Contents

[What’s a Hook? 2](#_Toc192359755)

[useState 2](#_Toc192359756)

[useEffect 3](#_Toc192359757)

[useContext 4](#_Toc192359758)

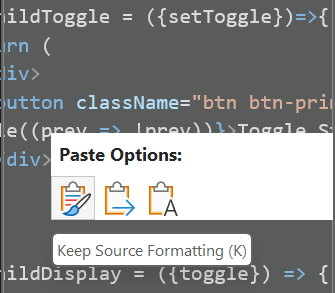
[useReducer 7](#_Toc192359759)

[useRef 9](#_Toc192359760)

[useImperativeHandle 13](#_Toc192359761)

[useCallback 17](#_Toc192359762)

Pasting in word with formatted code from vs code (right click in work and click on paste brush icon)



# What’s a Hook?

**What is a Hook?** A Hook is a special function that lets you “hook into” React features. For example, useState is a Hook that lets you add React state to function components.

## useState

The React useState Hook allows us to track state in a function component.

State generally refers to data or properties that need to be tracking in an application.

import React, { useState } from "react";

const UseStateExample = () => {

  //using vanila javascript, the dom value of cnt wont update on UI

  let cnt = 0;

  const incrementCnt = () => {

    cnt = cnt + 1;

    console.log(`cnt ${cnt}`);

  };

  // using useState,  the dom value of count  update on UI

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

  const increment = () => {

    setCount((previous) => previous + 1);

    console.log(`count ${count}`);

  };

  return (

    <div>

      <h1>{cnt}</h1>

      <button className="btn btn-primary" onClick={incrementCnt}>

        Increment Using vanila Javascript

      </button>

      <h1>{count}</h1>

      <button className="btn btn-primary" onClick={increment}>

        Increment Using React State

      </button>

    </div>

  );

};

export default UseStateExample;

## useEffect

If we want to run a function, every time a value changes, we can use useEffect hook.

It lets you run the code after the component renders, on a set of circumstances.

If I want to run a function every time state changes the value, we can use useEffect

Example of usage: We can use this for fetching data from an API, immediately when page renders

console.log("you will see useEffect running twice on page load, this is only for development  build, but for production build you see only once");

  useEffect(() => {

    console.log("runs every time  state changes");

  });

  useEffect(() => {

    console.log("runs only once during initial page render");

  }, []);

  let [value, setValue] = useState("N");

  useEffect(() => {

    console.log("runs on initial render and also if value changes");

  }, [value]);

## useContext

<https://medium.com/zestgeek/mastering-reacts-usecontext-hook-simplifying-state-management-65894e6dc431>

In the world of React development, efficient state management is key to building robust and scalable applications. While there are various state management solutions available, React provides its own built-in mechanism called the useContext hook, which offers a straightforward and elegant way to manage state across components

**Understanding the useContext Hook**

The useContext hook is a part of React’s hooks API introduced in React 16.8. It allows components to consume state or context without the need for prop drilling, which can lead to cleaner and more maintainable code. Context provides a way to share values like themes, user authentication status, or preferred language across the component tree without having to pass props down manually at every level.

Problem: If we want to pass parent values to child, then we need to pass them as part of props. Imagine we have 4 levels of parent -> child, then we need to pass 3 levels down

import UseContextExample from "./hooks/UseContextExample";

function App() {

  return (

    <div className="container-fluid mt-5 mx-5">

      <UseContextExample />

    </div>

  );

}

export default App;

import React, { useState } from 'react'

const UseContextExample = () => {

  const[toggle,setToggle]=useState(false);

    return (

    <div>

        <h1>Parent Component </h1>

        <ChildToggle setToggle={setToggle}/>

        <ChildDisplay toggle={toggle}/>

    </div>

  )

}

const ChildToggle = ({setToggle})=>{

    return (

      <div>

      <button className="btn btn-primary" type="button"   onClick={() => setToggle((prev => !prev))}>Toggle State</button>

      </div>

    )

}

const ChildDisplay = ({toggle}) => {

    return (

      <div>Current State is {toggle ?"ON":"OFF"}</div>

    )

}

export default UseContextExample;

Using Context

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

//Technically, all this components will be there in seperate files, but here we are storing in one file.

