International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 09 Issue: 12 | Dec 2022 www.irjet.net p-ISSN: 2395-0072

React state management and side-effects – A Review of Hooks

Krutika Patil[1], Sanath Dhananjayamurty Javagal[2]

[1]Software Engineer, JP Morgan Chase & Co., Palo Alto, California, USA
[2]Systems Engineer, Cruise LLC., San Francisco, California, USA

Abstract - *React is a front-end JavaScript library used to build User Interfaces (UIs) or UI components. ReactJS is a flexible, declarative, and efficient JavaScript library for building reusable UI components. We can also build complex components by nesting one or more simple components. This concept is the heart of any React application. One of the outstanding capabilities of React is Hooks. We can efficiently manage states and execute side effects using React Hooks without defining a JavaScript class. It is like taking advantage of the cleanliness and simplicity of a Pure Component and state and component lifecycle methods. This clarity, efficiency, and simplicity are possible because Hooks are regular JavaScript functions! In this paper, we dive deep into React Hooks and their capabilities.*

Key Words: react, react hooks, react state, react side-effects, JavaScript, frontend-frameworks, web development.

1. INTRODUCTION

React version **16.8** first introduced us to Hooks.

The functional components can access the state and other React features using React Hooks. React Hooks are functions using which we can introduce state management and side-effect logic, among others, in a React component. The React Hooks do not work with or inside classes; however, React Hooks let us perform these operations without using classes.

2. React Hooks are revolutionary

Hooks solve many seemingly unconnected roadblocks React developers face in their journey of building many React applications. Let us discuss a few of those problems in the following sections and understand how React Hooks fill those gaps and make React development more efficient and fun.

2.1 Reusing stateful logic among components is hard

React does not provide an out-of-the-box way to "hook" reusable behavior into components. The react developers during the early days of React usually preferred techniques like **higher-order components** and **render props** to achieve such behavior. However, these patterns require us to restructure our components when we use them, which can complicate the code and reduce its readability. A typical React class-based application consists of many wrapper components, making it challenging to analyze and troubleshoot. Even though we have Dev Tools to help us in this process, this brings a more pressing problem to sharp relief: we need to introduce a better alternative to manage states in React.

With Hooks, we can reuse the stateful logic without affecting the hierarchy of the components. The state logic can also be extracted and tested out independently.

2.2 Complex components are challenging to maintain

We have often dealt with a situation where we used components that seemed manageable and small to begin with but, during the development process, grew into complex pieces that were difficult to maintain and understand.

In many cases, splitting the application into smaller components gets challenging as the stateful logic gets shared throughout the application. Testing such components is also tricky, so React is usually combined with a third-party state-management library but might introduce too much abstraction making the code reuse difficult.

To mitigate this problem, React Hooks help us break down a complex component into smaller building blocks called functions based on relativity rather than lifecycle methods. We may also use the "**useReducer**" Hook to manage the local state of a component.

2.3 Learning classes is a steep learning curve

The concept of JavaScript classes is significantly different compared to other languages. Developers are known to struggle with classes in JavaScript since classes are complex pieces of logic. The complexity of classes also poses challenges to code-reusability and organization. Due to these reasons and the steep learning curve, a lack of enthusiasm is noticed among React developers to learn and use classes in React.

3. The State hook

Let us start understanding the **useState** Hook by comparing the below code in **Figure 1** with an equivalent class example. Reactivity is a common trait of any user-friendly front-end application. The application can achieve it by maintaining

the states and updating or rendering the components impacted by those states. A component's state needs to be updated whenever one or more variables in the component update. It is here that the useState Hook shines. Let us compare a state implementation in a React class component and a functional component.

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

function ClassComponent() {
  const [clickCount, setClickCount] = useState(0);

  return (
    <div>
      <p>You clicked {clickCount} times</p>
      <button onClick={() => setClickCount(clickCount + 1)}>Click me</button>
    </div>
  );
}
```

Fig -1: Sample React functional component with **useState** Hook

If we have used React classes before, the code below in **Figure 2** should look familiar.

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

  render() {
    return (
      <div>
        <p>You clicked {this.state.clickCount} times</p>
        <button
          onClick={() =>
            this.setState({ clickCount: this.state.clickCount + 1 })
          }
        >
          Click me
        </button>
      </div>
    );
  }
}
```

Fig -2: Sample React class component with state

Initially, when the state is declared, it has the value **{clickCount: 0}**, with every click action on the button **this.setState()** function gets executed, incrementing the previous value by 1.

