**7 Essential React Hooks & When To Use Them**

[[Ryan Lai](https://forreya.medium.com/?source=post_page-----bd18b843e0b0--------------------------------)](https://forreya.medium.com/?source=post_page-----bd18b843e0b0--------------------------------)

[Ryan Lai](https://forreya.medium.com/?source=post_page-----bd18b843e0b0--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F23030b811506&operation=register&redirect=https%3A%2F%2Fforreya.medium.com%2F7-essential-react-hooks-when-to-use-them-bd18b843e0b0&user=Ryan+Lai&userId=23030b811506&source=post_page-23030b811506----bd18b843e0b0---------------------post_header-----------)

29 min read

·

Oct 1

18

React Hooks has been one of the many groundbreaking additions to the ever-improving React library in recent times. To understand why they were, and still are, game-changing, we need to take a look at how React used to be written before hooks were introduced.

**A Brief History of Components**

*React components are reusable pieces of code that help build the user interface of a webpage. Each component can be thought of as an individual gear working together to form one large mechanical system of machinery.*



Re-colored images from the harrowing era of class-based components.

Now when I say “history”, I’m talking about pre-2019. During those dark ages, whenever you built projects that required some use of state, life-cycle methods, or other niche functionality, you would need to use a **class-based component.**

*Wait… what’s a state? And what’s a life-cycle method?*

Good question! Let me just finish this next paragraph and I’ll get back to you on that.

I’m not going to do a deep dive into class-based components, that isn’t the focus of this article, but just know that writing and structuring code in this manner was quite a bit more cumbersome. Hooks basically swooped in as an all-encompassing “savior”, simplifying these various complexities. They provide a way to tap into the functionalities of React in an efficient and intuitive way, making components easier to both write and debug.

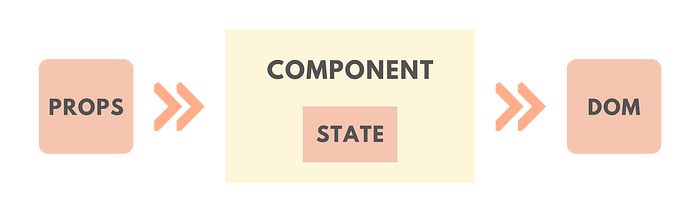
Now regarding your previous queries, let’s briefly cover an overview of **states and life-cycle methods.**

**State**

Each component in a React application can contain a state, a built-in object to store component-specific data.

*Sounds just like a variable. What’s the diff?*

It might seem like a variable, but there is a critical difference. A React component will re-render itself whenever its state changes, whereas changing a class variable does not trigger a re-render. This concept is vital as it allows the DOM to dynamically “repaint the user interface”, or in other words- re-render itself. A very useful feature indeed.



*The DOM is a programming interface that allows us to create, change, or remove elements from an HTML document. It defines the structure, content, and functionalities of the data displayed on a web page.*

**Life-Cycle Methods**

Every React component goes through different stages known as the “life-cycle.” These stages are Initialization, Mounting, Updating, and Unmounting, each having their respective methods.

1. **Initialization**- An instance of the component is defined with its initial props and its initial state.
2. **Mounting**- The component has its elements put into the DOM to be rendered on a webpage for the first time.
3. **Updating**- This stage occurs every time the component’s state or props change, ensuring that the component is displaying the most up-to-date information.
4. **Unmounting**- This stage occurs when the instance of the component is removed from the DOM.

Here’s where Hooks comes into the picture. One specific React hook known as “useEffect” combines the life-cycle methods of the mounting and updating stages, simplifying situations where an external procedure has to be triggered in reaction to a component event, such as a user clicking on a button. If this all sounds like mumbo jumbo to you right now, don’t worry, we have a whole section dedicated to “useEffect” later.

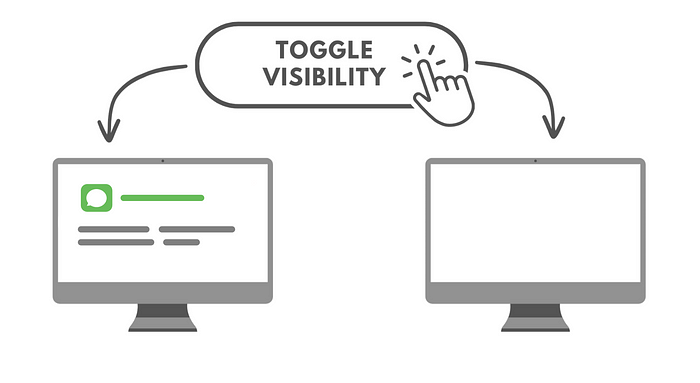
*Note: It’s important to remember that hooks do not replace pre-existing React concepts and classed-based approaches, they simply provide a more ergonomic way to approach the same situations.*

**#1 useState**

Let’s begin with the most popular React hook and understand why useState is so widely used.

The purpose of this hook is to handle “reactive” data, or in other words- **the** **state**. The term “reactive” means that, as mentioned before, changing the value will trigger a re-render of the component. This is important so that the UI is updated, ensuring the latest changes are reflected to the user.

To picture a concrete use-case of this hook, let’s build a simple component that displays a message on the screen, whose visibility can be toggled by a button.



To initialize the state, we need to first import the hook from React.

import {useState} from 'react'

Now, we must set an initial value for our “stateful variable”, which will hold information on the message’s current visibility status. Let’s create a state variable called isVisible and set its initial value to true .

const [isVisible, setIsVisible] = useState(true);

As you can see, the useState hook returns an array that contains 2 values you can utilize within your component. We can destructure these 2 values with JavaScript into the two variables- isVisible and setIsVisible.

isVisible acts similarly to a regular variable, it contains the “reactive” data of the state variable. If the value of isVisible changes at any point, React will automatically rebuild the UI to present the latest value.