// We should import GlobalStateContext in each component

export const GlobalStateContext = createContext();

const UseContextExample = () => {

  const [toggle, setToggle] = useState(false);

  return (

    <GlobalStateContext.Provider value={{toggle,setToggle}}>

      <div>

        <h1>Parent Component</h1>

        <ChildToggle/>

        <ChildDisplay/>

      </div>

    </GlobalStateContext.Provider>

  );

};

const ChildToggle = () => {

    const {setToggle}=useContext(GlobalStateContext)

    return (

    <div>

      <button

        className="btn btn-primary"

        type="button"

        onClick={() => setToggle((prev) => !prev)}

      >

        Toggle State

      </button>

    </div>

  );

};

const ChildDisplay = () => {

    const {toggle}=useContext(GlobalStateContext)

  return <div>Current State is {toggle ? "ON" : "OFF"}</div>;

};

export default UseContextExample;

**Conclusion**

The useContext hook in React is a versatile tool for managing global state and context in your applications. Whether you’re handling user authentication, language preferences, themes, or any other shared data, useContext simplifies your code by eliminating prop drilling and providing a cleaner way to access context values. Incorporate useContext into your React projects to enhance maintainability, readability, and scalability, making your development process more efficient and enjoyable.

## useReducer

In the world of React.js, managing state is a fundamental aspect of building dynamic and interactive user interfaces. While the useState hook serves as a versatile tool for managing simpler state needs, there comes a time in every developer's journey when more complex state management requirements emerge. Enter the unsung hero of React hooks: useReducer.

import { type } from "@testing-library/user-event/dist/type";

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

const UseReducerExample = () => {

  const myReducerFunction = (state, action) => {

    switch (action.type) {

      case "increment":

        return { count: state.count + 1 };

      case "multiply":

        return { count: state.count \* 2 };

      case "square":

        return { count: state.count \* state.count };

      default:

        return { count: 1 };

    }

  };

  // useReducer is similar to useState, but if we wan to handle complex logic

  const [state, dispatch] = useReducer(myReducerFunction, { count: 0 });

  return (

    <div>

      <h1>{state.count}</h1>

      <button className="btn btn-primary mx-2" onClick={() => dispatch({ type: "increment" })}>Increment By 1</button>

      <button className="btn btn-primary mx-2" onClick={() => dispatch({ type: "multiply" })}>Multiply By 2</button>

      <button className="btn btn-primary mx-2" onClick={() => dispatch({ type: "square" })}>Square</button>

      <button className="btn btn-primary mx-2" onClick={() => dispatch({ type: "clear" })}>Clear</button>

    </div>

  );

};

export default UseReducerExample;

Understanding the Basics: At its core, the useReducer hook provides a powerful alternative to useState for managing state in React functional components. Inspired by the concept of reducers in functional programming, useReducer offers a structured approach to handling state transitions by encapsulating them within a single function.

The Anatomy of useReducer: The useReducer hook takes two parameters: a reducer function and an initial state value. The reducer function, akin to a reducer in Redux or similar state management libraries, accepts the current state and an action, and returns the new state based on the action type.

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

Harnessing the Power of Reducers: Reducers serve as the backbone of useReducer, empowering developers to manage complex state logic with ease. By centralizing state transitions within a reducer function, developers can maintain a clear and predictable state management flow, making code maintenance and debugging a breeze.

const myReducerFunction = (state, action) => {

    switch (action.type) {

      case "increment":

        return { count: state.count + 1 };

      case "multiply":

        return { count: state.count \* 2 };

      case "square":

        return { count: state.count \* state.count };

      default:

        return { count: 1 };

    }

  };

Seamless Integration with Context: One of the key advantages of useReducer is its seamless integration with React context. By combining useReducer with context, developers can efficiently manage global state in their applications, facilitating communication between components and promoting code reusability.

