

[Open in app](#)

Asbar Ali

93 Followers

[About](#)[Follow](#)

You can now subscribe to get stories delivered directly to your inbox.

[Got it](#)

React Hooks — How To Use useState and useEffect Example



Asbar Ali Dec 10, 2018 · 10 min read

Today I am going to talk about the newly introduced react hook. So I thought, it would be easy for you guys to understand if I described under these subtopics.

1. What is React Hook?

2. Why React Hook?

3. Examples

4. Rules

01. What is react hook?

React hook is newly introduced at react conference and it's available in react's alpha version 16.7. The React team is collecting feedback for React Hooks which is your opportunity to contribute to this feature.

It mainly uses to handle the state and side effects in react functional component.

States in a functional component? So what does that mean? If you're familiar with react for a while, you must know that the functional component has been called as a functional stateless component.

In React 14 introduce this feature and said,

- *It Enforces best practices*
- *Easy to under understand*
- *easy to test*
- *It increases the performance and so on.*

Increase performance before React 16

```
import React from 'react';
import PropTypes from 'prop-types';
import {View, Text} from 'react-native';

import styles from './ClubCard.styles';

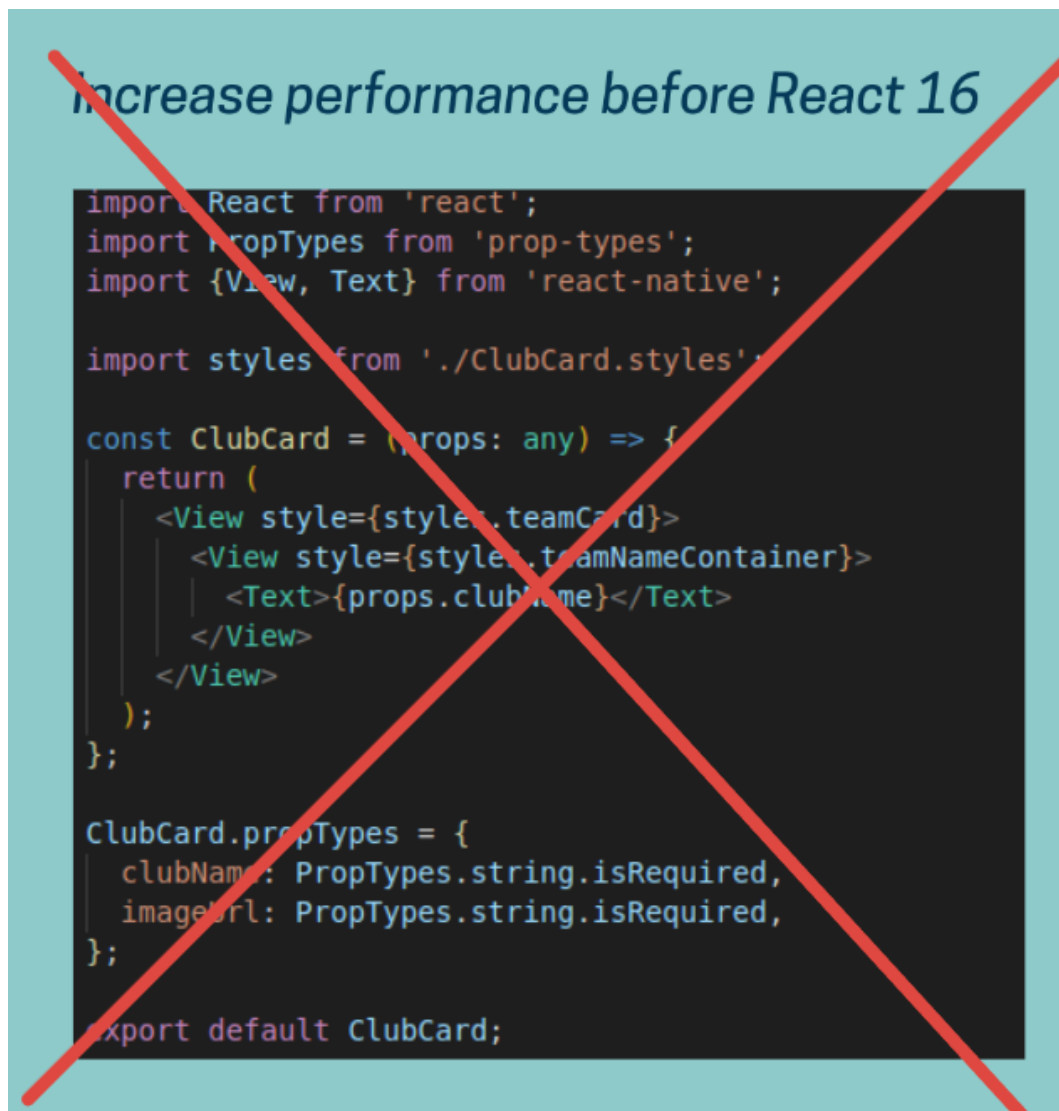
const ClubCard = (props: any) => {
  return (
    <View style={styles.teamCard}>
      <View style={styles.teamNameContainer}>
        <Text>{props.clubName}</Text>
      </View>
    </View>
  );
};

ClubCard.propTypes = {
  clubName: PropTypes.string.isRequired,
  imageUrl: PropTypes.string.isRequired,
};

export default ClubCard;
```

