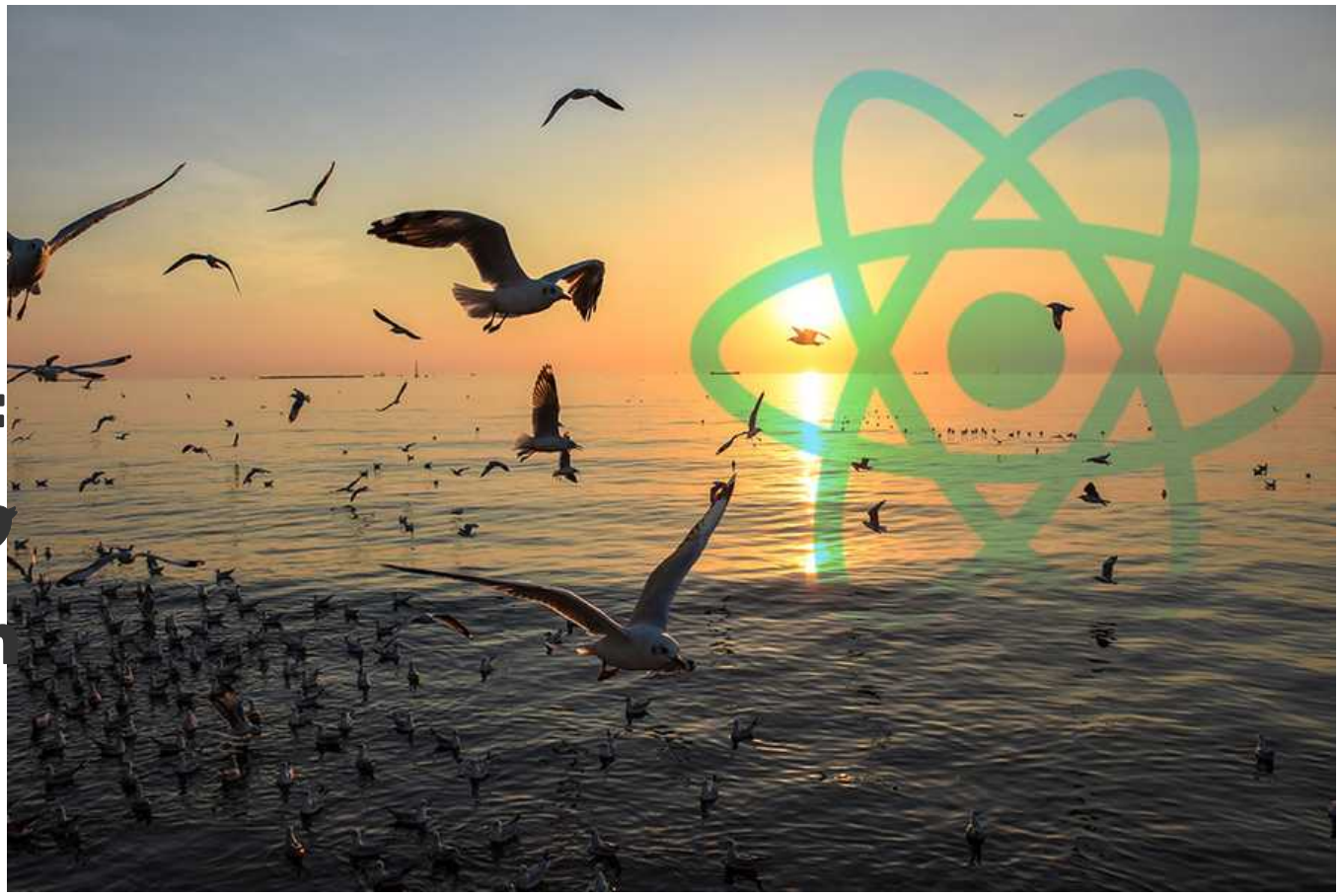
How to useCallback in React

JULY 06, 2020 BY ROBIN WIERUCH - EDIT THIS POST



React's useCallback Hook can be used to **optimize the rendering behavior** of your React function components. We will go through an example component to illustrate the problem first, and then solve it with **React's useCallback Hook**.

Keep in mind that most of the performance optimizations in React are premature. React is fast by default, so *every* performance optimization is opt-in in case something starts to feel slow.

Note: Don't mistake React's useCallback Hook with React's useMemo Hook. While useCallback is used to memoize functions, useMemo is used to memoize values.

Note: Don't mistake React's useCallback Hook with React's memo API. While useCallback is used to memoize functions, React memo is used to wrap React components to prevent re-renderings.

