upGrad ENTERPRISE

React Hooks, State & Effect

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).

What is Prop Drilling in React?

1. What is Prop Drilling in React?

- Prop drilling refers to the process of passing data (props) through multiple levels of nested components in a React application, even when intermediate components do not directly use that data.
- This creates unnecessary coupling and can make the codebase harder to maintain as the component tree grows deeper.

Key Characteristics:

- **Unavoidable in Small Apps**: In simple hierarchies, it's fine, but it becomes problematic in large-scale apps.
- **Issues Caused**:
  - Increases boilerplate code (repetitive prop passing).
  - Makes refactoring difficult (changing a prop name requires updates across all levels).
  - Reduces component reusability.
  - Can lead to performance issues if props are complex objects.

Example Scenario:

Imagine a UserProfile component that needs user data from the top-level App component, but it's buried 5 levels deep:

```
App → Header → Navigation → Sidebar → UserProfile
```

You'd pass `user` prop from App all the way down, even if Header, Navigation, etc., don't use it.

2. Different Solutions to Address Prop Drilling

To avoid prop drilling, React provides built-in and third-party tools for global state management. Common solutions include:

- **React Context API**: Built-in way to share state across the component tree without manual prop passing. (Ideal for medium-sized apps.)
- **State Management Libraries**:
  - **Redux**: Centralized store for predictable state management. (Best for complex apps with many interactions.)
  - **Zustand**: Lightweight alternatives to Redux for simpler needs.

Choose based on app complexity: Context for simple sharing, Redux for advanced logic.

3. Context API

Introduction

React Context API is a built-in feature (introduced in React 16.3) that allows you to share state across the entire component tree without prop drilling. It creates a "context" that components can subscribe to via a Provider and consume via a Consumer or the `useContext` hook.

- **Use Cases**: Theme switching, user authentication, localization—any data needed by many components.
- **Pros**: No external dependencies; simple for medium apps.
- **Cons**: Can cause unnecessary re-renders if not optimized; not ideal for highly complex state logic.

Steps to Implement Context API

1. **Create a Context**: Use `React.createContext()` to define the context object.
2. **Provide the Value**: Wrap your app (or subtree) in a `<Context.Provider value={...}>` and pass the state/value as `value`.
3. **Consume the Context**:
    - **Class Components**: Use `<Context.Consumer>` with a render prop.
    - **Functional Components**: Use the `useContext(Context)` hook.

upGrad ENTERPRISE

Q & A

Narasimha Rao T
tnrao.trainer@gmail.com