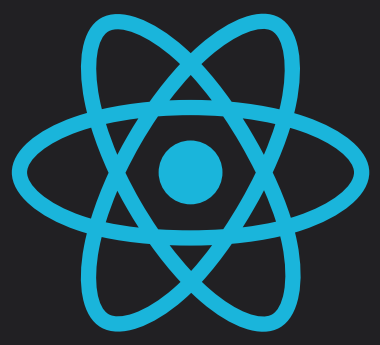

React Performance

Optimization

1. Minimizing Re-renders with React.memo()

- React.memo() is a higher-order component that memoizes the result of a component's rendering.

- It skips the unnecessary re-renders if the component's props remain the same.

```jsx
import React from "react";

const MyComponent = React.memo((props) => {
  // Component logic here
  console.log('Rendering MyComponent');

  return (
    <div>
      {/* Render component content using props */}
    </div>
  );
});

export default MyComponent;
```

- MyComponent will only re-render if its props change. This can help optimize performance our application.

2. Lazy Loading

- Large bundle sizes can increase the initial loading time of your application.

- With React.lazy, you can lazily load components to improve initial load times, which means the components are loaded only when they are needed.

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

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

function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <LazyComponent />
    </Suspense>
  );
}
```

- The Suspense component allows you to provide a fallback UI while the component is being loaded.

3. React Virtualized

- React Virtualized is a library for efficiently rendering large lists.
- It uses a windowing technique to render only the items that are currently visible on the screen.

```jsx
import React from 'react';
import { List } from 'react-virtualized';

function VirtualizedList({ items }) {
  return (
    <List
      width={300}
      height={400}
      rowCount={items.length}
      rowHeight={50}
      rowRenderer={({ index, key, style }) =>
        <div key={key} style={style}>
          {items[index]}
        </div>
      )}
    />
  );
}
```

- This reduces the memory usage and improves performance.

4. Memoize Costly Computations with useMemo

- The useMemo hook lets you cache the result of a calculation between re-renders.

- It's used to optimize performance by avoiding unnecessary re-computation of expensive operations.

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

function ExpensiveComponent({ a, b }) {
  const result = useMemo(() => {
    // Expensive computation
    return a * b;
  }, [a, b]);

  return <div>Result: {result}</div>;
}
```

- This example demonstrates how to optimize performance by avoiding unnecessary re-computation.

5. Code Splitting

- Code splitting is a technique that allows you to split your React application into smaller chunks, which are loaded on-demand.

- It helps reduce the initial bundle size and improves the loading performance of your application.

.

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

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

function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <LazyComponent />
    </Suspense>
  );
}
```

- With React.lazy(), you can dynamically import components, and Suspense provides a fallback UI while the component is being loaded.

6. Debouncing

- Debouncing is a technique used to delay the execution of a function until after a certain amount of time has passed since the last invocation.
- It is commonly used for handling expensive operations triggered by user events, such as input changes or search requests.
- Let's take an example of a search input field. When a user types in the search box, an event is triggered for every keystroke.
- Without debouncing, this can lead to excessive API calls or unnecessary processing.
- By debouncing the event handler, we can ensure that the search function is called only after the user has finished typing or paused for a specified duration.

Debouncing Example

- we only wanna make the API calls to the server when the user finishes typing their word and not on every input change.

```
// Define a debounce function
function debounce(func, delay) {
  let timeoutId;

  return function() {
    clearTimeout(timeoutId);
    timeoutId = setTimeout(func, delay);
  }
}

// Get the search input element by its id
const searchInput = document.getElementById('search-input');

// Define a function to handle the search operation
function handleSearch() {
  // Perform search operation
  console.log("Searching...");
}

// Apply debounce to the handleSearch function with a delay of 300 milliseconds
const debouncedSearch = debounce(handleSearch, 300);

// Add an event listener to the search input field for the input event
searchInput.addEventListener('input', debouncedSearch);
```

7. Content Delivery Network (CDN)

- A CDN is a distributed network of servers located in different geographical locations.

- It serves as an intermediary between your web application and its users, helping to optimize content delivery and improve performance.

- Reduced latency: With a CDN, content is served from servers located closer to the user's geographical location.

- Improved scalability: CDNs are designed to handle high traffic volumes and distribute content across multiple servers.

- Bandwidth offloading: By offloading the delivery of static assets, such as images, CSS files, and JavaScript files, to a CDN, you can reduce the bandwidth usage on your web servers. This frees up server resources.

8. Server-Side Rendering (SSR)

- SSR is a technique where the initial rendering of a React application is performed on the server, and the resulting HTML is sent to the client.
- It can improve the performance of your application by sending a pre-rendered HTML page to the client, which can be displayed quickly while the JavaScript bundle is being loaded and executed.
- Improved Performance: SSR can reduce the time-to-content for users, as they receive a fully-rendered page from the server
- SEO Optimization: Search engines can crawl and index the content of SSR pages more easily, as they receive the complete HTML content upfront.
- Additionally, SSR may not be suitable for all types of applications or use cases, particularly those that require highly dynamic or interactive interfaces.