What Is Memoization, and How Does the useMemo Hook Work?

As your React app gets larger, unnecessary re-renders and expensive calculations can slow down performance, leading to slow UI updates and increased resource usage.

This can be especially problematic in apps with complex state management, large lists, functions that require heavy computations, and many components with a single parent.

This gives rise to the need to optimize your React app for better performance by minimizing redundant computations and ensuring smoother interactions.

React solves this problem with a process called memoization, a technique which caches values and functions to prevent unnecessary recalculations, so your app can be faster and more responsive.

By definition, memoization is an optimization technique in which the result of expensive function calls are cached (remembered) based on specific arguments. When the same arguments are provided again, the cached result is returned instead of re-computing the function.

The memoization process happens this way:

- Store the results of function calls along with their input arguments.

- Before executing the function, check if the result for the current arguments already exists in the cache.
- If it exists, return the cached result instead of running the computation again.
- If it doesn't exist, compute the result, store it in the cache, and then return it.

To improve developer experience with memoization, React provides three tools – `React.memo` (or `memo`), `useMemo` and `useCallback`.

As you might guess, both `useMemo` and `useCallback` are hooks, but `React.memo` is a component wrapper, a higher-order component (HOC).

In the next lesson, we will take a look at how the `useCallback` hook and `React.memo` work.

`useMemo` lets you memoize computed values while `useCallback` does the same for function references.

If you're wondering what computed values and function references are, computed values refer to the result of executing a function, while function references are the pointers to functions – the function object in memory.

Let's see how to use the `useMemo` hook first. Here's the basic syntax of the `useMemo` hook:

```
const memoizedValue = useMemo(
  function () {
    return computeExpensiveValue(a, b);
  },
  [a, b]
);
```

You can see all that's needed is to wrap the `useMemo` hook around the function.

This `ExpensiveSquare` component will receive a `num` prop which it will use to calculate the square:

```
function ExpensiveSquare({ num }) {
  function calculateSquare(n) {
    console.log("Calculating square...");
    return n * n;
  }

  const squared = calculateSquare(num);
  return (
    <p>
      Square of {num}: {squared}
    </p>
  );
}
export default ExpensiveSquare;
```

Here's the `App` component where the `ExpensiveSquare` is being used:

```
import { useState, useEffect } from "react";
import ExpensiveSquare from "./components/ExpensiveSquare";

function App() {
 const [timer, setTimer] = useState(0);
 const [num, setNum] = useState(0);

 useEffect(() => {
   const interval = setInterval(() => setTimer((c) => c + 1), 1000);
   return () => clearInterval(interval);
 }, []);

 return (
   <div>
     <h1>Timer: {timer} seconds gone</h1>
     <ExpensiveSquare num={num} />
     <button onClick={() => setNum((n) => n + 1)}>Increase Number</button>
   </div>
 );
}

export default App;
```

The `timer` in the `useEffect`, running every second, will make the `calculateSquare` function run any time it runs, even when you don't increase the `num` state variable.

To solve this problem, we can use the `useMemo` hook by wrapping the function call in it and specifying the `num` variable as the dependency:

```
// import the useMemo hook
import { useMemo } from "react";

function ExpensiveSquare({ num }) {
  function calculateSquare(n) {
    console.log("Calculating square...");
    return n * n;
  }

  // const squared = calculateSquare(num);
  // Wrap the function call in useMemo instead
  const squared = useMemo(() => calculateSquare(num), [num]);

  return (
    <p>
      Square of {num}: {squared}
    </p>
  );
}

export default ExpensiveSquare;
```

This will make sure the function is memoized by caching the result, so calculation happens only when the `num` variable changes, not when anything changes in the component it's being used in.

The `calculateSquare` function call is not running any time `timer` changes anymore but on the initial render and when `num` changes.

Questions

What is memoization in React?

A technique that caches values and functions to prevent unnecessary recalculations.

A technique that lets you manage component state updates to prevent unnecessary recalculations.

A process of reconciling the Virtual DOM with the actual DOM.

A way to handle side effects in functional components.

What is the difference between computed values and function references?

Computed values are function objects, while function references are execution results.

Computed values are the result of executing a function, while function references are the function objects in memory.

Computed values and function references are the same thing.

Function references store computed values.

Which of these is NOT one of the tools React provides for memoization?

`React.memo`

`useMemo`

`useCallback`

`useEffect`