Understanding useMemo and useCallback

If you've struggled to make sense of `useMemo` and `useCallback`, you're not alone! I've spoken with ***so many*** React devs who have been scratching their heads about these two hooks.

My goal in this blog post is to clear up all of this confusion. We'll learn what they do, why they're useful, and how to get the most out of them.

Let's go!

> **Intended audience**
>
> This tutorial is written to help beginner/intermediate React developers get more comfortable with React. If you're taking your very first steps with React, you might wish to bookmark this one and come back to it in a few weeks!

The basic idea

Alright, so let's start with `useMemo`.

The fundamental idea with `useMemo` is that it allows us to *“remember”* a computed value between renders.

This definition requires some unpacking. In fact, it requires a pretty sophisticated mental model of how React works! So let's address that first.

The main thing that React does is keep our UI in sync with our application state. The tool that it uses to do this is called a “re-render”.

Each re-render is a snapshot of what the application's UI should look like at a given moment in time, based on the current application state. We can think of it like a stack of photographs, each one capturing how things looked given a specific value for every state variable.

— Snapshot #1 —
{ ticketIds: []
— Snapshot #2 —
{ ticketIds: ['abc']
— Snapshot #3 —
{ ticketIds: ['abc', 'def'] }

```
<p style="color: re
  Pleas
  (You
</p>
```

```
<p>
  ✓ 2 tickets selected.
</p>
```

Reset Cards

Each “re-render” produces a mental picture of what the DOM should look like, based on the current state. In the little demo above, it's pictured as HTML, but in reality it's a bunch of JS objects. This is sometimes called the ***“virtual DOM”***, if you've heard that term.

We don't directly tell React which DOM nodes need to change. Instead, we tell React what the UI *should be* based on the current state. By re-rendering, React creates a new snapshot, and it can figure out what needs to change by comparing snapshots, like playing a “find the differences” game.

Wait, what?

I recently published a blog post that explains what a “re-render” is, and why they occur. If you're feeling a bit lost, it might help to read that blog post first, and then come back to this one!

→ **“Why React Re-Renders”**

React is heavily optimized out of the box, and so ***in general***, re-renders aren't a big deal. But, in certain situations, these snapshots *do* take a while to create. This can lead to performance problems, like the UI not updating quickly enough after the user performs an action.

Fundamentally, `useMemo` and `useCallback` are tools built to help us optimize re-renders. They do this in two ways:

1. Reducing the amount of work that needs to be done in a given render.
2. Reducing the number of times that a component needs to re-render.

Let's talk about these strategies, one at a time.

Use case 1: Heavy computations

Let's suppose we're building a tool to help users find all of the prime numbers between 0 and `selectedNum`, where `selectedNum` is a user-supplied value. A prime number is a number that can only be divided by 1 and itself, like 17.

Here's one possible implementation:

Code Playground

```
          There are {allPrimes.length} prime(s) between 1 and {selectedNum}:
          {' '}
          <span className="prime-list">
            {allPrimes.join(', ')}
          </span>
        </p>
      </>
    );
}

// Helper function that calculates whether a given
// number is prime or not.
function isPrime(n){
  const max = Math.ceil(Math.sqrt(n));

  if (n === 2) {
    return true;
  }

  for (let counter = 2; counter <= max; counter++) {
    if (n % counter === 0) {
      return false;
    }
  }

  return true;
}

export default App;
```

I don't expect you to read every line of code here, so here are the relevant highlights:

→ We have a single piece of state, a number called `selectedNum`.

- Using a `for` loop, we manually calculate all of the prime numbers between 0 and `selectedNum`.
- We render a controlled number input, so the user can change `selectedNum`.
- We show the user all of the prime numbers that we calculated.

This code requires a significant amount of computation. If the user picks a large `selectedNum`, we'll need to go through *tens of thousands* of numbers, checking if each one is prime. And, while there *are* more efficient prime-checking algorithms than the one I used above, it's always going to be computationally intensive.

We do need to perform this calculation ***sometimes***, like when the user picks a new `selectedNum`. But we could potentially run into some performance problems if we wind up doing this work ***gratuitously***, when it doesn't need to be done.

For example, let's suppose our example also features a digital clock:

