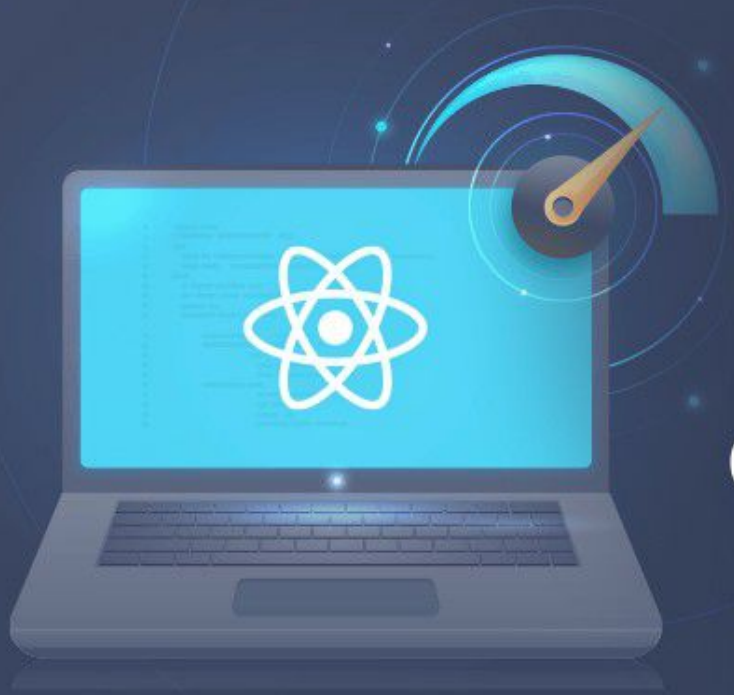

HOW TO IMPROVE THE PERFORMANCE

OF YOUR REACTJS APPS

Using React.memo()

React.memo is a great way of optimizing performance as it helps cache functional components.

In computing, memoization is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

Here’s how it works: When a function is rendered using this technique, it saves the result in memory, and the next time the function with the same arguments is called it returns the saved result without executing the function again, saving you bandwidth.

Using React.memo()

In the context of React, functions are the functional components, and arguments are props. Here's an example:

```
import React from 'react';


const MyComponent = React.memo(props =>  {
  /* render only if the props changed */
});
```

React.memo is a higher-order component and it's similar to React.PureComponent but for using function components instead of classes.

Function/Stateless Components and React.PureComponent

In React, function components and PureComponent provide two different ways of optimizing React apps at the component level.

Function components prevent constructing class instances while reducing the overall bundle size as it minifies better than classes.

On the other hand, in order to optimize UI updates, we can consider converting function components to a PureComponent class (or a class with a custom shouldComponentUpdate method). However, if the component doesn’t use state and other life cycle methods, the initial render time is a bit more complicated when compared to function components with potentially faster updates.

Function/Stateless Components and React.PureComponent

When should we use React.PureComponent?

React.PureComponent does a shallow comparison on state change. This means it compares values when looking at primitive data types, and compares references for objects. Due to this, we must make sure two criteria are met when using React.PureComponent:

- Component State/Props is an immutable object;

- State/Props should not have a multi-level nested object.

Pro Tip: All child components of React.PureComponent should also be a Pure or functional component.

Multiple Chunk Files

Your application always begins with a few components. You start adding new features and dependencies, and before you know it, you end up with a huge production file.

You can consider having two separate files by separating your vendor, or third-party library code from your application code by taking advantage of CommonsChunkPlugin for webpack. You'll end up with vendor.bundle.js and app.bundle.js. By splitting your files, your browser caches less frequently and parallel downloads resources to reduce load time wait.

Using the useCallback Hook

With the useCallback Hook, the incrementCount function only redefines when the count dependency array changes:

```
const incrementCount = React.useCallback(() => setCount(count + 1), [count]);
```

Windowing or list virtualization in React applications

Imagine we have an application where we render several rows of items on a page. Whether or not any of the items display in the browser viewport, they render in the DOM and may affect the performance of our application.

With the concept of windowing, we can render to the DOM only the visible portion to the user. Then, when scrolling, the remaining list items render while replacing the items that exit the viewport. This technique can greatly improve the rendering performance of a large list.

Lazy loading images in React

To optimize an application that consists of several images, we can avoid rendering all of the images at once to improve the page load time. With lazy loading, we can wait until each of the images is about to appear in the viewport before we render them in the DOM.

Similar to the concept of windowing mentioned above, lazy loading images prevents the creation of unnecessary DOM nodes, boosting the performance of our React application.

react-lazyload and react-lazy-load-image-component are popular lazy loading libraries that can be used in React projects.

Using a CDN

A CDN is a great way to deliver static content from your website or mobile application to your audience more quickly and efficiently.

CDN depends on user geographic location. The CDN server closest to a user is known as an “edge server”. When the user requests content from your website, which is served through a CDN, they are connected to edge server and are ensured the best online experience possible.

There are some great CDN providers out there. For example, CloudFront, CloudFlare, Akamai, MaxCDN, Google Cloud CDN, and others.

Use React.Fragments to Avoid Additional HTML Element Wrappers

React.fragments lets you group a list of children without adding an extra node.

```
class Comments extends React.PureComponent{
    render() {
        return (
            <React.Fragment>
                <h1>Comment Title</h1>
                <p>comments</p>
                <p>comment time</p>
            </React.Fragment>
        );
    }
}
```

Spreading props on DOM elements

You should avoid spreading properties into a DOM element as it adds unknown HTML attribute, which is unnecessary and a bad practice.

```
const CommentsText = props => {
    return (
      <div specificAttr={props.specificAttr}>
        {props.text}
      </div>
    );
};
```

Instead of spreading props, you can set specific attributes:

```
const CommentsText = props => {
    return (
      <div {...props}>
        {props.text}
      </div>
    );
  };
```

Avoid Async Initialization in componentWillMount()

componentWillMount() is only called once and before the initial render. Since this method is called before render(), our component will not have access to the refs and DOM element.

Here’s a bad example:

```
function componentWillMount() {
  axios.get(`api/comments`)
    .then((result) => {
      const comments = result.data
      this.setState({
        comments: comments
      })
    })
}
```

Avoid Async Initialization in componentWillMount()

Let’s make it better by making async calls for component initialization in the componentDidMount lifecycle hook:

```
function componentDidMount() {
  axios.get(`api/comments`)
    .then((result) => {
      const comments = result.data
      this.setState({
        comments: comments
      })
    })
}
```

The componentWillMount() is good for handling component configurations and performing synchronous calculation based on props since props and state are defined during this lifecycle method.

Analyzing and Optimizing Your Webpack Bundle Bloat

Before production deployment, you should check and analyze your application bundle to remove the plugins or modules that aren’t needed.

You can consider using Webpack Bundle Analyzer, which allows you to visualize the size of webpack output files with an interactive zoomable treemap.

This module will help you:

- Realize what's really inside your bundle
- Find out what modules take up the most size
- Find modules that got there by mistake
- Optimize it!

Enable Gzip Compression on Web Server

Gzip compression allows the web server to provide a smaller file size, which means your website loads faster. The reason gzip works so well is because JavaScript, CSS, and HTML files use a lot of repeated text with lots of whitespace. Since gzip compresses common strings, this can reduce the size of pages and style sheets by up to 70%, shortening your website's first render time.

If you are using Node/Express backend, you can use Gzipping to compress your bundle size with the compression module.

```
const express = require('express');
const compression = require('compression');
const app = express();

// Pass `compression` as a middleware!
app.use(compression());
```