

React Hooks Cheatsheet



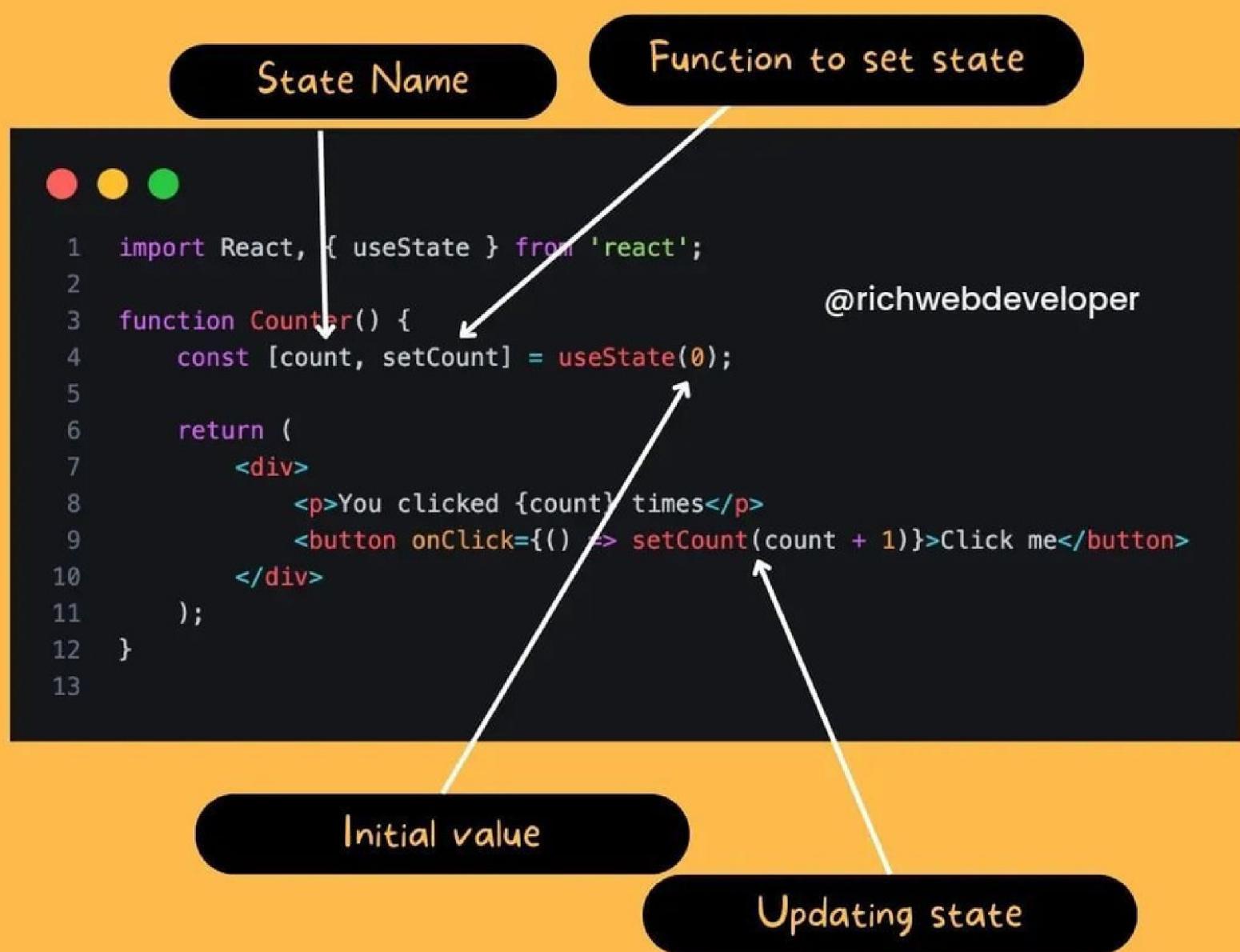
@richwebdeveloper



1. useState

This Hook allows you to add state to functional components. The useState function returns a pair: the current state and a function that updates it.

Example:



2. useEffect

This Hook lets you perform side effects in functional components. It's a close replacement for componentDidMount, componentDidUpdate, and componentWillUnmount in class components.

Example:


```
1 import React, { useState, useEffect } from 'react';
2
3 function Counter() {
4     const [count, setCount] = useState(0);          @richwebdeveloper
5
6     useEffect(() => {
7         document.title = `You clicked ${count} times`;
8
9         return () => {
10             document.title = `React App`;
11         };
12     }, [count]);
13
14     return (
15         <div>
16             <p>You clicked {count} times</p>
17             <button onClick={() => setCount(count + 1)}>Click me</button>
18         </div>
19     );
20 }
21
```

Dependancy

This code will be executed when
dependancy updated

3. useContext

This Hook lets you subscribe to React context without introducing nesting. It accepts a context object and returns the current context value for that context.

Example:

Setting up context



@richwebdeveloper

```
1 import React, { useContext } from 'react';
2 const ThemeContext = React.createContext('light');
3
4 function ThemedButton() {
5     const theme = useContext(ThemeContext);
6     return <button theme={theme}>I am styled by theme context!</button>;
7 }
8
```



Using context (You can use it any component)



@richwebdeveloper



4. useReducer

An alternative to useState. Accepts a reducer of type `(state, action) => newState`, and returns the current state paired with a dispatch method.

Example:

It is particularly useful when the state logic is complex and involves multiple sub-values, or when the next state depends on the previous one. useReducer is an alternative to useState.

While useState is great for handling independent pieces of state, useReducer excels at handling more complex, interconnected state that involves multiple changes in an atomic and predictable manner.

Example(next slide):



@richwebdeveloper



Code to update state of dispatched action

```
1 import React, { useReducer } from 'react';
2
3 const initialState = {count: 0};
4
5 function reducer(state, action) {
6     switch (action.type) {
7         case 'increment':
8             return {count: state.count + 1};
9         case 'decrement':
10            return {count: state.count - 1};
11        default:
12            throw new Error();
13    }
14 }
15
16 function Counter() {
17     const [state, dispatch] = useReducer(reducer, initialState);
18     return (
19         <>
20             Count: {state.count}
21             <button onClick={() => dispatch({type: 'decrement'})}>-</button>
22             <button onClick={() => dispatch({type: 'increment'})}>+</button>
23         </>
24     );
25 }
```

@richwebdeveloper

updated state value

Updating state with action

5. useRef

This Hook creates a mutable ref object whose .current property is initialized to the passed argument. It's handy for keeping any mutable value around similar to how you'd use instance fields in classes.

Example:

Initialising ref

```
1 import React, { useRef } from 'react'  
2  
3 function TextInputWithFocusButton() {  
4     const inputEl = useRef(null);  
5     const onButtonClick = () => {  
6         inputEl.current.focus();  
7     };  
8     return (  
9         <>  
10            <input ref={inputEl} type="text" />  
11            <button onClick={onButtonClick}>Focus the input</button>  
12        </>  
13    );  
14}  
15
```

Attaching the ref to access
the element

@richwebdeveloper

Updating value



@richwebdeveloper



DID YOU FIND IT USEFUL?



Follow @Ch Bappy