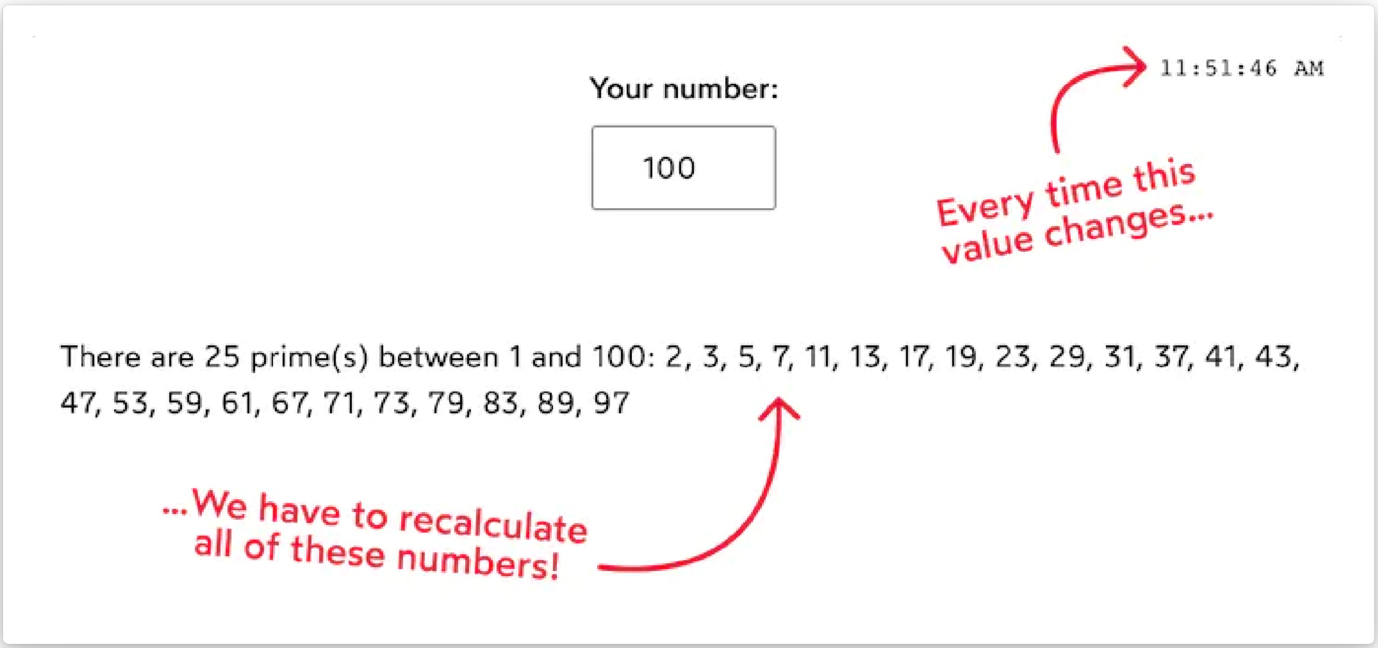
Code Playground

```
import React from 'react';
import format from 'date-fns/format';

function App() {
  const [selectedNum, setSelectedNum] = React.useState(100);

  // `time` is a state variable that changes once per second,
  // so that it's always in sync with the current time.
  const time = useTime();

  // Calculate all of the prime numbers.
  // (Unchanged from the earlier example.)
  const allPrimes = [];
  for (let counter = 2; counter < selectedNum; counter++) {
    if (isPrime(counter)) {
      allPrimes.push(counter);
    }
  }

  return (
    <>
      <p className="clock">
        {format(time, 'hh:mm:ss a')}
      </p>
```

```
<form>
  <label htmlFor="num">Your number:</label>
  <input
    type="number"
```

Our application now has two pieces of state, `selectedNum` and `time`. Once per second, the `time` variable is updated to reflect the current time, and that value is used to render a digital clock in the top-right corner.

Here's the issue: whenever *either* of these state variables change, we re-run all of those expensive prime-number computations. And because `time` changes once per second, it means we're ***constantly*** re-generating that list of primes, even when the user's selected number hasn't changed!


Your number:
100
11:51:46 AM
Every time this value changes...
...We have to recalculate all of these numbers!

There are 25 prime(s) between 1 and 100: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97

In JavaScript, we only have one main thread, and we're keeping it *super* busy by running this code over and over, every single second. It means that the application might feel sluggish as the user tries to do other things, especially on lower-end devices.

But what if we could "skip" these calculations? If we already have the list of primes for a given number, why not ***re-use*** that value instead of calculating it from scratch every time?

This is precisely what `useMemo` allows us to do. Here's what it looks like:

JS

```
const allPrimes = React.useMemo(() => {
  const result = [];

  for (let counter = 2; counter < selectedNum; counter++) {
    if (isPrime(counter)) {
      result.push(counter);
    }
  }

  return result;
}, [selectedNum]);
```

`useMemo` takes two arguments:

1. A chunk of work to be performed, wrapped up in a function
2. A list of dependencies

During mount, when this component is rendered for the very first time, React will invoke this function to run all of this logic, calculating all of the primes. **Whatever we return from this function is assigned to the `allPrimes` variable.**

For every subsequent render, however, **React has a choice to make.** Should it:

1. Invoke the function again, to re-calculate the value, or
2. Re-use the data it already has, from the *last* time it did this work.

To answer this question, React looks at the supplied list of dependencies. Have any of them changed since the previous render? If so, React will rerun the supplied function, to calculate a new value. Otherwise, it'll skip all that work and reuse the previously-calculated value.

`useMemo` is essentially like a lil' cache, and the dependencies are the cache invalidation strategy.

In this case, we're essentially saying "recalculate the list of primes *only when* `selectedNum` changes". When the component re-renders for *other* reasons (eg. the `time` state variable changing), `useMemo` ignores the function and passes along the cached value.

This is commonly known as ***memoization***, and it's why this hook is called "useMemo".

