Top 10 Optimizing Techniques Every React developer should know! Tips to optimize your ReactJS App.

Eli Elad Elrom Follow
Nov 7, 2020 · 27 min read ★

By the end of this article, you will know how to precache, lazy loading, code splitting, tree shaking, prefetching, sprite splitting as well as dead code and side effects cleaning among other things for React App — and I am not double-talking with you.

Optimizing code is the next level and an advanced topic and needed to ensure we deliver a quality product that lowers our footprint and loads our app faster, In this article, I will give you optimizing techniques efforts you should be aware of before you write your first line of code.

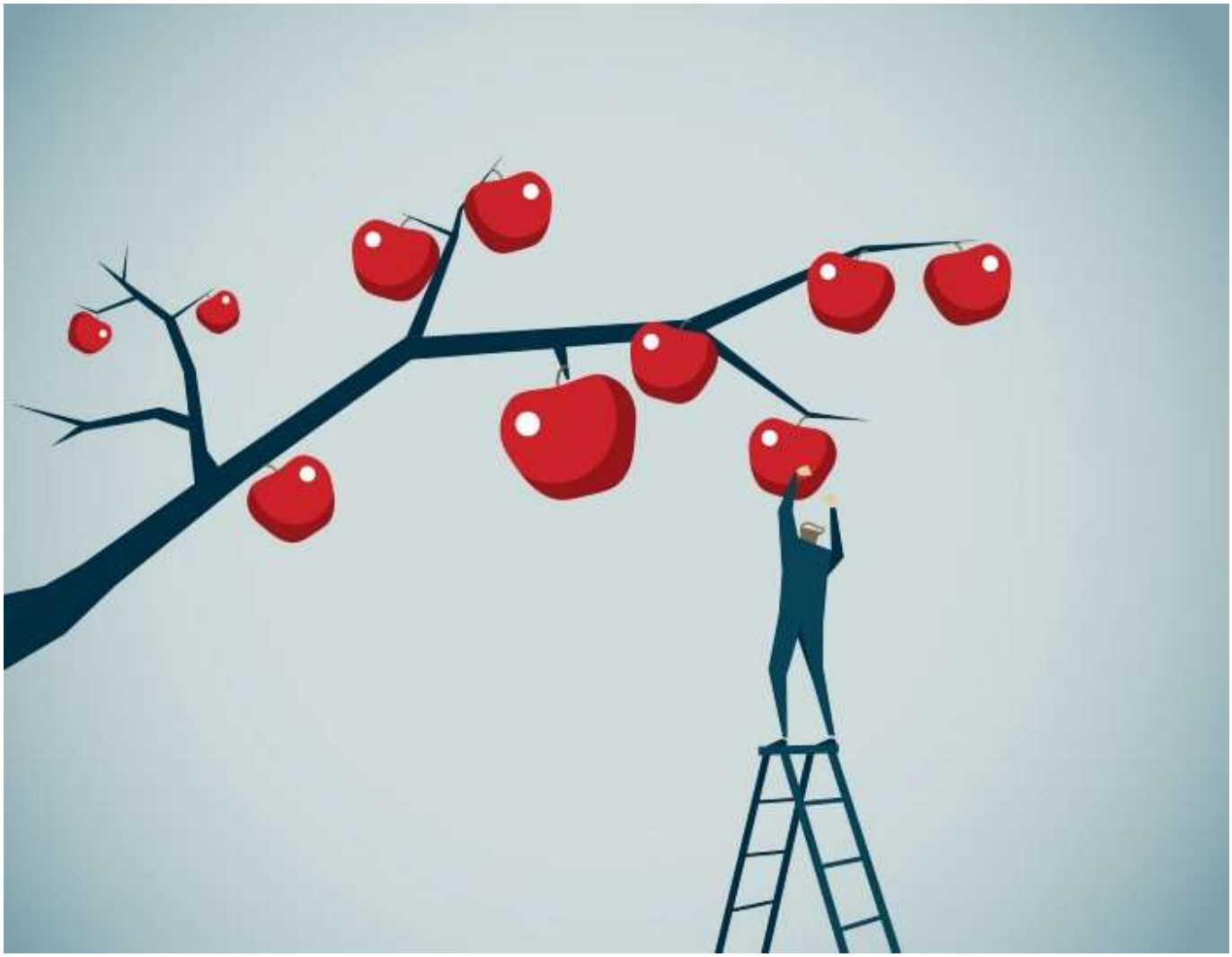


Photo credit: Getty images.

Why do we need to optimize?

CRA is a popular way to create a React App, it is opinionated and set to manage your configuration of our App using react-scrips, it utilizing WebPack techniques to optimize the production build and include minifying, uglify, gzipped, and you don't need to do much it's all out of the box.

On top of that, you should already be automating your development and deployment effort: setting ESlint, Huskey, Unit Testing, E2E testing, checking code quality, and other efforts to identify poor coding and weak spots (see here).