Then after in react 16, advice to use pure component instead of functional stateless component. because functional stateless component reduce the performance and it calls every time when component renders.

But in pure component do a shallow comparison. It means, when props or state change, a pure component will do a shallow comparison on both props and state.



After introduced react hook, we can use a functional component with a lot of benefits. So, do we need to rewrite them again for a functional component?

No!! Do not need it. They have no plans to remove classes from react.

They only recommend trying hooks in new code. If you're using it, you should know the exact reason for it. So let's move on to the next subtopic.

02. Why React Hook?

The first main reason is the **Introduce state in a functional component**. You know that the states cannot be used in functions. But with hooks, we can use states.

Another reason is the **handle side effect in react component**. It means, now you can use newly introduced state such as useEffect.

But do you know for some scenarios, there are 3 places where react fails.

1. *While Reuse logic between components*

2. *Has Huge components*

3. *Confusing classes*

Let's move into deep and understand these kinds of stuff in a brief.

1. While Reuse logic between components

We know that the re-usability and abstraction were introduced with, **Higher order component** and **RenderProp** Component in react. It is a really very useful and good thing.

But not for some cases. React context, its provider and consumer components. These things also introduce abstraction and invisible to the developers. in react conference, Sophie Alpert called it 'the wrapper hell'

it really doesn't visible to us until inspecting the components in the browser. So what if remove all wrapping components and make your component tree flat?

