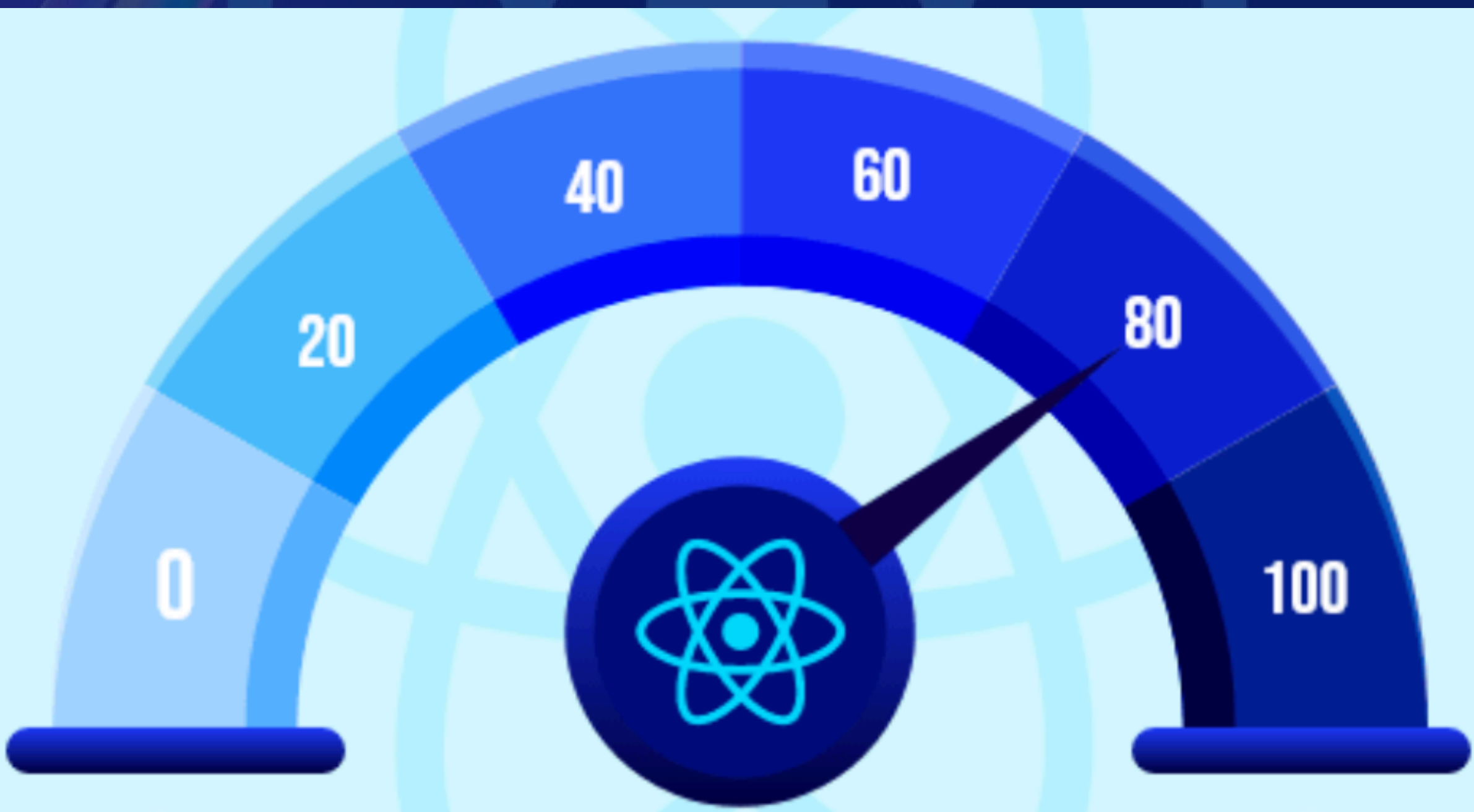
Day 17

React Perfomance Optimization Techniques (3)



@oluwakemi Oluwadahunsi

Optimizing Lists with Keys and Proper Rendering

In React, rendering lists is a common operation when you're working with **dynamic data (refer to Day 11 of this series to learn how lists and keys work in React)**. While this might seem straightforward, there are important performance considerations you must take into account to ensure your React app runs efficiently.

One of the key techniques for optimizing list rendering in React is the proper use of **keys**.

Why are Keys Important?

Whenever you render a list of elements, React needs a way to keep track of which items have **changed**, **been added**, or **removed** from the list. This is where **keys** come into play.

