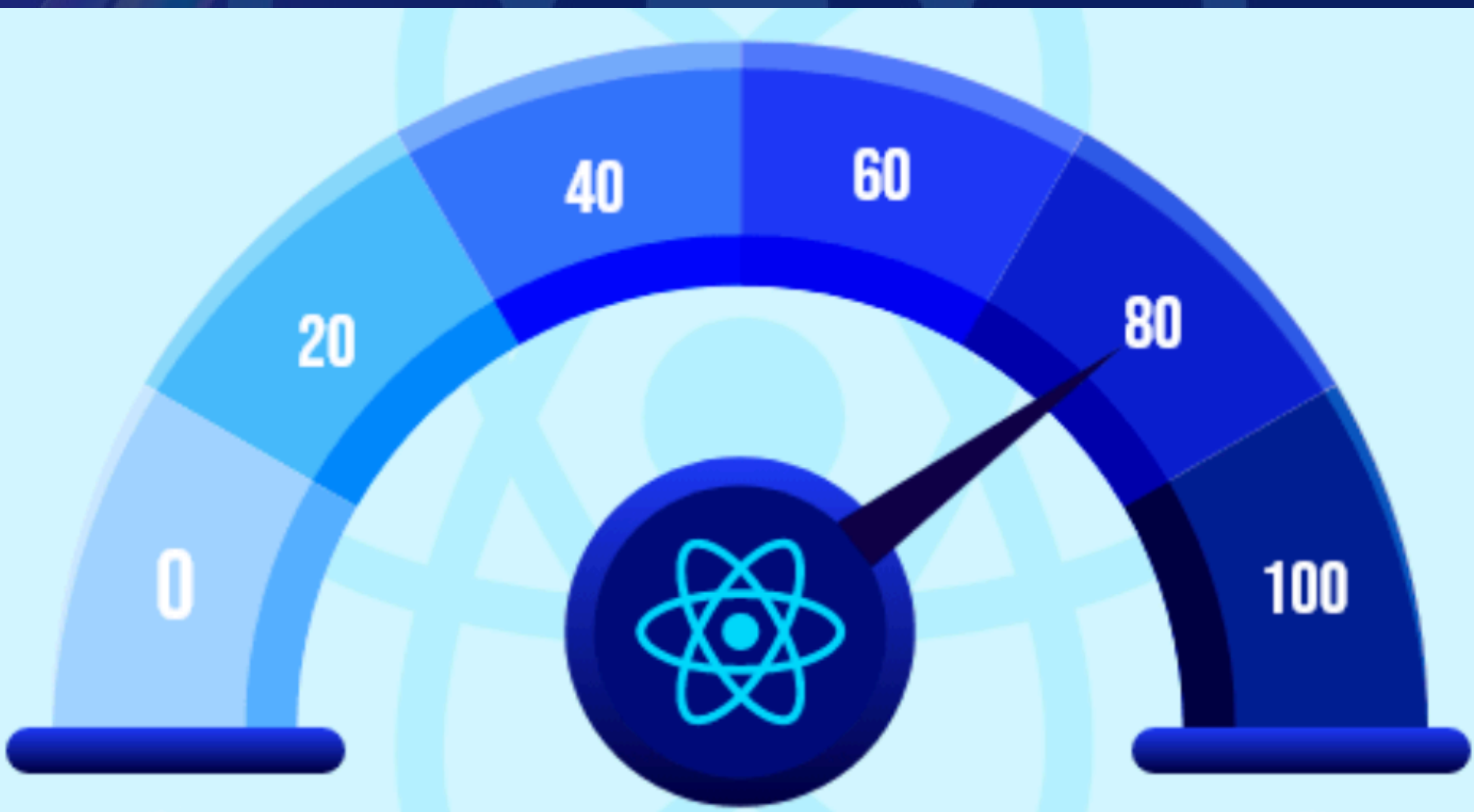
Day 15

React Perfomance Optimization Techniques (1)



@oluwakemi Oluwadahunsi

Introduction

React is renowned for its efficiency and flexibility in building dynamic user interfaces. However, as applications grow in **complexity** and **size**, ensuring optimal performance becomes important.

Poor performance can lead to **sluggish user experiences**, **increased load times**, and **higher bounce rates**.

This guide delves into various React performance optimization techniques, explaining each method thoroughly to help you build fast, efficient, and scalable React applications.