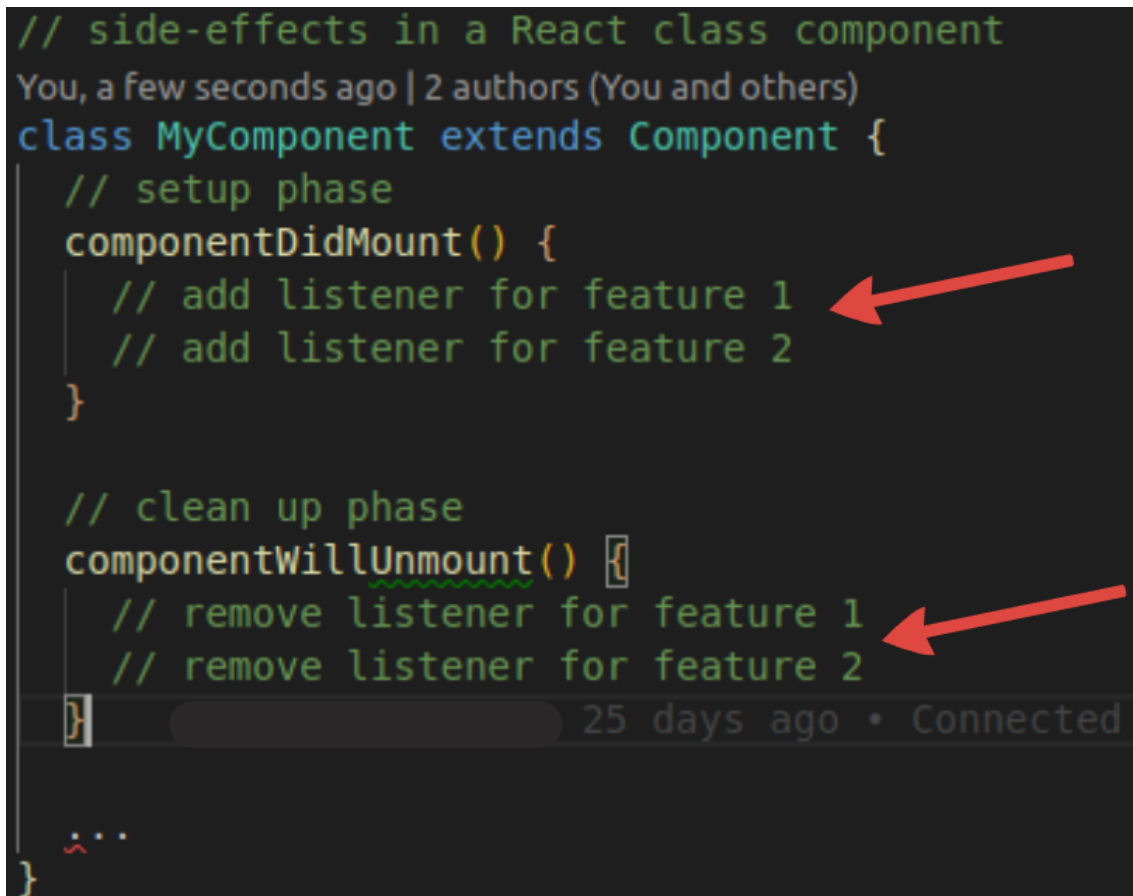
2. Has Huge components

If you already built the application, you will have experience with a thousand lines react component and your logic split into different lifecycle methods. So it makes very difficult for developers to understand the flow of the logic while reviewing.

Look at this example. This is a basic class component. inside the componentDidMount method, I initialize some listeners. It means, it can be a send-off network request, start a timer or subscribe to the data store.

```
// side-effects in a React class component
You, a few seconds ago | 2 authors (You and others)
class MyComponent extends Component {
  // setup phase
  componentDidMount() {
    // add listener for feature 1
    // add listener for feature 2
  }

  // clean up phase
  componentWillUnmount() {
    // remove listener for feature 1
    // remove listener for feature 2
  }
}
```



Not only these 2 methods, but you will also have to do some process in `componentDidUpdate` method. At there, compare with `newProps` with current props and do the same thing again in another method.

So what you can understand here? The same logic is split into different lifecycle methods. Sometime you will think this is really not a big issue and doesn't take more lines. But when app grows into a large and complex application and then you will get into stuck with these kinds of problems. So this kind of component includes,

- * **The mix of unrelated logic.**
- * **Inconsistency,**
- * **Difficult to test and**
- * **Introduce bugs easily**

3. Confusing classes

We know that the javascript mix of both object-oriented programming and functional programming.

Both react and redux uses functional programming. But react doesn't introduce these kinds of stuff. It came from JavaScript and that's why every react developers become a JavaScript developer.

JavaScript method like map, reducer, filter, functional composition, side effect and some terms like immutability. These functional programming concepts come from JavaScript to react.