Here's a live version of this solution:

Code Playground

```
          {allPrimes.join(', ')}
        </span>
      </p>
    </>
  );
}

function useTime() {
  const [time, setTime] = React.useState(new Date());

  React.useEffect(() => {
    const intervalId = window.setInterval(() => {
      setTime(new Date());
    }, 1000);

    return () => {
      window.clearInterval(intervalId);
    }
  }, []);

  return time;
}

function isPrime(n){
  const max = Math.ceil(Math.sqrt(n));

  if (n === 2) {
    return true;
  }
```

An alternative approach

So, the `useMemo` hook can indeed help us avoid unnecessary calculations here… but is it *really* the best solution here?

Often, we can avoid the need for `useMemo` by restructuring things in our application.

Here's one way we could do this:

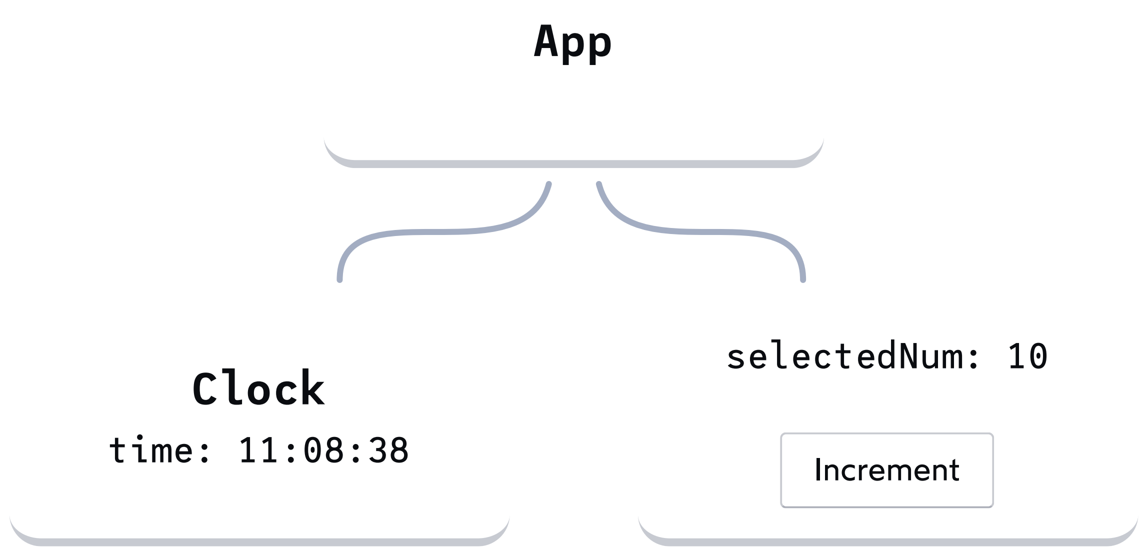
Code Playground

App.js PrimeCalculator.js Clock.js

```
import React from 'react';

import Clock from './Clock';
import PrimeCalculator from './PrimeCalculator';

function App() {
  return (
    <>
      <Clock />
      <PrimeCalculator />
    </>
  );
}

export default App;
```

I've extracted two new components, `Clock` and `PrimeCalculator`. By branching off from `App`, these two components each manage their own state. A re-render in one component won't affect the other.

Here's a graph showing this dynamic. Each box represents a component instance, and they flash when they re-render. Try clicking the “Increment” button to see it in action:



We hear a lot about lifting state up, but sometimes, the better approach is to ***push state down!*** Each component should have a single responsibility, and in the example above, `App` was doing two totally-unrelated things.

Now, this won't always be an option. In a large, real-world app, there's lots of state that ***needs*** to be lifted up pretty high, and can't be pushed down.

I have another trick up my sleeves for this situation.

Let's look at an example. Suppose we **need** the `time` variable to be lifted up, above `PrimeCalculator`:

