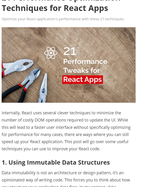
Codementor Blog Last updated Jan 17, 2020

21 Performance Optimization Techniques for React Apps

Optimize your React application's performance with these 21 techniques



Internally, React uses several clever techniques to minimize the number of costly DOM operations required to update the UI. While this will lead to a faster user interface without specifically optimizing for performance for many cases, there are ways where you can still speed up your React application. This post will go over some useful techniques you can use to improve your React code.

1. Using Immutable Data Structures

Data immutability is not an architecture or design pattern. It's an opinionated way of writing code. This forces you to think about how you structure your application data flow. In my opinion, data immutability is a practice that revolves around a strict unidirectional data flow.

Data immutability, which comes from the functional programming world, can be applied to the design of front-end apps. It can have many benefits, such as:

- Zero side-effects;
- Immutable data objects are simpler to create, test, and use;
- Helps prevent temporal coupling;
- Easier to track changes.

In the React landscape, we use the notion of `immutable` to maintain the internal state of components, and changes to the state can cause the component to re-render.

React builds and maintains an internal representation of the rendered UI (Virtual DOM). When a component's props or state changes, React compares the newly returned element with the previously rendered one. When the two are not equal, React will update the DOM. Therefore, we have to be careful when changing the state.

Let's consider a User List Component:

```
state = {
    users: []
}

addUserUser = () =>{
    // mutate array
    // if (!Array.isArray(this.state.users)) {
    //   this.state.users = [];
    // }
    const users = this.state.users;
    const newUser = {
        username: "random",
        email: "test@email.com"
    };
    this.setState({users: [...users]});
}
```

The concern here is that we are pushing new users onto the variable `users`, which is a reference to `this.state.users`.

Pro Tip: React state should be treated as immutable. We should never mutate `this.state` directly as calling `setState()` afterward may replace the mutation you made.

So what's wrong with mutating `state` directly? Let's say we overrode `shouldComponentUpdate` and are checking `nextProps` against `this.state` to make sure that we only re-render components when changes happen in the state.

```
shouldComponentUpdate(nextProps, nextState) {
    if (this.state.users !== nextState.users) {
        return true;
    }
    return false;
}
```

Even if changes happen in the user's array, React won't re-render the UI as it's the same reference.

The easiest way to avoid this kind of problem is to avoid mutating props or state. So the `addNewUser` method could be rewritten using `concat`:

```
addNewUser = () => {
    this.setState({state => ({
        users: state.users.concat([
            {
            username: "random",
            email: "test@email.com"
            }
        ])
    })});
}
```

For handling changes to state or props in React components, we can consider the following immutable approaches:

- For arrays use `[].concat` or ES6 `[...,param]`;
- For objects use `Object.assign({},...)` or ES6 `{...,param}`

These two methods go a long way when introducing immutability to your code base.

But it's better to use an optimized library which provides a set of immutable data structures. Here are some of the libraries you can use:

- Immutability Helper: This is a good library when it's comes to mutating a data copy without changing the source.
- Immutable.js: This is my favorite library as it provides a lot of persistent immutable data structures, including: List, Stack, Map, OrderedMap, Set, OrderedSet, and Record.
- Seamless-immutable: A library for immutable javaScript data structures that are backward-compatible with normal arrays and objects.
- React-copy-write: An immutable React state management library with a simple mutable API, memoized selectors, and structural sharing.

Pro Tip: React `setState` method is asynchronous. This means that rather than immediately mutating `this.state`, `setState()` creates a pending state transition. If you access `this.state` after calling this method, it would potentially return the existing value. To prevent this, use the callback function of `setState` to run code after the call is completed.

Additional Resources:

- Do you really need immutable data?
- The State of Immutability
- Pros and Cons of using immutability with React.js
- Handling State in React: Four Immediate Approaches to Consider

2. 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.

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 StatelProps is an immutable object;
- StatelProps should not have a multi-level nested object.

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

3. 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 CommonChunkPlugin 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.

Note: If you are using the latest version of webpack, you can also consider SplitChunksPlugin.

