Performance - Avoiding Heavy Computation

Learn how the performance can be optimized by avoiding heavy computation.

We'll cover the following

- Avoid re-run of expensive computations
- Exercises:

The `list` passed to the `List` component is the same, but the `onRemoveItem` callback handler isn't. If the App component re-renders, it always creates a new version of this callback handler. Earlier, we used React's `useCallback` Hook to prevent this behavior, by creating a function only on a re-render (if one of its dependencies has changed).

```
const App = () => {
  ...


  const handleRemoveStory = React.useCallback(item => {

    dispatchStories({
      type: 'REMOVE_STORY',
      payload: item,
    });

  }, []);


  ...

  console.log('B:App');

  return (... );
};
```

src/App.js

Since the callback handler gets the `item` passed as an argument in its function signature, it doesn't have any dependencies and is declared only once when the App component initially renders. None of the props passed to the List component should change now. Try it with the combination of `memo` and `useCallback`, to search via the SearchForm's input field. The "B:List" output disappears, and only

the App component re-renders with its “B:App” output.

While all props passed to a component stay the same, the component renders again if its parent component is forced to re-render. That’s React’s default behavior, which works most of the time because the re-rendering mechanism is fast enough. However, if re-rendering decreases the performance of a React application, `memo` helps prevent re-rendering.

Sometimes `memo` alone doesn’t help, though. Callback handlers are re-defined each time in the parent component and passed as *changed* props to the component, which causes another re-render. In that case, `useCallback` is used for making the callback handler only change when its dependencies change.

Avoid re-run of expensive computations

Sometimes we’ll have performance-intensive computations in our React components – between a component’s function signature and return block – which run on every render. For this scenario, we must create a use case in our current application first.

```
const getSumComments = stories => {
  console.log('C');

  return stories.data.reduce(
    (result, value) => result + value.num_comments,
    0
  );
};


const App = () => {
  ...

  const sumComments = getSumComments(stories);

  return (
    <div>

      <h1>My Hacker Stories with {sumComments} comments.</h1>

      ...
    </div>
  );
};
```

src/App.js

If all arguments are passed to a function, it’s acceptable to have it outside the component. It prevents creating the function on every render, so the `useCallback`

hook becomes unnecessary. The function still computes the value of summed

comments on every render, which becomes a problem for more expensive computations.

Each time text is typed in the input field of the SearchForm component, this computation runs again with an output of “C”. This may be fine for a non-heavy computation like this one, but imagine this computation would take more than 500ms. It would give the re-rendering a delay, because everything in the component has to wait for this computation. We can tell React to only run a function if one of its dependencies has changed. If no dependency changed, the result of the function stays the same. React’s useMemo Hook helps us here:

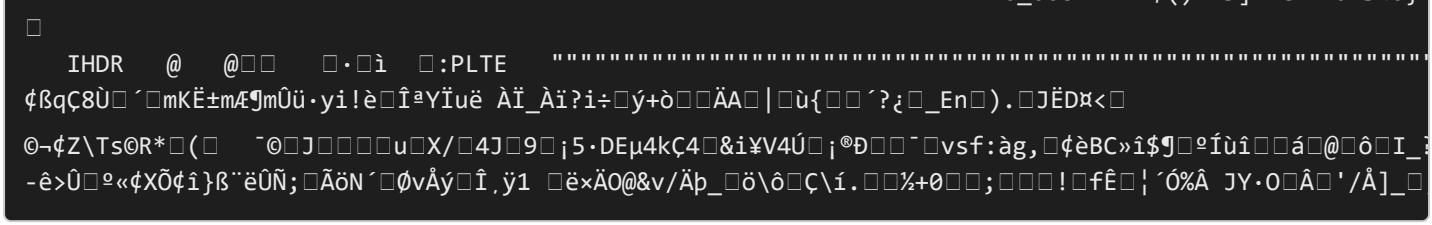
```
const App = () => {
  ...

  const sumComments = React.useMemo(() => getSumComments(stories), [
    stories,
  ]);


  return ( ... );
};
```

src/App.js

For every time someone types in the SearchForm, the computation shouldn’t run again. It only runs if the dependency array, here `stories`, has changed. After all, this should only be used for cost expensive computations which could lead to a delay of a (re-)rendering of a component.

Now, after we went through these scenarios for `useMemo`, `useCallback`, and `memo`, remember that these shouldn’t necessarily be used by default. Apply these performance optimization only if you run into a performance bottlenecks. Most of the time this shouldn’t happen, because React’s rendering mechanism is pretty efficient by default. Sometimes the check for utilities like `memo` can be more expensive than the re-rendering itself.


```
  □ □ □□  □   ã□  F   □□  □   □  )□     □   9□  5□  @@  □   °□  n□  □PNG
□
   IHDR   □   □□□   (-□S   äPLTE"""""""""""""""""""2PX=r□)7;*:>H□¤-BGE□□8do5Xb6[eK□®K□¯1MU9gs3S
□
   IHDR   □   □□□   ×©ÍÊ  □ePLTE"""""""""""""""""""""""""2RZN¢¹J□«3R[J□¬)59YÁÞ0KS4W`Q«ÄL□²%+-0JR
?^q÷ñíÛ□ï.},□ìsæÝ_TttÔ¾ □1#□□/(ì□-[□□□è`□è`Ì□ÚïÅðZ□d5□□□□?ÎebZ¿Þ□i.Ûæ□□□ìqÎ□+1°□}Â□5ù  ïçd
□
   IHDR       □□   D¤□Æ  □APLTE   """""""""""""""""""""""""2RZVºÖ ÔôU·Ñ=r□$()'25]ÎíC□□0LS<o}X
```



Exercises:

- Confirm the changes from the last section.
- Remove all performance optimizations to keep the application simple. Our current application doesn't suffer from any performance bottlenecks. Try to avoid premature optimizations. Use this section as reference, in case you run into performance problems.

Test your knowledge!

1 Does the `SearchForm` re-render when removing an item from the `List` with the "Dismiss" button?

2 Does each `Item` re-render when removing an item from the `List` with the "Dismiss" button?

Retake Quiz