**React Hooks:**

React hooks are functions introduced in React version 16.8, designed to empower functional components with capabilities for state management and handling side effects. By integrating hooks into functional components, Developers can achieve more concise and readable code.   
  
***Benefits of React Hooks:***  
1. React hooks simplify React code by enabling developers to use functional components for stateful logic and side effects.  
2. React hooks provide developers with control over state updates and side effects. This allows developers to precisely manage when state changes occur, and which side effects are executed in response to those changes.  
3. In this control, developers can define the exact behavior of their components, specifying which side effects should be triggered based on specific state changes.  
4. When we use hooks in our application, it often results in improved performance and optimization  
  
***Understanding State Management and Side Effects:***  
In the previous paragraph and point, we discussed state management and side effects extensively. Now, let's learn about state management and side effects in detail.  
  
***State :***  
***What is the State?***

- The state is a built-in react object of the react components that is used to contain data or information about the component.

- The state in only inside the components.

- State is mutable, it can change as per the need.

- Whenever the state object changes the component is re-rendered.

- To use state inside the functional component ‘useState’ hook is used.

**Why do we call it State?**

In React, we cannot directly update variables or data types such as objects, arrays, strings, etc. If we need to update them, we use state.

**Why do we use State?**

|  |
| --- |
| export default function Home() {  let x = 10;  function updateState() {   x = 10 + 1;   console.log(x);  }  console.log(x);  return (   **<div>**   **<h1>**State {x}**</h1>**   **<button** onClick={updateState}**>**ClickMe**</button>**   **</div>**  )} |

1. The `Home` component initializes a variable `x` with the value 10 and displays it in the UI.

2. Clicking the "Click Me" button triggers the `updateState` function.

3. Inside `updateState`, `x` is incremented to 11, but this change isn't reflected in the UI.

4. Both the initial and updated values of `x` are logged to the console.

5. React doesn't detect the change in `x` since it's directly modified without using state management.

6. Consequently, the UI doesn't update to show the new value of `x`, remaining at 10 instead of 11.

***State Management:***  
1. State management refers to the process of managing and updating data within components to control their behavior and appearance.

2. Each component can have its own state, which represents either the current condition or the updated condition of the component.

3. State management involves initializing, updating, and accessing this state data, typically in response to user interactions, events, or other factors.

***Apply State Management with the help of useState:***

The `useState` hook is a built-in function in React that allows functional components to manage local state. It provides a simple way to add state to functional components

Here's how useState works:   
1. You call the `useState` function within a functional component and pass in the initial state value as an argument.

2. The `useState` function returns an array with two elements. The first element is the current state value, and the second element is a function that allows you to update the state.

3. You can use array destructuring to assign names to these elements. Typically, you'll use a descriptive name for the state variable and a name like `setVariable` for the update function.

4. You can read the current state value and update it using the provided update function. Whenever you call the update function, react will re-render the component with the new state value.

***For Example:***

|  |
| --- |
| import { useState } from "react";  export default function Home() {  const [state, setState] = useState(10);  function updateState() {  setState(state + 1);  }  console.log(state)  return (   **<div>**   **<h1>**State {state}**</h1>**   **<button** onClick={updateState}**>**Click Me**</button>**   **</div>**   )} |

1. In the Home component, a state variable state is initialized using the useState hook, with an initial value of 10.  
2. The initial value of state (10) is displayed in the UI within an <h1> element.  
3. When the "Click Me" button is clicked, the updateState function is triggered  
4. Inside the updateState function, the setState function is called to update the value of state. It increments the current value of state by 1.  
5. React detects the state change triggered by setState and schedules a re-render of the component.  
6. After the re-render, the UI reflects the new value of state (11), and the <h1> element displays "State 11".

**So in this previous topic we can learn about state management so this time to learn about side effects.**

***SIDE EFFECT:***

A side effect refers to any additional action or behavior that occurs when our component renders, beyond the usual rendering process.

For example:

- We want that whenever our component renders, it fetches data from an API.

- Whenever a component renders, we want our DOM to update.

***How to Handle Side Effect?***

To handle side effects in React, we typically use the useEffect hook.

***UseEffect:***

• The **useEffect** hook allows us to perform side effects in functional components.

• Inside the useEffect hook, we can include code that executes side effects, such as   
 fetching data from an API, subscribing to events, or updating the DOM.  
• By encapsulating side effects within the useEffect hook, we ensure that they are handled in a controlled and efficient manner.  
• Additionally, we can specify dependencies for the effect, so that it only runs when certain values change.  
• This helps prevent unnecessary side effects and ensures that our component behaves   
 predictably.

**To use the useEffect hook in React, follow these steps:**  
1- Import the useEffect hook from the react

**import React, { useEffect } from 'react';**

2- Declare the Effect: Inside your functional component, declare the useEffect hook. Pass a function as the first argument, which will contain the code for your side effect.  
  
**useEffect(() => {**

**// Code for your side effect**

**}, [/\* Dependency array \*/]);**

***Behavior of useEffect Hook:***

1. **useEffect(() => { // Code for your side effect });**

• This **useEffect** hook will execute the side effect (the code inside the function) after every   
 render of the component.  
• Since there are no dependencies specified in the dependency array, the side effect will   
 run after every render, regardless of whether any specific values have changed.

**2. useEffect(() => { // Code for your side effect }, []);**

• This useEffect hook will execute the side effect only once, after the initial render of the   
 component.

• The empty dependency array indicates that there are no dependencies, so the effect will   
 run only after the initial render and not be triggered again on subsequent renders.

**3. useEffect(() => { // Code for your side effect }, [dependency1, dependency2 /\* , ... \*/]);**

• This useEffect hook will execute the side effect whenever any of the dependencies   
 (dependency1, dependency2, etc.) change.

• The side effect will run after the initial render and then again whenever any of the   
 specified dependencies change. If none of the dependencies change between renders,   
 the effect will not be triggered.

**Code Example:**

|  |
| --- |
| import { useEffect, useState } from "react";  export default function EffectHook() {  const [count1, setCount1] = useState(0);  const [count2, setCount2] = useState(0);   // This useEffect runs after every render, regardless of any state changes.  useEffect(() => {  console.log("UseEffect fired at full speed");  });   // This useEffect runs only once after the initial render.  useEffect(() => {  console.log("UseEffect fired at full speed");  }, []);   // This useEffect runs whenever either count1 or count2 changes.  useEffect(() => {  console.log("UseEffect fired at full speed");  }, [count1, count2]);   return (  **<div>**  **<h1>**Use Effect**</h1>**  **<h2>**React Hook**</h2>**  **<button** onClick={() =**>** setCount1(count1 + 1)}>  Increment Count {count1}  **</button>**  **<br** **/>**  **<button** onClick={() =**>** setCount2(count2 + 1)}>  Increment Count {count2}  **</button>**  **</div>**  ); } |

[More About Side Effect](https://medium.com/@remoteupskill/what-is-a-react-side-effect-a5525129d251)

**useRef hook:**

|  |
| --- |
| **sadasdassad** |

**useCallback hook:**

|  |
| --- |
| **dssfdsfsdfdsfds** |

**useMemo hook:**

|  |
| --- |
| **dfadfsfdsfsd** |

**context API**

**usecontext\_hook:**

|  |
| --- |
| **asdasassdasdsa** |