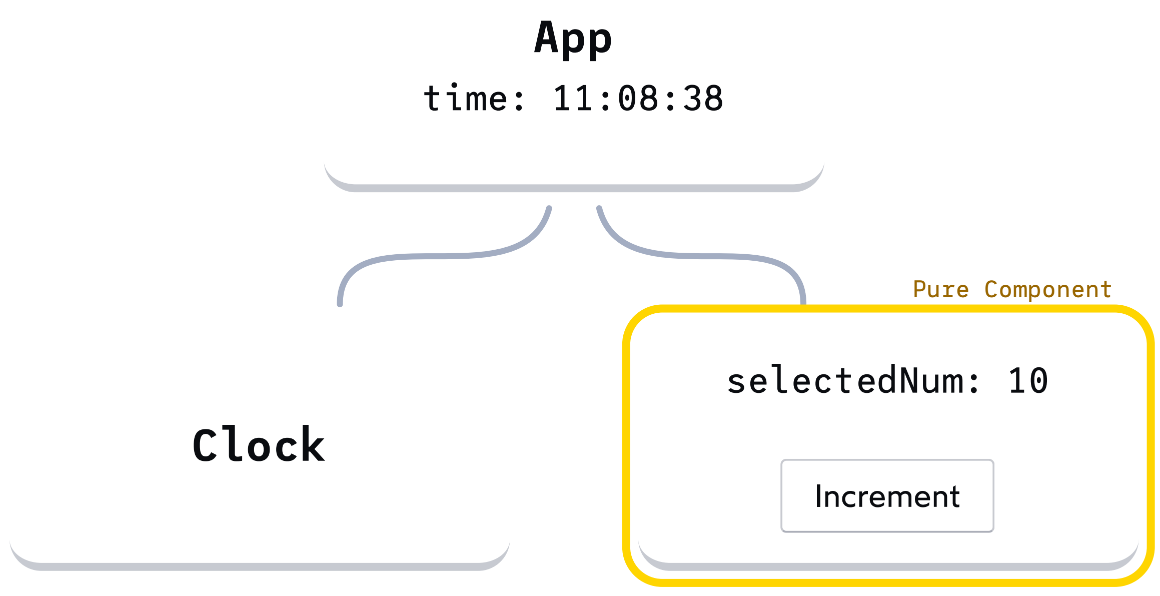
Code Playground

App.js PrimeCalculator.js Clock.js

```
import React from 'react';
import { getHours } from 'date-fns';

import Clock from './Clock';
import PrimeCalculator from './PrimeCalculator';

// Transform our PrimeCalculator into a pure component:
```

```
const PurePrimeCalculator = React.memo(PrimeCalculator);

function App() {
  const time = useTime();

  // Come up with a suitable background color,
  // based on the time of day:
  const backgroundColor = getBackgroundColorFromTime(time);

  return (
    <div style={{ backgroundColor }}>
      <Clock time={time} />
      <PurePrimeCalculator />
    </div>
  );
}

const getBackgroundColorFromTime = (time) => {
  const hours = getHours(time);
```

Here's an updated graph, showing what's happening here:



Like a force field, `React.memo` wraps around our component and protects it from unrelated updates. Our `PurePrimeCalculator` will only re-render when it receives new data, or when its internal state changes.

This is known as a ***pure component***. Essentially, we're telling React that this component will always produce the same *output* given the same *input*, and we can skip the re-renders where nothing's changed.

I wrote more about how `React.memo` works in my recent blog post, "Why React Re-Renders".

A more conventional approach

In the example above, I'm applying `React.memo` to the ***imported*** `PrimeCalculator` component.

In truth, this is a bit unusual. I chose to structure it this way so that everything was visible in the same file, to make it easier to understand.

In practice, I often apply `React.memo` to the component ***export***, like this:

JSX

```
// PrimeCalculator.js
function PrimeCalculator() {
  /* Component stuff here */
}

export default React.memo(PrimeCalculator);
```

Our `PrimeCalculator` component will now always be pure, without us having to tinker with it when we go to consume it.

Should we ever need a non-pure version of `PrimeCalculator`, we can export the underlying component as a named export. I don't think I've ever needed to do this though.

There's an interesting perspective-shift here: Before, we were memoizing the result of a specific computation, calculating the prime numbers. In this case, however, ***I've memoized the entire component instead.***

Either way, the expensive calculation stuff will only re-run when the user picks a new `selectedNum`. But we've optimized the parent component rather than the specific slow lines of code.

I'm not saying that one approach is better than the other; each tool has its spot in the toolbox. But in this specific case, I prefer this approach.

Now, if you've ever tried to use pure components in a real-world setting, you've likely noticed something peculiar: **Pure components often re-render quite a bit,** even when it seems like nothing's changed! 😬

This leads us nicely into the second problem that `useMemo` solves.

> **Even more alternatives**
>
> In his blog post **"Before you memo()"**, Dan Abramov shares another approach based around restructuring the app using `children`, to avoid needing to do any memoization.
>
> Thanks to Yuval Shimoni for pointing this out!

Use case 2: Preserved references

In the example below, I've created a `Boxes` component. It displays a set of colorful boxes, to be used for some sort of decorative purpose.

I also have a bit of unrelated state, the user's name.

App.js Boxes.js

```
import React from 'react';

import Boxes from './Boxes';

function App() {
  const [name, setName] = React.useState('');
  const [boxWidth, setBoxWidth] = React.useState(1);

  const id = React.useId();

  // Try changing some of these values!
  const boxes = [
    { flex: boxWidth, background: 'hsl(345deg 100% 50%)' },
    { flex: 3, background: 'hsl(260deg 100% 40%)' },
    { flex: 1, background: 'hsl(50deg 100% 60%)' },
  ];

  return (
    <>
      <Boxes boxes={boxes} />

      <section>
        <label htmlFor={`${id}-name`}>
          Name:
        </label>
        <input
          id={`${id}-name`}
```

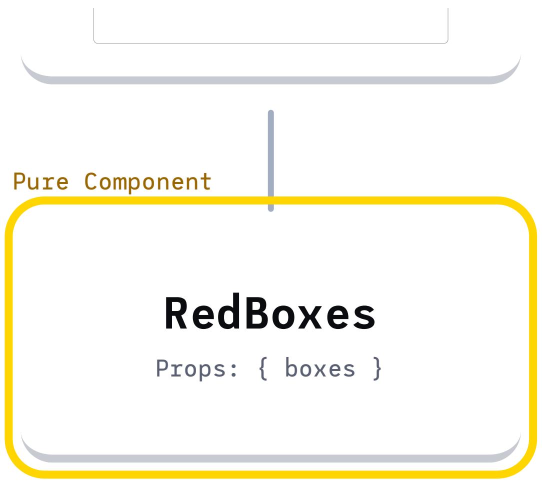
`Boxes` is a pure component, thanks to a `React.memo()` wrapping around the default export in `Boxes.js`. This means that it should only re-render whenever its props change.

