**ReactJS Hooks**

Hook are special functions in React introduced in React 16.8 that allow us to use state and react lifecycle methods in functional components. They make it possible to manage component state, handle side effects, and access context without writing class based components.

**Features of hooks**

1. **State Management in functional components**
2. **Access to lifecycle methods**
3. **Reusable logic**
4. **Backward compatibility**

|  |  |  |
| --- | --- | --- |
| Aspect | hooks | Normal functions |
| React integration | Hooks are tightly integrated with react features like state and lifecycles | Normal functions are for general purpose |
| State management | Hooks like useState enable managing state directly in functional components | Normal functions cant manage state unless passed explicitly as arguments. |
| Rules | Hooks follow strict rules (e.g., only call hooks at the top level of a component / inside custom hooks). | Normal functions can be called anywhere without constraints. |
| Side Effects | Hooks like useEffect allow side effects to be managed within components | Normal functions cant manage side effects in a react specific way. |
| Reusable logic | Custom hooks enable sharing stateful logic across components. | Normal functions can only share non stateful logic unless passed props / context. |
| Use in JSX | Hooks are designed to work seamlessly in JSX components. | Normal function are used as utilities and helpers, don’t directly state / react lifecycles |

Hooke are enable react developers to

Write cleaner, functional component without relying on classes

Manage state and lifecycle in a way that’s more intuitive and reusable

Improve code reusability through custom hooks.

useMemo

useMemo hook is use to memorize the complex calculations and it returns a memorized value.It ensure that expesive compotation are only performed when their dependency changes.It is used for optimizing the performance of our application.

When to use useMemo

When we have a state dependent on expensive calculation, but we don’t want to run the calculation on every render.

When we declare array / object inside a component, its references changes on every render, even though the value remains same. Wrapping the values inside useMemo maintains referential equality and prevents unnecessary re-renders. This is essential when theres useEffect dependent on the array / object.

When we are rendering lists using Array.map that do not need to change unless a certain state value changes.

Memorization is a technique that involves storing the result of a computation and reusing it later, instead of recomputing it everytim.

Feature of useMemo

1. Optimizes performance  
   Avoids recalculating values unnecessarily, especially for heavy computations.
2. Memorizes computed values

The memorized value is stored and reused until the dependencies changes.

1. Read only

The memorized value cannot be modified directly.

Pros of useMemo

* Prevents unnecessary calculation, improving app performance.
* Easy to use for optimizing computationally heavy operations.

Cons of useMemo

* Adds complexity if overused in simple components.
* Requires careful dependency management to avoid stale values.

useCallback

useCallback is a react hook that memorizes a callback function. It ensures that the same function reference is returned unless its dependencies change. This is useful for optimizing components that depend on stable function references.

Feature of useCallback

* Stable function references

Prevents recreation of functions, ensuring child components don’t rerender unnecessarily.

* Optimizes performance

Particularly effective when passing function as props to memorized child components.

Pros of useCallback

* Ensures stable function references, reducing unnecessary rerenders
* Useful when functions are passed as props to child components.

Cons of useCallback

* Adds complexity if overused
* Only provides benefits in components optimized with React.memo

useLayoutEffect

useLayoutEffect hook is used to perform side effects that require synchronous interaction with the DOM after rendering but before the browser paints the screen.

It runs synchronously after react renders the component and update the DOM.

Ideal for tasks requiring DOM measurement, layout adjustments, synchronous operations dependent on updated DOM.

Feature of useLayoutEffect

1. Execute before the browser repaints the screen.
2. Access to updated DOM elements and layout data
3. Runs synchronously, ensuring no flickering / intermediate states are visible.
4. Complements useEffect for cases where the UI must match the DOM immediately after rendering.