@oluwakemi Oluwadahunsi

Understanding React's Rendering Process

Before diving into optimization techniques, it's essential to understand how **React renders components**:

- **Virtual DOM:** React maintains a virtual representation of the real DOM. When a component's state or props change, React updates the virtual DOM first.
- **Reconciliation:** React compares the new virtual DOM with the previous one to identify changes.
- **Batch Updates:** React batches multiple updates to minimize direct DOM manipulations, enhancing performance.
- **Commit Phase:** React applies the necessary changes to the real DOM based on the differences detected.

@oluwakemi Oluwadahunsi

Techniques for Performance Optimization In React

There are sevral techniques involved in optimizing performance in React, which are:

- Function and Component Memoization
- Implementing Virtualization for Large Lists
- Optimizing Images and Assets
- Efficient State Management
- Optimizing Lists with Keys and Proper Rendering
- Code Splitting and Lazy Loading
- Throttling and Debouncing Input Events
- Avoiding Inline Functions and Objects
- Concurrent Mode and Suspense

@oluwakemi Oluwadahunsi

Function and Component Memoization

Memoization is a technique in programming that involves **caching** the results of **expensive** function calls and returning the cached result when the same inputs occur again.

In React, memoization allows you to cache component results or values, ensuring that the component **only re-renders** or **re-calculates** when its **dependencies change**.

Memoization is particularly useful in React for:

- Avoiding **unnecessary** re-renders of functional components.
- **Preventing expensive** calculations from running on every render.
- **Caching** event handlers and callback functions to avoid creating new references on every render.

@oluwakemi Oluwadahunsi

Memoization Techniques in React

React provides three primary hooks or functions for memoization:

- **React.memo( )**
- **useMemo( )**
- **useCallback( )**

1. React.memo( )

React.memo() is used to **memoize entire functional components**.

It works by **wrapping** the component and ensuring that it **only re-renders** when its **props** change.

If the **props** remain the same between renders, React uses the **memoized (cached)** version of the component instead of re-rendering it.

@oluwakemi Oluwadahunsi

For example: Memoizing a Child Component

```jsx
// Child component that only re-renders when 'count' prop changes
const ChildComponent = React.memo(({ count }) => {
  console.log("Child component re-rendered");
  return <div>Count: {count}</div>;
});

const ParentComponent = () => {
  const [count, setCount] = React.useState(0);
  const [text, setText] = React.useState("");

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Increment Count</button>
      <input
        type="text"
        value={text}
        onChange={(e) => setText(e.target.value)}
        placeholder="Type something"
      />
      <ChildComponent count={count} />
    </div>
  );
};

export default ParentComponent;
```

@oluwakemi Oluwadahunsi

In this example,

- **Without React.memo( ):** The **ChildComponent** would re-render every time the **ParentComponent** re-renders, even if only the text state changes.

- **With React.memo():** The ChildComponent only re-renders when its **prop (count)** changes, preventing unnecessary re-renders when text changes.

Note: By default, **React.memo()** performs a **shallow** comparison of props. For deeper comparisons, a custom comparison function can be provided as the second argument like this:

second arguement

```
React.memo(Component, (prevProps, nextProps) ⇒ {
  // Custom comparison logic
});
```

When to Use React.memo:

- **Pure Functional Components:** Components that render the same output given the same props.
- **Performance-Critical Components:** Components that are expensive to render or have frequent re-renders.

Benefits:

- **Reduced Re-Renders:** Prevents unnecessary updates, saving computational resources.
- **Improved Performance:** Enhances the overall efficiency of the application, especially with large component trees.

Considerations:

- **Shallow Comparison:** React.memo performs a shallow comparison of props. For deep prop comparisons, consider using a custom comparison function.
- **Overuse:** Using React.memo indiscriminately can lead to complexity without significant performance gains.

@oluwakemi Oluwadahunsi

2. useMemo( )

useMemo( ) is a **hook** used to memoize values that result from **expensive calculations**. It caches the computed result and only recomputes it when **one** of its dependencies changes.

In React, an **expensive function** or **component** refers to one that requires **significant computation** or resources, leading to slow performance, especially if it's invoked or re-rendered frequently.

An expensive component typically involves **heavy calculations**, **rendering large lists**, or **performing tasks** that consume significant browser resources.

Consider a component that performs complex data processing, such as filtering and sorting a large dataset. The operation can become expensive when it needs to happen on every render, which can significantly impact performance.

useMemo memoizes the result of the component's computation, preventing expensive calculations on every render.

