React Optimization Techniques

Sarim Iqbal

Techniques:

1. Lazy Loading
2. List Virtualization
3. Memoization
4. Throttling and Debouncing
5. Use Transition Hook

Lazy Loading:

What is it? Load components or resources only when needed, instead of all at once.

Why it's needed:

- Improve initial load time
- Reduce unnecessary downloads

Real-life example: Think of a long e-commerce page. Instead of loading all product pages upfront, only load pages when the user visits them.

- Reduces app size at startup
- Optimizes performance for large apps
- Especially useful for routes or images

Code Example:

```
import React, { lazy, Suspense } from "react";
import { BrowserRouter, Link,
Route, Routes } from "react-router-dom";

const Admin = lazy(() => import("./Admin"));

const LazyLoadingExample = () => {
return (
<BrowserRouter>
    <h1>Home Page</h1>
    <Link to={"/admin"}>Admin</Link>
    <Routes>
    <Route
      path="/admin"
      element={
        <Suspense>
         <Admin />
        </Suspense>
       }
       /   >
    </Routes>
 </BrowserRouter>
);
};

export default LazyLoadingExample;
```

```
// Admin.js

import React from "react";
const Admin = () => {
return (
   <div>
    <h1>Admin Page</h1>
    <p>Lorem ipsum*5000000....
    </p>
   </div>
   )
}

export default Admin;
```

04

List Virtualization:

What is it?Optimize the rendering of large lists by only displaying the items that are currently visible on the screen.

Why it's needed:

- Avoid slow performance on lists with thousands of items
- Only render visible items

Real-life example: Imagine a social media feed showing thousands of posts. With list optimizations, only visible posts are rendered, improving scroll performance.

- Prevents lag from large lists
- Only renders visible items
- Reduces memory and CPU usage

05

Code Example:

```
import React from "react";
import { List } from "react-virtualized";
import "react-virtualized/styles.css";

// Your list data
const list = Array(20000)
.fill()
.map((_, index) => ({
   id: index,
 name: `Item ${index}`,
}));

// Function to render each row
function rowRenderer({ index, key, style }) {
return (
   <div key={key} style={style}>
     {list[index].name}
   </div>
  );
}
```

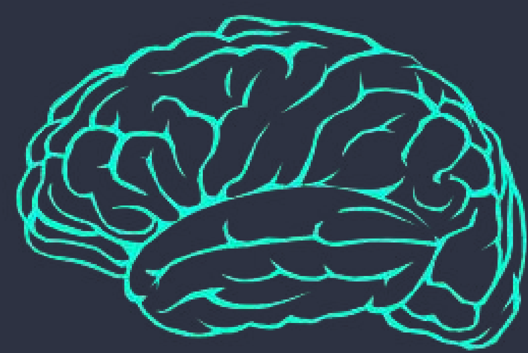
```
// Main component
function ListOptimization() {
return (
// Fast and efficient way
   <List
     width={300}
     height={300}
     rowCount={list.length}
     rowHeight={20}
     rowRenderer={rowRenderer}
   />

 // Normal Way (take time to load)
   // <>
   // <ul>
    //    {list.map((li) => (
//          <li>{li.name}</li>
   //    ))}
   // </ul>
   // </>
 );
}

export default ListOptimization;
```

Memoization:

What is it? Memoizes a computed value, recalculating only when dependencies change.



Why it's needed:

- Avoids unnecessary recalculations
- Boosts performance in components with expensive calculations

Real-life example: In a dashboard showing analytics, useMemo can cache expensive calculations like data summaries.

- Avoids re-rendering large components
- Reduces computation time
- Useful for data-heavy components

Code Example:

```
import React, { useMemo } from "react";

const MemoizedExample = () => {
const [count, setCount] = React.useState(0);
const [otherState, setOtherState] = React.useState("");

 const expensiveComputation = (num) => {
   let i = 0;
 while (i < 1000000000) i++;
return num * num;
};

 const memoizedValue = useMemo(() => expensiveComputation(count), [count]); // With Memo (Fast)
// const memoizedValue = expensiveComputation(count); // Without Memo (Time taking)

 return (
   <div>
     <p>Count: {count}</p>
     <p>Square: {memoizedValue}</p>
     <button onClick={() => setCount(count + 1)}>Increase Count</button>
     <input
       type="text"
       onChange={(e) => setOtherState(e.target.value)}
       placeholder="Type something to check rerendering"
       style={{ width: "100%" }}
     />
   </div>
 );
};

export default MemoizedExample;
```

Throttling & Debouncing:

08

What is it? Limits the rate at which a function is invoked (throttling) or ensures a function runs only after it hasn’t been called for a specified time (debouncing).

Why it's needed:

- Improves performance for frequent user actions
- Avoids overloading the browser with events

Real-life example: Imagine resizing a window or typing in a search bar. Throttling ensures the resize event doesn't fire continuously. Debouncing waits for the user to stop typing before making a search request.

- Reduces the number of API calls
- Prevents performance degradation
- Ideal for scroll, resize, or input events

Code Example:

09

```
// Throttling.js
import React, { useState, useEffect } from "react";


const Throttling = () => {
const [scrollPosition, setScrollPosition] = useState(0);

 const handleScroll = () => {
     setScrollPosition(window.scrollY);
 }; // Throttled to run once every 1 second

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

 useEffect(() => {
   window.addEventListener("scroll", throttle(handleScroll, 1000));
   return () => {
     window.removeEventListener("scroll", throttle(handleScroll, 1000));
   };
 }, []);

 return (
   <div>
     <h1>Scroll position: {scrollPosition}</h1>
     <div style={{ height: "200vh" }}>
       Scroll down to see throttling in action!
     </div>
   </div>
 );
};

export default Throttling;
```

Code Example:

101

```
// Debouncing.js
import React, { useState, useCallback } from "react";
import { debounce } from "lodash";

const Debouncing = () => {
const [searchTerm, setSearchTerm] = useState("");
const [displayTerm, setDisplayTerm] = useState("");

 const handleSearch = useCallback(
debounce((term) => {
    setDisplayTerm(term);
 }, 500), // Debounced to execute 500ms after the user stops typing
   []
 );

 const handleChange = (e) => {
const term = e.target.value;
setSearchTerm(term);
handleSearch(term);
};

 return (
   <div>
     <input
       type="text"
       value={searchTerm}
       onChange={handleChange}
       placeholder="Type to search"
     />
     <p>Search results for: {displayTerm}</p>
   </div>
 );
};

export default Debouncing;
```

useTransition()

What is it? Delays non-urgent updates to maintain smooth UI transitions.

Why it's needed:

- Prevents janky UI when multiple state updates occur
- Useful in apps with heavy state changes

Real-life example: Imagine typing in a search box that filters a list. With useTransition, the input remains responsive while filtering happens in the background.

- Keeps the UI responsive
- Prioritizes important state updates
- Ideal for input-heavy forms or searches

Code Example:


```
const UseTransition = () => {
const [search, setSearch] = useState("");
const [filteredUsers, setFilteredUsers] = useState(users);
const [isPending, startTransition] = useTransition();


 const handleChange = (e) => {
   setSearch(e.target.value);
   startTransition(() =>
     setFilteredUsers(users.filter((user) => user.includes(e.target.value)))
   );
 };
return (
   <>
     <label>Input: </label>
     <input onChange={handleChange} />
     {isPending && <p>Loading...</p>}
     {!isPending && filteredUsers.map((e) => <p>{e}</p>)}
   </>
 );
};

export default UseTransition;
```