Let us consider the same example using a functional component. As a reminder, **Figure 3** depicts an example of a functional component.

```
const ComponentFunction = (props) => {
  return <div />;
};
```

Fig -3: Sample React functional component

Depicted in **Figure 4** is another way to declare a functional component using the ES 5 JavaScript format.

```
function ComponentFunction(props) {
  return <div />;
}
```

Fig -4: Sample React functional component using ES5 JavaScript format

We have previously known these as **stateless components**. These are also known as **functional components** since these are functions with stateful logic in them. Hooks do not work inside classes. However, we can use them instead of writing classes.

Our example starts by adding a named import of the same name, useState**,** from the react library, as depicted below in **Figure 5**.

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

function ComponentFunction() {
  //...
}
```

Fig -5: Syntax to import the "**useState**" Hook

3.1 When do we use a React state Hook

During the early React days, we would convert a functional component to a React class component if we needed to add stateful logic. However, we can now instead use a state hook.

3.2 Introducing stateful logic in components

Continuing with our class example, let us initialize **clickCount** to **0** by setting the value of **this.state** to **{clickCount: 0}** in the constructor, as shown in **Figure 6**.

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

Fig -6: Declaring **state** in a React class

The concept of **this** object is not supported in functional components of React, making it impossible to read from **this.state**. However, we can directly execute the useState Hook within the functional component, as depicted below in **Figure 7**.

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

function FunctionalComponent() {
  const [clickCount, setClickCount] = useState(0);
```

Fig -7: Functional component state declaration using **useState**

3.3 What are the inputs to the useState Hook?

The **useState** Hook takes in one argument, the initial state. The state need not be an object, unlike the classes. If that solves the purpose, it can be as simple as a string or a number. Like in our example here, we initialize the state of the variable as 0, which is a number. The useState Hook can be called as many times as we need to store different states. We get two values returned by the React useState Hook that can be retrieved by **array de-structuring,** as depicted below in **Figure 8**.

```
const [val, setVal] = useState(0);
```

Fig -8: **useState** initial value and return values

In the example above, **val** is the state variable, and we can set the value of **val** using the setter function **setVal**.

3.4 What values are returned by useState?

The useState Hook returns two values: the existing or the current value and a setter function that sets or updates the state. Hence, the syntax depicted in **Figure 8** relates directly to the class example shown in **Figure 6** with **this.state.clickCount** and **this.setState**. With this understanding, the example depicted in **Figure 7** should make sense.

When the React application first renders, the **App** component function, the parent to all other components in the project, gets executed first. The App component then executes its child component functions, and so on. Hence, when a particular component variable changes, the change is not reflected on the DOM since the component function is not re-executed. It is here that the **useState** react Hook shines.

3.5 Detailed Working

Upon declaring a variable or a state using the useState Hook, React stores and maintains the state in the JavaScript memory. To set a value to the variable, use the setter function and pass in the new value. The setter function indicates to React that the component function needs to be re-executed, which then executes the useState function. The re-execution of the state Hook ensures the state variable gets updated with the latest value since every execution of the state Hook returns the latest value. As the state updates, it gets reflected on the DOM courtesy of React's virtual DOM. Once the component function is re-executed, the updated value gets reflected on the DOM. The execution of the state Hook in a component would not affect any other component except the component where the useState for this variable is initialized or used. Also, when the component function is re-executed, it would not reset the variable's value to its initial value since React keeps track of the first initialization of useState Hook. React also keeps track of the current or the latest state between renders and always provides us with the most current state. So, during every subsequent execution of the state Hook and except the first execution, the initial value is ignored.

The useState Hook helps us separate the state on a per-component basis. This way, useState helps maintain the states of React components.

4. The Side Effect Hook

We can perform side-effect logic in React functional components using the **useEffect** Hook.

4.1 What do side-effects mean in React?

We perform a side-effect when interacting outside our React component to achieve a particular goal. It is impractical to predict the outcome of side effects since they are actions performed with elements in the outside world. Some common side-effects include calling an API to fetch data, using browser APIs (window or document directly), and using unpredictable timer functions (**setTimeout** and **setInternal**).

4.2 useEffect – Champion of handling side-effects

We should refrain from using the side effects directly in the React component functions since those can impact the component's rendering. For example: If we have a logic to call an API in the React component's function, the API call will be executed upon every component render. Moreover, if the side-effect updates a state in the component function, it triggers a re-render of the component resulting in the side-effect's re-execution. This sequence of events may trigger an infinite render loop. The side-effects need to be separated from React's rendering logic and must consistently execute once the component renders, which is what the **useEffect** Hook gives us.