The second, setIsVisible, is a setter function that we can use to change the value of the state variable. Let’s now implement this state to create our mini-application.

import React, { useState } from 'react';  
  
const ToggleableMessage = () => {  
 const [isVisible, setIsVisible] = useState(true);  
  
 return (  
 <div>  
 <button onClick={setIsVisible(!isVisible)}>  
 Toggle Visibility  
 </button>  
  
 {isVisible && <p>NEW MESSAGE</p>}  
 </div>  
 );  
};  
  
export default ToggleableMessage;

In this example, we use useState to create a state variable called isVisible, initialized with a value of true. The component renders a button, and when the button is clicked, it uses the setIsVisible function to toggle the value of isVisible between true and false.

In short, the message is conditionally rendered based on the value of isVisible. If isVisible is set to true, the message is displayed; otherwise, it remains hidden.

From this example, it’s easy to see why being able to manipulate and handle state within a component is so useful- it allows us to re-render the UI based on changing data, which gives us the freedom to design a vast number of different dynamic websites and applications.

**#2 useEffect**

Second in line, we have useEffect. This particular hook perplexed me for a long time when I was first introduced to it, so let’s take this step by step to avoid any misunderstandings.

To really understand useEffect, it’s important that you clearly understand the basics of the aforementioned “component life-cycle”. These different stages outline the basis of the useEffect hook and how it handles different life cycle events.

Consider a component where one random image of a frog will be fetched from a “frog images” API when the component mounts.

Let’s see what this code would look like.

import React, { useEffect, useState } from 'react';  
  
const FrogFetcher = () => {  
 const [imageUrl, setImageUrl] = useState('');  
  
 useEffect(() => {  
 // Fetch data from Frog Images API;  
 setImageUrl(data.frogImage);  
 }, []);  
  
 return (  
 <div>  
 <h2>A Random Frog</h2>  
 <img src={imageUrl} alt="Random Frog" />  
 </div>  
 );  
};  
  
export default FrogFetcher;

Okay, this is a lot to take in. Let’s break this component down line-by-line.

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

Notice how we are importing **both** useEffect and useState in this component. The reason why I’m not using useEffect alone in this example is to demonstrate how different hooks can be used together in a component to achieve various functionalities. In this case, to display frogs.

const [imageUrl, setImageUrl] = useState('');

Next, we define our state variable imageUrl and set its initial value to an empty string. This variable will store a URL to the image of a frog- quite an important job if I must say so myself.

useEffect(() => {  
 // Code to fetch data from Frog Images API…  
 setImageUrl(data.frogImage);  
 }, []);

Now for the new & exciting bit, the new hook “useEffect”. In this example, useEffect is used to fetch a random frog image URL when the component mounts. A set of actions, declared inside the body of the useEffect, are carried out when the useEffect hook is run.

These actions include fetching the frog data from the API and then setting the fetched URL to the state variable imageUrl. Once the image URL string is set to the state variable imageUrl, the img tag is then able to display the image to the user.

<img src={imageUrl} alt="Random Frog" />

Now let’s focus our attention back on the useEffect hook once again.

useEffect(() => {  
 // Fetch data from Frog Images API;  
 setImageUrl(data.frogImage);  
 }, []);

Notice the [] as the second argument to the useEffect. This array is very important because it **determines when the useEffect is invoked.**Let me elaborate.

Currently, the only time when our useEffect function will execute is when the component is initially mounted.

*Why?*

**The elements of this array act as a range of dependencies** **that allow us to have fine-grained control over its execution.** Since the array is currently empty, we have no dependencies. However, in another scenario, we might want to re-run the useEffect every time some sort of stateful data changes. For clarity, let’s divert our attention away from frogs and take a look at another completely different component.

import React, { useState, useEffect } from 'react';  
  
const Greeting = () => {  
 const [name, setName] = useState('');  
 const [greeting, setGreeting] = useState('Hello');  
  
 useEffect(() => {  
 setGreeting(`Hello, ${name}!`);  
 }, [name]); // The effect will re-run whenever "name" changes  
  
 return (  
 <div>  
 <input  
 type="text"  
 value={name}  
 onChange={(event) => setName(event.target.value)}   
 placeholder="Enter your name"  
 />  
 <p>{greeting}</p>  
 </div>  
 );  
};  
  
export default Greeting;

In this second example, whenever the user enters their name into the input field, setName is used to update the name state. Since name is listed as a dependency in the useEffect, the effect will trigger again, and the new name will be appended onto greeting state. This in turn will display an updated p tag on the UI with the new name included in the greeting.

The important difference to note between this example and the frog example is that the Greeting component’s useEffect will run in two cases:

1. When the Greeting component mounts.
2. Whenever the value of the name state variable changes.

The FrogFetcher component, however, will **only** run in one default case:

1. When the FrogFetcher component mounts.

This all stems from the dependencies array and its contents. It’s important to understand this notion of state dependencies to fully comprehend the power of useEffect, so take some time to digest these two examples. When you’re ready, let’s continue onto our next hook.

**#3 useContext**

In React, passing data from a parent component to its child components is traditionally done through something known as “props”. This is short for properties, and they act as a communication channel for transferring data down a component tree in a unidirectional flow.

<App>  
 ├── <Header>  
 │ └── <WebsiteLogo>  
 │  
 ├── <TodoList>  
 │ ├── <TodoItem>  
 │ ├── <TodoItem>  
 │ └── ...  
 │  
 └── <Footer>

For example, in this component tree for a “Todo List” website, props can be passed down from above to below, based on information requirements and website functionality.