@oluwakemi Oluwadahunsi

If you don’t provide a **key** for each item in a list, React will log a warning message in the console. This isn't just a minor warning; it indicates that your list rendering may be inefficient.

In cases where there are complex lists or data updates, performance can suffer due to unnecessary re-renders.

For example:

```jsx
const ItemList = ({ items }) => {
  return (
    <ul>
      {items.map(item => (
       //Missing key prop
        <li>{item.name}</li>
      ))}
    </ul>
  );
}

export default ItemList;
```

n this example, React can’t efficiently track which li elements are **added**, **updated**, or **removed** because no keys are provided.

@oluwakemi Oluwadahunsi

To fix the issue, you need to provide a unique key for each item in the list. React recommends using a unique identifier from your data (such as an ID) as the key for each list item.

If you don't have a unique ID, avoid using indices (e.g index) unless absolutely necessary, because they may lead to performance bugs when the list order changes dynamically.

Do not use ‘index’ as an identifier

```
1 {items.map((item, index) ⇒ (
2    <li key={index}>{item.name}</li>
3 ))}
```

If the list order changes (e.g., an item is deleted), the keys will no longer match the original items, which could result in React reusing incorrect DOM elements, leading to performance and rendering bugs.

@oluwakemi Oluwadahunsi

Correct Example with Keys:

```jsx
import React from 'react';

const ItemList = ({ items }) => {
  return (
    <ul>
      {items.map(item => (
        <li key={item.id}>{item.name}</li> // Proper key usage
      ))}
    </ul>
  );
}

export default ItemList;
```

unique identifier provided

In this example, the **key** ensures that React can identify and track each **li** by the **item.id**.

This helps React optimize rendering when there are changes in the list. React knows exactly which items need to be added, updated, or removed based on the key.

@oluwakemi Oluwadahunsi

Rendering a List of Users with Keys:

Let’s say we have a data file containing a list of users called users.js, like this:

```
export const userData = [
    {
      id: 1,
      name: 'John'
    },
    {
      id: 2,
      name: 'Jane'
    },
    {
      id: 3,
      name: 'Smith'
    },
  ];
```

Now create a UserList component: UserList.jsx

```jsx
import { useState } from 'react';
import userData from './users' // import userData from data file

const UserList = () => {
  const [users, setUsers] = useState(userData);

  // Function to add a new user to the list
  const addUser = () => {
    const newUser = { id: users.length + 1, name: `User ${users.length + 1}` };
    setUsers([...users, newUser]);
  };

  return (
    <div>
      <ul>
        {users.map(user => (
          <li key={user.id}>{user.name}</li> // Correct key usage
        ))}
      </ul>
      <button onClick={addUser}>Add User</button>
    </div>
  );
}

export default UserList;
```

unique id passed for the new user to be added

on clicking the button, a new user is added to the userData, taking up the next id “4”

In this example:

- We have a list of users with unique **IDs**.
- Each **li** element has a **key** corresponding to the user’s id.
- When we add a new user, React knows exactly which list items need to be re-rendered based on the keys.

Throttling and Debouncing Input Events

Handling input events like **scroll**, **resize**, or **text input** can be performance-heavy if you update the state or trigger **expensive operations** on every event.

Throttling and **debouncing** are techniques that can help limit how often event handlers are called, leading to smoother interactions and better performance.

Throttling

Throttling is a technique that ensures a function is called **at most once** in a specified time period, even if the event triggers multiple times. It limits the rate at which a function can be executed.

Think of throttling as a way of regulating the execution of an expensive function so that it's only triggered at regular intervals.

@oluwakemi Oluwadahunsi

Let’s say you have a component that listens to the **scroll event** and fetches more data as the user scrolls down the page.

Without throttling, the event listener would fire hundreds of times per second, degrading performance.

Here is an example of how throttling saves a situation like this:

```jsx
import { useEffect } from 'react';

// Throttle utility function
function throttle(func, delay) {
  let lastCall = 0;
  return (...args) => {
    const now = new Date().getTime();
    if (now - lastCall < delay) return;
    lastCall = now;
    return func(...args);
  };
}

const ThrottleScrollComponent = () => {
  const handleScroll = () => {
    console.log('User is scrolling...');
    // Logic for handling scroll event
  };

  useEffect(() => {
    const throttledScroll = throttle(handleScroll, 200); // Throttle the function to run every 200ms
    window.addEventListener('scroll', throttledScroll);

    return () => {
      window.removeEventListener('scroll', throttledScroll); // Cleanup event listener on unmount
    };
  }, []);

  return (
    <div style={{ height: '2000px' }}>
      <h1>Scroll Down</h1>
    </div>
  );
};

export default ThrottleScrollComponent;
```

@oluwakemi Oluwadahunsi

In this example:

- **Throttle function:** The utility function **throttle** ensures that the **handleScroll** function is only called every **200 milliseconds**, no matter how frequently the scroll event is triggered.
- **UseEffect Hook:** The throttled scroll event listener is added when the component mounts and is **cleaned up** when the component unmounts.