However, with all of that, CRA is still vanilla flavor one size fits all and there are many cases that you would want to fine-tune and configure your App a bit more to your specific needs and be more aware of what's happening under the hood. Furthermore, if you not careful, your App can bloat and turn into jank.

> ***Note**: Jank is not a typo it's a low-quality processes that lead to poor response times for your App, or that blocks user interactions.*

One of the main performance advantages and challenges with single-page applications (SPA) that CRA is based on, is that the user needs to wait for the JS bundles, that make up the App to complete downloading before the user can see the content.

In case the JS bundles get bloated this can take valuable time for the user on a slow internet connection, making your App sluggish, or just not available to some users, resulting in losing visitors and business. Our goal is to build Apps that are Progressive Web Apps (PWA).

> Progressive Web Apps (PWA) are built and enhanced with modern APIs to deliver enhanced capabilities, reliability, and installability while reaching anyone, anywhere, on any device with a single codebase. — https://web.dev/what-are-pwas/

What will you learn?

Following the steps in this article, I will show you how you can reduce the amount of memory footprint, avoid memory leaks, reduce bundle file sizes, load resources only once in usage, decrease wait time to view content, increase performance and ensure it works anytime, anywhere even offline.

You also going to be more aware of what's happening under the hood in your App, so you can configure your App better instead of using the default settings. This subject is big and can take several articles to cover but this article aims to give you the main and most important ways to optimize.

How can I optimize my App?

I broke down the top ways to optimize your CRA React TS App into parts;

1. **PureComponent and React.memo() —** gain performance.

2. **Lazy loading —** break your JS bundles and serve once needed avoiding wait time to see your content.
3. **Prerender —** almost static HTML.
4. **Precache —** have your App work offline.
5. **Code Splitting —** split JS Bundle even more.
6. **Tree shaking —** dead code removal configure 'package.json' file.
7. **Reduce Media size —** reduce the size of the media resources with image sprite.
8. **Prefetching —** set loading hierarchy.
9. **Clean unused side effects event handlers —** avoid memory leaks.
10. · **Install modules instead of global imports —** reduce bundle size

Setup

But, before we start let's create a project that we can use to play around with and test.

I will be using my CRA starter template project that I use as my starter project and it is already including the must-have libraries such as TS, SCSS, React 17, Recoil & Redux, and much more.

```
$ yarn create react-app optimize-ts --template must-have-libraries
```

Let's create a page component, that we can use to experiment. I am using the Generate templates that are already included with the CRA MHL template project that can help us create these pages quickly with one command;

```
$ npx generate-react-cli component MyPage --type=page
```

The template generated three files (scss style, TS component, and Jest test file) automatically for us, using the templates sets inside the 'templates/page' folder, we get a

confirmation here;

```
Stylesheet "MyPage.scss" was created successfully at
src/pages/MyPage/MyPage.scss
Test "MyPage.test.tsx" was created successfully at
src/pages/MyPage/MyPage.test.tsx
Component "MyPage.tsx" was created successfully at
src/pages/MyPage/MyPage.tsx
```

Next, the CRA MHL template includes React Routing and Recoil/Redux Toolkit already out of the box, open '*AppRouter.tsx*' and let's add the page we created to our Router;

```
// src/AppRouter.tsx

import MyPage from './pages/MyPage/MyPage'

const AppRouter: FunctionComponent = () => {
  return (
    <Router>
      <RecoilRoot>
        <Suspense fallback={<span>Loading...</span>}>
          <Switch>
            <Route exact path="/" component={App} />
            <Route exact path="/MyPage" component={MyPage} />
          </Switch>
        </Suspense>
      </RecoilRoot>
    </Router>
  )
}
```

Test that everything working correctly (you should see just the spinner with the link);

```
$ yarn start
```

At this point, you can navigate to the page: http://localhost:3000/MyPage

Next, let's add a link to the page component we created so we can navigate from the main page.

Open 'src/App.tsx' and add the `NavLink` so we can link to our page menu. I am setting it as an array in case we want to add more pages.

```
<List>
  {[
    { name: 'MyPage', url: '/MyPage' }
  ].map((itemObject, index) => (
    <NavLink
      to={itemObject.url}
      key={itemObject.url}
    >
      <ListItem>{itemObject.name}</ListItem>
    </NavLink>
  ))}
</List>
```

You can see the final result;

Figure 1. Our CRA template starter project with a page and navigation.

Analyzer Bundle

Another tool good to know about is the Analyzer Bundle as well as knowing how to debug and profile your App well.

If we want to see what's under the hood when it comes to our JS bundle, we can just eject from CRA and uglify our code to see the content of our JS Bundles. Another option to tweak the create-react-app webpack config(s) without eject using this library: react-app-rewired (https://github.com/timarney/react-app-rewired).