And yet, whenever the user changes their name, `Boxes` re-renders as well!

Here's a graph showing this dynamic. Try typing in the text input, and notice how both components re-render:

App

name:



What the heck?! Why isn't our `React.memo()` force field protecting us here??

The `Boxes` component only has 1 prop, `boxes`, and it appears as though we're giving it the exact same data on every render. It's always the same thing: a red box, a wide purple box, a yellow box. We *do* have a `boxWidth` state variable that affects the `boxes` array, but we aren't changing it!

Here's the problem: every time React re-renders, we're producing a ***brand new array***. They're equivalent in terms of ***value***, but not in terms of ***reference***.

I think it'll be helpful if we forget about React for a second, and talk about plain old JavaScript. Let's look at a similar situation:

JS

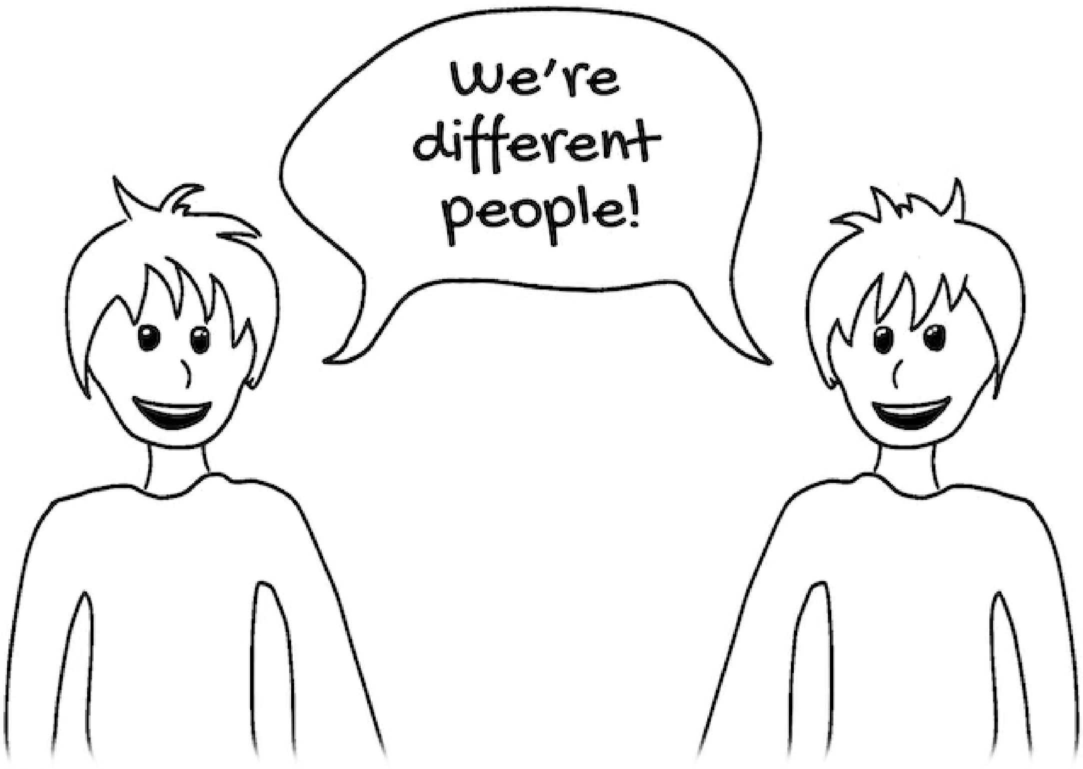
```js
function getNumbers() {
  return [1, 2, 3];
}

const firstResult = getNumbers();
const secondResult = getNumbers();

console.log(firstResult === secondResult);
```

What do you think? Is `firstResult` equal to `secondResult` ?

In a sense, they are. Both variables hold an identical structure, `[1, 2, 3]` . But that's not what the `===` operator is actually checking.

Instead, `===` is checking whether two expressions *are the same thing.*

We've created two different arrays. They may hold the same contents, but they're not the same array, in the same way that ***two identical twins are not the same person.***



Every time we invoke the `getNumbers` function, we create a brand-new array, a distinct thing held in the computer's memory. If we invoke it multiple times, we'll store multiple copies of this array in-memory.

Note that simple data types — things like strings, numbers, and boolean values — can be compared by value. But when it comes to arrays and objects, they're only compared by *reference.* For more information on this distinction, check out this wonderful blog post by Dave Ceddia: A Visual Guide to References in JavaScript.