Let's take the following example of a React application which renders a list of user items and allows us to add and remove items with callback handlers. We are using React's useState Hook to make the list stateful:

```
import React from 'react';
import { v4 as uuidv4 } from 'uuid';

const App = () => {
  const [users, setUsers] = React.useState([
    { id: 'a', name: 'Robin' },
    { id: 'b', name: 'Dennis' },
  ]);

  const [text, setText] = React.useState('');

  const handleText = (event) => {
    setText(event.target.value);
  };

  const handleAddUser = ()  =>{
    setUsers(users.concat({ id: uuidv4(), name: text }));
  };

  const handleRemove = (id) => {
    setUsers(users.filter((user) => user.id !== id));
  };

  return (
    <div>
      <input type="text" value={text} onChange={handleText} />
      <button type="button" onClick={handleAddUser}>
        Add User
      </button>

      <List list={users} onRemove={handleRemove} />
    </div>
  );
};

const List = ({ list, onRemove }) => {
  return (
    <ul>
      {list.map((item) => (
        <ListItem key={item.id} item={item} onRemove={onRemove} />
      ))}
    </ul>
  );
};

const ListItem = ({ item, onRemove }) => {
  return (
    <li>
      {item.name}
      <button type="button" onClick={() => onRemove(item.id)}>
```

```
        Remove
      </button>
    </li>
  );
};

export default App;
```

Using what we learned about React memo (if you don't know React memo, read the guide first and then come back), which has similar components to our example, we want to prevent every component from re-rendering when a user types into the input field.

```
const App = () => {
  console.log('Render: App');

  ...
};

const List = ({ list, onRemove }) => {
  console.log('Render: List');
  return (
    <ul>
      {list.map((item) => (
        <ListItem key={item.id} item={item} onRemove={onRemove} />
      ))}
    </ul>
  );
};

const ListItem = ({ item, onRemove }) => {
  console.log('Render: ListItem');
  return (
    <li>
      {item.name}
      <button type="button" onClick={() => onRemove(item.id)}>
        Remove
      </button>
    </li>
  );
};
```

Typing into the input field for adding an item to the list should only trigger a re-render for the App component, but not for its child components which don't care about this state change. Thus, React memo will be used to prevent the child components from updating:

```
const List = React.memo(({ list, onRemove }) => {
  console.log('Render: List');
  return (
    <ul>
      {list.map((item) => (
```

```
          <ListItem key={item.id} item={item} onRemove={onRemove} />
        ))}
      </ul>
    );
});

const ListItem = React.memo(({ item, onRemove }) => {
  console.log('Render: ListItem');
  return (
    <li>
      {item.name}
      <button type="button" onClick={() => onRemove(item.id)}>
        Remove
      </button>
    </li>
  );
});
```

However, perhaps to your surprise, both function components still re-render when typing into the input field. For every character that's typed into the input field, you should still see the same output as before:

```
// after typing one character into the input field

Render: App
Render: List
Render: ListItem
Render: ListItem
```

Let's have a look at the props that are passed to the List component.

```
const App = () => {
  // How we're rendering the List in the App component
  return (
    //...
    <List list={users} onRemove={handleRemove} />
  )
}
```

As long as no item is added or removed from the `list` prop, it should stay intact even if the App component re-renders after a user types something into the input field. So the culprit is the `onRemove` callback handler.

Whenever the App component re-renders after someone types into the input field, the `handleRemove` handler function in the App gets re-defined.

By passing this **new** callback handler as a prop to the List component, it notices **a prop changed compared to the previous render**. That's why the re-render for the List and ListItem components kicks in.

Finally we have our use case for React's useCallback Hook. We can use useCallback to **memoize a function**, which means that this function only gets re-defined if any of its dependencies in the dependency array change:

```
const App = () => {
  ...
  // Notice the dependency array passed as a second argument in useCallback
  const handleRemove = React.useCallback(
    (id) => setUsers(users.filter((user) => user.id !== id)),
    [users]
  );

  ...
};
```

If the `users` state changes by adding or removing an item from the list, the handler function gets re-defined and the child components should re-render.

However, if someone only types into the input field, the function doesn't get re-defined and stays intact. Therefore, the child components don't receive changed props and won't re-render for this case.

You might be wondering why you wouldn't use React's useCallback Hook on all your functions or why React's useCallback Hook isn't the default for all functions in the first place.

Internally, React's useCallback Hook has to compare the dependencies from the dependency array for every re-render to decide whether it should re-define the function. Often the computation for this comparison can be more expensive than just re-defining the function.

. . .

In conclusion, React's useCallback Hook is used to memoize functions. It's already a small performance gain when functions are passed to others components without worrying about the function being re-initialized for every re-render of the parent component. However, as you have seen, React's useCallback Hook starts to shine when used together with React's memo API.

Show Comments

KEEP READING ABOUT REACT›

HOW TO USE REACT MEMO

React's memo API can be used to optimize the rendering behavior of your React function components . We will go through an example component to illustrate the problem first, and then solve it with...

HOW TO USEMEMO IN REACT

React's useMemo Hook can be used to optimize the computation costs of your React function components . We will go through an example component to illustrate the problem first, and then solve it...