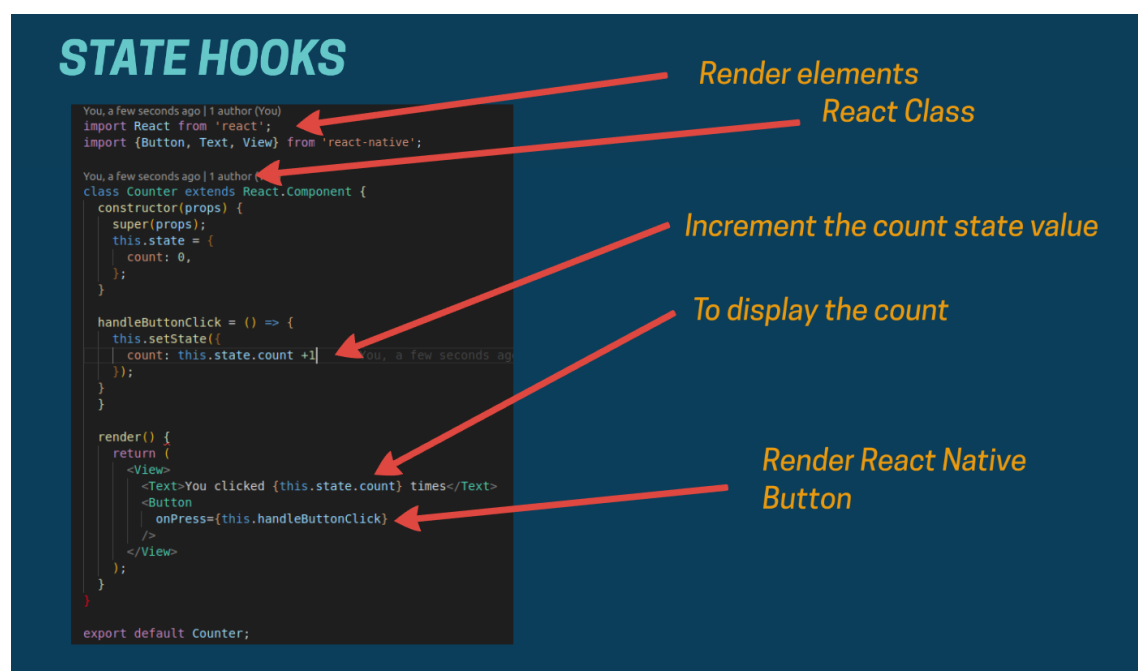
On the other side react uses OOP stuffs like a class instance of **this** object where use to interact with class methods. For an example, setState, forceupdate, and other custom class methods.

So it is really hard for developers where does not have OOP background. however, newly introduced hook API is a smoother learning curve for beginners and allow them to write JavaScript classes in the first place.

Now let's learn some hook types.

3. Hook Examples

(1) State Hook



React Class (1)

Above example is the simple react class and not include any hooks. Look at there, first import the react native render elements from react-native.

```

import React from 'react';
import {Text, View} from 'react-native';
  
```

It is a react class component and named it as a counter. Simply, I initialized the counter state and set the initial value to 0.

```
You, a few seconds ago | 1 author (You)  
class Counter extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      count: 0,  
    };  
  }  
}
```



Inside the render method, Text to display the counter value and button with onPress props. onPress props are to handle when user tap button. To handle this event, I used arrow function in prop.

```
render() {  
  return (  
    <View>  
      <Text>You clicked {this.state.count} times</Text>  
      <Button  
        onPress={this.handleClick}</Button>  
    </View>  
  );  
}
```

So inside the function, I mutate the state and increment the count state value to one for each button press.

```
handleButtonClick = () => {  
  this.setState({  
    count: this.state.count + 1  
  });  
}
```

So how to do these kinds of stuff in react hook?

STATE HOOKS

```
import React, {useState} from 'react';
import {Text, View, Button} from 'react-native';

function Counter() {
  // Declare a new state variable, which we'll call 'count'
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
  }

  return (
    <View>
      <Text>Count incremented {count} times</Text>
      <Button
        onPress={handleClick}
      >
        Click Me
      </Button>
    </View>
  );
}

export default Counter;
```

Import `useState` from `react`
Not the `react` component

```
this.state = {
  count: 0,
}
```

Use to Mutate the count state
Increment the state count by 1
To display the count state

Above example, simply import the `useState` from `react` other than `react` elements. This is the JavaScript function and not the `react` class component where I showed you an early example.