Taking this back to React: Our `Boxes` React component is also a JavaScript function. When we render it, we invoke that function:

JSX

```jsx
// Every time we render this component, we call this function...
function App() {
  // ...and wind up creating a brand new array...
  const boxes = [
    { flex: boxWidth, background: 'hsl(345deg 100% 50%)' },
    { flex: 3, background: 'hsl(260deg 100% 40%)' },
    { flex: 1, background: 'hsl(50deg 100% 60%)' },
  ];

  // ...which is then passed as a prop to this component!
  return (
    <Boxes boxes={boxes} />
  );
}
```

When the `name` state changes, our `App` component re-renders, which re-runs all of the code. We construct a brand-new `boxes` array, and pass it onto our `Boxes` component.

And `Boxes` re-renders, because we gave it a brand new array!

The *structure* of the `boxes` array hasn't changed between renders, but that isn't relevant. All React knows is that the `boxes` prop has received a freshly-created, never-before-seen array.

To solve this problem, we can use the `useMemo` hook:

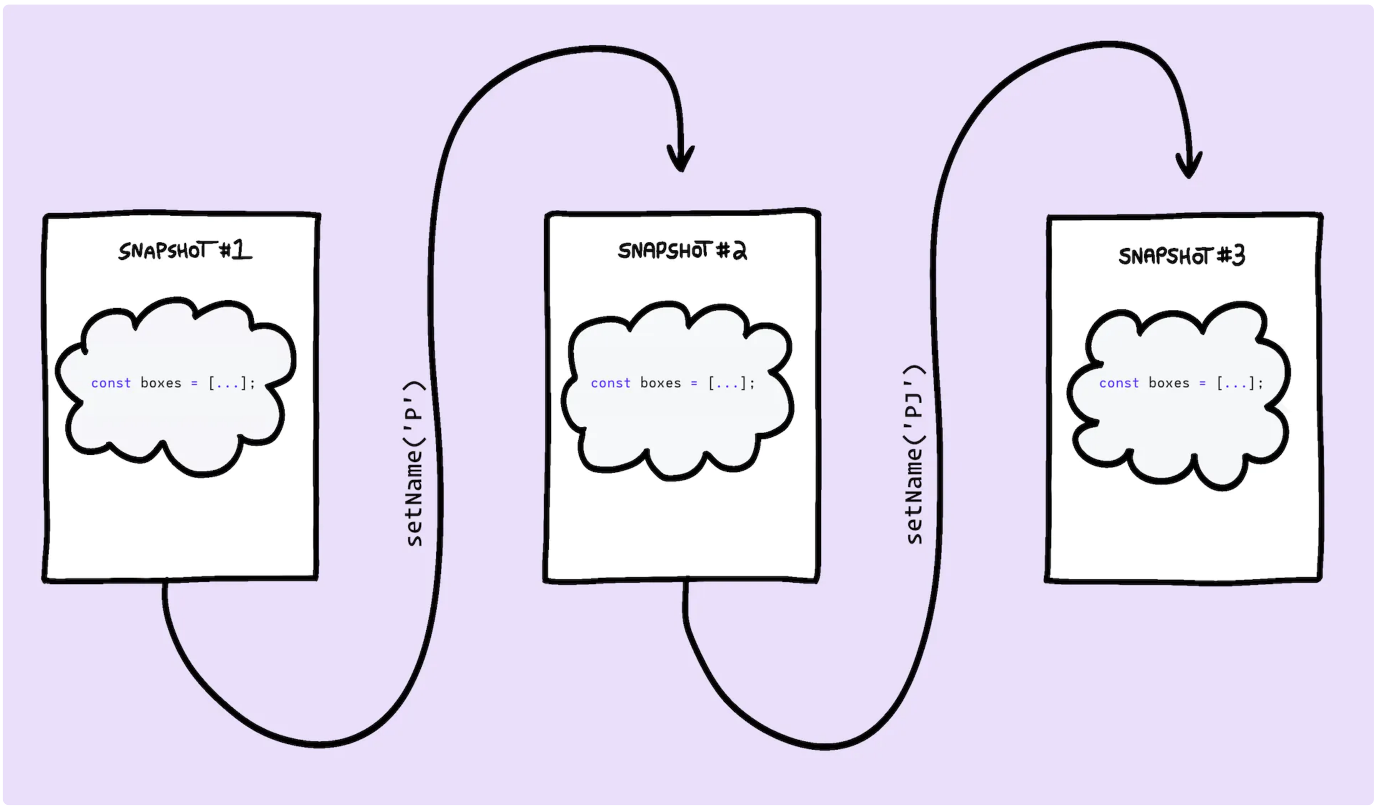
JS

```
const boxes = React.useMemo(() => {
  return [
    { flex: boxWidth, background: 'hsl(345deg 100% 50%)' },
    { flex: 3, background: 'hsl(260deg 100% 40%)' },
    { flex: 1, background: 'hsl(50deg 100% 60%)' },
  ];
}, [boxWidth]);
```