4.Using Production Mode Flag in Webpack

If you are using `webpack 4`, as a module bundler for your app, you can consider setting the mode option to **production**. This basically tells webpack to use the built-in optimization:

```
module.exports = {
    mode: 'production'
};
```

Alternatively, you can pass it as a CLI argument:

```
webpack --mode=production
```

Doing this will time optimizations, such as minification or removing development-only code, to libraries. It will not expose source code, file paths, and many more.

5.Dependency optimization

When considering optimizing the application bundle size, it's worth checking how much code you are actually utilizing from dependencies. For example, you could be using `moment.js` which includes localized files for multi-language support. If you don't need to support multiple languages, then you can consider using ignore locales webpack-plugin to remove unused locales for your final bundle.

Another example is `lodash`. Let's say you are only using 20 of the 100+ methods, then having all the extra methods in your final bundle is not optimal. So for this, you can use lodash-webpack-plugin to remove unused functions.

Here's an extensive list of dependencies which you can optimize.

6. Use `React.Fragments` To Avoid Additional HTML Element Wrappers

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

```
class Columns extends React.PureComponent{
    render(){
        return (
            <Aux>
            <Aux>.Fragment>
                <td>Column1 Title</td>
                <td>Column2 Title</td>
                <td>Column3 Title</td>
            </Aux>.Fragment>
        );
    }
}
```

But wait! There is the alternate and more concise syntax using `React.Fragments`:

```
class Columns extends React.PureComponent{
    render(){
        return (
            <>
                <td>Column1 Title</td>
                <td>Column2 Title</td>
                <td>Column3 Title</td>
            </>
        );
    }
}
```

7. Avoid Inline Function Definition in the Render Function.

Since functions are objects in JavaScript (`() => {}` is the inline function will always fail the prop diff when React does a diff check. Also, an arrow function will create a new instance of the function on each render if it's used in a JSX property. This might create a lot of work for the garbage collector.

```
default Class CommentList extends React.Component {
    render(){
        {comments: [],
        selectedCommentID: null}
    }

    render(){
        const { comments } = this.state;
        return (
            {comments.map((comment)=>{
                return <Comment onClick={()=>{
                    this.setState({selectedCommentID:comment.commentID})
                }} comment={comment} key={comment.id}/>
                })}
            )
    }
}
```

Instead of defining the inline function for props, you can define the arrow function.

```
default Class CommentList extends React.Component {
    render(){
        {comments: [],
        selectedCommentID: null}
    }

    onCommentClick = (commentID)=>{
        this.setState({selectedCommentID:commentID})
    }

    render(){
        const { comments } = this.state;
        return (
            {comments.map((comment)=>{
                return <Comment onClick={this.onCommentClick}
                    commentID={comment.commentID} key={comment.id}/>
                })}
            )
    }
}
```

8. Throttling and Debouncing Event Action in JavaScript

Event trigger rate is the number of times an event handler invokes in a given amount of time.

In general, mouse clicks have lower event trigger rates compare to scrolling and mousemove. Higher event trigger rates can sometimes crash your application, but it can be controlled.

Let's discuss some of the techniques.

First, identify the event handler that is doing the expensive work. For example, an RHR request or DOM manipulation that performs UI updates, processes a large amount of data, or perform computation expensive tasks. In these cases, throttling and debouncing techniques can be a savior without making any changes in the event listener.

Throttling

In a nutshell, throttling means delaying function execution. So instead of executing the event handler/function immediately, you'll be adding a few milliseconds of delay when an event is triggered. This can be used when implementing infinite scrolling, for example. Rather than fetching the next result set as the user is scrolling, you can delay the XHR call.

Another good example of this is Ajax-based instant search. You might not want to hit the server for every key press, so it's better to throttle until the input field is dormant for a few milliseconds

Throttling can be implemented a number of ways. You can throttle by the number of events triggered or by the delay event handler being executed.

Debouncing

Unlike throttling, debouncing is a technique to prevent the event trigger from being fired too often. If you are using `lodash`, you can wrap the function you want to call in `lodash's debounce function`.