@oluwakemi Oluwadahunsi

For example: Memoizing an Expensive Calculation

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

const ExpensiveCalculation = (num) => {
  console.log("Calculating...");
  // Simulate an expensive operation
  return num * 1000;
};

const MemoizedComponent = () => {
  const [number, setNumber] = useState(0);
  const [count, setCount] = useState(0);

  // Memoize the result of the expensive calculation
  const computedValue = useMemo(() => ExpensiveCalculation(number), [number]);

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={() => setCount(count + 1)}>Increment Count</button>

      <h1>Computed Value: {computedValue}</h1>
      <button onClick={() => setNumber(number + 1)}>Increment Number</button>
    </div>
  );
};

export default MemoizedComponent;
```

@oluwakemi Oluwadahunsi

In this example:

- **Without useMemo( ):** The **ExpensiveCalculation** function would run on every render, even when number hasn’t changed, leading to inefficient re-renders.

- **With useMemo( ):** The expensive calculation is only recomputed when the number state changes. The cached result is returned for subsequent renders unless number is updated.

Benefits:

- **Performance Optimization:** Reduces unnecessary computations and function recreations.

- **Stable References:** Provides stable references to functions and computed values, aiding in preventing unnecessary re-renders of child components.

@oluwakemi Oluwadahunsi

3. useCallback( )

useCallback() is used to memoize callback functions. It ensures that the same function reference is maintained across renders, preventing unnecessary re-renders of child components that receive the function as a prop.

For Example: Memoizing an Event Handler

```jsx
import React, { useCallback, useState } from "react";

const Button = React.memo(({ onClick }) => {
  console.log("Button re-rendered");
  return <button onClick={onClick}>Click me</button>;
});

const UseCallbackExample = () => {
  const [count, setCount] = useState(0);

  // Memoize the click handler
  const handleClick = useCallback(() => {
    setCount((prevCount) => prevCount + 1);
  }, []);

  return (
    <div>
      <h1>Count: {count}</h1>
      <Button onClick={handleClick} />
    </div>
  );
};

export default UseCallbackExample;
```

@oluwakemi Oluwadahunsi

In this example:

- **Without useCallback():** The Button component would re-render every time the parent

- **With useCallback():** The onClick function is memoized, ensuring the same function reference is passed to the Button component, preventing unnecessary re-renders.

Note: useCallback( ) is ideal for event handlers passed down to child components to prevent re-renders.

Implementing Virtualization for Large Lists

In React applications, dealing with large lists or datasets can pose significant performance challenges.

Rendering thousands of items at once can lead to slow page load times, janky scrolling, and an overall poor user experience

One effective solution for improving performance when working with large lists is virtualization (or windowing), a technique that only renders the visible portion of a list and dynamically loads items as the user scrolls.

@oluwakemi Oluwadahunsi

What is Virtualization in React?

Virtualization (or windowing) is a performance optimization technique that involves rendering **only the visible part** of a **list** and dynamically loading the remaining items as the user scrolls.

Instead of rendering all the items at once, virtualized lists render only a **subset of items based on the scroll position**, ensuring that the application performs efficiently even with thousands of items.

React has several popular libraries specifically designed for virtualizing large lists, with the two most widely used being:

- **react-window:** A **lightweight** library for rendering large lists with virtualization. It is easy to use and highly performant, with a minimal API.
- **react-virtualized:** A more **feature-rich** library that offers advanced features for handling large datasets. It's great for more **complex** use cases but comes with a steeper learning curve.

@oluwakemi Oluwadahunsi

Implementing Virtualization with react-window

react-window is a popular and lightweight library that provides an easy way to render virtualized lists. It focuses on **simplicity** and performance, making it an excellent choice for most use cases.

For Example:

Before we can use this library, we have to install it either using npm or yarn like this:

```
1 npm install react-window //windows system
2
3 //or
4
5 yarn add react-window
```

@oluwakemi Oluwadahunsi

Here’s a simple example of how to use react-window to virtualize a large list of items.