This is the most basic way to transfer information from one component to another. However, consider a scenario where you have a deeply nested component structure, and you need to share a specific piece of data or a function across multiple levels. Traditionally, you would have to pass this data as props through every intermediate component, even if those components don’t directly use that data. This is known as prop drilling, and it not only clutters the code but also makes it difficult to maintain as the application scales.

<App>  
 ├── <Header>  
 │ ├── <Navigation>  
 │ │ ├── <Home>   
 │ │ ├── <Products>   
 │ │ │ ├── <Electronics>   
 │ │ │ ├── <Fashion>   
 │ │ │ └── <Furniture>  
 │ │ ├── <AboutUs>   
 │ │ └── <ContactDetails>  
 │ ├── <SearchBar>  
 │ └── <Cart>  
 │  
 ├── <Main>  
 │ ├── <Home>  
 │ ├── <ProductList>  
 │ │ └── <ProductItem>  
 │ │ └── <AddToCartButton>  
 │ ├── <ProductDetail>  
 │ │ └── <ProductDetailInfo>  
 │ │ └── <AddToCartButton>  
 │ ├── <Checkout>  
 │ │ └── <CartItem>  
 │ │ │ └── <RemoveFromCartButton>  
 │ │ └── <PaymentMethod>  
 │ └── <AboutUs>  
 │  
 └── <Footer>  
 ├── <FooterLinks>  
 └── <FooterDisclaimer>

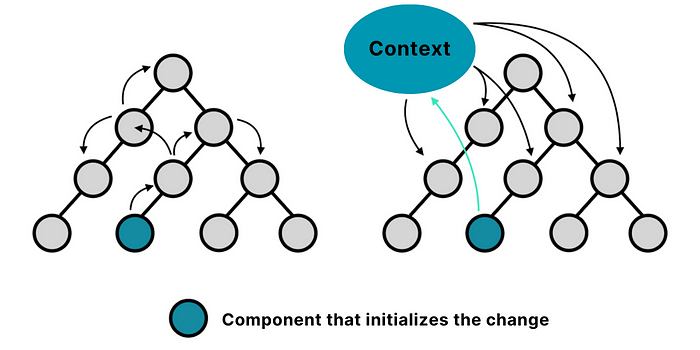
Imagine if you wanted to pass a prop down from the App component down to the Fashion component. This means we would have to go through Header, Navigation, and Products before we actually reached our desired component. So a lot of extra code would have to be written in between just to pass down a simple piece of information. Because of this, managing the flow of data in larger-scale applications can become the breeding ground for bugs and errors if we simply use props alone.

You’ve probably guessed it by now, but this is where useContext enters the picture. One of this hook’s sole purposes is to solve this idiosyncrasy. So what ingenious solution does it bring about?

At its core, it allows us to **share values throughout the entire component tree through a transport mechanism known as Context**.

*How does it manage that?*

Well, context provides a way to pass data through the tree without having to manually pass props down through each intermediate component. It does this by **enabling components to access data from a central location (context)**, providing a more efficient solution for information sharing across components in the component tree.



So, enough backstory. Let’s take a look at how we can utilize useContext.

**How is useContext used?**

Think of useContext in terms of the following example. We have this top-secret headquarters whose server acts as a storage system for a bunch of classified files. To access these files from our computer, we need a special USB stick inserted into our computer to allow a “connection” to be made to the headquarters’ servers. Now, different people might want different files from the server, so each computer is forced to specify what files specifically we want to access from the headquarters’ servers. For example, computer A might want just file 2, while computer B might want file 4, file 18, and file 281.

So we can break this down into three main steps:

1. **Headquarters’ Server-**This stores all the files to be accessed.
2. **USB Stick:** This acts as a connector from the computer to the headquarters’ servers.
3. **Specifying the Files to Access:** This means each computer will have to individually specify which file they are interested in accessing (it could be every single file if they wanted to).

Let’s convert these into *real*steps:

1. **Create the Context-** The React.createContext() method creates a new Context object, storing all the data you want to provide across the component tree.
2. **Provide the Context-** The Context object created includes a React component named Provider. This Provider component is used to “provide” the context value to the child components of the tree.
3. **Consume the Context-** Finally, the useContext Hook allows you to “consume” the context. In other words, to access the value of the Context object from any child component.

So to clarify, the headquarter’s servers are the Context object. The files are the values of the Context object. The individual computers are the child components. The USB sticks are the context providers. And finally, specifying which files to access is the equivalent of “consuming” the context.

**Day/Night Theme Toggler**



Let’s demonstrate this with a simple application where users can toggle between light and dark themes. We’ll create a context to hold the current theme and a function to switch between themes. The context will then be provided to the app, and any component in the app can access and modify the theme.

So let’s first take a look at our App component, where we will create and provide the context.

import React, { createContext, useContext, useState } from 'react';  
  
// Step 1: Create the context  
const ThemeContext = createContext();  
  
function App() {  
 const [theme, setTheme] = useState('light');  
  
 function toggleTheme() {  
 setTheme(prevTheme => prevTheme === 'light' ? 'dark' : 'light');  
 }  
  
 return (  
 // Step 2: Provide the context  
 <ThemeContext.Provider value={{ theme, toggleTheme }}>  
 <Navbar />  
 <MainContent />  
 </ThemeContext.Provider>  
 );  
}

Let’s look into this component in further detail.