4.3 Syntax

useEffect is a function that takes two arguments. The first argument is a function that executes the side effect logic. The second argument is an optional list of dependencies on which the side effect function depends, as depicted below in **Figure 9**. This list can be empty if we choose not to list any dependencies. When the list is empty, the useEffect function gets executed upon every component render.

```
React.useEffect(() => {...}, [dependencies])
```

Fig -9: **useEffect** declaration

By listing out the dependencies, we inform React that the side-effect function must execute ONLY if one or all the dependencies in the list change**, ensuring that the side-effect logic executes in specific circumstances unique to the side-effect**. When the react component re-renders, React compares the values of the dependencies from their previous values and only runs the useEffect function if a change is detected. The useEffect Hook does not execute in the case of no change in the values of the list of dependencies. This way, we may skip running the side-effect logic in cases where we do not need to.

4.4 Effects needing clean-up

Consider the below useEffect implementation depicted in **Figure 10**. There may be instances where our side effects include subscribing to an event or a timeout. In this case, for every execution of the useEffect function, we create a new subscription without clearing the older ones.

```
useEffect(() => {
  setInterval(() => {
    count++;
  }, 2000);
}, [count]);
```

Fig -10: **useEffect** with a subscription to **setInterval**

The above logic in **Figure 10** might lead to memory leaks and unexpected behavior. It is here that the **useEffect clean-up function** comes to the rescue.

We can have the useEffect Hook return a function, which essentially executes the clean-up logic as depicted below in **Figure 11**.

```
useEffect(() => {
  const interval = setInterval(() => {
    count++;
  }, 2000);

  return () => clearInterval(interval);
}, [count]);
```

Fig -11: **useEffect** with clean-up function

The clean-up function executes in the following scenarios: during every execution of the useEffect function, except the first component render, and upon the component getting unmounting from the DOM. In the example above, the clean-up function does not run during the first render of the component, but then it runs before every execution of the useEffect function. Since we save the interval value that was last created, the clean-up function clears the previously created interval using **clearInterval** before re-executing the useEffect function, which then creates a new interval. Since the clean-up function runs just before the component is unmounted from the DOM, the last interval also gets cleared.

5. CONCLUSIONS

We have attempted to highlight React's state management and the side-effect hooks. The state management hook **useState** helps us to efficiently manage React component updates and show appropriate feedback to the user by updating the respective components, which ensures a seamless user experience. The side-effect hook **useEffect** lets us perform side-effect operations without interfering with React's rendering logic or the performance of the React component. There are more hooks, for example, **useRef**, **useContext**, and **useCallback**. We also have an opportunity to build our hooks in React. These capabilities of React hooks can be used appropriately per the use case, leading to efficient and user-friendly front-end applications.

REFERENCES

[1] Bhupati Venkat Sai Indla | Yogeshchandra Puranik "Review on React JS" Published in International Journal of Trend in Scientific Research and Development (ijtsrd), ISSN: 2456-6470, Volume-5 | Issue-4, June 2021, pp.1137-1139, URL: https://www.ijtsrd.com/papers/ijtsrd42490.pdf.

[2] Rawat, Prateek, and Archana N. Mahajan. "ReactJS: A Modern Web Development Framework." International Journal of Innovative Science and Research Technology 5, no. 11 (2020).

[3] Maratkar, Pratik Sharad, and Pratibha Adkar. "Re act JS-An Emerging Frontend JavaScript Libr ary." *Iconic Research And Engineering Journal s* 4, no. 12 (2021): 99-102.

[4] https://reactjs.org/

[5] https://www.w3schools.com/react/default.asp/

[6] https://en.wikipedia.org/wiki/React_(JavaScript_library)#React_hooks

[7] https://www.udemy.com/course/react-the-complete-guide-incl-redux/learn/lecture/25600136#overview

BIOGRAPHIES

Ms. Krutika Patil is a Full-Stack Senior Software Engineer at JP Morgan Chase & Co. in Palo Alto, California, USA. Her interests include web development, Spring Boot, JavaScript frameworks (React, Vue and Angular), and other Computer Science concepts.

Mr. Sanath Dhananjayamurty Javagal is a Senior System Engineer at Cruise LLC in San Francisco, California, USA. Working mainly as an Autonomous Vehicle Network architecture and Systems Engineer, his interests include Computer Engineering, AV Architecture and Design, Simulation, and Analysis.