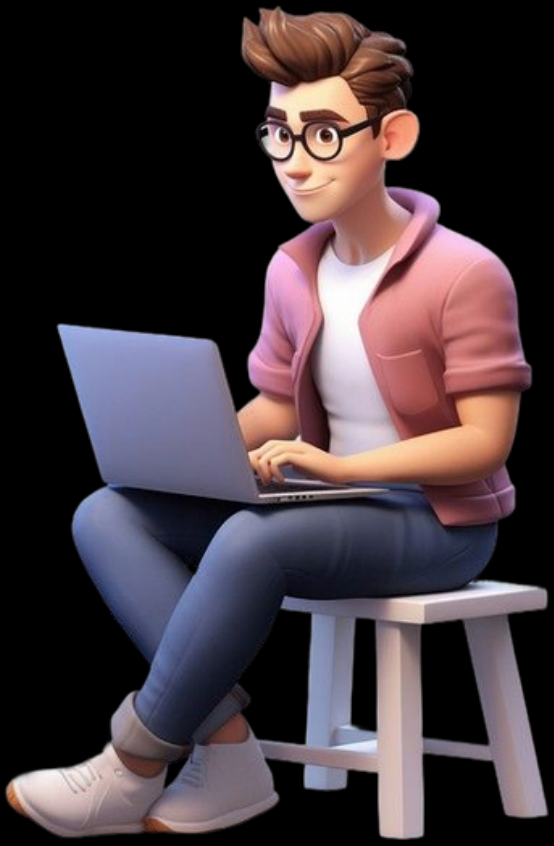
REACT HOOKS

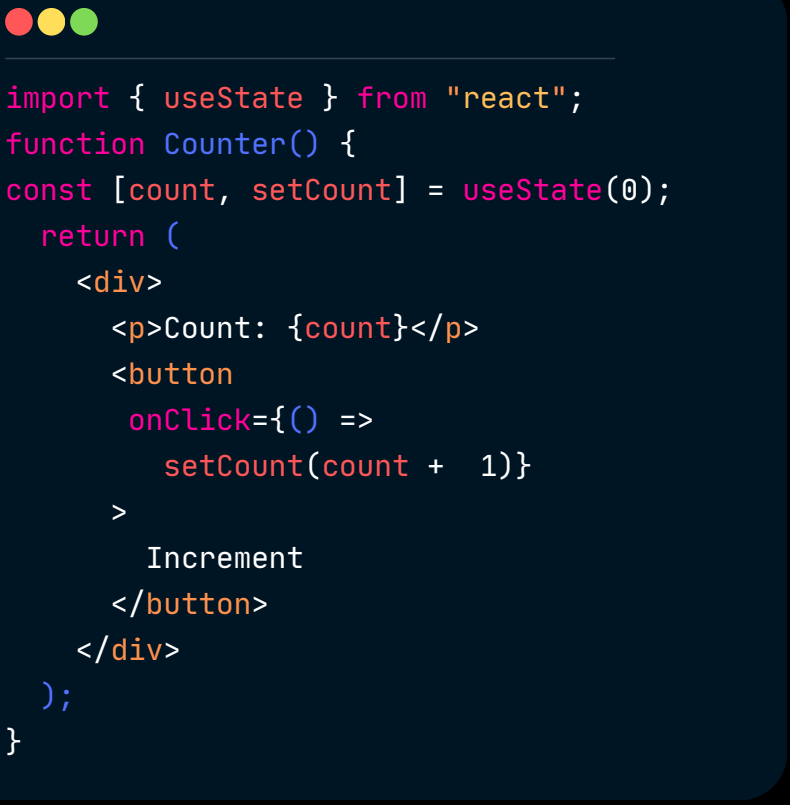


WEBGHOUL

USESTATE

Managing State in Functional Components

`useState` lets you add state to function components. It returns a stateful value and a function to update it.



```
import { useState } from "react";
function Counter() {
const [count, setCount] = useState(0);
  return (
    <div>
      <p>Count: {count}</p>
      <button
       onClick={() =>
         setCount(count +  1)}
      >
        Increment
      </button>
    </div>
  );
}
```

Handling Side Effects in React

`useEffect` runs side effects like fetching data, subscriptions, or manual DOM manipulations.

```
import { useEffect, useState } from
"react";
function FetchData() {
  const [data, setData] =
useState(null);
  useEffect(() => {
fetch("https://jsonplaceholder.typicode.
com/todos/1").then((response) =>
    response.json()
  ).then((json) => setData(json));
  }, []);
  return (
    <pre>
      {JSON.stringify(data, null, 2)}
    </pre>;
  )
}
```

USECONTEXT

Accessing Global State with Context

`useContext` helps to use values from React Context without prop drilling.

```
import { createContext, useContext }
from "react";
const ThemeContext =
createContext("light");
function ThemedButton() {
  const theme =
useContext(ThemeContext);
  return <button style={{ background:
theme === "dark" ? "#333" : "#fff"
}}>Click Me</button>;
}
function App() {
  return (
    <ThemeContext.Provider value="dark">
      <ThemedButton />
    </ThemeContext.Provider>
  );
}
```

USEREF

Managing DOM References and Preserving Values

`useRef` creates a mutable object that persists across renders without causing re-renders.