Pros of useLayoutEffect

1. Precise timing
2. Immediate effect
3. Enhance control

Cons of useLayoutEffect

1. Performance impact
2. Complex debugging
3. Overuse issue

When to use Each

useEffect

When the effect doesnt require precise control over the DOM and layout updates, like data fetching / subscriptions.

useLayoutEffect

when we need to synchronize UI elements with DOM changes to prevent layout inconsistencies / flickering.

useRef Hook

useRef hook use to create a reference to a DOM element and keep track of variables without causing rerendering.

useRef in react returns a mutable object that doesn’t trigger re rendering of the component when they change and persist throughout their lifetime. This object has a current property that hold a mutable value.

Feature of useRef

* Mutable reference

It stores a mutable value that doenst cause a rerenders when updated.

* Accessing DOM nodes  
  It helps to access a DOM element directly.
* Doesn’t trigger rerenders

The useRef react hook allows us to save values between renders

Use cases

* Accessing DOM elements

import React, { useRef } from 'react';

function InputFocus() {

  const inputRef = useRef(null);

  const handleFocus = () => {

    inputRef.current.focus(); // Focuses the input element

  };

  return (

    <div>

      <input ref={inputRef} type="text" />

      <button onClick={handleFocus}>Focus Input</button>

    </div>

  );

}

* Storing Mutable Data across renders

import React, { useRef } from 'react';

function Timer() {

  const count = useRef(0);

  const increment = () => {

    count.current += 1;

    console.log(count.current); // Logs updated count without re-rendering

  };

  return <button onClick={increment}>Increment</button>;

}

Pros of useRef

* Efficient for accessing and manipulating DOM elements
* Great for storing mutable data that persist across renders
* Prevents unnecessary rerenders.

Cons of useRef

* Misuse for state management can lead to inconsistent UI since useRef updates are don’t tracked by React.
* Overuse can lead to reduced readability and maintainability.

useReducers

useReducer hook use to handle complex state manipulation like handling multiple states that rely on complex logic and update.useReducer hook is alternative to useState hook and we use it when the react components need to be optimized / when the next state value is dependent upon the previous state value.

reducer it is a function that determines state changes based on the current state and action.

initialState The initial value of the state

dispatch A function to send actions to the reducer

Feature of useReducer

Centralized state logic

Encapsulates all state transition logic in the reducer function.

Efficient for complex state

Useful when managing multiple related state variables / intricate state transitions.

Immutable state updates

Ensure state changes follow react principle of immutability.

Use cases

Simple counter

import React, { useReducer } from 'react';

const initialState = { count: 0 };

function reducer(state, action) {

  switch (action.type) {

    case 'increment':

      return { count: state.count + 1 };

    case 'decrement':

      return { count: state.count - 1 };

    default:

      return state;

  }

}

function Counter() {

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

  return (

    <div>

      <p>Count: {state.count}</p>

      <button onClick={() => dispatch({ type: 'increment' })}>Increment</button>

      <button onClick={() => dispatch({ type: 'decrement' })}>Decrement</button>

    </div>

  );

}

Complex state management

import React, { useReducer } from "react";

const initialState = { tasks: [] };

function reducer(state, action) {

  switch (action.type) {

    case "add":

      return { tasks: [...state.tasks, action.payload] };

    case "remove":

      return {

        tasks: state.tasks.filter((task, index) => index !== action.index),

      };

    default:

      return state;

  }

}

function TaskManager() {

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

  const addTask = () => {

    const task = prompt("Enter a new task:");

    dispatch({ type: "add", payload: task });

  };

  const removeTask = (index) => {

    dispatch({ type: "remove", index });

  };

  return (

    <div>

      <button onClick={addTask}>Add Task</button>

      <ul>

        {state.tasks.map((task, index) => (

          <li key={index}>

            {task} <button onClick={() => removeTask(index)}>Remove</button>

          </li>

        ))}

      </ul>

    </div>

  );

}

Pros of useReducer

Better structure for manging complex state logic

Makes state transition predictiable and easier to debug

Encourages use of pure functions

Cons of useReducer

Can be overuse for simple state management

May introduce additional boilerplate.