Tips and Tricks for Optimizing React Performance

Yasith Wimukthi

Use React.memo or PureComponent

One of the easiest ways to optimize React performance is by using React.memo or PureComponent for functional components and class components, respectively. These React features help in memoizing components, which means they will only re-render when their props change. This can significantly reduce unnecessary re-renders and improve overall performance.

```
// Using React.memo
const MyComponent = React.memo(function MyComponent(props) {
  /* component logic */
});

// Using PureComponent
class MyComponent extends React.PureComponent {
  /* component logic */
}
```

Minimize Re-renders

To further minimize re-renders, ensure that the render method in your components is efficient and only renders necessary parts of the UI. For example, if a component depends on a state or prop that rarely changes, you can wrap the code that uses this state or prop in a React.useMemo hook to memoize the computed value.

```
const MyComponent = ({ value }) => {
  const memoizedValue = React.useMemo(() => {
    /* compute value */
    return /* computed value */;
  }, [value]);

  return <div>{memoizedValue}</div>;
};
```

Use Key Prop in Lists

When rendering lists in React, it's crucial to provide a unique key prop to each item. This helps React identify which items have changed, added, or removed, enabling it to perform more efficient updates.

```
const MyList = ({ items }) => (
  <ul>
    {items.map((item) => (
      <li key={item.id}>{item.text}</li>
    ))}
  </ul>
);
```

Implement Virtualized Lists

For large lists, consider implementing virtualized lists using libraries like react-window or react-virtualized. These libraries render only the items that are visible on the screen, reducing the number of DOM elements and improving rendering performance.

```
import { FixedSizeList as List } from "react-window";

const MyVirtualizedList = ({ items }) => (
  <List
    height={400}
    itemCount={items.length}
    itemSize={50}
    width={300}
  >
    {({ index, style }) => (
      <div style={style}>
        <p>{items[index]}</p>
      </div>
    )}
  </List>
);
```

Use Event Delegation

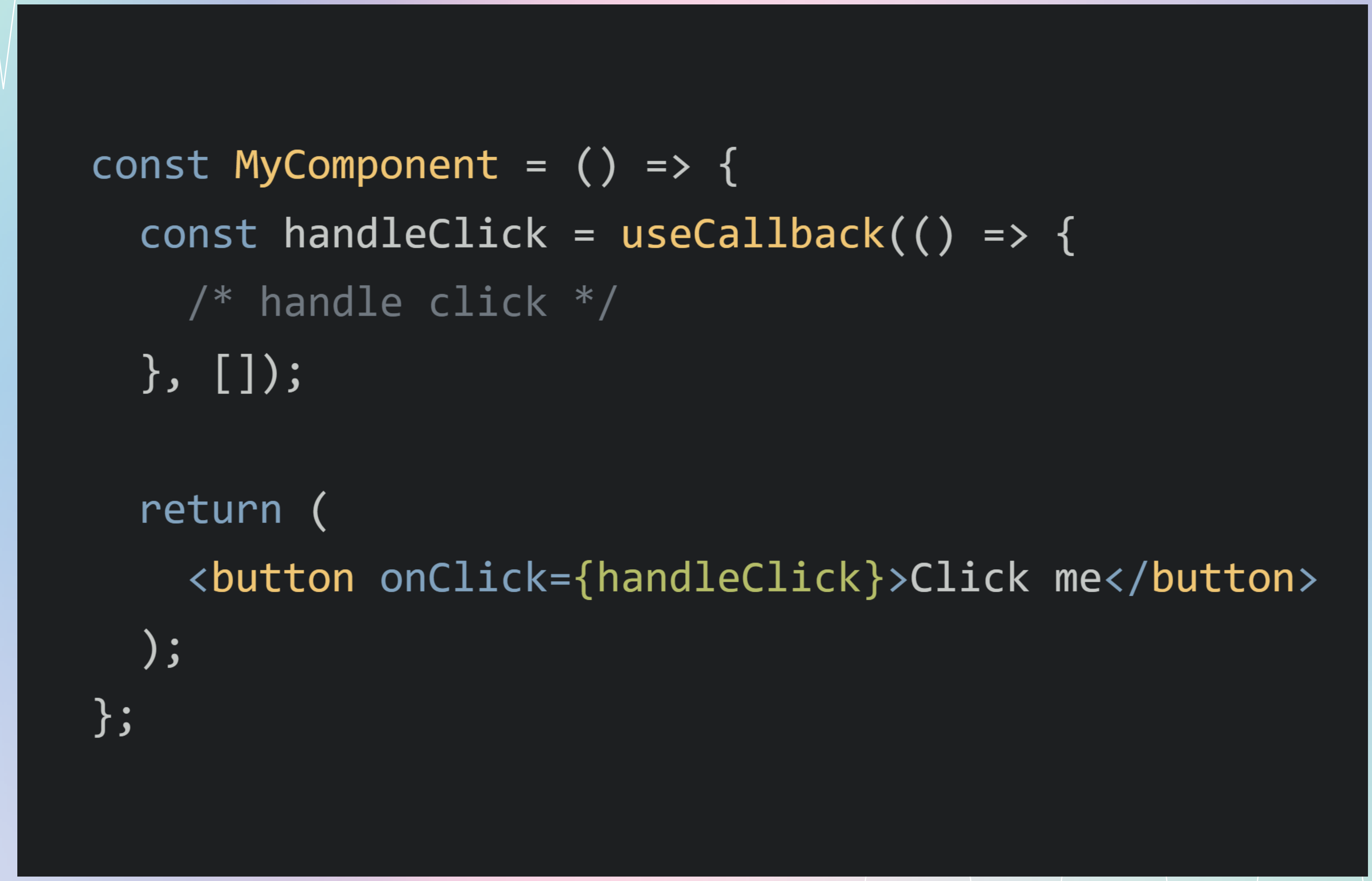
In React, event delegation can be used to optimize event handling by attaching event listeners to parent elements instead of individual elements. This reduces the number of event listeners and improves performance, especially in complex UIs with many interactive elements.

```
const MyComponent = () => {
  const handleClick = (event) => {
    /* handle click */
  };

  return (
    <div onClick={handleClick}>
      <button>Button 1</button>
      <button>Button 2</button>
      <button>Button 3</button>
    </div>
  );
};
```

Use useCallback for Callback Functions

When passing callbacks to child components, use the useCallback hook to memoize the function. This ensures that the function reference does not change on each render, which can lead to unnecessary re-renders of child components.



```
const MyComponent = () => {
  const handleClick = useCallback(() => {
    /* handle click */
  }, []);

  return (
    <button onClick={handleClick}>Click me</button>
  );
};
```

Code Splitting

Code splitting is a technique that allows a React application to load only the necessary JavaScript code for the current view, reducing the initial load time and improving overall performance. With code splitting, you can divide your application into multiple bundles (or chunks) and load them on-demand, rather than loading everything upfront.

```
import React, { Suspense } from 'react';

const MyLazyComponent = React.lazy(() => import('./MyLazyComponent'));

const MyComponent = () => (
  <Suspense fallback={<div>Loading...</div>}>
    <MyLazyComponent />
  </Suspense>
);
```