Optimizing Performance with Memoization: In addition to simplifying state management, useReducer empowers developers to optimize performance through memoization. By memoizing state transitions within reducers and selectively dispatching actions, developers can minimize unnecessary re-renders and enhance the responsiveness of their applications.

In conclusion, the useReducer hook represents a paradigm shift in React state management, offering developers a robust solution for handling complex state logic with elegance and efficiency. By embracing the power of reducers, developers can unlock new possibilities for building scalable and maintainable React applications. So, the next time you encounter a state management challenge in your React project, remember to reach for the useReducer hook and unleash the full potential of your application. Happy coding!

## useRef

<https://medium.com/zestgeek/understanding-the-useref-hook-in-react-real-life-examples-98339ab7f768>

The useRef hook is a powerful tool in React that often flies under the radar for many developers. While its primary purpose is to reference a DOM element, it can also be used to persist values across renders without causing a re-render.

**What is**useRef**?**

The useRef hook returns a mutable object with a .current property that you can use to store a value. Unlike useState, updating a useRef value does not trigger a component re-render. Here’s a basic example:

import React, { useRef } from 'react';

const UseRefExample =()=> {

  const inputRef = useRef(null);

  const focusInput = () => {

    inputRef.current.focus();

  };

  return (

    <div>

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

      <button className="btn btn-primary mx-2" onClick={focusInput}>Focus Input</button>

    </div>

  );

}

export default UseRefExample

**1.Accessing DOM Elements**

//click Tab button and see

import React, { useRef } from 'react';

const UseRefExample = () => {

    // Create a ref to hold a reference to the input element

    const firstNameRef = useRef(null);

    const lastNameRef = useRef(null);

    const emailRef = useRef(null);

    const phoneRef = useRef(null);

    // Function to focus on the input field

    const focusInput = (e) => {

        // Focus on the input field when the button is clicked

        console.log(e.target.name);

        if(e.target.name==='firstName'){

            console.log(e.target.name);

            lastNameRef.current.focus();

        }else if(e.target.name==='lastName'){

            console.log(e.target.name);

            emailRef.current.focus();

        }else if(e.target.name==='email'){

            console.log(e.target.name);

            phoneRef.current.focus();

        }

    };

    return (

        <div>

            <table>

                <tr>

                    <td><input type="text" name="firstName" onBlur={focusInput} ref={firstNameRef} className='mx-2' placeholder='FirstName' /></td>

                    <td><input type="text" name="email" onBlur={focusInput} ref={emailRef}  className='mx-2' placeholder='Email'/></td>

                </tr>

                <tr>

                    <td><input type="text" name="lastName" onBlur={focusInput} ref={lastNameRef}  className='mx-2' placeholder='LastName'/></td>

                    <td><input type="text" name="phone" onBlur={focusInput} ref={phoneRef}  className='mx-2' placeholder='Phone'/></td>

                </tr>

                <tr>

                    <td colSpan={2}>

                        <button onClick={()=>{console.log('submitting')}} className="btn btn-primary mx-2">Submit</button>

                    </td>

                </tr>

            </table>

        </div>

    );

};

**2.Persisting Values Across Renders**

Sometimes, you need to persist a value across renders without triggering a re-render. This is where useRef comes in handy.

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

const UseRefExample = () => {

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

    const renderCount = useRef(0);

    useEffect(() => {

      renderCount.current++;

    });

    return (

      <div>

        <p>Count: {count}</p>

        <p>This component has re-rendered {renderCount.current} times</p>

        <button onClick={() => setCount(count + 1)}>Increment</button>

      </div>

    );

  }

  export default UseRefExample;

Here, useRef is used to track how many times the component has re-rendered. Unlike state, updating renderCount.current does not cause the component to re-render.

