|  |  |
| --- | --- |
| **ch34.ReactJS Hooks** | **Date: 22-02-2022** |

**Topics**

React Props Validation,

# React Props Validation

# React Hooks

Hooks are the new feature introduced in the React 16.8 version. It allows you to use state and other React features without writing a class. Hooks are the functions which "hook into" React state and lifecycle features from function components. It does not work inside classes.

Hooks are backward-compatible, which means it does not contain any breaking changes. Also, it does not replace your knowledge of React concepts.

## **When to use a Hooks**

If you write a function component, and then you want to add some state to it, previously you do this by converting it to a class. But, now you can do it by using a Hook inside the existing function component.

## **Rules of Hooks**

Hooks are similar to JavaScript functions, but you need to follow these two rules when using them. Hooks rule ensures that all the stateful logic in a component is visible in its source code. These rules are:

Competitive questions on Structures in Hindi

Keep Watching

### **1. Only call Hooks at the top level**

Do not call Hooks inside loops, conditions, or nested functions. Hooks should always be used at the top level of the React functions. This rule ensures that Hooks are called in the same order each time a components renders.

### **2. Only call Hooks from React functions**

You cannot call Hooks from regular JavaScript functions. Instead, you can call Hooks from React function components. Hooks can also be called from custom Hooks.

## **Pre-requisites for React Hooks**

1. Node version 6 or above
2. NPM version 5.2 or above
3. Create-react-app tool for running the React App

## **React Hooks Installation**

To use React Hooks, we need to run the following commands:

1. $ npm install react@16.8.0-alpha.1 --save
2. $ npm install react-dom@16.8.0-alpha.1 --save

The above command will install the latest React and React-DOM alpha versions which support React Hooks. Make sure the **package.json** file lists the React and React-DOM dependencies as given below.

1. "react": "^16.8.0-alpha.1",
2. "react-dom": "^16.8.0-alpha.1",

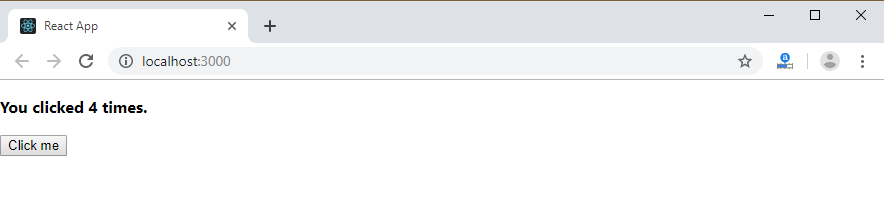
## **Hooks State**

Hook state is the new way of declaring a state in React app. Hook uses useState() functional component for setting and retrieving state. Let us understand Hook state with the following example.

**App.js**

1. **import** React, { useState } from 'react';
3. function CountApp() {
4. // Declare a new state variable, which we'll call "count"
5. **const** [count, setCount] = useState(0);
7. **return** (
8. <div>
9. <p>You clicked {count} times</p>
10. <button onClick={() => setCount(count + 1)}>
11. Click me
12. </button>
13. </div>
14. );
15. }
16. export **default** CountApp;

**output:**



In the above example, useState is the Hook which needs to call inside a function component to add some local state to it. The useState returns a pair where the first element is the current state value/initial value, and the second one is a function which allows us to update it. After that, we will call this function from an event handler or somewhere else. The useState is similar to this.setState in class. The equivalent code without Hooks looks like as below.

**App.js**

1. **import** React, { useState } from 'react';
3. **class** CountApp **extends** React.Component {
4. constructor(props) {
5. **super**(props);
6. **this**.state = {
7. count: 0
8. };
9. }
10. render() {
11. **return** (
12. <div>
13. <p><b>You clicked {**this**.state.count} times</b></p>
14. <button onClick={() => **this**.setState({ count: **this**.state.count + 1 })}>
15. Click me
16. </button>
17. </div>
18. );
19. }
20. }
21. export **default** CountApp;

## **Hooks Effect**

The Effect Hook allows us to perform side effects (an action) in the function components. It does not use components lifecycle methods which are available in class components. In other words, Effects Hooks are equivalent to componentDidMount(), componentDidUpdate(), and componentWillUnmount() lifecycle methods.