```jsx
import { FixedSizeList as List } from "react-window";

// A simple list item component
const Row = ({ index, style }) => (
  <div style={style}>Item #{index}</div>
);

const VirtualizedList = () => {
  return (
    <List
      height={400}  // Height of the container
      itemCount={1000}  // Total number of items in the list
      itemSize={35}  // Height of each item
      width={300}  // Width of the list
    >
      {Row}
    </List>
  );
};

export default VirtualizedList;
```

@oluwakemi Oluwadahunsi

In this example:

- **List:** This is the component from react-window that handles virtualization. It renders a windowed (virtualized) list of items.
  - **height:** The height of the list container, which defines the visible window.
- **itemCount:** The total number of items in the list (in this case, 1000).
- **itemCount:** The titemSize: The height of each item in pixels (in this case, 35px).otal number of items in the list (in this case, 1000).
- **Row:** A functional component that renders each list item.

Here, **react-window** ensures that only the visible items are rendered in the DOM. As the user scrolls, new items are dynamically rendered while offscreen items are removed, leading to improved performance.

@oluwakemi Oluwadahunsi

By default, **react-window** assumes that all items in the list have a **fixed** height. If your items have **variable** heights, you can use the **VariableSizeList** component likw this:

```
import React from "react";
import { VariableSizeList as List } from "react-window";

const Row = ({ index, style }) => (
  <div style={style}>Item #{index}</div>
);

const getItemSize = index => {
  // Return varying sizes based on item index
  return index % 2 === 0 ? 50 : 75;
};

const VirtualizedList = () => {
  return (
    <List
      height={400}
      itemCount={1000}
      itemSize={getItemSize}
      width={300}
    >
      {Row}
    </List>
  );
};

export default VirtualizedList;
```

In this case, the height of each item is determined dynamically using the **getItemSize** function.

@oluwakemi Oluwadahunsi

Implementing Virtualization with react-virtualized

react-virtualized is a **more advanced** library that provides a wide array of features, including grids, tables, masonry layouts, and more. While it offers more functionality than react-window, it can be **overkill** for simple use cases.

For Example:

Before we can use this library, we have to install it either using npm or yarn like this:

```
npm install react-virtualized //windows system

//or

yarn add react-virtualized
```

@oluwakemi Oluwadahunsi

Here’s an example of how to use react-virtualized to virtualize a list of items.

```jsx
import React from "react";
import { List } from "react-virtualized";
import "react-virtualized/styles.css";  // Optional CSS for styling

const rowRenderer = ({ key, index, style }) => (
  <div key={key} style={style}>
    Item #{index}
  </div>
);

const VirtualizedList = () => {
  return (
    <List
      width={300}  // Width of the list
      height={400}  // Height of the container
      rowHeight={35}  // Height of each row
      rowCount={1000}  // Total number of rows
      rowRenderer={rowRenderer}  // Function to render each row
    />
  );
};

export default VirtualizedList;
```

@oluwakemi Oluwadahunsi

In this example:

- **List:** The List component from react-virtualized handles virtualization.
- **rowRenderer:** A function that renders each row based on its index and key.
- **rowHeight**: Specifies the height of each row (35px in this case).
- **rowCount:** Specifies the total number of rows (1000 items).

Advanced features in react-virtualized

- **Grid:** For rendering large grids with virtualization.
- **Masonry:** For rendering a masonry layout (items with varying heights).
- **AutoSizer:** Automatically adjusts the size of the list based on its parent container.
- **InfiniteLoader:** Supports infinite scrolling and lazy loading of items.

@oluwakemi Oluwadahunsi

Benefits of Virtualization

The benefits of implementing virtualization in React include:

- **Reduced Initial Rendering Time:** Virtualization ensures only a small subset of items are rendered initially, speeding up the page load.
- **Improved Scrolling Performance:** By reducing the number of rendered DOM elements, virtualization leads to smoother, more responsive scrolling.
- **Lower Memory Usage:** Virtualized lists only keep a few elements in memory, reducing the overall memory footprint.
- **Optimized User Experience:** With faster load times and smoother interactions, virtualization helps maintain a consistent and performant user experience even with large datasets.

Optimizing Images and Assets

Just like every other web development tools, another key aspect of performance optimization technique in React is optimizing images and other static assets such as CSS, JavaScript files, fonts, and videos.

Large and unoptimized images or assets can significantly slow down page load times, increase bandwidth usage, and negatively impact the overall user experience.

There are several methods to optimize images and assets in React applications. These techniques are aimed at improving loading speed, reducing bandwidth usage, and enhancing overall application performance.

@oluwakemi Oluwadahunsi

Best Practices for Image Optimization in React

1. Choosing the Right Image Format

Selecting the appropriate image format is the first step in optimizing images:

- **JPEG (JPG):** Best suited for photographs and complex images with lots of colors. It offers good compression and balance between quality and size.
- **PNG:** Ideal for images with transparency or simple graphics. PNG is lossless but tends to have larger file sizes compared to JPEG.
- **WebP:** A modern image format that provides both lossy and lossless compression, offering superior compression than both JPEG and PNG.
- **SVG:** Scalable vector graphics are great for logos, icons, and illustrations because they are resolution-independent and lightweight.
- **GIF:** Used for simple animations. However, for static images or videos, other formats like WebP or MP4 are preferable due to better compression.

@oluwakemi Oluwadahunsi

For Example: Loading WebP Images with Fallback

