### What Are Hooks?

**Definition:** Hooks are functions introduced in React 16.8 that let you "hook into" React state and lifecycle features from function components. They provide a more direct API to the React concepts you already know and love. Hooks can only be used inside function components or custom hooks and they offer a way to reuse stateful logic without changing the component hierarchy, which makes it easier to share hooks among components or with the community.

**Purpose:** Hooks allow you to use state and other React features without writing a class. They simplify state management and side effects, making functional components more powerful.

**Why do we need hooks?**

Hooks solve two problems for developers.

* Clean code
* With less code, achieve similar class-based functionality.

1. **UseState**

**Definition:** ‘useState’ is a hook that lets you add state to functional components. It is the most basic and commonly used hook for state management in function components. When you call useState, it returns an array with two elements: the current state value and a function to update that state.

**Purpose:** Manage local state in a functional component, enabling components to remember information between renders.

**How it Works:** ‘useState’ takes an initial state value as an argument and returns an array where the first element is the current state value, and the second element is a function that allows you to update that state value. Each time you call this function, React will re-render the component with the new state value.

**Example**: Counter

import React, {useState} from 'react'; // Import React and the useState hook

function Counter() {

// Declare a state variable 'count' initialized to 0

// 'count' is the current state value

// 'setCount' is the function to update the state

const [count, setCount] = useState(0);

return (

<div>

<p>You clicked {count} times</p> {/\* Display the current count \*/}

<button onClick={() => setCount(count + 1)}> {/\* Increment count when button is clicked \*/}

Click me

</button>

</div>

);

}

export default Counter; // Export the Counter component

1. **UseEffect**

**Definition:** ‘useEffect’ is a hook that lets you perform side effects in function components. Side effects are operations that interact with external systems, like fetching data from an API, subscribing to a data stream, or manually changing the DOM. useEffect can replace the lifecycle methods componentDidMount, componentDidUpdate, and componentWillUnmount used in class components.

**Purpose:** Perform side effects in a functional component (e.g., data fetching, subscriptions).

**How it Works:** ‘useEffect’ takes two arguments: a function that contains the side effect, and an optional array of dependencies. The function runs after the render is committed to the screen. If the dependencies change, the effect runs again. If you provide an empty array, the effect runs only once after the initial render.

**Example: Update Document**

import React, { useState, useEffect } from 'react'; // Import React and the useState, useEffect hooks

function DocumentTitle() {

const [count, setCount] = useState(0); // Declare a state variable 'count' initialized to 0

useEffect(() => {

// This effect updates the document title whenever 'count' changes

document.title = `You clicked ${count} times`;

}, [count]); // Dependency array with 'count' ensures effect runs only when 'count' changes

return (

<div>

<p>You clicked {count} times</p> {/\* Display the current count \*/}

<button onClick={() => setCount(count + 1)}> {/\* Increment count when button is clicked \*/}

Click me

</button>

</div>

);

}

export default DocumentTitle; // Export the DocumentTitle component

1. **UseContext**

**Definition:** ‘useContext’ is a hook that lets you access context values directly from a function component, without needing to use a Consumer component. Context provides a way to pass data through the component tree without having to pass props down manually at every level.

**Purpose:** Simplify consuming context values in functional components, making it easier to read and update context values without intermediate components.

**How it Works:** ‘useContext’ accepts a context object (the value returned from React.createContext) and returns the current context value for that context. The current context value is determined by the nearest <Context.Provider> above the calling component in the tree.

**Example: Using a Theme Context**

import React, { useContext } from 'react'; // Import React and the useContext hook

const ThemeContext = React.createContext('light'); // Create a context with 'light' as the default value

function ThemedButton() {

const theme = useContext(ThemeContext); // Access the context value using useContext

return (

<button style={{ background: theme === 'light' ? '#fff' : '#333', color: theme === 'light' ? '#000' : '#fff' }}>

{/\* Button style changes based on the context value \*/}

Themed Button

</button>

);

}

export default ThemedButton; // Export the ThemedButton component

1. **UseReducer**

**Definition:** ‘useReducer’ is a hook that lets you manage complex state logic in function components. It is an alternative to useState for more complex state logic that involves multiple sub-values or when the next state depends on the previous one. It is similar to Redux but built into React.