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

First, we’re importing the tools we will be using. createContext is used to create a new context, useContext is the hook we use to access the context, and useState was the very first hook we discussed.

const ThemeContext = createContext();

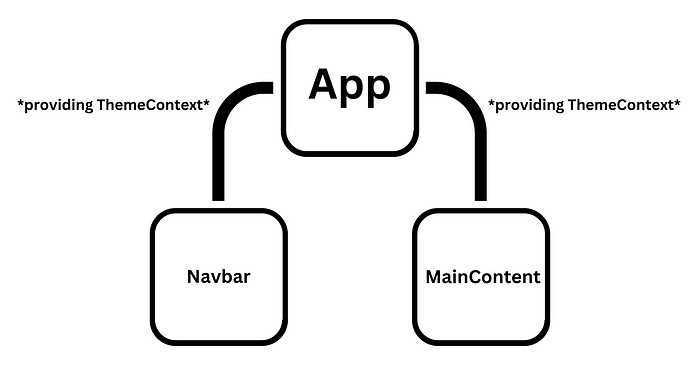
Here we’re creating a new context using createContext(). The returned context object is set to the variable ThemeContext.

const [theme, setTheme] = useState('light');  
  
 function toggleTheme() {  
 setTheme(prevTheme => prevTheme === 'light' ? 'dark' : 'light');  
 }

Within our App component, we're initializing a state variable, theme, with the default value 'light', using the useState hook. We're also defining a function, toggleTheme, that will switch our theme between 'light' and 'dark'.

return (  
 <ThemeContext.Provider value={{ theme, toggleTheme }}>  
 <Navbar />  
 <MainContent />  
 </ThemeContext.Provider>  
 );

The ThemeContext object contains a component called Provider that allows us to provide context to any other component down the component tree. In our case, that would be the Navbar and the MainContent components, and they would therefore have access to the current theme state and toggleTheme function.



In essence, the ThemeContext.Provider component allows other components that are children of the App component to essentially “subscribe” to the context changes, giving them access to the values stated within the ThemeContext.Provider.

Now let’s see how we can use this provided context in our Navbar.

function Navbar() {  
 const { theme, toggleTheme } = useContext(ThemeContext);  
  
 return (  
 <nav className={`navbar ${theme}`}>  
 <h1>The Navbar</h1>  
 <button onClick={toggleTheme}>Toggle Theme</button>  
 </nav>  
 );  
}

In the Navbar component, we're using the useContext hook to access our ThemeContext. This hook returns the value that was passed to the ThemeContext.Provider. In this case, the value is the object that contains our theme state and toggleTheme function. Let’s once again go line-by-line through this Navbar component for more clarity.

const { theme, toggleTheme } = useContext(ThemeContext);

Here we’re using object destructuring to extract theme and toggleTheme from the context.

<nav className={`navbar ${theme}`}>

We then use the theme to dynamically set the CSS class of our nav element.

<button onClick={toggleTheme}>Toggle Theme</button>

We can also use an onClick handler on our button that calls the toggleTheme function when clicked, effectively allowing us to toggle the theme from the Navbar component.

As you can see, context gives us the ability to utilize data and information without having to pass in theme and toggleTheme as props.

Next, we’ll see how the MainContent component uses the context in a similar way.

function MainContent() {  
 const { theme } = useContext(ThemeContext);  
  
 return (  
 <main className={`content ${theme}`}>  
 <p>This is the main content area of the App.</p>  
 </main>  
 );  
}

The MainContent component similarly uses the useContext hook to access the ThemeContext and extract the theme state. The theme state is then used to dynamically set the CSS class of our main element, allowing the appearance of our main content to respond to theme changes.

So to summarize it all, ThemeContext provides a shared state to Navbar and MainContent — two sibling components that wouldn't be able to share state directly otherwise. useContext then allows them to both read and update the state.

As I stated before, a lot of people who are new to React are often confused by how context works due to the term being rarely mentioned in most other libraries or frameworks. Therefore it’s essential you get a complete comprehension of the logistics behind this transport mechanism so you can conscientiously determine when and when not to use Context in your own applications.

**#4 useRef**

A ref is very similar to state in the sense that it persists a value in between renders of a component. However, unlike state, a ref does not trigger a component re-render when its value changes. And the useRef hook essentially allows us to use these so-called “refs”. Simple enough, right?

Let’s see how we can use this hook:

import { useRef } from "react";

As always, the first thing we do is import it into our component.

const count = useRef(0);

Let’s say we instantiate a ref variable called count, using useRef, with an initial value of 0. If we logged the value of count right now, this is what it would look like:

{ current: 0 }

This means that useRef returns an object with a singular property called current, and by default, this will be set as the initial value of 0. When we update that property, that current value is what gets persisted between the different renders.

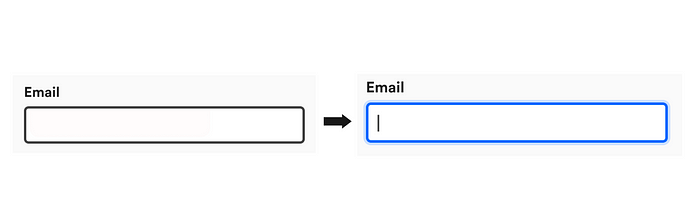
**Typical Uses**

useRef is typically used for two main reasons:

1. **Accessing DOM elements directly-** Often, we may need to interact directly with the DOM, be it to manage focus, text selection, or triggering animations. This is where useRef comes in handy. It provides us with a way to directly interact with the DOM element associated with a given ref.
2. **Persisting data without triggering a re-render-** We’ve already briefly discussed this. But just to reiterate, useRef allows us to persist data across re-renders without causing the component to update. This can be particularly useful for keeping track of previous props or state.

Let’s take a look at an example for each of these typical use cases.