But you don't need to do all of that because ejecting would force you to maintain the configuration files, we can just use `source-map-explorer` to view a map of our bundles;

```
$ npm install -g source-map-explorer
```

Now you can open the libraries and intuitively view the libraries;

```
$ source-map-explorer optimize-ts/build/static/js/[my chunk].chunk.js
```

Alternative use 'bundle-analyzer' (https://github.com/svengau/cra-bundle-analyzer), it's more colorful and includes all the bundles in one page instead of calling them one by one with `source-map-explorer`.

```
$ yarn add -D cra-bundle-analyzer
```

Now we can create the report:

```
$ npx cra-bundle-analyzer
```

This command will generate '`webpack-bundle-analyzer`' report in `build/report.html`, once you '`$ yarn build`'.

#1 Use PureComponent and `React.memo()` as much as possible

One of the most common issues in React applications is when components re-render unnecessarily. There are two tools provided by React that are helpful in these situations:

- `PureComponent`: prevents unnecessary re-rendering of class components
- `React.memo()`: prevents unnecessary re-rendering of function components

For React Classes;

React offers two main options when it comes to creating React Component classes;

- `React.Component` - This prevents unnecessary re-rendering of function
- `React.PureComponent` – This prevents unnecessary re-rendering of class components

When you don't need `shouldComponentUpdate` it's better to use, `PureComponent` instead. This prevents unnecessary re-rendering of class components.

```
extends React.PureComponent
```

`React.PureComponent` gives performance boost in some cases in exchange for losing `shouldComponentUpdate` lifecycle event. You can read about it more in the React Docs (https://reactjs.org/docs/react-api.html#reactpurecomponent).

In our code, we don't need access to `shouldComponentUpdate` so we can use PureComponent. Here is the initial file that just displays the name of the page.

```
import React from 'react'
import './MyPage.scss'
import { RouteComponentProps } from 'react-router-dom'
import Button from '@material-ui/core/Button

// or React.Component
export default class MyPage extends React.PureComponent<IMyPageProps,
IMyPageState> {
  constructor(props: IMyPageProps) {
```

```
    super(props)
    this.state = {
      name: this.props.history.location.pathname
        .substring(1, this.props.history.location.pathname.length)
        .replace('/', ''),
      results: 0
    }
  }

  render() {
    return (
      <div className="TemplateName">
        {this.state.name} Component
      </div>)
    )
  }
}

interface IMyPageProps extends RouteComponentProps<{ name: string }>
{
  // TODO
}

interface IMyPageState {
  name: string
  results: number
}
```

React.memo()

Similarly, if you using Function Component and your component is is a higher-order (HOC) component check `React.memo()`. This prevents unnecessary re-rendering of function components;

```
const MyComponent = React.memo(function MyComponent(props) {
  /* render using props */
})
```

Caveat

Both `PureComponent` and `React.memo()` rely on a shallow comparison of the props passed into the component. In case props are not changed, then the component will not re-render.

While both tools are very useful, the shallow comparison brings with it an additional performance penalty, so both can have a negative performance impact if used incorrectly.

By using the React Profiler, performance can be measured before and after using these tools to ensure that performance is actually improved and not degraded.

#14 Component StrictMode

`<StrictMode />` is a component helper included with React and provides additional visibility of potential issues that may exist in your components. If the application is running in development mode, any issues are logged to the development console, but these warnings are not shown if the application is running in production mode.

`<StrictMode />` is great to use when in bug hunting mode to find problems such as deprecated lifecycle methods and legacy patterns, to ensure that all React components follow current best practices and side effect free. It's also good to use when you unfamiliar with the codebase.

`<StrictMode />` can be applied at any level of an application component hierarchy, which allows it to be adopted incrementally within a codebase.

```
<React.StrictMode>
  <MyComponent />
</React.StrictMode>
```

Keep in mind, if you set linting correctly, that should do the same job as StrictMode and point out the same concerns so it may be redundancy depends on how you configure your project.

Re-re-re-re-render

With that being said, there are times where `shouldComponentUpdate` is needed because we can use that method to let React know that the component is not affected by a state change from a parent component, and no need to re-render. In this case, you would need to set your class as `React.Component` and then you can access `shouldComponentUpdate`;

```
public shouldComponentUpdate(nextProps: IProps, nextState: IState) {
  return false // prevent rendering
}
```

These cases where we need to control the component and want to stop the re-render. Using `React.Component` can give better performance as we can stop the re-render process.

To find misbehaving citizens, you can use the Chrome DevTools React developer tool extension “Highlight Updates” checkbox to find components that misbehave and abuse by looking out for over-rendering components.

