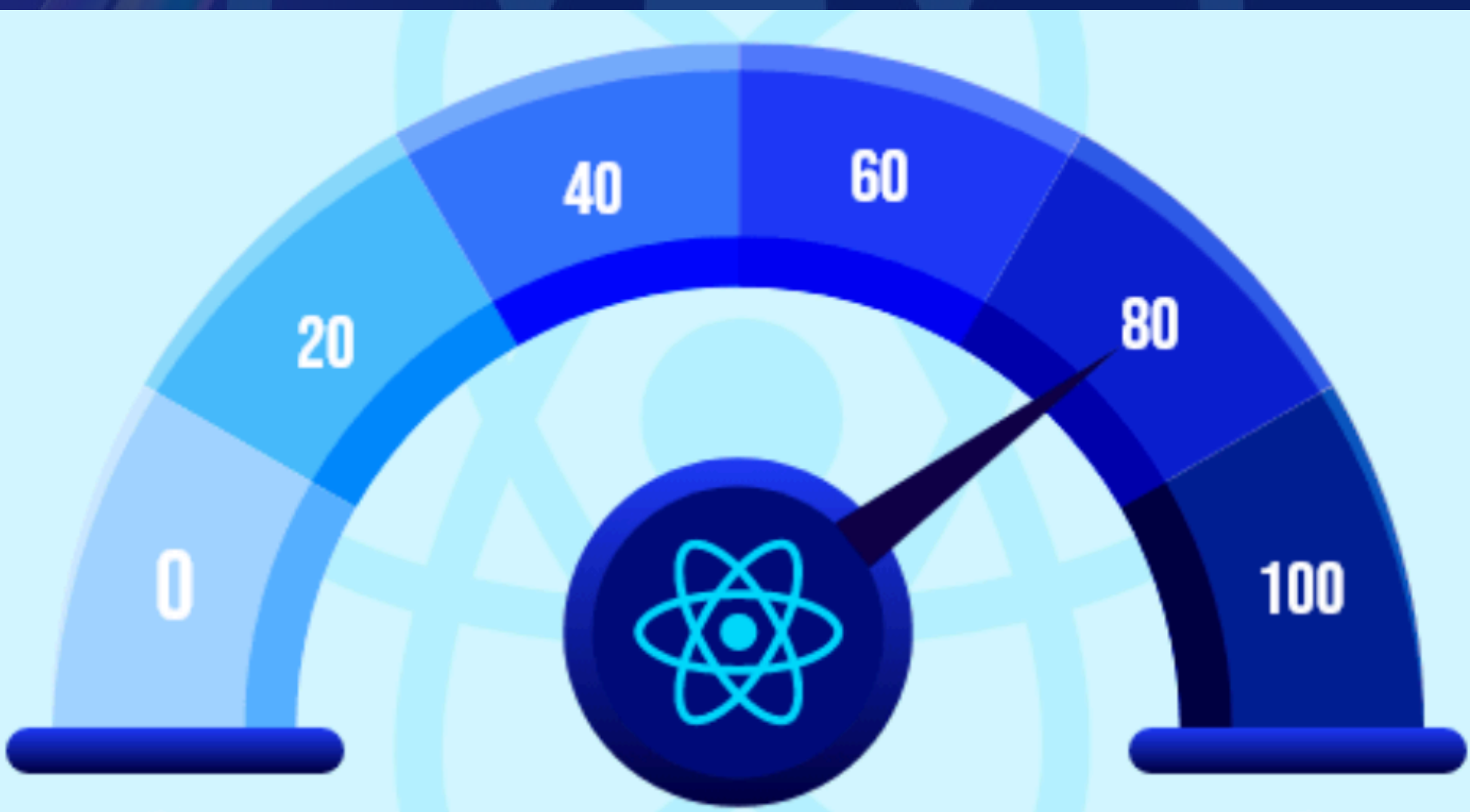
Day 16

React Perfomance Optimization Techniques (2)



@oluwakemi Oluwadahunsi

Efficient State Management

Efficient state management is crucial in React applications to ensure optimal performance, scalability, and maintainability.

