

React Hooks, State & Effect, Props

By

Narasimha Rao T

Microsoft.Net FSD Trainer

Professional Development Trainer

tnrao.trainer@gmail.com

1. Introduction to React Hooks

- React Hooks are functions that allow you to "hook into" React state and lifecycle features from functional components.
- Introduced in React 16.8 (February 2019), Hooks enable developers to use state and other React features without writing class components.
- Key Hooks include `useState` , `useEffect` , `useContext` , `useReducer` , `useRef` , etc.

Rules of Hooks

- Hooks must be called at the top level of a functional component to ensure consistent order of execution.
- Not inside loops, conditions, or nested functions

Example Code Snippet:

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

function Example() {
  const [count, setCount] = useState(0); // Hook usage
  return <button onClick={() => setCount(count + 1)}>Count: {count}</button>;
}
```

2. Why React Hooks Are Important?

- **Simplifies Code:** Hooks eliminate the need for class components, lifecycle methods like `componentDidMount`).
- **Reusability:** Logic can be extracted into custom Hooks, making it shareable across components.
- **Better Organization:** Group related code (state + effects) together instead of scattering it across lifecycle methods.
- **Functional Paradigm:** Encourages functional programming, making components easier to test, reason about, and compose.
- **Performance and Readability:** Avoids wrapper hell (e.g., multiple HOCs) and improves tree shaking in bundlers.

3. useState Hook for State Management

- `useState` is a Hook that lets you add state to functional components.
- It returns an array: `[currentState, setterFunction]`.
- Initial state can be a value or a function .
- Multiple `useState` calls can manage different pieces of state independently.

Example Code Snippet:

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

function Counter() {
  const [count, setCount] = useState(0); // Initial state: 0
  const increment = () => setCount(prev => prev + 1); // Functional update

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
```

- **Common Use:** Managing form inputs, toggles, counters, or local UI state.

useEffect Hook for Side Effects

4. useEffect Hook for Side Effects

- `useEffect` handles side effects in functional components (e.g., data fetching, subscriptions, manual DOM manipulations).
- It runs after every render by default, but can be controlled with a dependency array.
- **Syntax:** `useEffect(callback, [dependencies])`
 - No dependencies: Runs after every render.
 - Empty array `[]`: Runs once on mount.
 - With dependencies: Runs when those values change.
- Side effects include API calls (e.g., via `fetch` or Axios), setting timers (`setTimeout`), or updating document title.

Example Code Snippet (API Call):

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

function DataFetcher() {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(setData);
  }, []); // Empty array: Fetch once on mount

  return <div>{data ? JSON.stringify(data) : 'Loading...'}</div>;
}
```

Example (DOM Update):

```
useEffect(() => {  
  document.title = `Count: ${count}`; // Update title on count change  
}, [count]);
```

5. Component Lifecycle in Functional Components

- Functional components with Hooks mimic class lifecycle methods:
 - **Mounting:** Initial render; `useEffect(() => {}, [])` acts like `componentDidMount`.
 - **Updating:** Re-renders on state/prop changes; `useEffect(() => {}, [deps])` acts like `componentDidUpdate`.
 - **Unmounting:** Component removal; Cleanup function in `useEffect` acts like `componentWillUnmount`.
- Rendering is synchronous
- Order: Render → Commit (DOM update) → Effects run.

Lifecycle Flow Example:

1. Component mounts → Render → `useEffect` callback (if deps met).
2. State changes → Re-render → `useEffect` cleanup (if any) → `useEffect` callback.

6. Cleanup Functions in useEffect

- Return a function from `useEffect` callback to perform cleanup.
- Runs before the next effect or on unmount.
- Useful for: Clearing timers, unsubscribing from events/WebSockets, canceling API requests.
- Prevents memory leaks (e.g., lingering event listeners).

Example Code Snippet (Timer with Cleanup):

```
useEffect(() => {  
  const timer = setInterval(() => {  
    console.log('Tick');  
  }, 1000);  
  
  return () => clearInterval(timer); // Cleanup on unmount or deps change  
}, []);
```

Example (Event Listener):

```
useEffect(() => {  
  const handleResize = () => console.log('Resized');  
  window.addEventListener('resize', handleResize);  
  
  return () => window.removeEventListener('resize', handleResize);  
}, []);
```


Q & A

Narasimha Rao T

tnrao.trainer@gmail.com