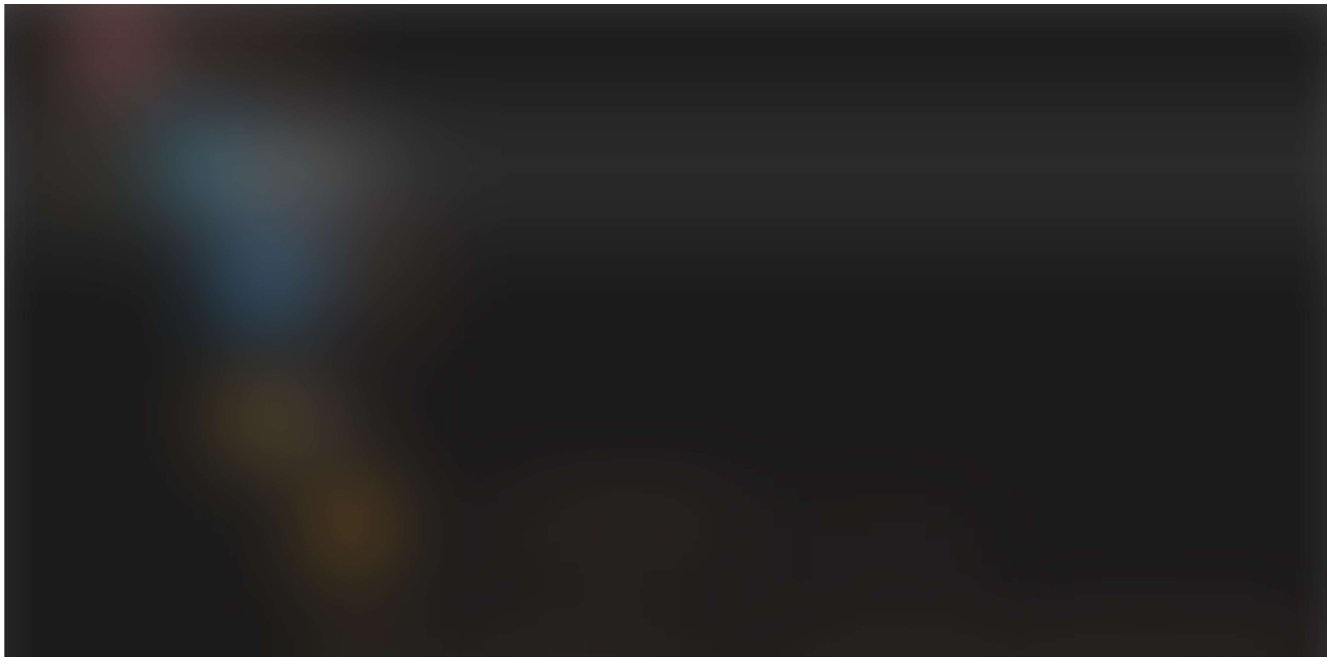
#2 Lazy loading

Lazy loading is one of the easiest ways to increase performance and see significant results quickly. I would say that this effort is the low-hanging fruit.

The best place to start is on the router. Let’s create an optimized production build;

```
$ yarn build
```

If you navigate to the ‘build/static’ folder that got created for us, you can see that we have three chuck JS files and a license file, see Figure 2.



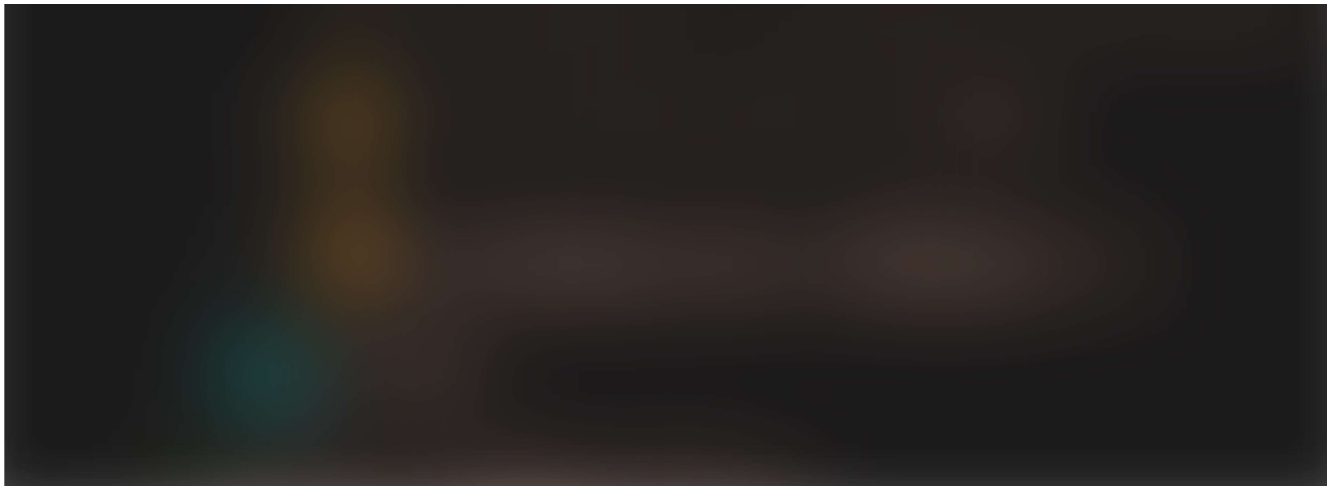


Figure 2. CRA Production builds before utilizing lazy loading.

Now update the code to include lazy loading. The Suspense component will be showing during the loading phase of the component and importing the component using the lazy method will ensure the component is only loading once used.