**Example 1: Accessing DOM elements directly**



Let’s create a simple React component that automatically focuses on an input field when the component loads. This is a direct manipulation of a DOM element using useRef.

import React, { useRef, useEffect } from 'react';  
  
function AutoFocusInput() {  
 const inputRef = useRef(null);  
  
 useEffect(() => {  
 // Auto-focus on the input element when the component is mounted  
 inputRef.current.focus();  
 }, []);  
  
 return (  
 <input ref={inputRef} type="text" placeholder="Input" />  
 );  
}  
  
export default AutoFocusInput;

In this example, useRef is used to create a ref (inputRef) that is connected to the input field using a prop. Its initial value is set to null. Below are the two relevant lines in question.

const inputRef = useRef(null); // #1  
  
<input ref={inputRef} type="text" placeholder="Input" /> // #2

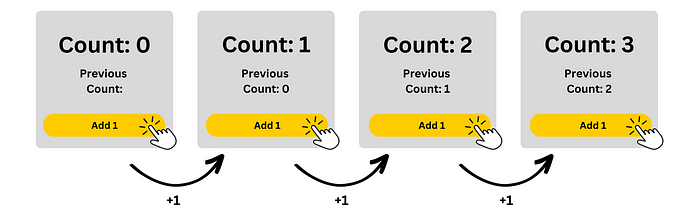
Then, using useEffect, we tell React to focus on this input field whenever the component mounts.

inputRef.current.focus();

In this component, the useRef hook helps us access the input DOM element directly and persists the reference across re-renders without triggering an update.

**Example 2: Persisting data without triggering a re-render**

Now let’s write a counter application that not only updates the count but also tells you the previous count, without causing a re-render when the previous count is updated. This can be a bit confusing over words, so let’s see a step-by-step diagram of what we want to achieve.



Let’s put this idea into code now. You don’t have to understand everything as of now, just have a rough skim over and we’ll go through it together in more detail after.

import React, { useState, useRef, useEffect } from 'react';  
  
function Counter() {  
 const [count, setCount] = useState(0);  
 const previousCountRef = useRef(null);  
  
 useEffect(() => {  
 previousCountRef.current = count;  
 });  
  
 const incrementCount = () => {  
 setCount(count + 1);  
 };  
  
 return (  
 <div>  
 <p>Count: {count}</p>  
 <p>Previous Count: {previousCountRef.current}</p>  
 <button onClick={incrementCount}>Increment Count</button>  
 </div>  
 );  
}  
  
export default Counter;

In this Counter component, we’re managing two values: the current count, and the previous count. The current count is “stateful” and changes over time, triggering re-renders of our component. The previous count is also a piece of data that we want to remember and have access to, however, changes to it should not cause our component to re-render. Keeping track of the previous count is therefore a perfect use case for a ref.

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

Here, previousCountRef is a ref variable, which contains an object where we will store the value of the previous count. In the beginning, this object is empty, since we didn't pass any initial value to useRef().

useEffect(() => {  
 previousCountRef.current = count;  
});

Here, we use useEffect without passing any dependency array, which means this effect will run after every completed render of the component. After each render, we update previousCountRef.current (the property in our ref object) with the current count.

The most important thing to understand is that although the value of previousCountRef.current is updated with every component render, the UI does not reflect this change until the count is incremented in the next cycle. This is, again, because updating a ref value **does not** trigger a re-render of the component.

const incrementCount = () => {  
 setCount(count + 1);  
};

The incrementCount function is used to increment the count by 1 by using setCount, which triggers a re-render with the new count value. After that re-render, our useEffect runs again, updating previousCountRef.current with the new count.

**#5 useReducer**

Another extremely useful hook is useReducer. It allows us to manage and control state within our components, akin to useState. However, while these two hooks may share certain qualities on the surface, that’s where their similarities end.

As we’ll see in just a second, useReducer requires a bit more setup and understanding initially, but it definitely pays off in the long run. It makes your code cleaner, easier to debug, and more maintainable, especially in scenarios involving multiple interrelated states or complex state changes.

Let’s look at a specific case where using the useState hook too much in a single component might lead to messy and hard-to-read code, followed by how we can use useReducer instead to elegantly clean up and simplify that said codebase.

Have a skim over the following component. For context, its purpose is to manage a user profile.

import React, { useState } from 'react';  
  
function UserProfile() {  
 const [firstName, setFirstName] = useState("");  
 const [lastName, setLastName] = useState("");  
 const [age, setAge] = useState(0);  
 const [email, setEmail] = useState("");  
 const [username, setUsername] = useState("");  
 const [loggedInStatus, setLoggedInStatus] = useState(false);  
  
 const incrementAge = () => {  
 setAge(prevAge => prevAge + 1);  
 };  
  
 const updateName = (newFirstName, newLastName) => {  
 setFirstName(newFirstName);  
 setLastName(newLastName);  
 };  
  
 const updateEmail = (newEmail) => {  
 setEmail(newEmail);  
 };  
  
 const updateUsername = (newUsername) => {  
 setUsername(newUsername);  
 };  
  
 const toggleLogin = () => {  
 setLoggedInStatus(prevStatus => !prevStatus);  
 };  
  
 // Other logic and JSX here  
}

As you can see, due to the sheer number of interrelated state variables, the component is quite cluttered and therefore difficult to maintain. Each state variable has its own setter function to update its value, leading to scattered state management logic throughout the component. If we add even more state logic on top of this, the code structure would only get harder to manage and even more convoluted.

**Let’s solve this atrocity by refactoring it with useReducer.**

To begin with, we need to understand the idea behind useReducer. The way this hook works is that it allows you to centralize all your state logic into a single **reducer function**. This single source of truth acts as the ‘mind’ behind everything that is relevant to your component’s state. This makes your code more readable and cleaner in general because everything is systemized in one location, rather than scattered all over your code.

**The Reducer Function**

So, let’s make a separate file to set up this so-called “reducer function”. To stick to convention, let’s call it userProfileReducer.js.