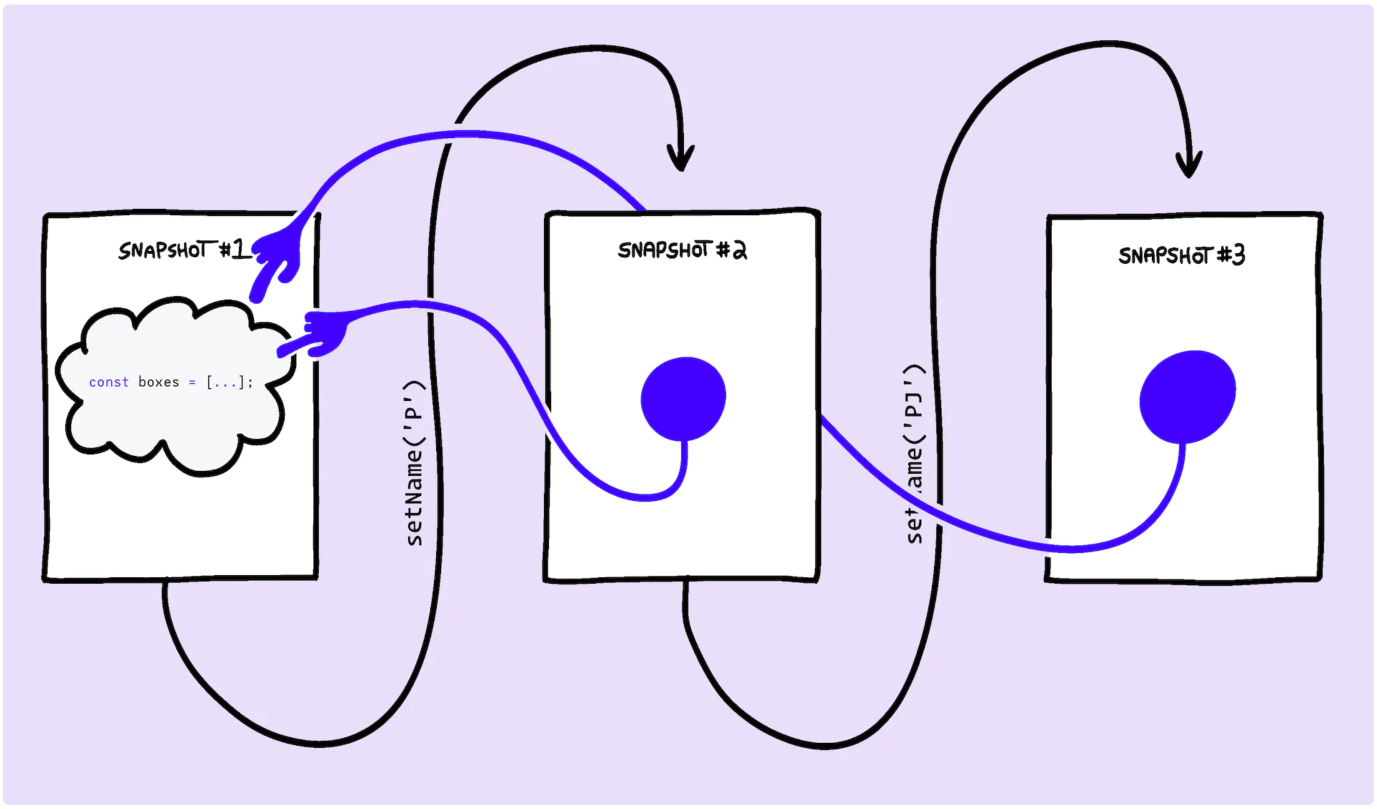
Unlike the example we saw earlier, with the prime numbers, we're not worried about a computationally-expensive calculation here. Our only goal is to ***preserve a reference*** to a particular array.

We list `boxWidth` as a dependency, because we *do* want the `Boxes` component to re-render when the user tweaks the width of the red box.

I think a quick sketch will help illustrate. Before, we were creating a brand new array, as part of each snapshot:



With `useMemo`, however, we're re-using a previously-created `boxes` array:



By preserving the same reference across multiple renders, we allow pure components to function the way we want them to, ignoring renders that don't affect the UI.

The useCallback hook

Alright, so that just about covers `useMemo` … what about `useCallback`?

Here's the short version: It's the exact same thing, but for ***functions*** instead of arrays / objects.

Similar to arrays and objects, functions are compared by reference, not by value:

JS

```
const functionOne = function() {
  return 5;
};
const functionTwo = function() {
  return 5;
};

console.log(functionOne === functionTwo); // false
```

This means that if we define a function within our components, it'll be re-generated on every single render, producing an identical-but-unique function each time.

Let's look at an example:

Code Playground

App.js MegaBoost.js

```
import React from 'react';

import MegaBoost from './MegaBoost';

function App() {
  const [count, setCount] = React.useState(0);

  function handleMegaBoost() {
    setCount((currentValue) => currentValue + 1234);
  }

  return (
    <>
      Count: {count}
      <button
        onClick={() => {
          setCount(count + 1)
        }}
      >
        Click me!
      </button>
      <MegaBoost handleClick={handleMegaBoost} />
    </>
  );
}
```

This sandbox depicts a typical counter application, but with a special “Mega Boost” button. This button increases the count by a large amount, in case you're in a hurry and don't want to click the standard button a bunch of times.