The change is updating how we import our component from;

```
import MyPage from './pages/MyPage/MyPage'
```

To;

```
const MyPage = lazy(() => import('./pages/MyPage/MyPage'))
```

The Suspense component includes a fallback while the component is loading. Take a look at the complete code;

```
import React, { FunctionComponent, lazy, Suspense } from 'react'
import { BrowserRouter as Router, Route, Switch } from 'react-router-
dom'
import { RecoilRoot } from 'recoil'
import App from './App'

// Normal
// import MyPage from './pages/MyPage/MyPage'

// Lazy loading
```

```
const MyPage = lazy(() => import('./pages/MyPage/MyPage'))

const AppRouter: FunctionComponent = () => {
  return (
    <Router>
      <RecoilRoot>
        <Suspense fallback={<span>Loading...</span>}>
          <Switch>
            <Route exact path="/" component={App} />
            <Route exact path="/MyPage" component={MyPage} />
            <Redirect to="/" />
          </Switch>
        </Suspense>
      </RecoilRoot>
    </Router>
  )
}
```

Run `$ yarn build` again. You can see that the build script broke down our bundle chunks from 3 files to 4 files due to the lazy loading we put in place. Figure 3.

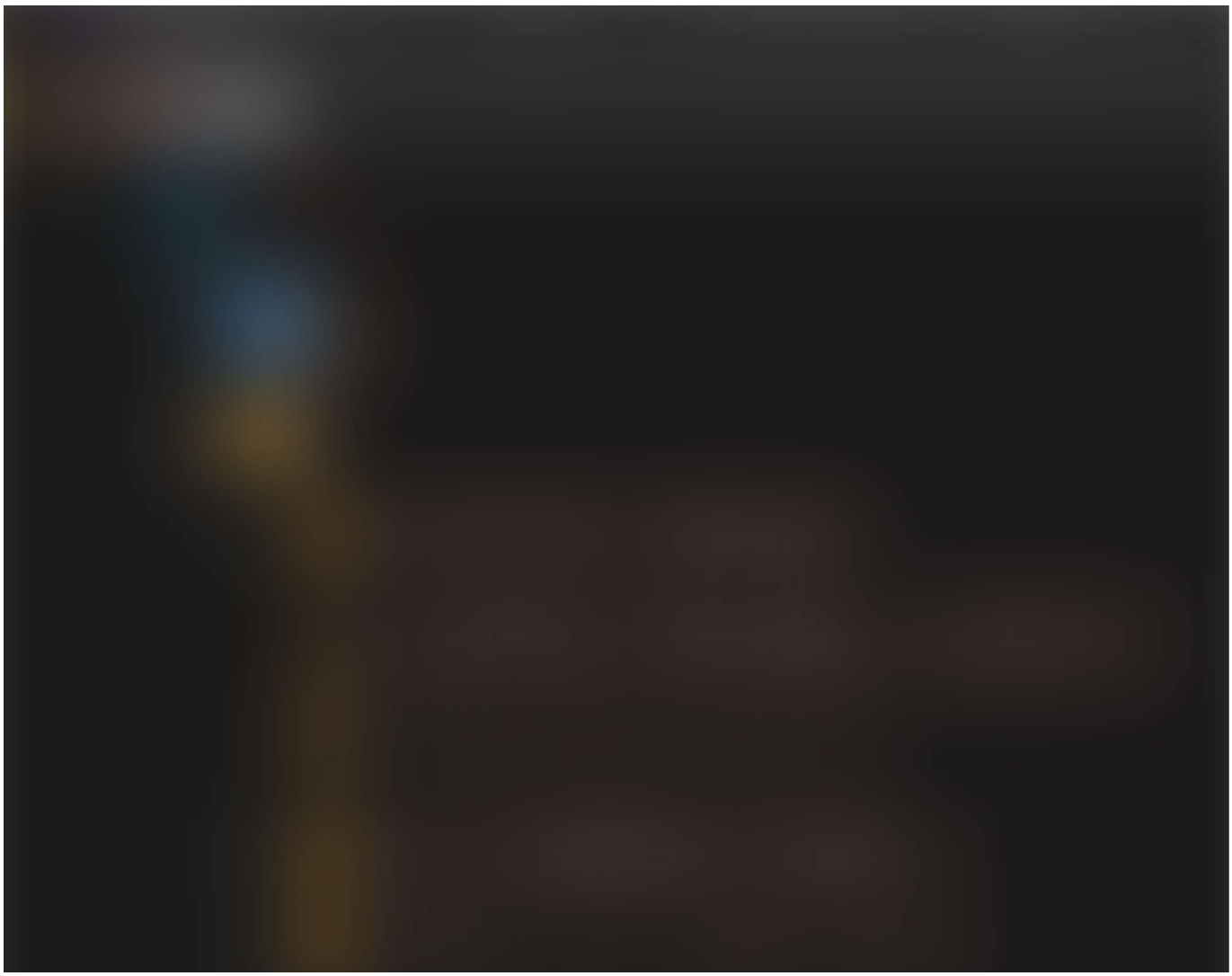


Figure 3. CRA Production build optimized.

