React Hooks

Hooks is special function that allows to use states and other react features in functional compomnent.

React Hooks are simple JavaScript functions that we can use to isolate the reusable part from a functional component. Hooks can be stateful and can manage side-effects

Although Hooks generally replace class components, there are no plans to remove classes from React.

**Note:** Hooks will not work in React class components.

Why are hooks important in React?

React hooks make component development easier while also improving readability and organization. They allow for code reuse, improve performance, and are compatible with functional programming.

What are the benefits of hooks?

With Hooks, you can extract stateful logic from a component so it can be tested independently and reused. Hooks allow you to reuse stateful logic without changing your component hierarchy. This makes it easy to share Hooks among many components or with the community.

What are the rules of Hooks?

**Don't call Hooks inside loops, conditions, or nested functions.**

* ✅ Call them at the top level in the body of a function component.
* ✅ Call them at the top level in the body of a custom Hook.

The most common React hooks are:

UseState useCallback useReducer

UseEffect useMemo customhooks

UseRef useContext

UseState

UseState is react hook, that create an state variables. which helps to track the state in the components and update the user interface when the state value is changed.

**Structure of React useState hook**

This hook takes some initial state and returns two value. The first value contains the state and the second value is a function that updates the state.

**Syntax:**

const [var, setVar] = useState(0);

**Internal working of useState hook**

* **useState()**creates a new cell in the functional component’s memory object.
* New state values are stored in this cell during renders.
* The stack pointer points to the latest cell after each render.
* Deliberate user refresh triggers stack dump and fresh allocation.
* The memory cell preserves state between renders, ensuring persistence.

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

**function** App() {

**const** [click, setClick] = useState(0);

**return** (

<div>

<p>You've clicked {click} times!</p>

<p>The number of times you have clicked

is {click % 2 == 0 ? 'even!' : 'odd!'}</p>

<button onClick={() => setClick(click => click + 1)}>

Click me

</button>

</div>

);

}

**export** **default** App;

**React useEffect** hook handles the effects of the dependency array. The useEffect Hook allows us to perform side effects on the components. fetching data, directly updating the DOM and timers are some side effects. It is called every time any state if the dependency array is modified or updated.

**What is useEffect hook in React?**

The useEffect in ReactJS is used to handle the side effects such as fetching data and updating DOM. This hook runs on every render but there is also a way of using a dependency array using which we can control the effect of rendering.

**How does it work?**

* You call useEffect with a callback function that contains the side effect logic.
* By default, this function runs after every render of the component.
* You can optionally provide a dependency array as the second argument.
* The effect will only run again if any of the values in the dependency array change.

**Structure of useEffect hook**

The useEffect hook syntax accepts two arguments where the second argument is optional

**React useEffect Hook Syntax:**

useEffect(<FUNCTION>, <DEPENDECY>)

**React useEffect Hook ShortHand for:**

* **FUNCTION:** contains the code to be executed when useEffect triggers.
* **DEPENDENCY:**is an optional parameter, useEffect triggers when the given dependency is changed

import { useState, useEffect } from "react";

function HookCounterOne() {

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

useEffect(() => {

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

}, [count]);

return (

<div>

<button onClick={() => setCount((prevCount) => prevCount + 1)}>

Click {count} times{" "}

</button>

</div>

);

}

export default HookCounterOne;

**useRef Hook**

The useRef is a hook that allows to directly create a reference to the DOM element in the functional component.

**Reasons to use useRef hook**

The main use of useRef hook is to access the DOM elements in a more efficient way as compared to simple refs. Since useRef hooks preserve value across various re-renders and do not cause re-renders whenever a value is changed they make the application faster and helps in caching and storing previous values.

**Structure of useRef hook**

It accepts only one initial value.

**Syntax:**

const refContainer = useRef(initialValue);

The useRef returns a mutable ref object. This object has a property called .current. The value is persisted in the refContainer.current property. These values are accessed from the current property of the returned object. The .current property could be initialised to the passed argument initialValue e.g. useRef(initialValue). The object can persist a value for a full lifetime of the component.

function App() {

  // Creating a ref object using useRef hook

  const focusPoint = useRef(null);

  const onClickHandler = () => {

    focusPoint.current.value =

      "The quick brown fox jumps over the lazy dog";

      focusPoint.current.focus();

  };

  return (

    <Fragment>

      <div>

        <button onClick={onClickHandler}>

         ACTION

        </button>

      </div>

      <label>

       Click on the action button to

       focus and populate the text.

      </label><br/>

      <textarea ref={focusPoint} />

    </Fragment>

  );

};

export default App;

**useMemo Hook**

useMemo hook is used to get the memoized value of a function in react components. It works on the concept of memoization which refers to caching the output of a function for a given argument to save the computation time.

**useMemo Hook in React**

The **useMemo Hook returns a memoized value**and **prevents the application from unnecessary re-renders**. It is useful in heavy computations and processes when using functional components.