Debouncing

Debouncing is a technique that ensures a function is called **only after a certain amount of time has passed** since the last time the event was triggered.

Unlike throttling, which runs at regular intervals, **debouncing delays the execution** of the function until after the user has stopped triggering the event for a specified amount of time.

Use Cases for Debouncing in React:

- **Form input validation:** You may want to debounce a function that validates a form field while the user is typing, ensuring that validation occurs only after the user has stopped typing for, say, 500 milliseconds.
- **Search input:** Debouncing can be used to limit the number of API calls made as the user types into a search bar, ensuring that the search function is triggered only after the user pauses typing.

Here’s an example of a simple search input field where we debounce the input value to delay API calls until the user stops typing:

```jsx
import React, { useState, useEffect } from 'react';

// Debounce utility function
function debounce(func, delay) {
  let timeout;
  return (...args) => {
    clearTimeout(timeout);
    timeout = setTimeout(() => {
      func(...args);
    }, delay);
  };
}

const DebounceSearchComponent = () => {
  const [searchTerm, setSearchTerm] = useState('');

  const handleSearch = (value) => {
    console.log('Fetching results for:', value);
    // Logic for fetching search results based on searchTerm
  };

  const debouncedSearch = debounce(handleSearch, 500);

  const handleInputChange = (e) => {
    const value = e.target.value;
    setSearchTerm(value);
    debouncedSearch(value); // Call the debounced function
  };

  return (
    <div>
      <h2>Debounced Search Input</h2>
      <input
        type="text"
        value={searchTerm}
        onChange={handleInputChange}
        placeholder="Search here..."
      />
    </div>
  );
};

export default DebounceSearchComponent;
```

In this example:

- **Debounce function:** The **debounce** utility function ensures that the **handleSearch** function is only called **500 milliseconds** after the user has stopped typing.
- **Debounced Search:** The **debouncedSearch** function is called within **handleInputChange**, so the actual search only occurs when the user stops typing for half a second.

Key Differences Between Throttling and Debouncing:

The benefits of implementing virtualization in React include:

1. When does it trigger?

- **Throttling:** At regular intervals during the event firing.
- **Debouncing:** Only after the event has stopped triggering.

2. Use Case:

- **Throttling:** When you need to limit the rate of execution (e.g., scroll or resize events).
- **Debouncing:** When you want to execute a function after the event stops (e.g., search input).

@oluwakemi Oluwadahunsi

3. Effect on Performance:

- **Throttling:** Controls frequent executions, enhancing performance by spacing out function calls.
- **Debouncing:** Reduces unnecessary function executions, improving performance by waiting for user action to complete.

While we can write our own custom debouncing and throttling functions, libraries like **Lodash** provide easy-to-use, well-tested utility functions.

Lodash offers built-in debounce and throttle methods, which can simplify your code and ensure reliability.

Documentation:

Avoiding Inline Functions and Objects

One of the subtle but significant contributors to performance issues from unnecessary re-renders is the use of **inline functions** and **inline objects** in component renders.

While they might seem harmless, inline functions and objects can create **new instances** on every render, causing React to treat them as **new references**, which leads to unnecessary re-renders of components.

Why this is to be avoided as much as possible?

When React encounters a function or object that is defined inline within a render method, it creates a **new instance** of that function or object on every render.

Even though the logic might be the same, React sees it as a **new entity**, causing child components that rely on this function or object to re-render.

@oluwakemi Oluwadahunsi

This happens because functions and objects in JavaScript are compared by **reference** rather than by **value**. Thus, each new instance is considered different from the previous one, even if the contents or logic are identical.

For example:

```jsx
import React, { useState } from 'react';

function ParentComponent() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Increment Count</button>
      <ChildComponent handleClick={() => console.log('Child clicked!')} />
    </div>
  );
}

function ChildComponent({ handleClick }) {
  console.log('ChildComponent rendered');
  return <button onClick={handleClick}>Click me</button>;
}

export default ParentComponent;
```

In this example:

- **ChildComponent** accepts a **handleClick** function prop.