When you `$ yarn start`, you don't see the *fallback* loading message, because it happens too quickly as our page components only include the name of the page, however, you can see an example in the wild on my site for much of the efforts you see in this article, where I placed lazy loading for the login page and an image spinner for the *fallback*. Navigate the pages and you can see the spinner the first time you navigate to the login page, if not you can reduce the internet connection DevTools.

If you use Chrome DevTools or Charles (see my debugging article), you can actually measure the results. It depends on the size of your app, and what you doing, but this simple method can gain you as much as a few seconds.

> ***Note:*** *A bit of a caveat about Lazy Loading. There are times that it won't make sense to do lazy loading on all pages, because pages are light. Splitting and loading multiple bundle files takes longer.*
>
> *You need to experiment and it's on a case-to-case basis. Also it may be good to wrap these lazy loading only on some components. For instance, a login member area that is not used by all users. While other pages can be loaded together. The best approach is to experiment and take profiling and throttle the network speed to find out what is the best user experince. It's not one size fits all sort of things.*

More information on the React docs (https://reactjs.org/docs/code-splitting.html#route-based-code-splitting).

#3 Pre-render static pages

CRA is based on a single-page application (SPA) and that's great because you don't get page refresh and the experience feels like you are inside of a mobile App.

The pages are meant to be rendered on the client-side. CRA doesn't support server-side rendering (SSR) out of the gate. There are ways to configure the routing etc and get CRA to work as SSR, but that may involve ejecting and maintaining configuration on your own and may not be worth the effort.

If you're build something that needs SSR it's better to just work with a different React library that is already configured out of the box with SSR such as Next.js framework, Razzle, or Gatsby (include a pre-render website into HTML at build time).

> *If you want to do server rendering with React and Node.js, check Next.js, Razzle or Gatsby. Create React App is agnostic of the backend, and only produces static HTML/JS/CSS bundles. — https://github.com/facebook/create-react-app*

Check my article here to set a minimum starter Nextjs SSL project for d3js & TypeScript project.

With that being said, with CRA we can do pre-render, which is the closest you can get to SSR at this time, see CRA docs: https://create-react-app.dev/docs/pre-rendering-into-static-html-files/

There are many options to generate HTML pages for each route or relative link. Here are a few;

- react-snap
- react-snapshot
- Webpack Static Site Generator Plugin

I recommend react-snap (https://github.com/stereobooster/react-snap) is more popular with 4K stars on Github and it works seamlessly with CRA. react-snap uses Puppeteer to create pre-rendered HTML files of different routes in your application automatically. The biggest benefit is that once we use react-snap, the App doesn't care if the JS bundle is successfully loaded or not because each page we set would be on its own. Keep in mind that for each page to load on its own some bundle may have redundant code so it does come with a price.

Step 1: To get started;

```
yarn add --dev react-snap
```

Step 2: Next, add the `inlineCss` and the script run;

```
// package.json
"scripts": {
  ...
  "postbuild": "react-snap"
},
```

Step 3: the static HTML rendered almost immediately, they are unstyled by default and can cause an issue of showing a "flash of unstyled content" (FOUC). This can be especially noticeable if you are using a CSS-in-JS library to generate selectors since the JavaScript bundle will have to finish executing before any styles can be set.

`react-snap` uses another third-party library under the hood, `minimalcss` to extract any critical CSS for different routes. You can enable this by specifying the following in your `package.json` file:

```
// package.json

"scripts": {
  ...
  "postbuild": "react-snap"
},
"reactSnap": {
  "inlineCss": true
},
```

Step 4: Now in `src/index.tsx` is where we will be hydrating and we can also register for precaching there as well with this: serviceWorker.*register*(). More about precache in the next section.

```
// src/index.tsx

import React from 'react'
import { hydrate, render } from 'react-dom'
import './index.scss'
import AppRouter from './AppRouter'
import * as serviceWorker from './serviceWorker'
```

```
const rootElement = document.getElementById('root')
if (rootElement && rootElement!.hasChildNodes()) {
  hydrate(<AppRouter />, rootElement
  serviceWorker.register()
} else {
  render(<AppRouter />, rootElement)
}
```

Step 5: Now run `$ yarn build` and `postbuild` will be called automatically via npm script that is configured in CRA. You should see successful results;

$ react-snap
✅ crawled 1 out of 3 (/)
✅ crawled 2 out of 3 (/404.html)
✅ crawled 3 out of 3 (/MyPage)
✨ Done in 29.29s.

Open the build folder and you can see the static pages created for us automatically;

Figure 4. static pages

> ***Note:*** *running pre-render and serving static pages is not necesserly always the best approach, it can actually create undesire experience for your users as each page will load and the component load will be distributed across pages. For light Apps it may be better to wait half a second and all the content loads and no more wait time vs wait a bit on each page loads. You need to test and see for yourself but be aware of this feature.*

Step 5: to spin off a local production build locally, run the CRA template run script, it's using serve library, so you don't even need to install or configure 'pacakge.json' if you using CRA MHL template.