**Syntax:**

const memoizedValue = useMemo(functionThatReturnsValue, arrayDependencies)

**import** React, { useState } **from** "react";

**function** App() {

**const** [number, setNumber] = useState(0);

*// Using useMemo*

**const** squaredNum = useMemo(() => {

**return** squareNum(number);

}, [number]);

**const** [counter, setCounter] = useState(0);

*// Change the state to the input*

**const** onChangeHandler = (e) => {

setNumber(e.target.value);

};

*// Increases the counter by 1*

**const** counterHander = () => {

setCounter(counter + 1);

};

**return** (

<div className="App">

<h1>Welcome to Geeksforgeeks</h1>

<input

type="number"

placeholder="Enter a number"

value={number}

onChange={onChangeHandler}>

</input>

<div>OUTPUT: {squaredNum}</div>

<button onClick={counterHander}>

Counter ++

</button>

<div>Counter : {counter}</div>

</div>

);

}

*// Function to square the value*

**function** squareNum(number) {

console.log("Squaring will be done!");

**return** Math.pow(number, 2);

}

**export** **default** App;

**Conclusion**

**useMemo hook**is used to enchance the performance in react applications. It stores the output for given arguments as cache and directly return it for same values, preventing unnecessary rerenders and computations.

**useReducer Hook**

The **useReducer** Hook is the better alternative to the [**useState**](https://www.geeksforgeeks.org/reactjs-usestate-hook/) hook and is generally more preferred over the **useState** hook when you have complex state-building logic or when the next state value depends upon its previous value or when the components are needed to be optimized.

The **useReducer** hook takes three arguments including reducer, initial state, and the function to load the initial state lazily.

**Syntax:**

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

**Parameters:**

* **state**: The current state value.
* **dispatch:** A function that allows you to trigger state transitions by dispatching actions.
* **reducer:** A function that specifies how the state should change in response to dispatched actions.

## **When to use useState ?**

* For simple state management with independent updates.
* When the state transitions are not complex and do not depend on the previous state.
* When dealing with a single piece of state.

## **When to use useReducer ?**

* For more complex state logic.
* When state transitions depend on the previous state.
* When managing multiple related pieces of state.

## Difference between useReducer and useState hook:

| **Feature** | **useState** | **useReducer** |
| --- | --- | --- |
| Complexity | Simple and straightforward. | More complex, suitable for intricate state transitions. |
| Use Case | Ideal for independent state updates. | Suitable for managing complex state logic, especially when transitions depend on the previous state. |
| Syntax | `const [state, setState] = useState(initialState);` | `const [state, dispatch] = useReducer(reducer, initialState);` |
| State Logic | Limited for simple state scenarios. | Allows for centralized state transition logic using a reducer function. |
| Dependencies | No dependencies on the previous state. | Can take advantage of the previous state in the reducer function. |
| Performance | Generally performs well for simple scenarios. | Can be more optimal for complex state logic by preventing unnecessary re-renders. |

**Props Drilling**:

Prop drilling is basically a situation when the same data is being sent at almost every level due to requirements in the final level.

import React, { useState } from "react";

function Parent() {

    const [fName, setfName] = useState("firstName");

    const [lName, setlName] = useState("LastName");

    return (

        <>

            <div>This is a Parent component</div>

            <br />

            <ChildA fName={fName} lName={lName} />

        </>

    );

}

function ChildA({ fName, lName }) {

    return (

        <>

            This is ChildA Component.

            <br />

            <ChildB fName={fName} lName={lName} />

        </>

    );

}

function ChildB({ fName, lName }) {

    return (

        <>

            This is ChildB Component.

            <br />

            <ChildC fName={fName} lName={lName} />

        </>

    );

}

function ChildC({ fName, lName }) {

    return (

        <>

            This is ChildC component.

            <br />

            <h3> Data from Parent component is as follows:</h3>

            <h4>{fName}</h4>

            <h4>{lName}</h4>

        </>

    );

}

export default Parent;

**Why not to use prop drilling?**

1. **Code Complexity:** As components grow, prop drilling increases code complexity as it is difficult to track the flow of data through various components.
2. **Reduced Maintainability:** It will become very challenging to maintain the code with prop drilling. When changes are required in the data flow, you need to make changes in many components.
3. **Performance Overhead:** We have to pass props through unnecessary intermediary components which can impact performance.
4. **Decreased Component Reusability:** Components used in prop drilling become tightly coupled to the structure of the parent components, so it very difficult to use it on other parts of the application.
5. **Increased Development Time:** Prop drilling often requires additional planning to implement. This can slow down the development process, especially when the component hierarchies is complex.

## **Solve prop drilling with UseContext Hooks**

**useContext Hook:**

React Context is a way to manage state globally.

It can be used together with the useState Hook to share state between deeply nested components more easily than with useState alone.