```html
<img
  srcSet="image.webp 1x, image@2x.webp 2x"
  src="image.jpg"
  alt="Example"
/>
```

fallback image format

In this example, **srcSet** provides a **WebP** version of the image, and if the browser does not support WebP, it falls back to using the **JPEG** version.

@oluwakemi Oluwadahunsi

2. Responsive Image Handling: Using srcset and sizes

For **mobile-first** performance, images should be responsive, meaning different image sizes are served based on the **device's screen size** and **resolution**.

The **<img>** tag can take advantage of the **srcset** and **sizes** attributes to serve different image versions.


```
<img
  srcSet="image-small.jpg 320w, image-medium.jpg 768w, image-large.jpg 1200w"
  sizes="(max-width: 768px) 100vw, 50vw"
  src="image-large.jpg"
  alt="Responsive image example"
/>
```

- **srcSet:** Defines a set of images with different resolutions (320w, 768w, 1200w) for various screen sizes.
- **sizes:** Instructs the browser on which image to load based on the viewport's width. Here, for screens below 768px, the image takes 100% of the screen width, and for larger screens, it occupies 50%.

@oluwakemi Oluwadahunsi

3. Image Compression and Tools

Image compression is essential for reducing file sizes without sacrificing quality. There are two types of compression:

- **Lossy compression:** Reduces image size by removing some image data. This results in a smaller file size but may slightly reduce image quality.
- **Lossless compression:** Reduces image size without losing any data. The file size reduction is less significant compared to lossy compression.

Tools for Image Compression:

- **TinyPNG or TinyJPG:** Online tools that compress PNG and JPEG images using lossy compression.
- **ImageOptim:** A tool for macOS that compresses images in various formats.
- **Squoosh:** An open-source image compression web app created by Google.

For automated compression in build processes, you can use libraries like **imagemin**.

@oluwakemi Oluwadahunsi

4. Lazy Loading Images in React

Lazy loading defers the loading of off-screen images until the user scrolls to them, improving the initial load time.

In React, you can achieve lazy loading using native browser support or by utilizing a third-party library such as react-lazyload.

Example: Native Lazy Loading

```
1 <img src="image.jpg" alt="Lazy loaded image" loading="lazy" />
```

loading attribute set to “lazy”

With the **loading="lazy"** attribute, images will only load when they are about to enter the viewport.

Example: Lazy Loading with **react-lazyload**

To use react-lazyLoad library, you need to install it first using npm or yarn depending on your OS:

```
1 npm install react-lazyload
2
3 // or
4
5 yarn add react-lazyload
```

```jsx
import LazyLoad from 'react-lazyload';

const LazyImage = () => (
  <LazyLoad height={200} offset={100} once>
    <img src="image.jpg" alt="Lazy loaded image" />
  </LazyLoad>
);

export default LazyImage;
```

Using this method, images are loaded as the user scrolls, improving performance by reducing the number of images loaded initially.

@oluwakemi Oluwadahunsi

Optimizing Other Static Assets in React

1. Minification of CSS and JavaScript Files

Minification involves removing unnecessary characters (like spaces, comments, and newlines) from CSS and JavaScript files, making them smaller and faster to load.

Both Webpack (used in CRA) and Vite automatically minify assets during the production build process.

React applications are built for production by running this command in your terminal:

```
npm run build
```

This command in React creates an optimized production build where all assets are minified.

@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.

3. Video Optimization

If your React app serves videos:

- **Use modern formats (e.g., MP4, WebM):** These offer better compression.
- **Lazy load videos:** Ensure videos are only loaded when they're about to be viewed. You can use the native loading by including the **loading="lazy"** attribute or using the **react-lazyLoading** library just like we did for images.
- **Use poster images:** Provide a poster image for video elements to show a preview while the video is loading.

```
1  const VideoWithPoster = () => {
2    return (
3      <div>
4      <video controls width="600"
         poster="https://example.com/poster-image.jpg">
5        <source src="https://www.example.com/sample-video.mp4"
         type="video/mp4" />
6        Your browser does not support the video tag.
7        </video>
8      </div>
9    );
10 };
11
12 export default VideoWithPoster;
13
```

poster image provided

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