# React useMemo Hook

The React useMemo Hook returns a memoized value.

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

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

This can improve performance.

The useMemo and useCallback Hooks are similar. The main difference is that useMemo returns a memoized value and useCallback returns a memoized function. You can learn more about useCallback in the [useCallback chapter](https://www.w3schools.com/react/react_usecallback.asp).

## **Performance**

The useMemo Hook can be used to keep expensive, resource intensive functions from needlessly running.

In this example, we have an expensive function that runs on every render.

When changing the count or adding a todo, you will notice a delay in execution.

### **Example:**

A poor performing function. The expensiveCalculation function runs on every render:

import { useState } from "react";

import ReactDOM from "react-dom/client";

const App = () => {

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

const [todos, setTodos] = useState([]);

const calculation = expensiveCalculation(count);

const increment = () => {

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

};

const addTodo = () => {

setTodos((t) => [...t, "New Todo"]);

};

return (

<div>

<div>

<h2>My Todos</h2>

{todos.map((todo, index) => {

return <p key={index}>{todo}</p>;

})}

<button onClick={addTodo}>Add Todo</button>

</div>

<hr />

<div>

Count: {count}

<button onClick={increment}>+</button>

<h2>Expensive Calculation</h2>

{calculation}

</div>

</div>

);

};

const expensiveCalculation = (num) => {

console.log("Calculating...");

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

num += 1;

}

return num;

};

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

root.render(<App />);

## **Use useMemo**

To fix this performance issue, we can use the useMemo Hook to memoize the expensiveCalculation function. This will cause the function to only run when needed.

We can wrap the expensive function call with useMemo.

The useMemoHook accepts a second parameter to declare dependencies. The expensive function will only run when its dependencies have changed.

In the following example, the expensive function will only run when count is changed and not when todo's are added.

### **Example:**

Performance example using the useMemo Hook:

import { useState, useMemo } from "react";

import ReactDOM from "react-dom/client";

const App = () => {

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

const [todos, setTodos] = useState([]);

const calculation = useMemo(() => expensiveCalculation(count), [count]);

const increment = () => {

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

};

const addTodo = () => {

setTodos((t) => [...t, "New Todo"]);

};

return (

<div>

<div>

<h2>My Todos</h2>

{todos.map((todo, index) => {

return <p key={index}>{todo}</p>;

})}

<button onClick={addTodo}>Add Todo</button>

</div>

<hr />

<div>

Count: {count}

<button onClick={increment}>+</button>

<h2>Expensive Calculation</h2>

{calculation}

</div>

</div>

);

};

const expensiveCalculation = (num) => {

console.log("Calculating...");

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

num += 1;

}

return num;

};

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

root.render(<App />);

**React useRef Hook**  
The useRef Hook allows you to persist values between renders.

It can be used to store a mutable value that does not cause a re-render when updated.

It can be used to access a DOM element directly.

## **Does Not Cause Re-renders**

If we tried to count how many times our application renders using the useState Hook, we would be caught in an infinite loop since this Hook itself causes a re-render.

To avoid this, we can use the useRef Hook.

### **Example:**

Use useRef to track application renders.

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

import ReactDOM from "react-dom/client";

function App() {

const [inputValue, setInputValue] = useState("");

const count = useRef(0);

useEffect(() => {

count.current = count.current + 1;

});

return (

<>

<input

type="text"

value={inputValue}

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

/>

<h1>Render Count: {count.current}</h1>

</>

);

}

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

root.render(<App />);

useRef() only returns one item. It returns an Object called current.

When we initialize useRef we set the initial value: useRef(0).

It's like doing this: const count = {current: 0}. We can access the count by using count.current.

Run this on your computer and try typing in the input to see the application render count increase.

## **Accessing DOM Elements**

In general, we want to let React handle all DOM manipulation.

But there are some instances where useRef can be used without causing issues.

In React, we can add a ref attribute to an element to access it directly in the DOM.

### **Example:**

Use useRef to focus the input:

import { useRef } from "react";

import ReactDOM from "react-dom/client";

function App() {

const inputElement = useRef();

const focusInput = () => {

inputElement.current.focus();

};

return (

<>

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

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

</>

);

}

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

root.render(<App />);

## **Tracking State Changes**

The useRef Hook can also be used to keep track of previous state values.

This is because we are able to persist useRef values between renders.

### **Example:**

Use useRef to keep track of previous state values:

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

import ReactDOM from "react-dom/client";

function App() {

const [inputValue, setInputValue] = useState("");

const previousInputValue = useRef("");

useEffect(() => {

previousInputValue.current = inputValue;

}, [inputValue]);

return (

<>

<input

type="text"

value={inputValue}

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

/>

<h2>Current Value: {inputValue}</h2>

<h2>Previous Value: {previousInputValue.current}</h2>

</>

);

}

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

root.render(<App />);

This time we use a combination of useState, useEffect, and useRef to keep track of the previous state.

In the useEffect, we are updating the useRef current value each time the inputValue is updated by entering text into the input field.

# React useReducer Hook

The useReducer Hook is similar to the useState Hook.