Side effects have common features which the most web applications need to perform, such as:

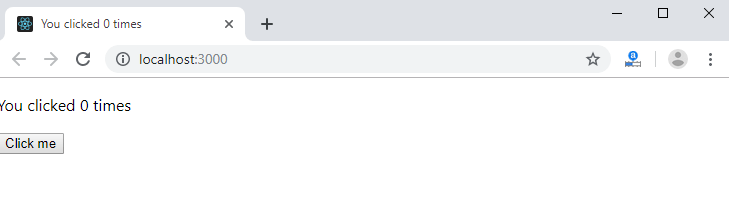
* Updating the DOM,
* Fetching and consuming data from a server API,
* Setting up a subscription, etc.

Let us understand Hook Effect with the following example.

1. **import** React, { useState, useEffect } from 'react';
3. function CounterExample() {
4. **const** [count, setCount] = useState(0);
6. // Similar to componentDidMount and componentDidUpdate:
7. useEffect(() => {
8. // Update the document title using the browser API
9. document.title = `You clicked ${count} times`;
10. });
12. **return** (
13. <div>
14. <p>You clicked {count} times</p>
15. <button onClick={() => setCount(count + 1)}>
16. Click me
17. </button>
18. </div>
19. );
20. }
21. export **default** CounterExample;

The above code is based on the previous example with a new feature which we set the document title to a custom message, including the number of clicks.

**Output:**



In React component, there are two types of side effects:

1. Effects Without Cleanup
2. Effects With Cleanup

### **Effects without Cleanup**

It is used in useEffect which does not block the browser from updating the screen. It makes the app more responsive. The most common example of effects which don't require a cleanup are manual DOM mutations, Network requests, Logging, etc.

### **Effects with Cleanup**

Some effects require cleanup after DOM updation. For example, if we want to set up a subscription to some external data source, it is important to clean up memory so that we don't introduce a memory leak. React performs the cleanup of memory when the component unmounts. However, as we know that, effects run for every render method and not just once. Therefore, React also cleans up effects from the previous render before running the effects next time.

## **Custom Hooks**

A custom Hook is a JavaScript function. The name of custom Hook starts with "use" which can call other Hooks. A custom Hook is just like a regular function, and the word "use" in the beginning tells that this function follows the rules of Hooks. Building custom Hooks allows you to extract component logic into reusable functions.

Let us understand how custom Hooks works in the following example.

1. **import** React, { useState, useEffect } from 'react';
3. **const** useDocumentTitle = title => {
4. useEffect(() => {
5. document.title = title;
6. }, [title])
7. }
9. function CustomCounter() {
10. **const** [count, setCount] = useState(0);
11. **const** incrementCount = () => setCount(count + 1);
12. useDocumentTitle(`You clicked ${count} times`);
13. // useEffect(() => {
14. //   document.title = `You clicked ${count} times`
15. // });
17. **return** (
18. <div>
19. <p>You clicked {count} times</p>
20. <button onClick={incrementCount}>Click me</button>
21. </div>
22. )
23. }
24. export **default** CustomCounter;

In the above snippet, useDocumentTitle is a custom Hook which takes an argument as a string of text which is a title. Inside this Hook, we call useEffect Hook and set the title as long as the title has changed. The second argument will perform that check and update the title only when its local state is different than what we are passing in.

#### **Note: A custom Hook is a convention which naturally follows from the design of Hooks, instead of a React feature.**

## **Built-in Hooks**

Here, we describe the APIs for the built-in Hooks in React. The built-in Hooks can be divided into two parts, which are given below.

**Basic Hooks**

* useState
* useEffect
* useContext

**Additional Hooks**

* useReducer
* useCallback
* useMemo
* useRef
* useImperativeHandle
* useLayoutEffect
* useDebugValue

Next Topic[React Flux Concept](https://www.javatpoint.com/react-flux-concept)