**ReactJS useRef Hook**

Last Updated : 08 Feb, 2025

The useRef Hook is a built-in React Hook that returns a mutable reference object (ref) that persists across renders. Unlike state variables, updating a ref does not trigger a component re-render.

**Syntax**

const refContainer = useRef(initialValue);

* useRef returns an object { current: initialValue }.
* The .current property can be updated without re-rendering the component.

**Implementing the useRef hook**

**1. Accessing the DOM using useRef hook.**

In this example, we have a button called ACTION, whenever we click on the button the onClickHandler gets triggered and it focuses the textarea with the help of useRef hook.

**import** React, { Fragment, useRef } **from** 'react';

**function** App() {

**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} />

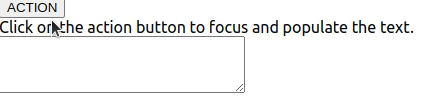
</Fragment>

);

};

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

**Output**



*React JS useRef Hook*

**In this example**

* useRef creates a reference focusPoint, which allows direct manipulation of the DOM element.
* Clicking the “ACTION” button triggers onClickHandler, which sets text in the textarea and focuses it.
* <Fragment> (<>…</>) is used to group multiple elements without adding extra wrappers in the DOM.

**2. Persisting Values Across Renders**

In addition to accessing DOM elements, useRef is useful for storing values that persist across renders. A common use case is storing a previous value, such as the previous state or props.

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

**function** PreviousValue() {

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

**const** prevCountRef = useRef();

useEffect(() => {

prevCountRef.current = count;

}, [count]);

**return** (

<div>

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

<p>Previous count: {prevCountRef.current}</p>

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

</div>

);

}

**export** **default** PreviousValue;

**Interesting Things About useRef Hook**

* **Does Not Cause Re-renders:**Unlike state variables, updating a useRef value does not trigger a component re-render. This makes it useful for storing values that persist across renders without causing unnecessary updates.
* **Accessing DOM Elements:**useRef is commonly used to reference DOM elements directly, allowing operations such as focusing input fields, managing animations, and interacting with elements without causing re-renders.
* **Tracking State Changes:** It can be used to store previous state values and track changes between renders, which is helpful for comparing current and previous values without affecting component updates.

**When to Use useRef?**

**You should use useRef when**

* Accessing and manipulating DOM elements without triggering re-renders.
* Persisting values across renders without causing re-renders.
* Storing previous state values to compare changes between renders.
* Optimizing performance by avoiding unnecessary state updates.

**useRef vs useState**

While both useRef and useState can store values, they behave differently:

* useRef does not trigger re-renders when updated, making it ideal for persisting values between renders.
* useState triggers re-renders whenever the state value is updated.
* Use useRef for storing references and preserving values, and useState for UI updates.

**Performance Considerations**

Using useRef correctly can enhance performance, but excessive use may introduce unnecessary complexity.

* **Use it for non-rendered values:** Ideal for persisting values like timers, previous states, or DOM elements.
* **Avoid using it as state replacement:** If UI updates are needed, use useState instead.
* **Measure before optimizing:** Use React DevTools to analyze performance.

**ReactJS useMemo Hook**

Last Updated : 08 Feb, 2025

The useMemo Hook is a built-in React Hook that helps optimize performance by memoizing the result of a computation and reusing it unless its dependencies change. This prevents expensive computations from being re-executed unnecessarily during component re-renders.

**Syntax**

const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);

* The first argument is a function that returns the computed value.
* The second argument is an array of dependencies. When any of these dependencies change, the function gets re-evaluated.

**Let’s now see some of the Practical Applications of useMemo**

**1. Optimizing Expensive Calculations**

Let’s take an example where we have an expensive computation that should not be recalculated on every render.

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

**function** App() {

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

**const** squaredNum = useMemo(() => squareNum(number), [number]);

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

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

setNumber(e.target.value);

};

**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** squareNum(number) {

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

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

}

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

import ReactDOM from "react-dom/client";

import useFetch from "./useFetch";

const Home = () => {

const [data] = useFetch("https://jsonplaceholder.typicode.com/todos");

return (

<>

{data &&

data.map((item) => {

return <p key={item.id}>{item.title}</p>;

})}

</>

);

};

const root = ReactDOM.createRoot(document.getElementById('root'));

Custom hooks in React allow you to extract and reuse logic across multiple components, making your code more modular and maintainable. They are essentially JavaScript functions that start with "use" and can call other hooks inside them.

**Creating a Custom Hook**

To create a custom hook, you need to define a function that encapsulates the logic you want to reuse. For example, let's create a custom hook called useOnlineStatus that tracks whether the user is online or offline:

import { useState, useEffect } from 'react';

function useOnlineStatus() {

const [isOnline, setIsOnline] = useState(true);

useEffect(() => {

function handleOnline() {

setIsOnline(true);

}

function handleOffline() {

setIsOnline(false);

}

window.addEventListener('online', handleOnline);

window.addEventListener('offline', handleOffline);

return () => {

window.removeEventListener('online', handleOnline);

window.removeEventListener('offline', handleOffline);

};

}, []);

return isOnline;

}

**Using the Custom Hook**

Once you have created the custom hook, you can use it in your components just like any other hook. For example, you can use useOnlineStatus in a StatusBar component to display the online status:

import React from 'react';

import { useOnlineStatus } from './useOnlineStatus';

function StatusBar() {

const isOnline = useOnlineStatus();

return <h1>{isOnline ? '✅ Online' : '❌ Disconnected'}</h1>;

}

Similarly, you can use the same hook in a SaveButton component to disable the button when the user is offline:

import React from 'react';

import { useOnlineStatus } from './useOnlineStatus';

function SaveButton() {

const isOnline = useOnlineStatus();

function handleSaveClick() {

console.log('✅ Progress saved');

}

return (

<button disabled={!isOnline} onClick={handleSaveClick}>

{isOnline ? 'Save progress' : 'Reconnecting...'}

</button>

);

}

**Benefits of Custom Hooks**

Custom hooks help you avoid code duplication and make your components more focused on their primary responsibilities. By extracting logic into custom hooks, you can hide the implementation details and make your components more readable and maintainable.

**Naming Conventions**

Custom hooks must start with the prefix "use" followed by a capital letter. This convention helps React identify hooks and enforce the rules of hooks. For example, useOnlineStatus is a valid custom hook name.

**Example: useFormInput Hook**

Let's create another custom hook called useFormInput to manage form input state:

import { useState } from 'react';

export function useFormInput(initialValue) {

const [value, setValue] = useState(initialValue);

function handleChange(e) {

setValue(e.target.value);

}

return {

value,

onChange: handleChange

};

}

You can use this hook in a Form component to manage multiple form inputs:

import React from 'react';

import { useFormInput } from './useFormInput';

function Form() {

const firstNameProps = useFormInput('Mary');

const lastNameProps = useFormInput('Poppins');

return (

<>

<label>

First name:

<input {...firstNameProps} />

</label>

<label>

Last name:

<input {...lastNameProps} />

</label>

<p><b>Good morning, {firstNameProps.value} {lastNameProps.value}.</b></p>

</>

);

}

**Conclusion**

Custom hooks are a powerful feature in React that allow you to encapsulate and reuse logic across multiple components. By following the naming conventions and best practices, you can create custom hooks that make your code more modular, readable, and maintainable.