Counter 1: {counter1}</h1>
      <h2>Computed Value: {computedValue}</h2>
      <button onClick={() => setCounter1(counter1 + 1)}>Increment Counter 1</button>

      <h1>Counter 2: {counter2}</h1>
      <button onClick={() => setCounter2(counter2 + 1)}>Increment Counter 2</button>
    </div>
  );
}

export default App;
```

In the above snippet, we perform a heavy computation which makes the UI slower. The value computed depends only on the counter1. But, it makes it slower even when counter2 is updated. To avoid this unnecessary computation, we can use the useMemo hook.

useMemo Hook

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

function App() {
      const [counter1, setCounter1] = useState(0);
      const [counter2, setCounter2] = useState(0);

    // Simulate a heavy computation for counter1
      const heavyComputation = (count) => {
        console.log("Running heavy computation...");
        let i = 0;
        while (i < 2000000) i++; // Simulates delay
        return count * 2;
      };

      const computedValue = useMemo(() => heavyComputation(counter1), [counter1]);

      return (
        <div>
          <h1>Counter 1: {counter1}</h1>
          <h2>Computed Value (Counter 1 * 2): {computedValue}</h2>
          <button onClick={() => setCounter1(counter1 + 1)}>Increment Counter 1</button>

          <h1>Counter 2: {counter2}</h1>
          <button onClick={() => setCounter2(counter2 + 1)}>Increment Counter 2</button>
        </div>
      );
    }

export default App;
```

Now, there will be no delay when counter2 is updated. We have specified in the useMemo as, calculate only when counter1 is updated.

OPTIMIZATION HOOKS

The useRef is another hook by means of which we can persist or preserve values between re-renders. The useRef hook takes an initial value of any type as argument and returns an object with a single current property. We can also use it for accessing DOM Elements.

useRef - DOM Method

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

export default function App( ) {
    const inputRef = useRef(null)
    const [letMeType, setLetMeType] = useState(false);
    useEffect(()=>{
        inputRef.current.focus()
    }, [letMeType])
    return (
        <>
            <button onClick={()=>{letMeType?setLetMeType(false):setLetMeType(true)}}>CHANGE
LET ME TYPE</button>
            <input ref={inputRef} placeHolder="Name" type="text"/>
        </>
    )
}
```

Here, the input box is focused when the state, letMeType is changed by the button click.

CHANGE LET ME TYPE | Ari

But, this is not only for calling the DOM Methods. As we have already discussed, the main purpose of using useRef is to persist or preserve mutable values across re-renders. In the following example, the value is changing every second. We can't persist the interval because it is re-created on each render.

useRef - VALUES PERSISTING ACROSS RERENDERS

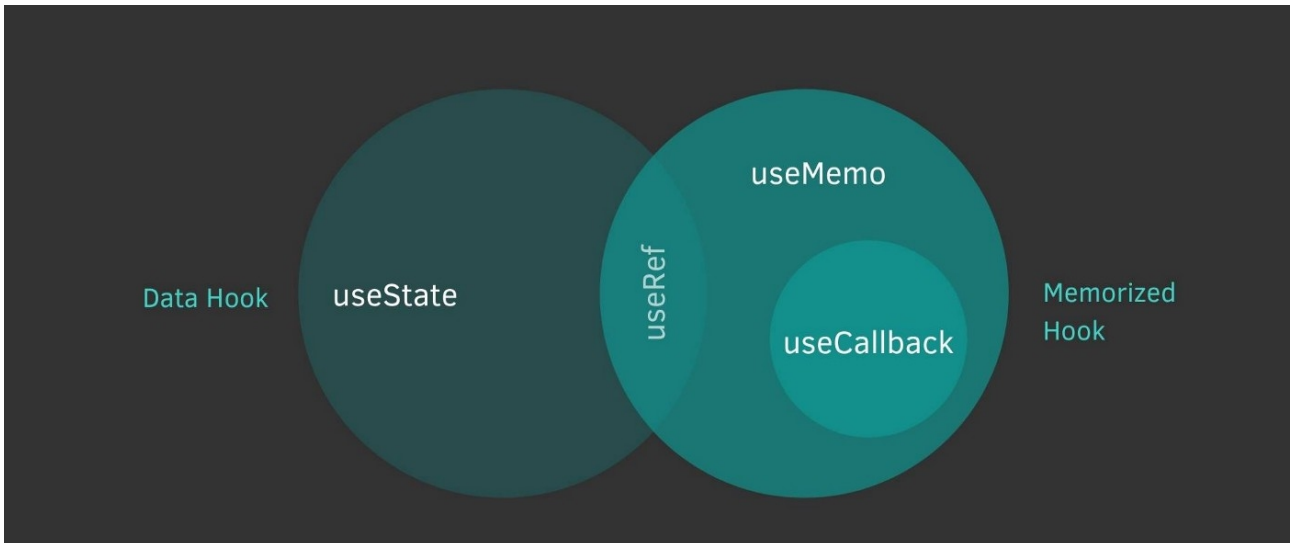
```
import { useState, useEffect } from "react";