**3.Storing Previous State Values**

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

const UseRefExample = () => {

  const [name, setName] = useState('David');

  const prevNameRef = useRef('');

  useEffect(() => {

    prevNameRef.current = name;

  }, [name]);

  return (

    <div>

      <p>Current Name: {name}</p>

      <p>Previous Name: {prevNameRef.current}</p>

      <input

        type="text"

        value={name}

        onChange={(e) => setName(e.target.value)}

      />

    </div>

  );

}

import React, { useEffect, useRef, useState } from 'react'

const UseRefExample = () => {

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

      const previousCount = useRef(0);

      useEffect(()=>{

        previousCount.current=count;

      },[count]);

  return (

    <div>

      <h1>Count:{count}</h1>

      <h1>Previous Count {previousCount.current}</h1>

      <button className="btn btn-primary" onClick={()=>setCount(prev=>prev+1)}>Increment</button>

    </div>

  )

}

export default UseRefExample;

## useImperativeHandle

The useImperativeHandle hook in React allows a child component to expose certain functions or properties to its parent component, giving the parent component more control over the child component. It was introduced in React 16.3 and provides a more explicit way for parent components to interact with child components. It's typically used in situations where a parent component needs to interact with a child component directly, such as for form validation or handling of user input. When compared to useRef, useImperativeHandle allows for more direct interaction with child components. However, it can add complexity to code and be error-prone if not used correctly.

The useImperativeHandle hook is a feature introduced in React 16.3 as a means to communicate from a child component to a parent component in a more explicit way. This hook allows a child component to expose certain functions or properties to its parent component, giving the parent component more control over the child component.

In simpler terms, the useImperativeHandle hook is used to create a custom interface between a child and its parent component. It is commonly used in situations where a parent component needs to interact with a child component directly, such as for form validation or handling of user input.

**How it Works**

In order to understand how useImperativeHandle works, it's important to first understand how refs work in React. Refs are a way to reference an instance of a component or an HTML element directly. They are commonly used to interact with the DOM or to access the state of a child component.

However, the useRef hook is used to create a ref that can be accessed by the parent component. However, the ref only provides access to the current state of the child component, and does not allow for direct interaction with the child component.

This is where useImperativeHandle comes in. It allows the child component to expose certain functions or properties to the parent component, which can be accessed through the ref. This provides a more explicit way for the parent component to interact with the child component.

Here’s an example of how useImperativeHandle works:

import React,{useState,useRef,forwardRef,useImperativeHandle} from 'react'

const UseImperativeHandleExample= () => {

    const childRef = useRef();

    const handleClick = () => {

      childRef.current.increment();

    };

    return (

      <>

        <ChildComponent ref={childRef} />

        <button onClick={handleClick}>Parent button controlling child</button>

      </>

    );

  };

export default UseImperativeHandleExample;

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

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

    useImperativeHandle(ref, () => ({

      increment() {

        setCount(count + 1);

      },

      getCount() {

        return count;

      }

    }));

    return <div>Count on Child: {count}</div>;

  });

In this example, the child component exposes two functions to the parent component through the use of useImperativeHandle: increment and getCount. These functions can be accessed by the parent component through the childRef ref. The parent component can then call the increment function when the button is clicked, which will update the count state of the child component.

**When to Use**useImperativeHandle

useImperativeHandle is typically used in situations where a parent component needs to interact with a child component directly. Some common use cases for useImperativeHandle include:

* Form validation: A child component can expose a function that validates form data to the parent component, which can be used to display error messages or prevent form submission.
* Handling of user input: A child component can expose a function that handles user input to the parent component, which can be used to update other parts of the application state.
* Accessing child state: useImperativeHandle can be used to expose certain state values of a child component to the parent component, allowing the parent component to access and use that state.

useImperativeHandle**vs**useRef

While both useImperativeHandle and useRef are used to interact with child components from a parent component, there are some key differences between the two hooks.

useRef creates a reference to a component or DOM element, which can then be accessed by the parent component. This allows the parent component to access the current state of the child component, but it does not provide a way for the parent component to directly interact with the child component.

useImperativeHandle, on the other hand, allows the child component to expose certain functions or properties to the parent component, which can then be accessed through the ref. This provides a more explicit way for the parent component to interact with the child component.