This is how we should declare state in hooks at below,

```
// Declare a new state variable, which we'll call 'count'
const [count, setCount] = useState(0);
```

Do u remember, *React Class (1)* example, how to declare state variable. ***SetCount*** variable is to mutate the variable like `setState`.

Then assign the ***useState*** element and put the initial value of count state variable as 0.

```
<Text>Count incremented {count} times</Text>
```

To display the count variable in text, you can call simply count state instead of ***this.state.count***.

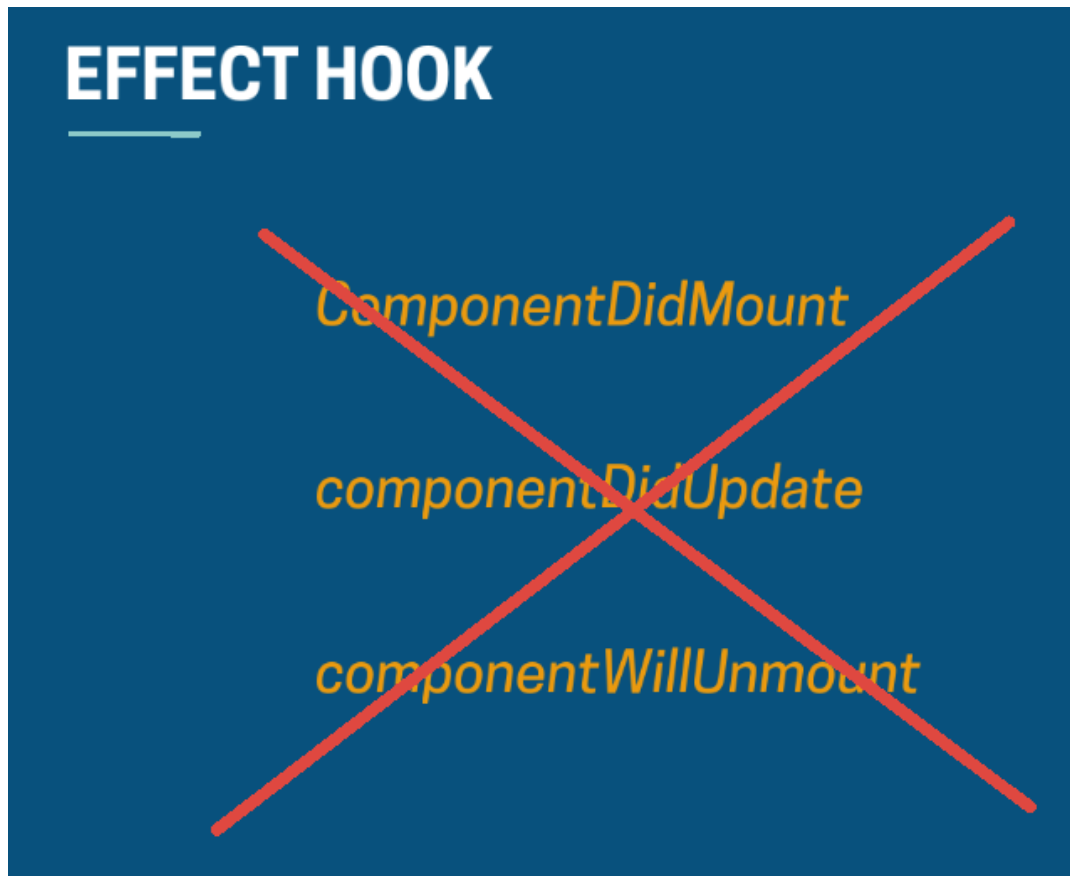
```
function handleClick() {
  setCount(count + 1);
}
```

Mutate the state inside the function and ***setCount*** variable use to increment the count state to 1 (Use to mutate the state).