 The useContext hook is based on Context API and works on the mechanism of Provider and Consumer. Provider needs to wrap components inside Provider Components in which data have to be consumed.

import React, { useState, useContext } from "react";

let context = React.createContext(null);

function Parent() {

    const [fName, setfName] = useState("firstName");

    const [lName, setlName] = useState("LastName");

    return (

        <context.Provider value={{ fName, lName }}>

            <div>This is a Parent component</div>

            <br />

            <ChildA />

        </context.Provider>

    );

}

function ChildA() {

    return (

        <>

            This is ChildA Component.

            <br />

            <ChildB />

        </>

    );

}

function ChildB() {

    return (

        <>

            This is ChildB Component.

            <br />

            <ChildC />

        </>

    );

}

function ChildC() {

    const { fName, lName } = useContext(context);

    return (

        <>

            This is ChildC component.

            <br />

            <h3> Data from Parent component is as follows:</h3>

            <h4>{fName}</h4>

            <h4>{lName}</h4>

        </>

    );

}

export default Parent;

**What is Context API?**

Context API is used to pass global variables anywhere in the code. It helps when there is a need for sharing state between a lot of nested components. It is light in weight and easier to use, to create a context just need to call React.createContext(). No need to install other dependencies or third-party libraries like redux for state management.

**Why is context API used?**

Context API solves the problem of prop drilling in React. Prop Drilling occurs when data is to be passed between multiple layers before finally sending it to the required component. This makes the application slower. This problem is solved by Context API as it creates global variables to be used throughout the application without any middle components involved.It is also easier to use than React Redux.

**Working of Context API**

To work with Context API we need React.createContext. It has two properties Provider and Consumer. The Provider acts as a parent it passes the state to its children whereas the Consumer uses the state that has been passed.

**Benefits of Context API over React Redux**

* In Redux we have to manipulate or update multiple files to add even a single feature but in Context it can be done in much lesser lines of code
* One way data binding in React is maintained using Context whereas Redux violates it.
* Multiple stores/contexts can be created using Context whereas Redux creates just a single store

**Difference between Redux and Context API:**

| Feature | Redux | Context API |
| --- | --- | --- |
| Middleware | Middlewares present. | Middlewares absent. |
| State management approach | Centralized | Decentralized |
| Data Flow | Unidirectional flow of data. | Bidirectional flow of data. |
| API | Actions, reducers, middleware | Context.Provider, Context.Consumer |
| Debugging | Dedicated Redux development tools for debugging. | No tools for debugging. |

**Conclusion:**

* Redux is preferred for managing larger and more complex states in applications.
* Context API is more suitable for managing smaller and localized states.
* It’s important to choose the appropriate state management solution based on the specific requirements and complexity of your application.

**useCallback Hook**

The React useCallback Hook returns a memoized callback function.

Think of memoization as caching a value so that it does not need to be recalculated.

This allows us to isolate resource intensive functions so that they will not automatically run on every render.

The useCallback Hook only runs when one of its dependencies update.

This can improve performance.

The useCallback and useMemo Hooks are similar. The main difference is that useMemo returns a memoized value and useCallback returns a memoized function.

This useCallback hook is used when you have a component in which the child is rerendering again and again without need.

**React useCallback Hook Syntax:**

const memoizedCallback = useCallback(  
 () => {  
 doSomething(a, b);  
 }, [a, b],  
);

**React useCallback Hook Usage**

* This is useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders.
* It improves the performance by reducing the unnecessary computations and providing already stored callback.
* It is similar to[**React useMemo Hook**](https://www.geeksforgeeks.org/react-js-usememo-hook/), the difference is it returns a callback and **useMemo** returns a value.

***Note:****Memoization is like storing the values as cache that need not to be calculated again and again.*

import React, { useState, useCallback } from "react";

var funccount = new Set();

const App = () => {

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

const [number, setNumber] = useState(0);

const incrementCounter = useCallback(() => {

setCount(count + 1);

}, [count]);

const decrementCounter = useCallback(() => {

setCount(count - 1);

}, [count]);

const incrementNumber = useCallback(() => {

setNumber(number + 1);

}, [number]);

funccount.add(incrementCounter);

funccount.add(decrementCounter);

funccount.add(incrementNumber);

return (

<div

style={{

display: "flex",

flexDirection: "column",

textAlign: "center",

justifyContent: "end",

margin: "auto",

marginTop: "20px",

width: "350px",

padding: "50px",

height: "300px",

fontSize: "20px",

boxShadow: "0px 2px 8px 4px grey",

borderRadius: "5px",

}}

>

<h4>React Example for useCallback Hook</h4>

<p>Count: {count}</p>

<p>Function Count: {funccount.size}</p>

<button onClick={incrementCounter}>

Increase counter

</button>

<button onClick={decrementCounter}>

Decrease Counter

</button>

<button onClick={incrementNumber}>

Increase number

</button>

</div>

);

};

export default App;

Custom Hooks

Hooks are reusable functions.

When you have component logic that needs to be used by multiple components, we can extract that logic to a custom Hook.

Custom Hooks start with "use".