In general, you would use useRef if you only need to access the state of the child component, and you would use useImperativeHandle if you need to interact with the child component in a more direct way.

**Pros and Cons of**useImperativeHandle

Like any React hook, there are pros and cons to using useImperativeHandle.

**Pros**

* Explicit interface: useImperativeHandle provides a more explicit way for child components to expose certain functions or properties to parent components.
* Improved performance: By allowing parent components to interact with child components directly, useImperativeHandle can improve performance in certain scenarios.
* Improved code organisation: By separating the interface between parent and child components, useImperativeHandle can improve the organisation of your code.

**Cons**

* Can be complex: useImperativeHandle can add complexity to your code, especially when dealing with multiple child components and multiple functions or properties to expose.
* Can be error-prone: If not used correctly, useImperativeHandle can lead to errors and bugs in your application.
* Not always necessary: In some cases, using useImperativeHandle may not be necessary, and it may be simpler to use other hooks like useRef.

The useImperativeHandle hook is a powerful feature in React that allows child components to expose certain functions or properties to parent components. This provides a more explicit way for parent components to interact with child components, which can improve performance and code organization in certain scenarios.

However, it’s important to use useImperativeHandle correctly and only when necessary, as it can add complexity to your code and be error-prone if not used correctly. By understanding the pros and cons of useImperativeHandle, you can decide when and how to use it in your own React applications.

## useCallback

**What is**useCallback**?**

The useCallback hook is a performance optimization mechanism in React that helps prevent unnecessary re-renders of child components when their parent component re-renders. It achieves this by memoizing (caching) callback functions based on their dependencies. If the dependencies haven't changed, the same function reference is returned, avoiding the creation of a new function object on every render.

**When to Use**useCallback**:**

* **Passing callbacks as props to child components:** When a parent component passes a callback function as a prop to a child component, and the child component relies on the same callback reference across renders (e.g., for event handlers), useCallback can prevent the child from re-rendering unnecessarily due to a change in the parent's state or props.
* **Callbacks that are expensive to create:** If creating the callback function involves complex calculations or fetching data, using useCallback can improve performance by ensuring it's only created when its dependencies change.

**Example:**

JavaScript

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

const UseCallbackExample = () => {

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

  const handleClick = useCallback(() => {

    setCount(count + 1);

  }, [count]); // Only recreate handleClick when count changes

  return (

    <div>

      <p>Count: {count}</p>

      <ChildComponent onClick={handleClick} />

    </div>

  );

};

export default UseCallbackExample;

const ChildComponent = ({ onClick }) => {

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

};

In this example, handleClick is memoized using useCallback with the dependency [count]. This ensures that the child component only receives a new function reference when count changes, preventing unnecessary re-renders even if the parent component re-renders due to other state changes.

**When Not to Use**useCallback**:**

* **Simple callbacks without dependencies:** If a callback function is simple and doesn’t have any dependencies that change frequently, using useCallback might not be necessary. The overhead of memoization could outweigh the potential performance benefit.
* **Callbacks used within the same component:** If a callback is only used within the same component where it’s created, there’s no need for useCallback as React already handles component re-renders efficiently.

**Example (where**useCallback**is not needed):**

JavaScript

function MyComponent() {  
 const handleClick = () => {  
 console.log('Clicked!');  
 };

return (  
 <button onClick={handleClick}>Click me</button>  
 );  
}

Here, handleClick is a simple function without dependencies and is only used within the same component. Using useCallback wouldn't provide any performance benefit in this case.

**Additional Considerations:**

* The dependency array in useCallback is crucial. If you omit required dependencies, the function might be recreated unexpectedly, leading to unintended behavior.
* Overusing useCallback can introduce complexity and potentially make your code harder to maintain. Use it judiciously in scenarios where it demonstrably improves performance.

By understanding the use cases and limitations of useCallback, you can effectively leverage it to optimize your React components and enhance their performance.