How re-rendering works with Context

In this reading you will learn about the default behavior of React rendering and when context is used. You will discover how to prevent unnecessary top-level re-renders with `React.memo` and how object references work in JavaScript. You will also learn how to utilize the `useMemo` hook to guarantee object references don't change during re-rendering.

So far, you have learned that when a component consumes some context value and the value of this context changes, that component re-renders.

But what happens with all components in between? Is React wise enough to only re-render the consumers and bypass the intermediary components in the tree? Well, as it turns out, that doesn't always happen and extra care should be taken when designing your React Context.

When it comes to the default behavior of React rendering, if a component renders, React will recursively re-render all its children regardless of props or context. Let's illustrate this point with an example that uses some context.

Imagine the following component structure, where the top level component injects a Context provider at the top:

`App (ContextProvider) > A > B > C`

```
const App = () => {
return (
   <AppContext.Provider>
     <ComponentA />
   </AppContext.Provider>
 );
};

const ComponentA = () => <ComponentB />;
const ComponentB = () => <ComponentC />;
const ComponentC = () => null;
```

If the outermost App component re-renders for whatever reason, all `ComponentA`, `ComponentB` and `ComponentC` components will re-render as well, following this order:

`App (ContextProvider) -> A -> B -> C`

If some of your top level components are complex in nature, this could result in some performance hit. To mitigate this issue, you can make use of the top level API `React.memo()`.

If your component renders the same result given the same props, you can wrap it in a call to `React.memo` for a performance boost by memoizing the result.

Memoization is a programming technique that accelerates performance by caching the return values of expensive function calls.

This means that React will skip rendering the component, and reuse the last rendered result. This is a trivial case for `ComponentA`, since it doesn't receive any props.

`const ComponentA = React.memo(() => <ComponentB />);`

React.memo takes the component definition as a first argument. An optional second argument can be included if you would like to specify some custom logic that defines when the component should re-render based on previous and current props.

After that little adjustment, you will prevent any rendering from happening in all `ComponentA`, `ComponentB` and `ComponentC` if the App component re-renders.

```
const App = () => {
return (
   <AppContext.Provider>
     <ComponentA />
   </AppContext.Provider>
 );
};

const ComponentA = React.memo(() => <ComponentB />);
const ComponentB = () => <ComponentC />;
const ComponentC = () => null;
```

A good rule of thumb is to wrap the React component right after your context provider with `React.memo`.

In real-life applications, you will find yourself in need of passing several pieces of data as context value, rather than a single primitive like a string or number, so you'll be working most likely with JavaScript objects.

Now, according to React context rules, all consumers that are descendants of a provider will re-render whenever the provider's value prop changes.

Let's go through the following scenario built upon the previous example, where the context value that gets injected is defined as an object called `value` with two properties, 'a' and 'b', being both strings. Also, `ComponentC` is now a consumer of context, so any time the provider `value` prop changes, `ComponentC` will re-render.

```
const App = () => {
  const value = {a: 'hi', b: 'bye'};
  return (
    <AppContext.Provider value={value}>
      <ComponentA />
    </AppContext.Provider>
  );
};

const ComponentA = React.memo(() => <ComponentB />);
const ComponentB = () => <ComponentC />;
const ComponentC = () => {
  const contextValue = useContext(AppContext);
  return null;
};
```

Imagine that the value prop from the provider changes to `{a: 'hello', b: 'bye'}`.

If that happens, the sequence of re-renders would be:

`App (ContextProvider) -> C`

That's all fine and expected, but what would happen if the App component re-renders for any other reason and the

provider value doesn't change at all, being still `{a: 'hi', b: 'bye'}`?

It may be a surprise to you to find out that the sequence of re-renders is the same as before:

```
App (ContextProvider) -> C
```

Even though the provider value doesn't seem to change, `ComponentC` gets re-rendered.

To understand what's happening, you need to remember that in JavaScript, the below assertion is true:

```
{a: 'hi', b: 'bye'} !== {a: 'hi', b: 'bye'}
```

That is because object comparison in JavaScript is done by reference. Every time a new re-render happens in the App component, a new instance of the `value` object is created, resulting in the provider performing a comparison against its previous value and determining that it has changed, hence informing all context consumers that they should re-render.

This problem can be resolved by using the `useMemo` hook from React as follows.

```
const App = () => {
  const a = 'hi';
  const b = 'bye';
  const value = useMemo(() => ({a, b}), [a, b]);

  return (
    <AppContext.Provider value={value}>
      <ComponentA />
    </AppContext.Provider>
  );
};

const ComponentA = React.memo(() => <ComponentB />);
const ComponentB = () => <ComponentC />;
const ComponentC = () => {
  const contextValue = useContext(AppContext);
  return null;
};
```

Hooks will be covered in depth in the next module, so don't worry too much if this is new for you.

For the purpose of this example, it suffices to say that `useMemo` will memoize the returned value from the function passed as the first argument and will only re-run the computation if any of the values are passed into the array as a second argument change.

With that implementation, if the App re-renders for any other reason that does not change any of 'a' or 'b' values, the sequence of re-renders will be as such:

```
App (ContextProvider)
```

This is the desired result, avoiding an unnecessary re-render on `ComponentC`. `useMemo` guarantees keeping the same object reference for the `value` variable and since that's assigned to the provider's value, it determines that the context has not changed and should not notify any consumer.

Conclusion

You have learned about how re-rendering works in React when context is used and how `React.memo` and `useMemo` APIs from React can help you perform some optimizations to avoid unnecessary re-renders in your components tree.

Go to next item Completed

Like Dislike Report an issue

coach

coach