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

    useEffect(() => {
        const interval = setInterval(() => {
            setCount(prevCount => prevCount + 1);
        }, 1000);
        return () => clearInterval(interval);
    }, []);

    return (
        <>
            <h1>COUNT : {count}</h1>
            <button onClick={()=>{clearInterval(interval)}}>REMOVE EVENT LISTENER</button>
        </>
    );
}
```

Therefore, we can't remove it with a button click, as the interval is defined inside the useEffect hook, and the reference changes with each render. To persist the interval across re-renders, we can use the useRef hook.

useRef - VALUES PERSISTING ACROSS RERENDERS

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

export default function App() {
    const [count, setCount] = useState(0);
    const intervalRef = useRef(null);

    useEffect(() => {
        intervalRef.current = setInterval(() => {
            setCount(prevCount => prevCount + 1);
        }, 1000);

        return () => clearInterval(intervalRef.current);
    }, []);

    const handleClick = () => {
        clearInterval(intervalRef.current);
    };

    return (
        <>
            <h1>COUNT : {count}</h1>
            <button onClick={handleClick}>REMOVE INTERVAL</button>
        </>
    );
}
```

Now, the we can access the intervalRef.

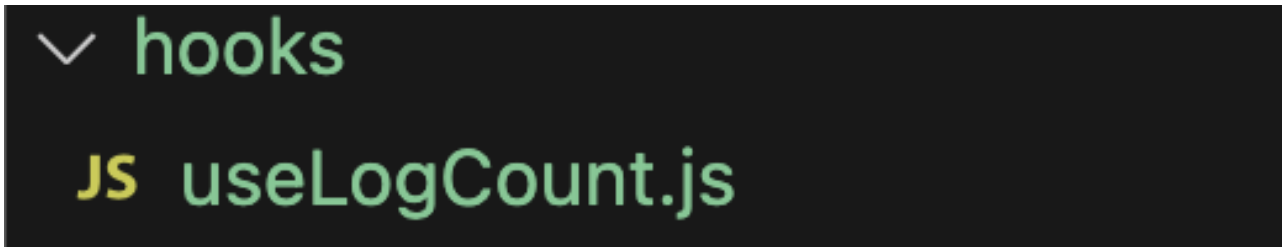


The useRef hook can be seen as a combination of useState and useCallback/useMemo because it allows you to persist a mutable value across re-renders without causing re-renders like useState does. Unlike useState, which triggers a re-render when its value changes, useRef holds a value that doesn't affect the component's render cycle.

Similar to how useCallback and useMemo memoize functions or computed values to prevent unnecessary re-creations, useRef preserves a reference to a mutable value or DOM element across renders, offering a stable and persistent reference throughout the component's lifecycle.

CUSTOM HOOKS

We can write our own hooks. If we want to write a hook that alerts the count whenever the count is updated, yes! We can! Let's write it under the hooks folder.



