useMemo Hook

In the lifecycle of a component, React re-renders the component when an update is made. When React checks for any changes in a component, it may detect an unintended or unexpected change due to how JavaScript handles equality and shallow comparisons. This change in the React application will cause it to re-render unnecessarily.
Additionally, if that re-rendering is an expensive operation, like a long `for loop`, it can hurt performance. Expensive operations can be costly in either time, memory, or processing.
In addition to potential technical issues, this may lead to poor user experience.
If one part re-renders, it re-renders the entire component tree.
Thus, React released the `memo` idea to fix this.

Understanding Memoization

Memoization is an optimization technique that passes a complex function to be memoized. In memoization, the result is "remembered" when the same parameters are passed-in subsequently.
If we have a function compute `1 + 1`, it will return `2`. But if it uses memoization, the next time we run `1`'s through the function, it won't add them up; it will just remember the answer is `2` without executing the adding function.
From the official React documentation, `useMemo`'s signature looks like this:
`const memoizedValue = React.useMemo(() => computeExpensiveValue(a, b), [a, b]);`
`useMemo` takes in a function and an array of dependencies.
The dependencies act similar to arguments in a function. The dependency's list are the elements `useMemo` watches: if there are no changes, the function result will stay the same. Otherwise, it will re-run the function. If they don't change, it doesn't matter if our entire component re-renders, the function won't re-run but instead return the stored result. This can be optimal if the wrapped function is large and expensive. That is the primary use for `useMemo`.

Creating a `useMemo` Example

Here is an abstract example of using `useMemo` for an array of items that uses two computationally expensive functions:

```
const List = React.useMemo(() =>

  listOfItems.map(item => ({

    ...item,

    itemProp1: expensiveFunction(props.first),

    itemProp2: anotherPriceyFunction(props.second)

  })), [listOfItems]

)
```

In the above example, the `useMemo` function would run on the first render. It would block the thread until the expensive functions complete, as `useMemo` runs in the first render. Initially, this will not look as clean as `useEffect`, since `useEffect` can render a loading spinner until the expensive functions finish and the effects fire off.
However, in subsequent renders, the expensive functions would not need to run again as long as `listOfItems` never changed. `useMemo` would “remember” the return value of each function.
It would make these expensive functions appear to render instantaneous. This is ideal if you have an expensive, synchronous function or two.

When to Use `useMemo`

Write the code first and then revisit it to see if you can optimize it. If you implement `useMemo` too often in an application, it can harm the performance.
When looking to implement `useMemo`, you can check with profiling tools to identify expensive performance issues. *Expensive* means it is using up a lot of resources (like memory). If you are defining a good number of variables in a function at render, it makes sense to memoize with `useMemo`.

Using the Right Hook for the Job

In addition to `useMemo`, there is also `useCallback`, `useRef`, and `useEffect`.
The `useCallback` hook is similar to `useMemo`, but it returns a memoized function, while `useMemo` has a function that returns a value.
If your dependencies array is not provided, there is no possibility of memoization, and it will compute a new value on every render. You could use the `useRef` hook in that instance. The advantage `useMemo` offers over `useRef` is a re-memoizing if the dependencies change.
You won't want to have `useMemo` fire off any side effects or any asynchronous calls. In those instances, you should use `useEffect`.

Conclusion

This article explored the `useMemo` hook and when it is appropriate to use it in a React application.`useMemo` can help the performance of an application by "remembering" expensive functions and preventing a re-render every time there is a change in the application.

While performance can be improved by using this hook, it can also slow down your application if you overuse it. The more you use the hook, the more your application has to allocate memory.