- Serve ($ yarn add serve) adds a local server. CRA scrips have a run script to publish your App. It will generate a folder called 'build'. We want to be able to test our build code before going to production. We can build and create the local server on port 5000 (default for serve) using the build folder with one command: `$ yarn serve -s build` I already included this library and set a run script `$ yarn publish:serve` to build and serve.

```
$ yarn build:serve
```

This run script is already included in 'package.json';

```
// package.json
"build:serve": "yarn build && serve -s build",
```

Another option beside `serve` is to run the App with NodeJS and Express using the MERN stack or whatever server you choose.

I want to point out that another BIG reason to use pre-render is the need for static pages beyond optimizing, Search Engine Optimization (SEO). If you pre-rendering pages and want to generate a different title, description, metadata, etc, for each page due to SEO reasons, or need sharing individual pages via social media, check react-helmet that can help to set a unique header for each React page component;

How to get react-helmet to generate a title for each page?

Step 1: install react-helmet and types for TS;

```
$ yarn add react-helmet @types/react-helmet
```

Step 2: Now, we can refactor our *MyPage.tsx* and add the Helmet component;

```
import Helmet from 'react-helmet'

render() {
  return (
    <div className="MyPage">
      <Helmet>
        <title>My Page</title>
      </Helmet>
      {this.state.name} Component
    </div>)
}
```

Notice that in our code level that the state store the name of the page, which is extracted from React Router, so we need ***.replace('/', '')*** for *this*.state.name because in case the user refreshes the static page it will have 'about/' at the end.

```
constructor(props: IMyPageProps) {
  super(props);
  this.state = {
    name: this.props.history.location.pathname.substring(
      1,
      this.props.history.location.pathname.length
    ).replace('/', '')
  }
}
```

Now if you do 'view source' it will have the title once you click the 'MyPage' link.

#4 precache — work offline

Be able to go offline is a core functionality of PWA. We can do that with a serviceWorker.

CRA include serviceWorker, you must have seen this code inside the index file;

```
// If you want your app to work offline and load faster, you can
change
// unregister() to register() below. Note this comes with some
pitfalls.
```

```
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister()
```

What does it mean?

CRA includes `workbox` for production build (https://developers.google.com/web/tools/workbox/modules/workbox-webpack-plugin).

To enable the feature just change the 'serviceWorker' state to *register;*

```
serviceWorker.register()
```

We already added the `serviceWorker` to when we build for production, in the previous section;

```
const rootElement = document.getElementById('root')
if (rootElement && rootElement!.hasChildNodes()) {
  hydrate(<AppRouter />, rootElement)
  serviceWorker.register()
} else {
  render(<AppRouter />, rootElement)
}

// serviceWorker.unregister()
```

Now when you build again (`$ yarn build`). New file appears: '**build/precache-manifest.[string].js**'

See Figure 5;