Poorly managed state can lead to unnecessary re-renders, memory leaks, and an overall sluggish user experience.

Understanding React's State System

In React, state refers to the data that **drives** the behavior of a component. Each component can have its **local state**, and components can also receive state data from **parent** components as **props**.

@oluwakemi Oluwadahunsi

The two **key state hooks** in React are:

- **useState:** Manages local state in functional components.lculations from running on every render.

- **useReducer:** Useful for managing more complex state logic, especially when dealing with multiple related state variables.

State updates in React trigger **re-renders**, which is essential for maintaining dynamic user interfaces.

However, inefficient state management can cause **unnecessary re-renders**, affecting performance, particularly in large applications or complex UIs.

@oluwakemi Oluwadahunsi

Techniques for Efficient State Management

1. Local vs Global State Management

Local state refers to state that is scoped to a **particular** component and its children. It’s typically managed using React's **useState** or **useReducer** hooks. Here are the key points:

- **Where to Use It:** Local state is ideal when you only need to manage data that concerns a specific component or a small part of the UI.
- **Performance Benefits:** Using local state ensures that state updates only trigger re-renders of the component holding the state, preventing unnecessary re-renders of other parts of the application.

@oluwakemi Oluwadahunsi

For Example:

```jsx
function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

In this case, only the **Counter** component re-renders when the state changes.

@oluwakemi Oluwadahunsi

Global state refers to state that is a**ccessible across multiple components** in an application. You often manage global state with tools like **React Context**, **Redux**, **Zustand**, or even **custom hooks**.

- **Where to Use It:** Global state is useful when multiple components need access to shared data, like authentication status, user data, or theme preferences.

- **Potential Performance Issues:** If not managed properly, global state can lead to unnecessary re-renders of components that don’t directly use the updated state. This happens because updating global state often triggers re-renders for **all components** subscribed to that state, whether or not they actually need the update.

Example (Using Context for Global State):

@oluwakemi Oluwadahunsi

Let’s create a global theme Context logic that affects the entire application in our App.jsx

```jsx
const ThemeContext = React.createContext();

const App = () => {
  const [theme, setTheme] = useState('light');

  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      <Header />
      <Body />
    </ThemeContext.Provider>
  );
}

export default App;
```

exported global props

components that uses the exported props

Header.jsx component:

```jsx
const Header = () => {
  const { theme, setTheme } = useContext(ThemeContext);
  return (
    <div>
      <h1>Current Theme: {theme}</h1>
      <button onClick={() => setTheme(theme === 'light' ? 'dark' : 'light')}>
        Toggle Theme
      </button>
    </div>
  );
}
export default Header;
```

React hook for accessing the context states

exported props from context being used in header component

@oluwakemi Oluwadahunsi

In this example:

- In the App.jsx, a global context was sset for the component Header and Body. Any of these components can use the exported props from the context API.
- When there is an update in the Header component, for example, both Header and Body (if used) will re-render when the theme changes, even if Body doesn't need that state.

Tips To Optimize Performance when working with states:

- Minimize Global State: Not all state needs to be global. Keep as much state local as possible to limit the number of components that re-render when state changes.
- Memoization: Use memoization (e.g., useMemo, React.memo) to prevent unnecessary re-renders of components that do not depend on the changing global state, just like we learned yesterday.

@oluwakemi Oluwadahunsi

- Selector Functions: When using global state with tools like Redux, use selector functions to only subscribe to the part of the state that the component needs, minimizing the impact of global state changes.

- Split State: If possible, split large global state objects into smaller, independent pieces. This way, only components using the specific piece of state that has changed will re-render.

PS: Many terms may not make too much sense now because we haven't learned about state global state in React.

You can come back to review this part after we learn the global state management lesson.

@oluwakemi Oluwadahunsi

2. Using useReducer for Complex State Logic

While **useState** is typically used for simpler state management, React's **useReducer** hook provides a more powerful alternative for managing more complex state logic.

It is particularly useful for situations where state logic is complex, involving multiple state transitions, or when state updates depend on previous states.

How useReducer Optimizes Performance

- Minimized Re-Renders:

By dispatching actions to a **single reducer**, React re-renders only when necessary. With complex state logic, managing state updates with multiple **useState** calls can trigger multiple re-renders, which can degrade performance

useReducer reduces the number of re-renders by grouping related state updates under a single dispatch.

@oluwakemi Oluwadahunsi

- Improved Readability for Complex Logic:

When you have multiple states that depend on one another, **useReducer** helps keep your state management code clean and less error-prone.

This indirectly improves performance by reducing the chance of bugs and making your codebase easier to maintain.

- Memoized Dispatch Function:

The **dispatch** function provided by **useReducer** is guaranteed to have a stable identity, meaning it won't change between renders.

This ensures that components that use it as a dependency for effects or callbacks won't unnecessarily re-render.

Note: You can refer to the Day 5 of this series to learn more about how the usereducer hook works with its example.

@oluwakemi Oluwadahunsi

React Tools and Libraries for State Management

React has a number of external libraries for managing state more efficiently, especially in larger applications.

a. Redux

Redux is the most popular state management library in the React ecosystem. It helps manage global state in a predictable and scalable way by maintaining a single state tree (a single source of truth). It enforces immutability and state updates through pure functions (reducers), leading to more predictable performance.

Documentation: **https://redux.js.org/**

b. Zustand

Zustand is a lightweight state management library that provides a simple API for managing state outside of React's component tree. It's efficient because it avoids the need to rerender components unnecessarily.

Documentation: **https://zustand.docs.pmnd.rs/**

@oluwakemi Oluwadahunsi

c. Recoil

Recoil is another state management library for React that allows fine-grained control over state updates. It works with "atoms" (units of state) and "selectors" (derived state) to allow for more flexible and performant updates compared to Redux.

Documentation: https://recoiljs.org/

d. React Query

React Query is specialized for managing server-side state (data fetching, caching, and synchronization) but can be a great complement to other state management tools, especially in applications that rely heavily on external data.

Documentation: https://tanstack.com/query/v3

e. Jotai

Jotai is a minimalist state management library designed for React. It is atomic-based, meaning that each piece of state is treated as an atom, making it highly performant by preventing unnecessary re-renders in large component trees.

Documentation: https://jotai.org/docs/introduction

Code Splitting and Lazy Loading

Users expect fast, responsive web experiences, and this is where code splitting and lazy loading come in as performance optimization techniques in React.

What is Code Splitting?

Code splitting is the process of **breaking down** your application's code into **smaller**, more manageable chunks or bundles.

Instead of loading the entire application at once, only the necessary parts of the code are loaded, improving performance and load times.

When a React app grows larger, it results in a large JavaScript bundle. If this bundle is loaded all at once, it can slow down the initial load time.

Code splitting allows us to load **only the parts of the app needed at a given time**, reducing the initial bundle size.

@oluwakemi Oluwadahunsi

What is Lazy Loading?

Lazy loading is a technique where certain parts of your app are only loaded when they are actually needed, instead of loading everything upfront.

This works hand-in-hand with code splitting because the chunks created by code splitting can be loaded on demand.

For instance, if you have a component or route that a user may not visit immediately, you can lazy load that component to ensure it only loads when the user navigates to it.

Code Splitting & Lazy Loading in React

React, along with **Webpack**, supports code splitting out of the box. React provides a way to split your code and load it on-demand using the **React.lazy( )** function in combination with **Suspense**.

a. Webpack and Code Splitting

Webpack is the most commonly used module bundler for React apps, and it automatically splits the code into smaller bundles when you use **dynamic import()** statements or **React's lazy loading**.

b. React's React.lazy() and Suspense

React introduced the **React.lazy()** function to allow components to be loaded lazily, meaning they are only loaded when they are rendered for the first time.

@oluwakemi Oluwadahunsi

How Webpack Handles Code Splitting

Webpack can automatically split your code into separate chunks. When you use dynamic imports (like React.lazy), Webpack recognizes these imports and creates separate chunks for those components.

This means Webpack will load the necessary bundles when required, rather than bundling everything together upfront.

Webpack uses two main techniques for splitting code:

- **Dynamic Imports:** Webpack creates chunks based on **dynamic imports**, and these chunks are loaded **asynchronously**.
- **Splitting by Route or Feature:** You can split your code by routes, meaning different sections of your application are split into different bundles. For example, if you have an admin section or a dashboard that is not immediately needed, you can delay its loading until the user navigates to that page.

@oluwakemi Oluwadahunsi

Example of code splitting and lazy loading concepts using **dynamic imports**, **routes splitting**, **React.lazy()** and **Suspense**:

```
1  import { Suspense } from 'react';
2  import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
3
4  //Lazy loading route-based components
5  const HomePage = React.lazy(() => import('./HomePage'));
6  const AboutPage = React.lazy(() => import('./AboutPage'));
7
8  function App() {
9    return (
10     <Router>
11     {/* Suspense is used to show a fallback while the component is loading */}
12       <Suspense fallback={<div>Loading ... </div>}>
13         <Switch>
14           <Route exact path="/" component={HomePage} />
15           <Route path="/about" component={AboutPage} />
16         </Switch>
17       </Suspense>
18     </Router>
19   );
20 }
21
22 export default App;
```

components are loaded only when needed

Note

@oluwakemi Oluwadahunsi

In the example above:

- **React.lazy():** This function allows you to **dynamically import** a component only when it's needed, deferring the loading of the component. This ensures that the initial bundle size is smaller, and additional components are loaded on-demand.

- **Suspense:** While the component is being loaded asynchronously, React will show a fallback UI (like a loading spinner) until the component has fully loaded. This improves the user experience by indicating that something is happening in the background.

Best Practices for Code splitting & Lazy loading

- **Lazy Load Non-Essential Features:** Focus on lazy-loading only non-essential components (e.g., settings pages or dashboards) while keeping core functionality readily available.

- **Optimize Fallbacks:** Ensure that your Suspense fallback components are user-friendly (e.g., show spinners or skeleton loaders).

- **Bundle Analysis:** Use tools like Webpack's bundle analyzer to ensure your code is split efficiently. You can use it to check how much each component contributes to the final bundle size.

- **Prefetching:** Use dynamic imports with **webpackPrefetch: true** to prefetch important chunks before they are needed.
  Syntax:

```
1 const LazyComponent = React.lazy(() ⇒
2   import(/* webpackPrefetch: true */ './LazyComponent')
3 );
```

@oluwakemi Oluwadahunsi

2. Font Optimization

Using web fonts can be resource-heavy if not handled correctly. Here are some tips for optimizing fonts:

- **Use modern formats like WOFF2:** It offers better compression than older formats like TTF.
- **Preload critical fonts:** Ensure that important fonts are loaded quickly using the rel="preload" attribute.
- **Use system fonts:** For applications where performance is crucial, using system fonts (Arial, Helvetica, etc.) can reduce the time required to download and render custom fonts.

Stay Tuned as we continue the Third part of React Performance Optimization Tomorrow.

@oluwakemi Oluwadahunsi

I hope you found this material useful and helpful.

Remember to:

Like

Save for future reference

&

Share with your network, be helpful to someone 👌

@oluwakemi Oluwadahunsi

Hi There!

Thank you for reading through

Did you enjoy this knowledge?

💼 Follow my LinkedIn page for more work-life balancing and Coding tips.

🌐 LinkedIn: Oluwakemi Oluwadahunsi

KodeMaven

kodemaven-portfolio.vercel.app