The `MegaBoost` component is a pure component, thanks to `React.memo`. It doesn't depend on `count`… **But it re-renders whenever `count` changes!**

Like we saw with the `boxes` array, the problem here is that we're generating a brand new function on every render. If we render 3 times, we'll create 3 separate `handleMegaBoost` functions, breaking through the `React.memo` force field.

Using what we've learned about `useMemo`, we could solve the problem like this:

JS

```js
const handleMegaBoost = React.useMemo(() => {
  return function() {
    setCount((currentValue) => currentValue + 1234);
  }
}, []);
```

Instead of returning an array, we're returning a ***function***. This function is then stored in the `handleMegaBoost` variable.

This works… but there's a better way:

JS

```js
const handleMegaBoost = React.useCallback(() => {
  setCount((currentValue) => currentValue + 1234);
}, []);
```

`useCallback` serves the same purpose as `useMemo`, but it's built specifically for functions. We hand it a function directly, and it memoizes that function, threading it between renders.

Put another way, these two expressions have the same effect:

JS

```js
// This:
React.useCallback(function helloWorld(){}, []);

// ...Is functionally equivalent to this:
React.useMemo(() => function helloWorld(){}, []);
```

useCallback is syntactic sugar. It exists purely to make our lives a bit nicer when trying to memoize callback functions.

When to use these hooks

Alright, so we've seen how `useMemo` and `useCallback` allow us to thread a reference across multiple renders, to reuse complex calculations or to avoid breaking pure components. The question is: **how often should we use it?**

In my personal opinion, it would be a waste of time to wrap every single object/array/function in these hooks. The benefits are negligible in most cases; React is highly optimized, and re-renders often aren't as slow or expensive as we often think they are!

The best way to use these hooks is in response to a problem. If you notice your app becoming a bit sluggish, you can use the React Profiler to hunt down slow renders. In some cases, you'll be able to improve performance by restructuring your application. In other cases, `useMemo` and `useCallback` can help speed things up.

(If you're not sure how to use the React profiler, I covered it in my recent blog post, "Why React Re-Renders"!)

That said, there are a couple of scenarios in which I *do* pre-emptively apply these hooks.

This could change in the future!

The React team is actively investigating whether it's possible to "auto-memoize" code during the compile step. It's still in the research phase, but early experimentation appears promising.

It could be that in the future, all of this stuff is done for us automatically. Until then, though, we still need to optimize things ourselves.

For more information, check out this talk by Xuan Huang called **"React without memo"**.

Inside generic custom hooks

One of my favourite lil' custom hooks is `useToggle`, a friendly helper that works almost exactly like `useState`, but can only toggle a state variable between `true` and `false`:

JS

```
function App() {
  const [isDarkMode, toggleDarkMode] = useToggle(false);

  return (
    <button onClick={toggleDarkMode}>
      Toggle color theme
    </button>
  );
}
```

Here's how this custom hook is defined:

JS

```
function useToggle(initialValue) {
  const [value, setValue] = React.useState(initialValue);

  const toggle = React.useCallback(() => {
    setValue(v => !v);
  }, []);

  return [value, toggle];
}
```

Notice that the `toggle` function is memoized with `useCallback`.

When I build custom reusable hooks like this, I like to make them as efficient as possible, because **I don't know where they'll be used in the future.** It might be overkill in 95% of situations, but if I use this hook 30 or 40 times, there's a good chance this will help improve the performance of my application.

Inside context providers

When we share data across an application with context, it's common to pass a big object as the `value` attribute.

It's generally a good idea to memoize this object:

JS

```js
const AuthContext = React.createContext({});

function AuthProvider({ user, status, forgotPwLink, children }){
  const memoizedValue = React.useMemo(() => {
    return {
      user,
      status,
      forgotPwLink,
    };
  }, [user, status, forgotPwLink]);

  return (
    <AuthContext.Provider value={memoizedValue}>
      {children}
    </AuthContext.Provider>
  );
}
```

Why is this beneficial? There might be dozens of pure components that consume this context. Without `useMemo`, all of these components would be forced to re-

render if `AuthProvider`'s parent happens to re-render.

The Joy of React

Phew! You made it to the end. I know that this tutorial covers some pretty rocky ground. 😅

I know that these two hooks are thorny, that React itself can feel really overwhelming and confusing. It's a difficult tool!

But here's the thing: If you can get over the initial hump, React is an *absolute joy* to use.

I started using React back in 2015, and it's become my absolute favourite way to build complex user interfaces and web applications. I've tried just about every JS framework under the sun, and I'm just not as productive with them as I am with React.

If you've struggled with React, I'm here to help!

For the past year, I've been hard at work on a course called The Joy of React. It's a beginner-friendly self-paced interactive online course that teaches you how to build next-level cool stuff with React.

If you found this blog post even a little bit helpful, I think you'll get ***so much*** out of my course. Unlike this blog post, the course combines interactive articles like this one with videos, exercises, mini-games, and even some real-world-style projects. It's a hands-on adventure.

The course isn't released yet, but you can learn more and sign up for updates on the course homepage:

→ www.joyofreact.com

LAST UPDATED

August 30th, 2022

HITS

102624