Here's a demo code for searching comments:

```
import debounce from 'lodash.debounce';

class SearchComponents extends React.Component {
    constructor(props) {
        super(props);
        this.state = { searchQuery: "" };
    }

    setSearchQuery = debounce(v => {
        this.setState({ searchQuery: v.target.value });
    }, 1000);

    render() {
        return (
            <div>
                <Input/>onChange={this.
                    setSearchQuery.bind(this)}
                />
            </div>
        );
    }
}
```

If you are not using `lodash`, you can use the minified debounced function to implement it in javaScript.

```
function debounce(a,b,c){var d,e;return function(){function h(){d=null,c||(e=a.
```

RELATED ARTICLES

JS Angular.js Interview Questions and Answers You Should Know

JS PHP Interview Questions and Answers You Should Know

FEATURED POST

Beer Cities to Launch a Startup Outside of the U.S.

"Array" Methods,
Handling Events.

9. Avoid using Index as Key for map

You often see indexes being used as a key when rendering a list.

```
{
  comments.map((comment, index) => {
    return (
      <Comment
        key={index} />
    )
  })
}
```

But using the key as the index can show your app incorrect data as it is being used to identify DOM elements. When you push or remove an item from the list, if the key is the same as before, React assumes that the DOM element represents the same component.

It's always advisable to use a unique property as a key, or if your data doesn't have any unique attributes, then you can think of using the `shortid` module which generates a unique key.

```
import shortid from "shortid";

{
  comments.map((comment, index) => {
    return (
      <Comment
        key={shortid.generate()} />
    )
  })
}
```

However, if the data has a unique property, such as an ID, then it's better to use that property.

```
{
  comments.map((comment, index) => {
    return (
      <Comment
        key={comment.id} />
    )
  })
}
```

In certain cases, it's completely okay to use the index as the key, but only if below condition holds:

- The list and items are static
- The items in the list don't have IDs and the list is never going to be reordered or filtered
- List is immutable

References and Related Articles:
Consider providing a default key for dynamic children #1342,
The importance of component keys in React.js,
Why you need keys for collections in React

10. Avoiding Props in Initial States

We often need to pass initial data with props to the React component to set the initial state value.

Let's consider this code:

```
class EditPanelComponent extends Component {

  constructor(props)
    super(props);

  this.state =
    isEditMode: false,
    applicationProps: props.applicationProps
  }

  render()
    return <div>
              {this.state.applicationProps.NameOfApplication}
              <button Coupons:{Coupons}></div>
    </div>
  }
}
```

Everything looks good in the snippet, right?

But what happens when `props.applicationProps` changes? Will it be reflected in the state? If the props are changed without refreshing the component, the new prop value will never be assigned to the state's `applicationProps`. This is because the construction function is only called when `EditPanelComponent` is first created.

To quote **React docs**:

> *Using props to initialize a state in constructor function often leads to duplication of "source of truth", i.e. where the real data is. This is because construction function is only invoked when the component is first created.*

Workaround:

1. Don't initialize state with props which can be changed later. Instead, use props directly in the component.

```
class EditPanelComponent extends Component {

  constructor(props)
    super(props);

  this.state =
    isEditMode: false
  }

  render()
    return <div>{this.props.applicationProps.NameOfApplication}
              <button Coupons:{this.props.Coupons}></div>
    </div>
  }
}
```

2. You can use `componentDidReceiveProps` to update the state when props change.

```
class EditPanelComponent extends Component {

  constructor(props)
    super(props);

  this.state =
    isEditMode: false,
    applicationProps: props.applicationProps
  }

  // reset state if the needed prop is updated
  componentDidReceiveProps(newProps)
    if (newProps.applicationProps !== this.props.applicationProps) {
      this.setState({ applicationProps: newProps.applicationProps })
    }

  render()
    return <div>{this.props.applicationProps.NameOfApplication}
              <button Coupons:{this.props.Coupons}></div>
    </div>
  }
}
```

References and Related Articles:
React(JS): Why is passing the component initial state a prop an anti-pattern?,
React Anti-Patterns: Props in Initial State

11. 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 CommentToast = props => {
  return (
    <div {...props}>
      {props.text}
    </div>
  )
};
```

Instead of spreading props, you can set specific attributes:

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

12. Use Reselect in Redux to Avoid Frequent Re-render

Reselect is a simple selector library for Redux, which can be used for building memoized selectors. You can define selectors as a function, retrieving snippets of the Redux state for React components.

Let's take a look at this code that isn't using Reselect:

```
const App = ({ comments, socialDetails }) => {
  return
    <CommentsContainer data={comments} />
    <SharedContainer socialDetails={socialDetails} />
  </div>
};

const mapStateToProps = social => {
  {socialDetails: [...state.lists[social].listedIds]
};

const mapDispatchToProps = {};
export
  connect(mapStateToProps, mapDispatchToProps)
  SocialDetails.displayName(SocialDetails)
];

App = connect(mapState => {
  return {
    comments: state.comments,
    socialDetails: mapStateToProps(state.social)
  }
})(App);
```

In this code, every time the `comments` data in state changes, both `CommentsContainer` and `SharedContainer` will be re-rendered. This happens even when `socialDetails` doesn't make any data changes to `socialDetails` as `socialDetails` will return a new object with a different identity (`===`). Here, if we rewrite `mapStateToProps` with Reselect, the issue will go away as Reselect will return the last function result until it is passed new inputs.

```
import { createSelector } from 'reselect';
const mapStateToProps = (state, social);
const mapDispatchToProps = {};
socialDetailsSelector,
  state => {
  [...state.lists[social].listedIds]
: {};
];
```

References and Related Articles:
Reselect: Motivation for Memoized Selectors,
Improving React and Redux performance with Reselect,
Using reselect

13. 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(comments => {
      // assume response contains a result data
      this.setState({
        comments: comments
      })
    })
}
```

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

```
function componentDidMount() {
  axios.get('api/comments')
    .then(comments => {
      // assume response contains a result data
      this.setState({
        comments: comments
      })
    })
}
```

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

14. Memoize React Components

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. A memoized function is usually faster because if the function is called with the same values as the previous one then instead of executing function logic it would fetch the result from cache.

Let's consider below simple stateless UserDetails React component.

```
const UserDetails = ({lower, width}) => {
  const {title, full_name, profile_img} = user;

  return (
    <div classNamen="user-detail-wrapper">
      
```

You can also choose Netlify or Surge.sh to host your static content on CDN. Surge is a free CLI tool that deploys your static projects to production-quality CDN.

References and Related Articles:
Why you should use a content delivery network

17. Using Web Workers for CPU Extensive Tasks

Web Workers make it possible to run a script operation in a web application's background thread, separate from the main execution thread. By performing the laborious processing in a separate thread, the main thread, which is usually the UI, is able to run without being blocked or slowed down.

In the same execution context, as javaScript is single threaded, we will need to parallel compute. This can be achieved two ways. The first option is using pseudo-parallelism, which is based on `setTimeout` function. The second option is to use Web Workers.

Web Workers works best when executing computation extensive operation since it executes code independent of other scripts in a separate thread in the background. This means it doesn't affect the page's performance.

We can take advantage of Web Workers in React to execute computation extensive tasks.

Here's a code without using Web Workers:

```
// calculate average price per point by the number of comments
const getPostsPerPoint = (posts) =>
  for (let index = 0, len = posts.length - 1; index < len; index++) {
    for (let count = index+1; count < posts.length; count++) {
      if (posts[index].points == posts[count].points && posts[index].commentCount !
        const temp = postsCopy[count];
        postsCopy[count] = postsCopy[index];
        postsCopy[index] = temp;
      }
    }
  }
  return postsCo;
}

export default Posts extends React.Component{

  constructor(props){
    super(props);
  }

  state = {
    posts: this.props.posts
  }

  doesHighlightCount = (j) => {
    if(this.state.posts && this.state.posts.length){
      const sortedPosts = sortThisArray(this.posts);
      if(sortedPosts){
        posts: sortedPosts
      }{j}
    }
  }

  render(){
    const posts = this.state.posts;
    return(
      <ul>{j=>{posts}
        {this.doesHighlightCount(j)
          key={j}
          {j+1}.
        }</ul>{j=>{posts}
      </Result.Fragment>
    )
  }
}
```

What happens when we have 20,000 posts? It will slow down the rendering since the `sort` method time complexity is `O(n^2)`, which will run in the same thread.