The hook is written to alert whenever the count is updated.

useLogCount: A CUSTOM HOOK

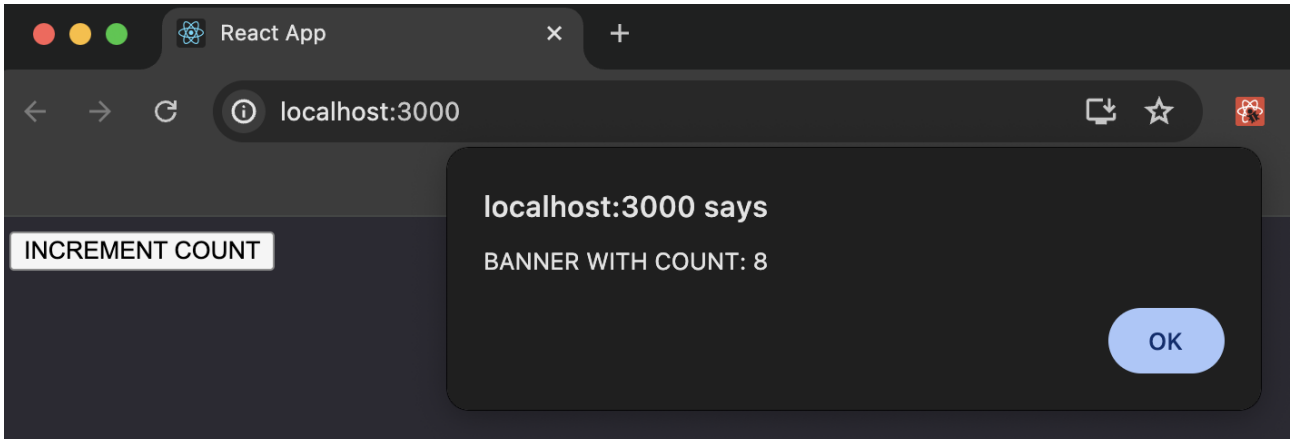
```
import { useEffect } from "react";

export function useLogCount(count){
    useEffect(()=>{
        alert(`BANNER WITH COUNT: ${count}`);
    }, [count])
}
```

Now, we can use this in our components.

Using Custom Hook: useLogCount

```
import { useLogCount } from "./hooks/useLogCount";
import { useState } from "react";

function App() {
  const [count, setCount] = useState(0)
  useLogCount(count)
  return (
    <button onClick={()=>{setCount(count+1)}}>INCREMENT COUNT</button>
  );
}

export default App;
```

It will alert whenever the count is updated.



Just like we make complex state logics using basic GATES, we can create complex hooks using basic hooks.

Following demonstrates creating a simple counter hook.

A Custom Hook: useCounter

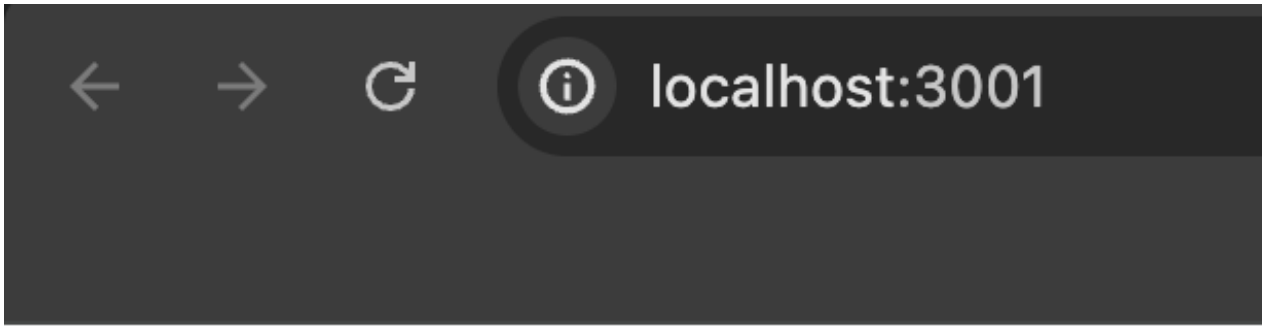
```
import {useState} from "react";

export function useCounter(_count, value) {
    const [count, setCount] = useState(_count);
    const increment = ()=>{
        setCount(count+value)
    }
    const decrement = ()=>{
        setCount(count+value)
    }
    const reset = ()=>{
        setCount(_count)
    }
    return [count, increment, decrement, reset];
}
```

One of the cool stuff about hooks is it's reusability (A hook being a function).

We can use this in our component.