To begin with, we need to specify the initial values of our state variables. Notice how this is very similar to useState. This all-encompassing object, which we will appropriately name as initialState, will describe all the initial values for every user profile property when the component first renders.

export const initialState = {  
 firstName: "",  
 lastName: "",  
 age: 0,  
 email: "",  
 username: "",  
 loggedInStatus: false  
};

Now for the tricky part, defining the actual reducer function. This function takes the current state, along with an action, and returns a new state based on the action type. It's the core logic that dictates how the state should update in response to different actions. Think of each action as a more intricate and user-defined version of the setState function from the useState hook.

export const userProfileReducer = (state, action) => {  
 // Reducer logic  
}

The action is typically an object with a type field and sometimes contains additional data. Based on the action type, the reducer will return a new state object. If it doesn't recognize the action type, it throws an error as a default case. As a single reducer function could potentially have countless different actions, it’s better to use “switch” statements to determine the action type as otherwise we would have a long nest of “if” statements, which wouldn’t be ideal.

export const userProfileReducer = (state, action) => {  
 switch (action.type) {  
 case 'incrementAge':  
 return { ...state, age: state.age + 1 };  
 case 'updateName':  
 return { ...state, firstName: action.firstName, lastName: action.lastName };  
 case 'updateEmail':  
 return { ...state, email: action.email };  
 case 'updateUsername':  
 return { ...state, username: action.username };  
 case 'toggleLogin':  
 return { ...state, loggedInStatus: !state.loggedInStatus };  
 default:  
 throw new Error();  
 }  
}

It’s important to understand that the reducer is a pure function that describes how the state of the component should change in response to different actions. This is the basic principle behind every reducer function.

Okay, great! We have our userProfileReducer.js file all set up. All we have to do is integrate what we’ve written into our main UserProfile component.

**The UserProfile Component**

First and foremost, let’s import everything we need for this file.

import React, { useReducer } from 'react';  
import { userProfileReducer, initialState } from './userProfileReducer.js';

Now onto the part where we’re finally using the hook itself. Let’s call the useReducer function with the userProfileReducer function and initialState as arguments.

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

The useReducer hook returns an array of two elements: the current state and a dispatch function. The current state (state) is an object that contains the current state of the component. The dispatch function (dispatch) is a function that you'll call when you want to update the state.

Let’s take a look at how we can apply this knowledge to our code. I’m going to show you the finished product for our component first so you can get a feel for the grand scheme of things, and then we’ll dive deeper into each line afterward.

function UserProfile() {  
 const [state, dispatch] = useReducer(reducer, initialState);  
  
 const incrementAge = () => {  
 dispatch({ type: 'incrementAge' });  
 };  
  
 const updateName = (newFirstName, newLastName) => {  
 dispatch({ type: 'updateName', firstName: newFirstName, lastName: newLastName });  
 };  
  
 const updateEmail = (newEmail) => {  
 dispatch({ type: 'updateEmail', email: newEmail });  
 };  
  
 const updateUsername = (newUsername) => {  
 dispatch({ type: 'updateUsername', username: newUsername });  
 };  
  
 const toggleLogin = () => {  
 dispatch({ type: 'toggleLogin' });  
 };  
  
 return (  
 <div>  
 <h2>User Profile</h2>  
 <p>Name: {state.firstName} {state.lastName}</p>  
 <p>Age: {state.age}</p>  
 <p>Email: {state.email}</p>  
 <p>Username: {state.username}</p>  
 <p>Logged in: {state.loggedInStatus ? "Yes" : "No"}</p>  
 {/\* Buttons or inputs to call the update functions \*/}  
 </div>  
 );  
}

In this component, there are two things we need to focus on.

**Dispatch**

Think back to when we were dealing with "actions" during the creation of the reducer function. Well, dispatch is an external function that you call to **send actions to the reducer**. Remember, an "action" is simply an object that describes a change that you want to make to the state. This action object should have a type field, which the reducer uses to determine how the state should change. It can also have additional data that the reducer might need to compute the next state, but we don’t use any in this case.

For example, when the incrementAge function is called, it dispatches an action of type 'incrementAge'.

const incrementAge = () => {  
 dispatch({ type: 'incrementAge' });  
 };

Recall that we already defined this incrementAge action type in the userProfileReducer.js file and therefore the reducer will respond to this action by returning a new state object where the age field has been incremented by one. For a reminder, here is the relevant section of code from our reducer function:

switch (action.type) {  
 case 'incrementAge':  
 return { ...state, age: state.age + 1 };  
 // Other actions...

**State**

As explained before, state is the current state of your component, which is a snapshot of the user data at a specific point in time. In our component, the state is used directly in our JSX to display the data to the user:

return (  
 <div>  
 <h2>User Profile</h2>  
 <p>Name: {state.firstName} {state.lastName}</p>  
 <p>Age: {state.age}</p>  
 <p>Email: {state.email}</p>  
 <p>Username: {state.username}</p>  
 <p>Logged in: {state.loggedInStatus ? "Yes" : "No"}</p>  
 {/\* Buttons or inputs to call the update functions \*/}  
 </div>  
 );

If all of this is a lot to take in, just remember that everything we have done essentially falls into three categories: **defining our initial state, updating our current state, and utilizing our current state values to display them on the UI.**The useReducer function simply acts as the “source of power” from which we are given the ability to effectuate these operations.

Alrighty! You’re officially 62.5% of the way through this article. It’s been a lot of information to take in at once, especially if you’re brand new to React hooks, so don’t forget to take some time off to properly digest everything you’ve learned. Remember, it’s about mastery- not memorization.