**Purpose:** Manage complex state logic with a reducer function.

**How it Works:** ‘useReducer’ takes two arguments: a reducer function and an initial state. It returns an array with the current state and a dispatch function. The reducer function determines the next state based on the current state and the action dispatched.

**Example: Simple Counter with Reducer**

import React, { useReducer } from 'react'; // Import React and the useReducer hook

const initialState = { count: 0 }; // Initial state with 'count' set to 0

function reducer(state, action) {

// Reducer function that handles state changes based on action type

switch (action.type) {

case 'increment':

return { count: state.count + 1 }; // Increment count

case 'decrement':

return { count: state.count - 1 }; // Decrement count

default:

throw new Error(); // Throw error for unknown action types

}

}

function CounterWithReducer() {

// Use reducer to manage state

const [state, dispatch] = useReducer(reducer, initialState);

return (

<div>

<p>Count: {state.count}</p> {/\* Display the current count \*/}

<button onClick={() => dispatch({ type: 'increment' })}> {/\* Dispatch 'increment' action \*/}

+

</button>

<button onClick={() => dispatch({ type: 'decrement' })}> {/\* Dispatch 'decrement' action \*/}

-

</button>

</div>

);

}

export default CounterWithReducer; // Export the CounterWithReducer component

1. **UseRef**

**Definition:** ‘useRef’ is a hook that lets you create a mutable object which persists for the lifetime of the component. It can be used to store a reference to a DOM element or a value that does not cause re-renders when updated.

**Purpose:** Access and manipulate DOM elements or store any mutable value that persists across renders without causing re-renders.

**How it Works:** ‘useRef’ returns a mutable ref object whose .current property is initialized to the passed argument. This object persists for the entire lifetime of the component and can be updated without causing a re-render.

**Example: Focus Input Field**

import React, { useRef } from 'react'; // Import React and the useRef hook

function FocusInput() {

const inputRef = useRef(null); // Create a ref with initial value null

const handleFocus = () => {

// Focus the input element when the button is clicked

inputRef.current.focus();

};

return (

<div>

<input ref={inputRef} type="text" /> {/\* Attach ref to the input element \*/}

<button onClick={handleFocus}> {/\* Call handleFocus when button is clicked \*/}

Focus the input

</button>

</div>

);

}

export default FocusInput; // Export the FocusInput component

1. **UseMemo**

**Definition:** ‘useMemo’ is a hook that lets you memoize the result of a calculation to optimize performance. It prevents expensive calculations on every render by caching the result based on the dependencies.

**Purpose:** Memoize expensive calculations to improve performance.

**How it Works:** ‘useMemo’ takes a function and an array of dependencies. It returns a memoized value that only recalculates if one of the dependencies has changed since the last render.

**Example: Expensive Calculation**

import React, { useState, useMemo } from 'react'; // Import React, useState, and useMemo hooks

function ExpensiveComponent() {

const [count, setCount] = useState(0); // Declare a state variable 'count' initialized to 0

const expensiveCalculation = useMemo(() => {

// Simulate an expensive calculation

let result = 0;

for (let i = 0; i < 1000000000; i++) {

result += i;

}

return result;

}, []); // Memoize the calculation, run only once

return (

<div>

<p>Expensive Calculation: {expensiveCalculation}</p> {/\* Display the result of the calculation \*/}

<p>Count: {count}</p> {/\* Display the current count \*/}

<button onClick={() => setCount(count + 1)}> {/\* Increment count when button is clicked \*/}

Increment

</button>

</div>

);

}

export default ExpensiveComponent; // Export the ExpensiveComponent

1. **UseCallback**

**Definition:** ‘useCallback’ is a hook that lets you memoize callback functions to prevent unnecessary re-renders. It returns a memoized version of the callback that only changes if one of the dependencies has changed.