It allows for custom state logic.

If you find yourself keeping track of multiple pieces of state that rely on complex logic, useReducer may be useful.

## **Syntax**

The useReducer Hook accepts two arguments.

useReducer(<reducer>, <initialState>)

The reducer function contains your custom state logic and the initialStatecan be a simple value but generally will contain an object.

The useReducer Hook returns the current stateand a dispatchmethod.

Here is an example of useReducer in a counter app:

### **Example:**

import { useReducer } from "react";

import ReactDOM from "react-dom/client";

const initialTodos = [

{

id: 1,

title: "Todo 1",

complete: false,

},

{

id: 2,

title: "Todo 2",

complete: false,

},

];

const reducer = (state, action) => {

switch (action.type) {

case "COMPLETE":

return state.map((todo) => {

if (todo.id === action.id) {

return { ...todo, complete: !todo.complete };

} else {

return todo;

}

});

default:

return state;

}

};

function Todos() {

const [todos, dispatch] = useReducer(reducer, initialTodos);

const handleComplete = (todo) => {

dispatch({ type: "COMPLETE", id: todo.id });

};

return (

<>

{todos.map((todo) => (

<div key={todo.id}>

<label>

<input

type="checkbox"

checked={todo.complete}

onChange={() => handleComplete(todo)}

/>

{todo.title}

</label>

</div>

))}

</>

);

}

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

root.render(<Todos />);

This is just the logic to keep track of the todo complete status.

All of the logic to add, delete, and complete a todo could be contained within a single useReducer Hook by adding more actions.

## **What are Reducers in Redux ?**

In Redux, reducers are pure functions that handle state logic, accepting the initial state and action type to update and return the state, facilitating changes in React view components.

(State,action) => newState

This was about the reducer syntax and its definition. Now we will discuss the term pure function that we have used before.

## **Pure function**

A function is said to be pure if the return value is determined by its input values only and the return value is always the same for the same input values or arguments. A pure function has no side effects.

### Pure Function Example:

Below is an example of a pure function:

const multiply= (x, y) => x \* y;  
multiply(5,3);

In the above example, the return value of the function is based on the inputs, if we pass 5 and 3 then we’d always get 15, as long as the value of the inputs is the same, the output will not get affected.

### **Redux Reducer Syntax:**

Here is an example of a reducer function that takes state and action as parameters.

const initialState = {};  
const Reducer = (state = initialState, action) => {  
 // Write your code here  
}

### **Redux Reducer Parameters:**

* Redux State
* Redux Action

### **Redux State**

The reducer function contains two parameters one of them is the state. The State is an object that holds some information that may change over the lifetime of the component. If the state of the object changes, the component has to re-render.

In redux, Updation of state happens in the reducer function. Basically reducer function returns a new state by performing an action on the initial state. Below, is an example of how we can declare the initial state of an application.

#### Redux State Syntax:

const INITIAL\_STATE = {  
 userID: '',  
 name: '',  
 courses: []  
}

### **Redux Actions**

The second parameter of the reducer function is actions. Actions are JavaScript object that contains information. Actions are the only source of information for the store. The Actions object must include the type property and it can also contain the payload(data field in the actions) to describe the action.

#### Redux Action Syntax:

For example, an educational application might have this action:

{  
 type: 'CHANGE\_USERNAME',  
 username: 'GEEKSFORGEEKS'  
}  
{  
 type: 'ADD\_COURSE',  
 payload: ['Java with geeksforgeeks',  
 'Web Development with GFG']  
}

## **Redux Reducer Working Example:**

We have created two buttons one will increment the value by 2 and another will decrement the value by 2 but, if the value is 0, it will not get decremented we can only increment it. With Redux, we are managing the state **state-managing**

// Filename - App.js

import React from 'react';

import './index.css';

import { useSelector, useDispatch } from 'react-redux';

import { incNum, decNum } from './actions/index';

function App() {

const mystate = useSelector((state) => state.change);

const dispatch = useDispatch();

return (

<>

<h2>Increment/Decrement the number by 2,

using Redux.</h2>

<div className="app">

<h1>{mystate}</h1>

<button onClick={() => dispatch(incNum())}>

+</button>

<button onClick={() => dispatch(decNum())}>

-</button>

</div>

</>

);

}

export default App;

//src/index.js

import React from 'react';

import ReactDOM from 'react-dom';

import './index.css';

import App from './App.jsx'

import store from './store';

import { Provider } from 'react-redux';

ReactDOM.render(

<Provider store={store}>

<App />

</Provider>

, document.getElementById("root")

);

//index.js

export const incNum = () => {

return { type: "INCREMENT" }

}

export const decNum = () => {

return { type: "DECREMENT" }

}

//reducers/func.js

const initialState = 0;

const change = (state = initialState, action) => {

switch (action.type) {

case "INCREMENT": return state + 2;

case "DECREMENT":

if (state == 0) {

return state;

}

else {

return state - 2;

}

default: return state;

}

}

export default change;

//reducer/index.js

import change from './func'

import { combineReducers } from 'redux';

const rootReducer = combineReducers({

change

});

export default rootReducer;