**#6 useMemo**

This hook is a tricky one. For all intents and purposes, it’s simply a tool to optimize your React application’s performance. However, a true understanding of useMemo lies on a sweet spot between premature usage (which could cause unexpected bugs and behavior) and not using it when you should be (which would lead to some potentially critical performance issues). Let’s take a look.

**What Problems Does useMemo Solve?**

There are two situations where useMemo comes in handy.

**1. Performance Optimization**

When you have an expensive function in your component that runs every time the component renders, it can significantly affect the performance of your application. useMemo solves this problem by "remembering" the previous result and only re-running the calculation if the dependencies change.

**2. Reference Equality**

In JavaScript, every time a new object or array is declared, a new reference in memory is created. The term “reference equality” refers to whether two variables refer to the same object in memory (i.e., they have the same reference). Let’s demonstrate this with real code:

let a = 5;  
let b = a;  
b = 3;  
  
console.log(a); // Outputs: 5  
console.log(b); // Outputs: 3

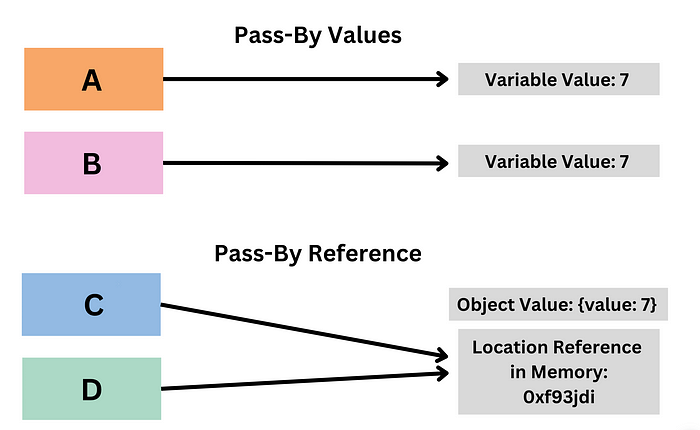
In this example, when b is set to a, a **copy** of the value in a is created and stored in b. Thus, when b is later changed to 3, a remains 5. The reason for this behavior is due to the fact that the variables are assigned a primitive data type (like a number, string, boolean, null, or undefined). **These are known as pass-by-values, which means the value is stored directly in the location that the variable accesses.**

Contrarily, when a variable is assigned a non-primitive data type (like an object, array, or function), the variable doesn’t actually contain the object. **This is known as pass-by-reference, which means the variables contain a reference to the location in memory where the object is stored, instead of storing the object directly.**

let obj1 = { value: 5 };  
let obj2 = obj1;  
obj2.value = 3;  
  
console.log(obj1.value); // Outputs: 3  
console.log(obj2.value); // Outputs: 3

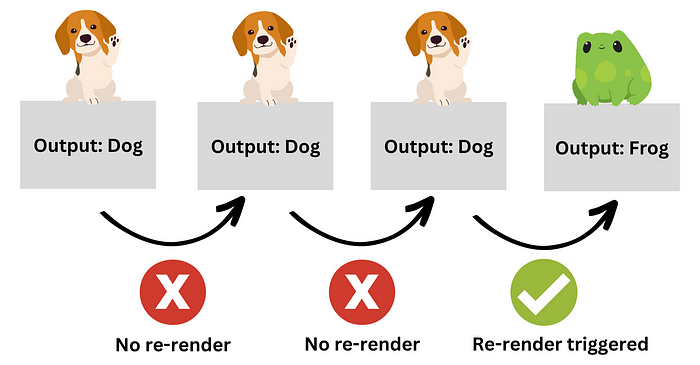
In this example, when obj2 is set to obj1, it doesn't create a new object; rather, obj2 now **refers to the same object** that obj1 does. Thus, when the value of obj2 is changed to 3, it's actually **changing the original object in memory**, and so obj1 outputs 3 in the end as well.

Just to make sure we’re all clear, here’s a diagram illustrating this concept:



*Why is this important in the context of React and useMemo?*

React's re-rendering mechanism relies heavily on comparing props and state to see if they have changed. When objects or arrays are involved, React uses reference equality for this comparison. If a new object is created on each render, even if its contents are the same, it will have a different reference, and React will consider it as changed, causing an unnecessary re-render. useMemo can help prevent this by maintaining reference equality until the actual contents change.



Okay, let’s see how we can utilize useMemo in a simple scenario.

**Sorting A Numbers List**



Suppose we have a component that displays a list of numbers in order of smallest to greatest. Sorting is an expensive operation, and doing it on every render can affect performance, especially when dealing with large lists.

Let’s see what this component might look like without useMemo:

import React from 'react';  
  
const NumberList = ({ numbers }) => {  
 const sortedNumbers = numbers.sort((a, b) => a - b);  
  
 return (  
 <div>  
 {sortedNumbers.map(number => (  
 <p key={number}>{number}</p>  
 ))}  
 </div>  
 );  
};  
  
export default NumberList;

In this numberList component, we’re accepting a numbers prop, which is the array of numbers. We sort this array before rendering it. However, the sorting operation happens on every render, regardless of whether the numbers prop has changed.

Here’s where useMemo comes into the picture. We can use useMemo to store the sorted list and only re-sort it when the list changes. In fact, this is exactly why it’s called useMemo, it’s literally a digital memo to remember the values of previous render iterations.

So let’s optimize this component, shall we?

import React, { useMemo } from 'react';  
  
const NumberList = ({ numbers }) => {  
 const sortedNumbers = useMemo(() => {  
 return numbers.sort((a, b) => a - b);  
 }, [numbers]);  
  
 return (  
 // ... same as before ...  
 );  
};  
  
export default NumberList;

Take a closer look at how we are using useMemo. The acute of you might have noticed that useMemo is structured very similarly to how we might structure a useEffect. Its first argument is a function and the second is an array.