Using useCounter

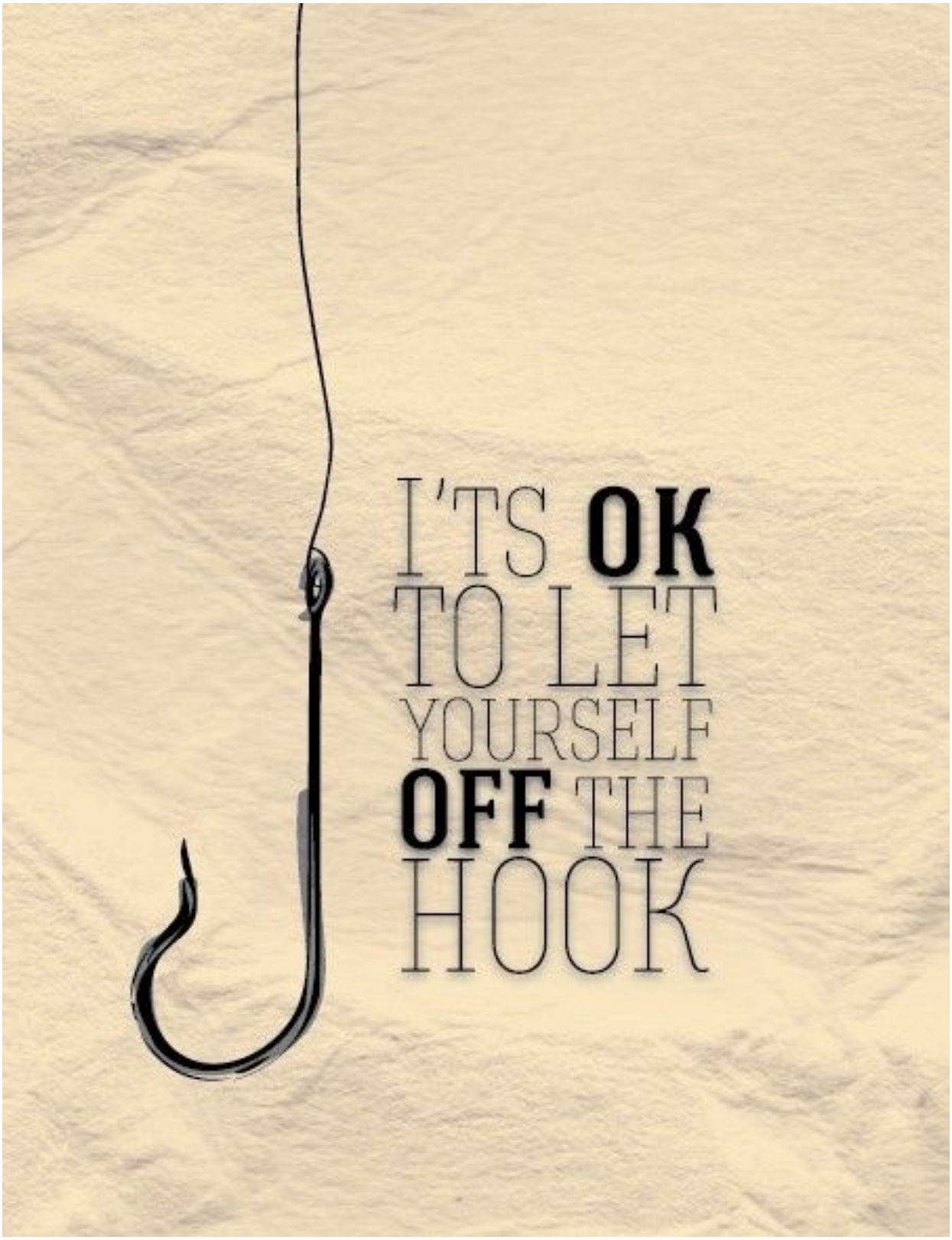
```
import { useCounter } from "./hooks/useCounter"

function App() {
  const [count, inc, dec, res] = useCounter(0,2)
  return (
    <>
      <h1>COUNT: {count}</h1>
      <button onClick={inc}>INCREMENT</button>
      <button onClick={dec}>DECREMENT</button>
      <button onClick={res}>RESET</button>
    </>
  );
}

export default App;
```



COUNT: 2

INCREMENT DECREMENT RESET

Hope you guys really enjoyed HOOKS!



MERCI