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

function FocusInput() {
  const inputRef = useRef(null);
  return (
    <div>
      <input ref={inputRef} />
      <button onClick={() =>
inputRef.current.focus()}>
       Focus
      </button>
    </div>
  );
}
```

State Management with Reducers

`useRef` is an alternative to `useState` for complex state logic.

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

function FocusInput() {
  const inputRef = useRef(null);
  return (
    <div>
      <input ref={inputRef} />
      <button onClick={() =>
inputRef.current.focus()}>
       Focus
      </button>
    </div>
  );
}
```

USEMEMO

Optimizing Performance with Memoization

`useMemo` caches expensive calculations between renders.

```
import { useMemo, useState } from
"react";
function ExpensiveCalculation({ num }) {
  const squared = useMemo(() => {
    console.log("Calculating...");
    return num * num;
  }, [num]);
  return <p>Squared: {squared}</p>;
}
function App() {
  const [num, setNum] = useState(2);
  return (
    <div>
      <ExpensiveCalculation num={num} />
      <button onClick={() => setNum(num
+ 1)}>Increase</button>
    </div>
  );
}
```

USECALLBACK

Memoizing Functions to Prevent Unnecessary Re-Renders

`useCallback` memoizes functions to avoid unnecessary recreations.

```
import { useState, useCallback } from
"react";
function Button({ handleClick }) {
  return <button onClick=
{handleClick}>Click Me</button>;
}
function App() {
  const [count, setCount] = useState(0);
  const increment = useCallback(() => {
    setCount((prev) => prev + 1);
  }, []);

  return (
    <div>
      <p>Count: {count}</p>
      <Button handleClick={increment} />
    </div>
  );
}
```

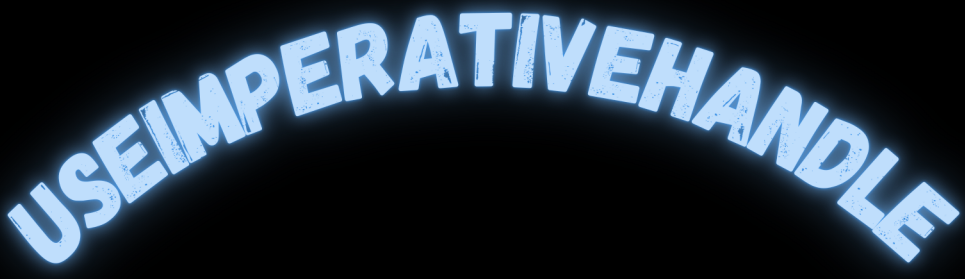
Syncing with the DOM Before Paint

`useLayoutEffect` runs synchronously after DOM mutations but before the browser repaints.

```
import { useLayoutEffect, useRef } from
"react";

function Box() {
  const boxRef = useRef(null);
  useLayoutEffect(() => {
    boxRef.current.style.transform =
"scale(1.2)";
  }, []);

  return <div ref={boxRef} style={{
width: 100, height: 100, background:
"blue" }} />;
}
```



Syncing with the DOM Before Paint

`useLayoutEffect` runs synchronously after DOM mutations but before the browser repaints.

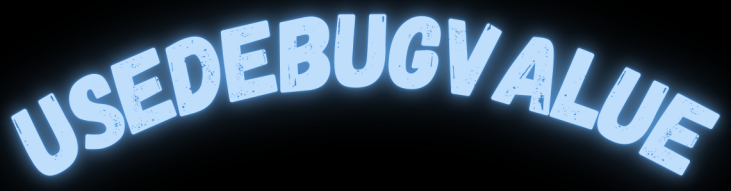
```
import { forwardRef,
useImperativeHandle, useRef } from
"react";

const CustomInput = forwardRef((_, ref)
=> {
  const inputRef = useRef();

  useImperativeHandle(ref, () => ({
    focus: () =>
inputRef.current.focus(),
  }));

  return <input ref={inputRef} />;
});

function App() {
  const inputRef = useRef();
  return (
    <div>
      <CustomInput ref={inputRef} />
      <button onClick={() =>
inputRef.current.focus()}>Focus
Input</button>
    </div>
  );
}
```



Debugging Custom Hooks

`useDebugValue` helps to display debug information in React DevTools.

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

function useCustomHook(value) {
  const [state, setState] =
useState(value);
  useDebugValue(state > 5 ? "High" :
"Low");
  return [state, setState];
}
function App() {
  const [count, setCount] =
useCustomHook(3);
  return <button onClick={() =>
setCount(count + 1)}>Count: {count}
</button>;
}
```

USEID

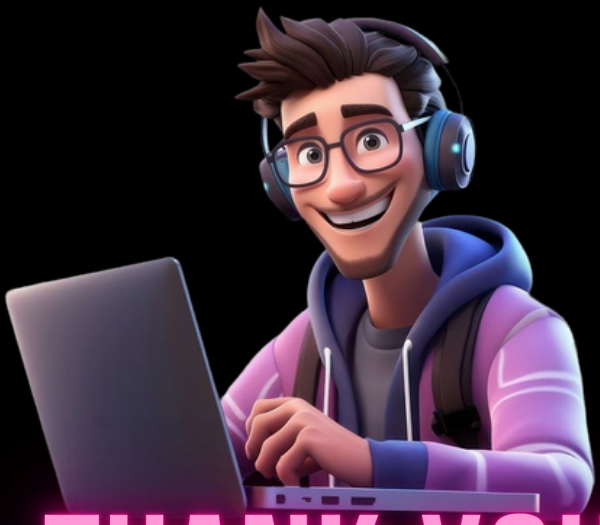
generates stable unique IDs for elements.

`useId` generates stable unique IDs for elements.

```tsx
import { useId } from "react";

function Form() {
  const id = useId();

  return (
    <div>
      <label htmlFor={id}>Name:</label>
      <input id={id} type="text" />
    </div>
  );
}
```



THANK YOU