In this case, we pass a function to useMemo that returns the sorted list of numbers. The second argument is the array of dependencies. When any of these dependencies change, useMemo will recalculate the value. In this case, we only have one dependency: numbers. This means that in subsequent renders, the function will only run again when the numbers prop changes.

This entire process is known as **“memoization”**, and it helps us optimize performance by remembering the outcomes of expensive calculations and only re-running them when necessary.

**#7 useCallback**

While useMemo is great for **memoizing return values**, useCallback allows us to **memoize entire functions**. Keeping this in mind, you’ll probably be able to familiarize yourself with the syntax of useCallback relatively quickly.

The basic syntax of useCallback is as follows:

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

useCallback takes two parameters: a function and an array of dependencies. It returns a memoized version of the callback function that only changes if one of the dependencies has changed.

**What problem does useCallback solve?**

In React, a function is re-created in memory every time a component renders. For simple functions, this may not be an issue. However, for complex functions that run frequently, this could be detrimental to performance. useCallback solves this problem by returning a memoized version of the callback function, which ensures that unless the dependencies change, the same function reference is used, reducing unnecessary renders.

**Comparing useCallback and useMemo**

For many people, even those with extensive React experience, the nuances distinguishing these two hooks often create an air of uncertainty regarding their fundamental differences. So, let’s dedicate this sub-section to dispelling any lingering confusion.

In short, the answer lies in **what they return and their use cases.**

**useCallback** returns a memoized version of the callback function itself.

**useMemo**, on the other hand, returns the value of the function.

Let’s explore a component that uses both hooks to illustrate the differences.

Suppose we have an app that displays a list of users. When a user is selected, a detailed description of that user is shown. We’ll need to filter through the usersData array to find the selected user. Filtering can be a heavy operation if the user list is large, so we use useMemo to memoize the result of this operation. Additionally, a button will update a count when clicked, and the click handler function will be memoized using useCallback.

import React, { useState, useCallback, useMemo } from 'react';  
  
// Mock data of users  
const usersData = [  
 { id: 1, name: 'John', details: 'Details about John...' },  
 { id: 2, name: 'Jane', details: 'Details about Jane...' },  
 // ... more users  
];  
  
function App() {  
 const [count, setCount] = useState(0);  
 const [selectedUserId, setSelectedUserId] = useState(1);  
  
 // Increment count (memoized using useCallback)  
 const incrementCount = useCallback(() => {  
 setCount(count + 1);  
 }, [count]);  
  
 // Filter through users to find the selected user (memoized using useMemo)  
 const selectedUser = useMemo(() => {  
 console.log('Filtering users...'); // To show when this operation is performed  
 return usersData.find((user) => user.id === selectedUserId);  
 }, [selectedUserId]);  
  
 return (  
 <div>  
 <p>Count: {count}</p>  
 <button onClick={incrementCount}>Increment count</button>  
 <h2>Selected User</h2>  
 {selectedUser && (  
 <div>  
 <h3>{selectedUser.name}</h3>  
 <p>{selectedUser.details}</p>  
 </div>  
 )}  
 {/\* Buttons to select users \*/}  
 <button onClick={() => setSelectedUserId(1)}>Select John</button>  
 <button onClick={() => setSelectedUserId(2)}>Select Jane</button>  
 </div>  
 );  
}  
  
export default App;

The two main points to note from the code above are:

1. incrementCount is a function that increments the count state. This function is passed to the button's onClick handler. Each time the App component renders, a new incrementCount function would be created. But since we're using useCallback, the same incrementCount function is used unless its dependencies (count) change.

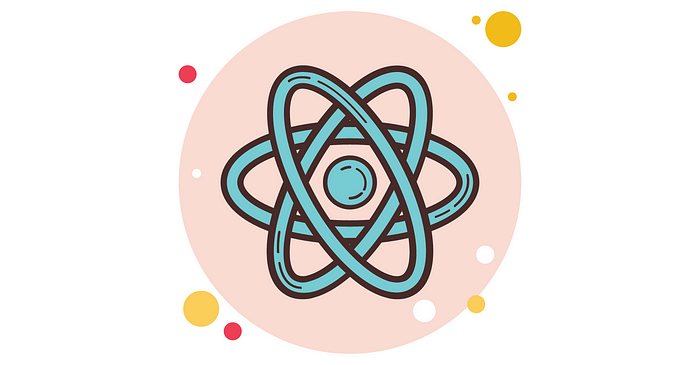
const incrementCount = useCallback(() => {  
 setCount(count + 1);  
 }, [count]);

2. selectedUser is the result of filtering through a list of users to find the user with the id that matches selectedUserId. This is an operation that could be expensive if the list of users were large. By using useMemo, we ensure that this operation is only performed when selectedUserId changes, not every time the App component renders.

const selectedUser = useMemo(() => {  
 console.log('Filtering users...'); // To show when this operation is performed  
 return usersData.find((user) => user.id === selectedUserId);  
 }, [selectedUserId]);

As you can see, both hooks intrinsically perform similar operations and were made to make your components more performant. The only difference is that one deals with pure values (useMemo) and the other deals with the actual functions (useCallback).

**The End**



As you become more proficient with these hooks, you’ll find that knowing when to use each one is as crucial as understanding how they work. Assessing your application’s requirements and considering factors such as state complexity, reusability, and performance, can help you make more judicious decisions on which hooks to implement in different situations.

It’s also important to note that React hooks aren’t rigid templates, but instead versatile tools designed to make your development process smoother and more enjoyable. So don’t forget to embrace experimentation as you integrate them into your projects. If you are really ambitious, you could even try creating your own React hook!

Anyways, till’ next time!