This is what we call state hook. Next one is the effect hook.

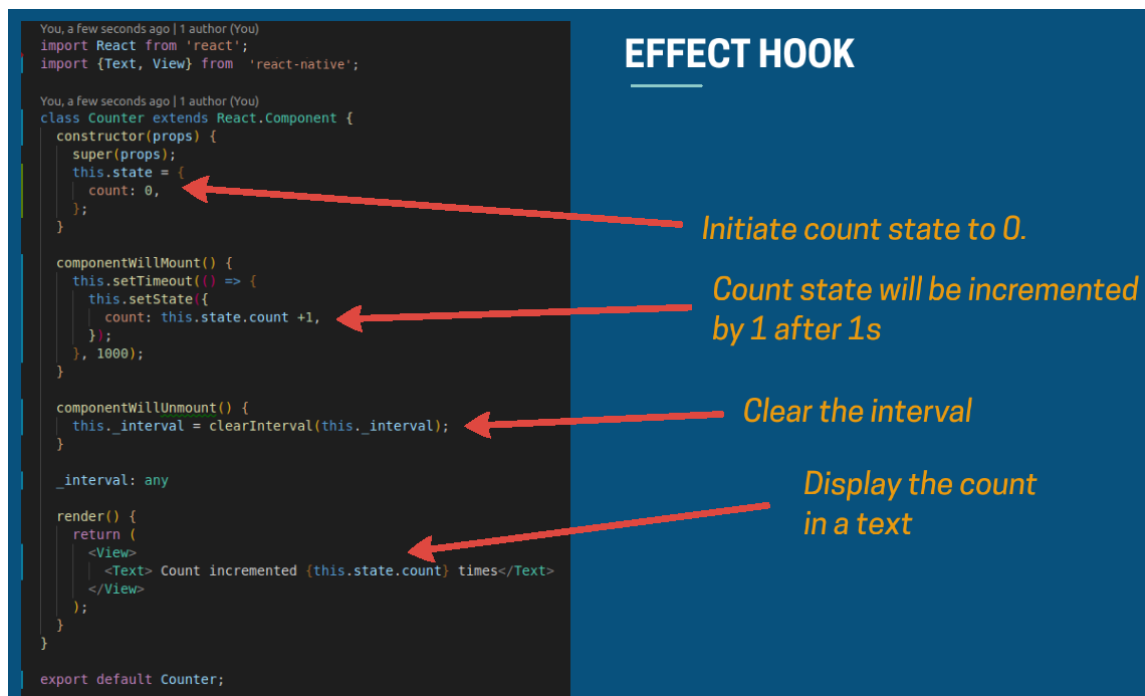
(2) State Hook

setEffect is used to replace the lifecycle hooks such as componentDidMount, componentDidUpdate and componentWillUnmount.



For an example, if your goal is to trigger data fetching upon clicking on a button, then there's no need to use useEffect.

Before move into the effect hook code, just look at this following example,



At the top imported the render elements and after initiated the count state to 0 as below.

```

this.state = {
  count: 0,
};

```

This code is inside the componentWillMountMethod. In here, simply I just initiate JavaScript time interval. It means each and every second count state will be incremented by one after componentWillMount method is called.

```

componentWillMount() {
  this.setTimeout(() => {
    this.setState({
      count: this.state.count + 1,
    });
  }, 1000);
}

```

So the count state will be printed in the text render element where count will be incremented for every 1 second.

```

<View>
  <Text> Count incremented {this.state.count} times</Text>
</View>

```

When the component is unmounting, we need to clear the interval. Otherwise, reference of the interval will be created for each time while the component is mounting.

```
componentWillUnmount() {
  this._interval = clearInterval(this._interval);
}
```

So look at the above examples, the same logic of time interval is split into multiple lifecycle methods. This is the one of example, you will have a lot of scenarios where splitting logic into different life cycle hook.

So how can we implement this with hooks?