- Every time **ParentComponent** re-renders (when the button is clicked), the inline **function () => console.log('Child clicked!')** is re-created, leading to **ChildComponent** being re-rendered unnecessarily even though nothing has changed for the child.

One way to avoid this problem is by moving the function definition **outside the render method** so that it doesn't get re-created on every render.

Another approach is to use **useCallback hook** to **memoize** (**Check part 1 on Day 15**) the function.

Here’s how we can optimize our previous example:

Option 1: Define Function Outside the Render

```jsx
import { useState } from 'react';

function ParentComponent() {
  const [count, setCount] = useState(0);

  // Define the function outside the return/render
  const handleClick = () => {
    console.log('Child clicked!');
  };

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Increment Count</button>
      <ChildComponent handleClick={handleClick} />
    </div>
  );
}

function ChildComponent({ handleClick }) {
  console.log('ChildComponent rendered');
  return <button onClick={handleClick}>Click me</button>;
}

export default ParentComponent;
```

In this optimized version:

- The **handleClick** function is now declared **outside** the render method, so it doesn’t get re-created on every render.
- **ChildComponent** will only re-render if the **handleClick** function or other props change.

@oluwakemi Oluwadahunsi

Option 2: Use useCallback Hook

```jsx
import { useState, useCallback } from 'react';

function ParentComponent() {
  const [count, setCount] = useState(0);

  // Memoize the function using useCallback
  const handleClick = useCallback(() => {
    console.log('Child clicked!');
  }, []);

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Increment Count</button>
      <ChildComponent handleClick={handleClick} />
    </div>
  );
}

function ChildComponent({ handleClick }) {
  console.log('ChildComponent rendered');
  return <button onClick={handleClick}>Click me</button>;
}

export default ParentComponent;
```

- **useCallback** ensures that the **handleClick** function is only re-created when its dependencies (empty in this case) change.
- This prevents unnecessary re-renders of **ChildComponent** since React will treat **handleClick** as the same function between renders.

@oluwakemi Oluwadahunsi

The same issue occurs when **objects** are defined inline within the render method. Since objects in JavaScript are compared by **reference**, an inline object will be treated as a **new object** on each render, even if the values are the same.

Example of Inline Object Problem:

```jsx
import React, { useState } from 'react';

function ParentComponent() {
  const [isActive, setIsActive] = useState(true);

  return (
    <div>
      <button onClick={() => setIsActive(!isActive)}>Toggle Active</button>
      <ChildComponent style={{ color: isActive ? 'green' : 'red' }} />
    </div>
  );
}

function ChildComponent({ style }) {
  console.log('ChildComponent rendered');
  return <p style={style}>This is a child component!</p>;
}

export default ParentComponent;
```

object declared inline

In this example:

- The **style** object is created **inline** inside the render method.

- Every time **ParentComponent** re-renders, the **style** object is treated as a **new object**, causing **ChildComponent** to re-render even when the style has not changed.

To optimize this, you can define the style object **outside of the render method** or **memoize** it using the useMemo hook just like we did for inline functions.

Option 1: Define Object Outside the Render

```jsx
import { useState } from 'react';

function ParentComponent() {
  const [isActive, setIsActive] = useState(true);

  const activeStyle = { color: 'green' };
  const inactiveStyle = { color: 'red' };

  return (
    <div>
      <button onClick={() => setIsActive(!isActive)}>Toggle Active</button>
      <ChildComponent style={isActive ? activeStyle : inactiveStyle} />
    </div>
  );
}

function ChildComponent({ style }) {
  console.log('ChildComponent rendered');
  return <p style={style}>This is a child component!</p>;
}

export default ParentComponent;
```

object declared outside of the render method

Option 2: Use useMemo Hook

Alternatively, you can use the **useMemo** hook to **memoize** the object.

@oluwakemi Oluwadahunsi

```jsx
import React, { useState, useMemo } from 'react';

function ParentComponent() {
  const [isActive, setIsActive] = useState(true);

  const style = useMemo(() => ({ color: isActive ? 'green' : 'red' }), [isActive]);

  return (
    <div>
      <button onClick={() => setIsActive(!isActive)}>Toggle Active</button>
      <ChildComponent style={style} />
    </div>
  );
}

function ChildComponent({ style }) {
  console.log('ChildComponent rendered');
  return <p style={style}>This is a child component!</p>;
}

export default ParentComponent;
```

In this version:

- The **style** object is memoized with **useMemo**, so it is only re-created when isActive changes.
- This prevents unnecessary re-renders of **ChildComponent**.

Stay Tuned as we continue the second 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-portfolio.vercel.app