**10 Expert Performance Tips Every Senior JS React Developer Should Know**

[[Evelyn Taylor](https://medium.com/@Evelyn.Taylor?source=post_page-----f8a78eebcb2e--------------------------------)](https://medium.com/@Evelyn.Taylor?source=post_page-----f8a78eebcb2e--------------------------------)

[Evelyn Taylor](https://medium.com/@Evelyn.Taylor?source=post_page-----f8a78eebcb2e--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Ffd2770b746ea&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40Evelyn.Taylor%2F10-expert-performance-tips-every-senior-js-react-developer-should-know-f8a78eebcb2e&user=Evelyn+Taylor&userId=fd2770b746ea&source=post_page-fd2770b746ea----f8a78eebcb2e---------------------post_header-----------)

4 min read

·

Aug 6

731

7

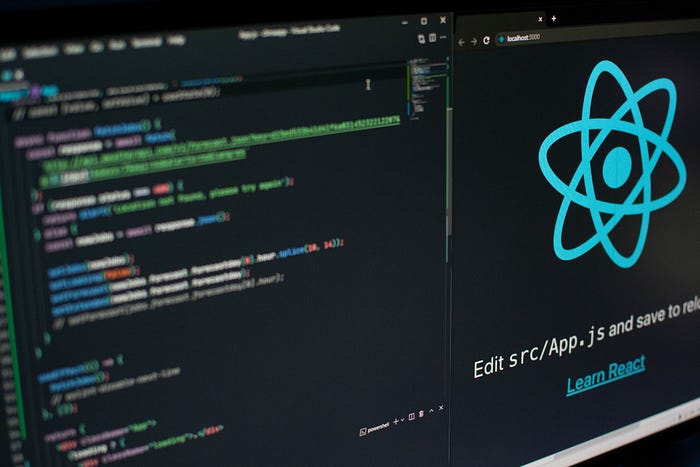


Photo by [Lautaro Andreani](https://unsplash.com/pt-br/@lautaroandreani?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral)

Hey, senior JS React developers! Are you looking to take your skills to the next level and optimize your React applications for top-notch performance?

You’re in the right place!

In this article, I’ll share with you 10 expert performance tips that will supercharge your React development.

Get ready to optimize, streamline, and make your apps lightning-fast. Let’s dive in!

1. **Use Functional Components and React Hooks:**

Functional components with React Hooks offer better performance compared to class components.

They are lighter and don’t carry the overhead of managing instance properties. Let’s see an example:

import React, { useState } from 'react';  
  
function MyComponent() {  
 const [count, setCount] = useState(0);  
  
 const increment = () => {  
 setCount(count + 1);  
 };  
  
 return (  
 <div>  
 <p>Count: {count}</p>  
 <button onClick={increment}>Increment</button>  
 </div>  
 );  
}

By using functional components and React Hooks, you avoid unnecessary re-renders and boost your app’s performance.

**2. Memoize Expensive Computations with useMemo:**

If you have expensive computations or complex data transformations in your components, you can optimize them with the useMemo hook.

It memoizes the result of the computation, preventing unnecessary re-calculations. Check out this example:

import React, { useMemo } from 'react';  
  
function MyComponent({ data }) {  
 const processedData = useMemo(() => {  
 // Expensive computation or data transformation  
 return processData(data);  
 }, [data]);  
  
 // Render the component using processedData  
  
 return <div>{processedData}</div>;  
}

With useMemo, you ensure that the expensive computation is only performed when the dependencies (in this case, data) change.

**3. Optimize Re-rendering with React.memo:**

Use React.memo to memoize functional components and prevent unnecessary re-renders.

It's similar to the PureComponent for class components. Here's an example:

import React from 'react';  
  
const MyComponent = React.memo(({ prop1, prop2 }) => {  
 // Render the component  
  
 return <div>{prop1} - {prop2}</div>;  
});

With React.memo, the component will only re-render if its props change, preventing unnecessary updates.

**4. Use Key Props in Lists:**

When rendering a list of items, make sure to provide a unique key prop to each item. This helps React efficiently update the list when items are added, removed, or reordered. Here's an example:

import React from 'react';  
  
function MyListComponent({ items }) {  
 return (  
 <ul>  
 {items.map((item) => (  
 <li key={item.id}>{item.name}</li>  
 ))}  
 </ul>  
 );  
}

By using a unique key prop for each item, React can efficiently track and update the list when changes occur.

**5. Implement Virtualized Lists for Large Data Sets:**

For large lists with a significant number of items, consider using virtualization techniques to improve performance.

Libraries like react-virtualized and react-window allow you to render only the visible items, reducing the DOM size and improving scrolling performance.

**6. Use Code Splitting and Lazy Loading:**

Leverage code splitting and lazy loading to load only the necessary code when it’s needed, reducing the initial bundle size and improving load times.

React provides the React.lazy function for dynamically loading components. Here's an example:

import React, { lazy, Suspense } from 'react';  
  
const LazyComponent = lazy(() => import('./LazyComponent'));  
  
function MyComponent() {  
 return (  
 <Suspense fallback={<div>Loading...</div>}>  
 <LazyComponent />  
 </Suspense>  
 );  
}

With code splitting and lazy loading, your app loads faster, improving the overall performance.

**7. Optimize Expensive Operations with Web Workers:**

For computationally intensive tasks that might block the main thread, consider offloading the work to Web Workers.

Web Workers run in the background, leaving the main thread available for user interactions.

This prevents the app from becoming unresponsive. Here’s a basic example:

// In your component  
const worker = new Worker('worker.js');  
worker.postMessage(data);  
  
worker.onmessage = (event) => {  
 const result = event.data;  
 // Process the result  
};  
  
// In worker.js  
self.onmessage = (event) => {  
 const data = event.data;  
 // Perform the expensive operation  
 self.postMessage(result);  
};

By using Web Workers, you can enhance performance by parallelizing heavy operations.

**8. Implement Memoization with Libraries:**

To optimize performance-intensive functions, consider using memoization libraries like memoize-one or reselect.

These libraries cache the result of expensive function calls, preventing unnecessary recalculations. Here's a simple example using memoize-one:

import memoize from 'memoize-one';  
  
const computeExpensiveValue = (a, b) => {  
 // Expensive computation here  
};  
  
const memoizedValue = memoize(computeExpensiveValue);  
  
// Usage  
const result = memoizedValue(a, b);

By memoizing expensive functions, you avoid redundant computations and improve overall performance.

**9. Optimize CSS and Render Performance:**

Minimize the use of inline styles and prefer external stylesheets for better caching and performance.

Also, be mindful of unnecessary re-renders caused by frequently changing styles.

Consider using CSS-in-JS libraries like styled-components or emotion that generate optimized CSS.

**10. Profile and Analyze Performance:**

Lastly, use performance profiling tools like React DevTools Profiler or Chrome DevTools Performance tab to identify performance bottlenecks.

Analyze render times, component lifecycles, and expensive operations to optimize your app’s performance.

**Conclusion:**

Congratulations, senior JS React developers! You’ve learned 10 expert performance tips to level up your React applications.

From using functional components with React Hooks to optimizing re-rendering, implementing code splitting, and leveraging Web Workers, these tips will help you build lightning-fast apps.

Remember to profile and analyze performance to fine-tune your optimizations.

Now go forth and create high-performing, responsive, and user-friendly React applications.

Happy coding!