**Purpose:** Optimize performance by memorizing callback functions to avoid creating new functions on every render, which can prevent unnecessary re-renders of child components.

**How it Works:** ‘useCallback’ takes a function and an array of dependencies. It returns a memoized callback that only changes if one of the dependencies has changed since the last render.

**Example: Memoized Callback**

import React, { useState, useCallback } from 'react'; // Import React, useState, and useCallback hooks

function CallbackComponent() {

const [count, setCount] = useState(0); // Declare a state variable 'count' initialized to 0

const handleClick = useCallback(() => {

// Log message when button is clicked

console.log('Button clicked');

}, []); // Memoize the function

return (

<div>

<p>Count: {count}</p> {/\* Display the current count \*/}

<button onClick={() => setCount(count + 1)}> {/\* Increment count when button is clicked \*/}

Increment

</button>

<button onClick={handleClick}> {/\* Call handleClick when button is clicked \*/}

Log to Console

</button>

</div>

);

}

export default CallbackComponent; // Export the CallbackComponent

1. **UseLayoutEffect**

**Definition:** ‘useLayoutEffect’ is a hook that lets you perform side effects that need to happen before the DOM is painted. It runs synchronously after all DOM mutations and before the browser paints, which can be useful for measuring layout or making DOM adjustments.

**Purpose:** Perform side effects that need to happen before the DOM is painted.

**How it Works:** ‘useLayoutEffect’ takes two arguments: a function that contains the side effect, and an optional array of dependencies. The function runs synchronously after the DOM updates but before the paint.

**Example: Measure DOM Element Size**

import React, { useLayoutEffect, useRef, useState } from 'react'; // Import React, useLayoutEffect, useRef, and useState hooks

function LayoutEffectComponent() {

const divRef = useRef(null); // Create a ref with initial value null

const [height, setHeight] = useState(0); // Declare a state variable 'height' initialized to 0

useLayoutEffect(() => {

// Set the height of the div element after it is rendered

setHeight(divRef.current.offsetHeight);

}, []); // Empty dependency array ensures effect runs only once

return (

<div>

<div ref={divRef} style={{ height: '100px', background: 'lightblue' }}>

Measure my height {/\* The div element to measure \*/}

</div>

<p>Height: {height}px</p> {/\* Display the measured height \*/}

</div>

);

}

export default LayoutEffectComponent; // Export the LayoutEffectComponent

1. **UseImperativeHandle**

**Definition:** ‘useImperativeHandle’ is a hook that lets you customize the instance value that is exposed when using ref in parent components. It can be useful for exposing imperative methods to parent components.

**Purpose:** Customize the ref object that is exposed to parent components, allowing you to define methods or properties that the parent can call.

**How:** ‘useImperativeHandle’ takes three arguments: a ref object, a function that returns an object containing the imperative handle, and an optional dependency array. The function runs whenever the dependencies change and allows you to customize the value exposed by the ref.

**Example: Customizing Ref Behavior**

import React, { useRef, forwardRef, useImperativeHandle } from 'react'; // Import React, useRef, forwardRef, and useImperativeHandle hooks

const CustomInput = forwardRef((props, ref) => {

const inputRef = useRef(); // Create a ref with initial value null

useImperativeHandle(ref, () => ({

focus: () => {

inputRef.current.focus(); // Define a custom focus method

},

clear: () => {

inputRef.current.value = ''; // Define a custom clear method

}

}));

return <input ref={inputRef} type="text" />; // Attach ref to the input element

});

function ImperativeHandleComponent() {

const inputRef = useRef(); // Create a ref with initial value null

return (

<div>

<CustomInput ref={inputRef} /> {/\* Attach ref to the CustomInput component \*/}

<button onClick={() => inputRef.current.focus()}> {/\* Call the custom focus method \*/}

Focus Input

</button>

<button onClick={() => inputRef.current.clear()}> {/\* Call the custom clear method \*/}

Clear Input

</button>

</div>

);

}

export default ImperativeHandleComponent; // Export the ImperativeHandleComponent