Below is the modified code, which uses Web Workers to process sorting:

```
// sort.worker.js
// calculate the function per post per by number of comments
export default function sort() {

  self.addEventListener('message', e => {
    if (!e) return;
    let posts = e.data;

    for (let index = 0, len = posts.length - 1; index < len; index++) {
      for (let count = index+1; count < posts.length; count++) {
        if (posts[index].points == posts[count].points && posts[index].commentCount !
          const temp = postsCopy[count];
          postsCopy[count] = postsCopy[index];
          postsCopy[index] = temp;
        }
      }
    }
    postMessage(posts);
  });
}

export default Posts extends React.Component{

  constructor(props){
    super(props);
  }

  state = {
    posts: this.props.posts
  }

  componentDidMount() {
    this.worker = new Worker('sort.worker.js');

    this.worker.addEventListener('message', event => {
      const sortedPosts = event.data;
      this.setState({
        posts: sortedPosts
      });
    });
  }

  doesHighlightCount = (j) => {
    if(this.state.posts && this.state.posts.length){
      this.worker.postMessage(this.state.posts);
    }
  }

  render(){
    const posts = this.state.posts;
    return (
      <Result.Fragment>
        {this.doesHighlightCount(j)
          key={j}
          {j+1}.
        }</ul>{j=>{posts}
      </Result.Fragment>
    )
  }
}
```

In this code, we are running the `sort` method in separate thread, which will ensure we're not blocking the main thread.

You can consider using Web Workers for tasks like image processing, sorting, filtering, and other CPU extensive tasks.

Reference: Using Web Workers

18. Virtualize Long Lists

List virtualization, or windowing, is a technique to improve performance when rendering a long list of data. This technique only renders a small subset of rows at any given time and can dramatically reduce the time it takes to re-render the components, as well as the number of DOM nodes created.

There are some popular React libraries out there, like react window and react-virtualized, which provides several reusable components for displaying lists, grids, and tabular data.

References and Related Articles:
You don't want to build your own list virtualization, Optimizing Performance

19. Analyzing and Optimizing Your Webpack Bundle Bloat

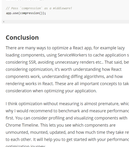
Before production deployment, you should check and analyze your application bundle to remove the plugins or modules that aren't needed.

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!

The best thing? It supports minified bundles! It parses them to get the real size of bundled modules and also shows the gzipped sizes! Here's an example of running the webpack-bundle-analyzer:


bundle.js
node_modules
lodash
react
react-dom
index.js
index.js
index.js

negatively affect overall performance.

0 ^ · Reply · Share ›

Edit · 3 year ago

I think the guy copied your article.

https://medium.com/twitter-p...

0 ^ · Reply · Share ›

Bernard Gitanda ★ · 3 year ago

Hi Edit, I read the article on medium https://medium.com/twitter-p... was published in July 2016 and the Jan 05, 2016, I am impressed by this article felt bad seeing a shadow copycat

0 ^ · Reply · Share ›

Charles Anze Cabezuelo ★ Bernard Gitanda · 3 year ago

Sounds like a great opportunity for plagiarism. Too bad it is currently looking into this issue. Glad to see that the features are not overrated. Thank you again :)

0 ^ · Reply · Share ›

Edit · 3 year ago

Your example number 7 will NOT work, you might want to check out what react passes down on the event handler.

0 ^ · Reply · Share ›

Anton Kapitonov · 3 year ago · edited

Nice article. Rank you! You only have confused throttle with debounce. throttle guarantees that a function will be executed regularly, at most once every X milliseconds. debounce guarantees that a function will never execute at some point in the future after N milliseconds after the event stops happening. Your debounce example is in fact what you describe in the throttle section.

0 ^ · Reply · Share ›

Sruren Mishra · 3 year ago

Great article. Great use of Stack people platform. Hire, engage and nurture right to grow business. Watch this video to see the power of insights and analytics

0 ^ · Reply · Share ›

Filip Tengnemark · 3 years ago

Nice article. Just a small point about #10. componentWillReceiveProps is deprecated. The new way of handling updates should be used instead of it. This is a static function and gets called even when we do setState.

0 ^ · Reply · Share ›

Tod Rizer · 3 years ago · edited

#20 - Don't forget the awesome React Static!

0 ^ · Reply · Share ›

Filip Tengnemark ★ Tod Rizer · 3 years ago

I have a blog post which summarizes the challenges with SSR in React :

0 ^ · Reply · Share ›

Edit ★ Filip Tengnemark · 7 months ago

Great article

0 ^ · Reply · Share ›

Subscribe · Add Disqus to your site · Do Not Sell My Data · DISQUS