EFFECT HOOK

```
import React, {useState, useEffect} from 'react';
import {Text, View} from 'react-native';

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

  useEffect(() => {
    setInterval(() => {
      setCount(count + 1);
    }, 1000);
  });

  return (
    <View>
      <Text>Count incremented {count} times</Text>
    </View>
  );
}

export default Counter;
```

- Import the react native elements
- Implement the simple JavaScript function named as counter
- Initiate the count state to 0
- SetCount variable is use to mutate the state
- Introduce new side effect
- Set interval is used to increase the count state every second
- Count state is displayed in the text element

At the top, simply I import the useState and useEffect in react.

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

on the next statement, imported the simple render elements text and view.

```
import {Text, View} from 'react-native';
```


Instead of react class, Implemented the JavaScript function and named as Counter.

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

Implement the simple JavaScript function named as counter


Inside there, I initialized state hook as we defined earlier.

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




setCount variable is simply used to mutate the count state variable.

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



This useEffect method call when any of state is changed. So inside there, I print the count in a text element.


```
useEffect(() => {  
  setInterval(() => {  
    setCount(count + 1);  
  }, 1000);  
});
```



Now I need to clear the interval when component unmount. So how I do this with effect hook. Without react hook, do you remember, we just clear the interval in another life-cycle method called componentWillUnmount? But in react hook, we can simply do it inside the useEffect.

Inside the return, clear the interval. So interval will be cleared when the component unmounted.

```
You, a few seconds ago | 2 authors (You and others)  
import React, {useState} from 'react';  
import {Text, View} from 'react-native';  
  
function Counter() {  
  const [count, setCount] = useState(0);  
  
  useEffect(() => {  
    const interval = this.setTimeout(() => {  
      setCount(count + 1);  
    }, 1000);  
    return () => {  
      clearInterval(this._interval);  
    }  
  });  
  
  return (  
    <View>  
      <Text> Count incremented {count} times</Text>  
    </View>  
  );  
}  
  
export default Counter;
```



Clean up the interval when unmounting the component

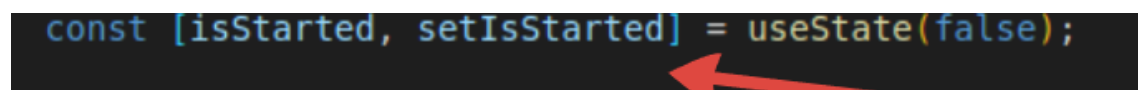
But Now each time when any of state is updated, this hook method is calling.

but we need to call this only at component will mount and unmount. So how can we fix it?

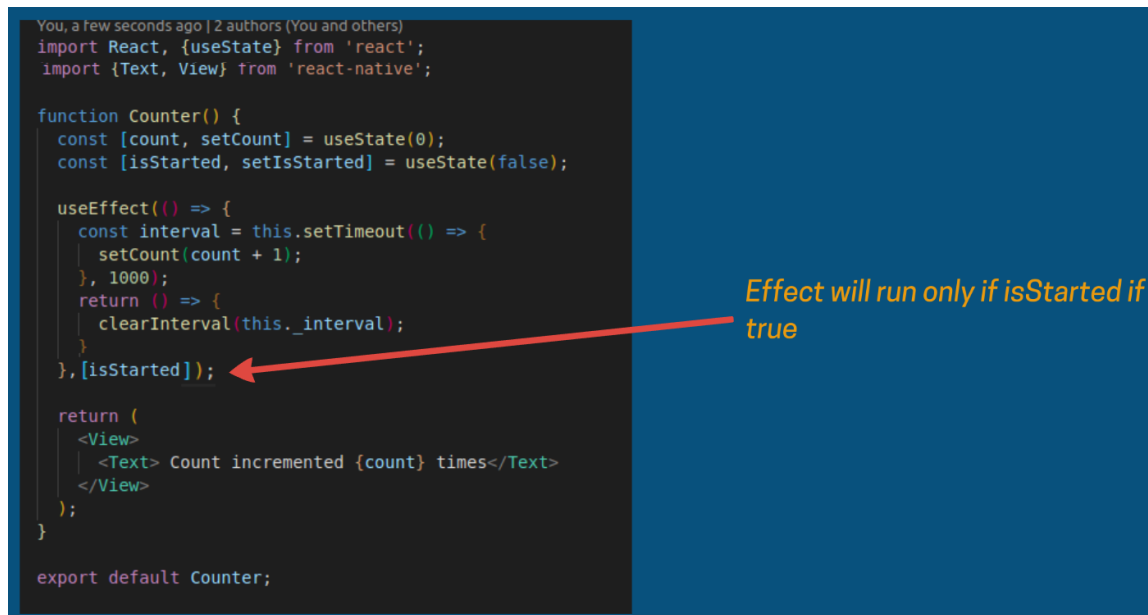
Simply, you can pass the empty array as a second argument. By doing this, this useEffect will call only at the component mount and unmount.



Now, what if I want to call this side effect only when for some state is changed? Let assume, If I have another state called **isStarted** and initial value of it is false.



If I want to trigger this useEffect when only **isStarted** state variable is true, then we can pass the **isStarted** state instead of passing empty array.



likewise, you can pass any state variable to check inside this array. Now, let's move to another subtopic, that is a custom hook.

(3) Custom Hook

Let's say, if I want to reuse the count state in another JavaScript function, we would copy & paste similar logic we wrote. But it wouldn't be ideal. Instead, make this code shareable code to use between other functions.

```
You, a few seconds ago | 2 authors (You and others)
import React, {useState} from 'react';
import {Text, View} from 'native-base';

function Counter() {
  const [count, setCount] = useState(0);
  const [isStarted, setIsStarted] = useState(false);

  useEffect(() => {
    const interval = this.setTimeout(() => {
      setCount(count + 1);
    }, 1000);
    return () => {
      clearInterval(this._interval);
    }
  }, isStarted);

  return (
    <View>
    <Text> Count incremented {count} times</Text>
    </View>
  );
}

export default Counter;
```

So extract it to 3rd function and name that function as *useCounterValue*.

A custom hook is a JavaScript function whose name starts with *use*. finally, it returns the count state.

```
You, a few seconds ago | 2 authors (You and others)
import React, {useState} from 'react';
import {Text, View} from 'native-base';

function Counter() {
  const count = useCounterValue();

  return (
    <View>
      <Text> Count incremented {count} times</Text>
    </View>
  );
}

export default Counter;

function useCounterValue() {
  const [count, setCount] = useState(0);
  const [isStarted, setIsStarted] = useState(false);

  useEffect(() => {
    const interval = this.setTimeout(() => {
      setCount(count + 1);
    }, 1000);
    return () => {
      clearInterval(this._interval);
    };
  }, isStarted);

  return count;
}
```

Then after assign that custom hook function to a constant variable and use it. Finally, print the count state.



Apart from these 3 hooks, there are some other important hooks. They are,

- * useContext,
- * useReducer,
- * useCallback,
- * useMemo,
- * useImperativeMethods
- * useMutationEffect,
- * useRef,
- * useEffect

Let discuss these hook briefly in another set of articles. Now, let us move to the last subtopic.

04. Rules

There are 2 important rules here.

1. **Don't call hooks inside *the loop, condition or nested function*.**

Instead, always use Hooks at the top level of your React function. This rule, you ensure that Hooks are called in the same order each time a component renders.

2. **Call hooks from react function. not the regular function.**

So you can Call Hooks from React functional components or from the custom hooks as we discussed early. By following this rule, you ensure that all stateful logic in a component is clearly visible from its source code.

More articles of the author -

1. [JavaScript Promises and asynchronous programming](#)
2. [Javascript Equality — Advanced Javascript Tutorial](#)

[React](#) [React Native](#) [React Hook](#) [JavaScript](#) [Reactjs](#)

[About](#) [Write](#) [Help](#) [Legal](#)

Get the Medium app