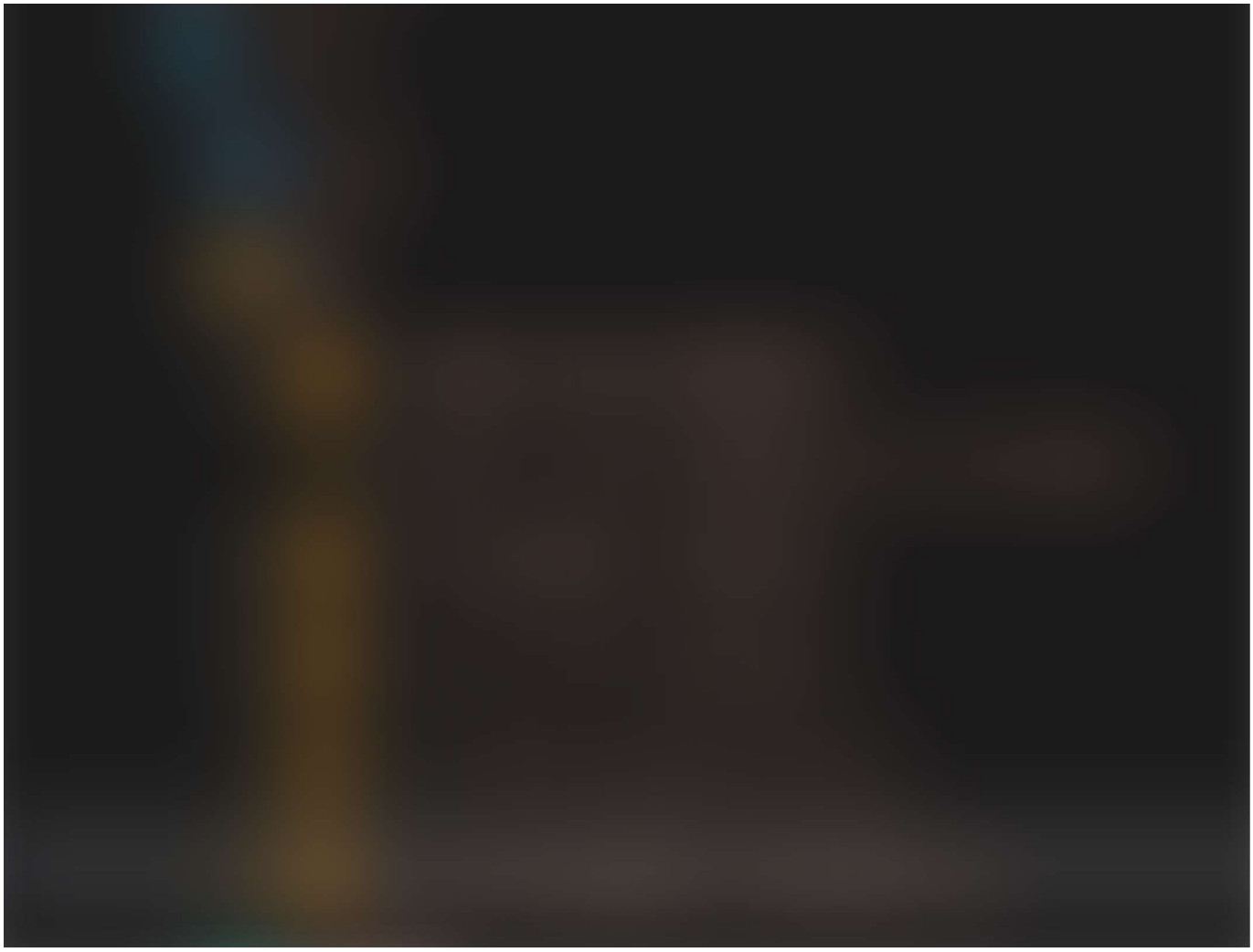


Figure 5. runtime-main bundle file was added to our static folder.

To see the worker in action you need to run a publish build again (`$ yarn build:serve`);

Take a look at the Chrome DevTools network tab under the size column you can see it says ('ServiceWorker'), see Figure 5;

Figure 6. precache ServiceWorker shows in Chrome DevTools Network

You can now simulate offline experience either by turning off your network connecting or in the Chrome DevTools Network tab, check the Offline checkbox. Refresh the App and it still works!

How is your App working offline?

Workbox default precaching strategy for CRA is `CacheFirst`. Static assets retrieved from the service worker cache mechanism and on fail — a network request is made.

Workbox supports different strategies such as `CacheOnly`, `NetworkFirst` etc, but CRA may need to be ejected to use a different strategy.

Read more about this feature on CRA: https://create-react-app.dev/docs/making-a-progressive-web-app/

#5 Code Splitting

When we used Lazy loading we were able to break down our JS Bundles into multiple chunks and only serve them when they are needed.

Dynamically import

We can do more. CRA handles the code-splitting task with WebPack. We can tell Webpack to split our JS bundle even more and dynamically import these modules. Let's take a look;

Create this file `src/page/MyPage/math.tsx;`

```
// src/page/MyPage/math.tsx

export function square(x: number) {
  return x * x
}

export function cube(x: number) {
  return x * x * x
}

export function add(x: number, y: number) {
  return x + y
}
```

To use the add method we normally would write our code like so;

```
import { add } from './math'
console.log(add(1, 2))
```

However, if we want to split the code so the js bundle only be retrieved once it's needed and only bundle what's used, we can do this;

```
import("./math").then(math => {
  console.log(math.add(1, 2))
})
```

Let's create an actual working example. We can have a state with variable results and one-click update, the results using the math add function. Take a look at *MyPage.tsx* code:

```
// src/page/MyPage/MyPage.tsx

render() {
  const onClickHandler = (event: React.MouseEvent) => {
    event.preventDefault()
    import('./math').then((math) => {
      this.setState({
        results: (math.add(1, 2))
      })
    })
  }
  return (
    <div className="MyPage">
      <Helmet>
        <title>My Page</title>
      </Helmet>
      {this.state.name} Component
      <Button type="submit" onClick={onClickHandler}>
        Math.add
      </Button>
      {this.state.results}
    </div>
  )
}
```

Or better yet, let's add the result for every click;

```
import('./math').then((math) => {
  this.setState(prevState => {
    const newState = prevState.results + (math.add(1, 2))
    return ({
      ...prevState,
      results: newState
    })
  })
})
```

Now build the production code and run it on our local machine. `$ yarn build:serve`

> *Keep in mind our build is set to precache so you would need to force clearing cache on a Mac it's with Shift+Refresh to clears your cache, or you can find a fancy plugin on the Chrome Web Store. Otherwise you may be serving old pages and pulling your hair off. Otherwise, you may be serving old pages and pulling your hair off.*

See Figure 7,

Figure 7. 'Math.add' code-splitting running production build locally.

Notice that our js bundle increase in one and that bundle file will be retrieved when we call the MyPage and the button instead of having the user wait for the bundle when we first load the page. Figure 8;

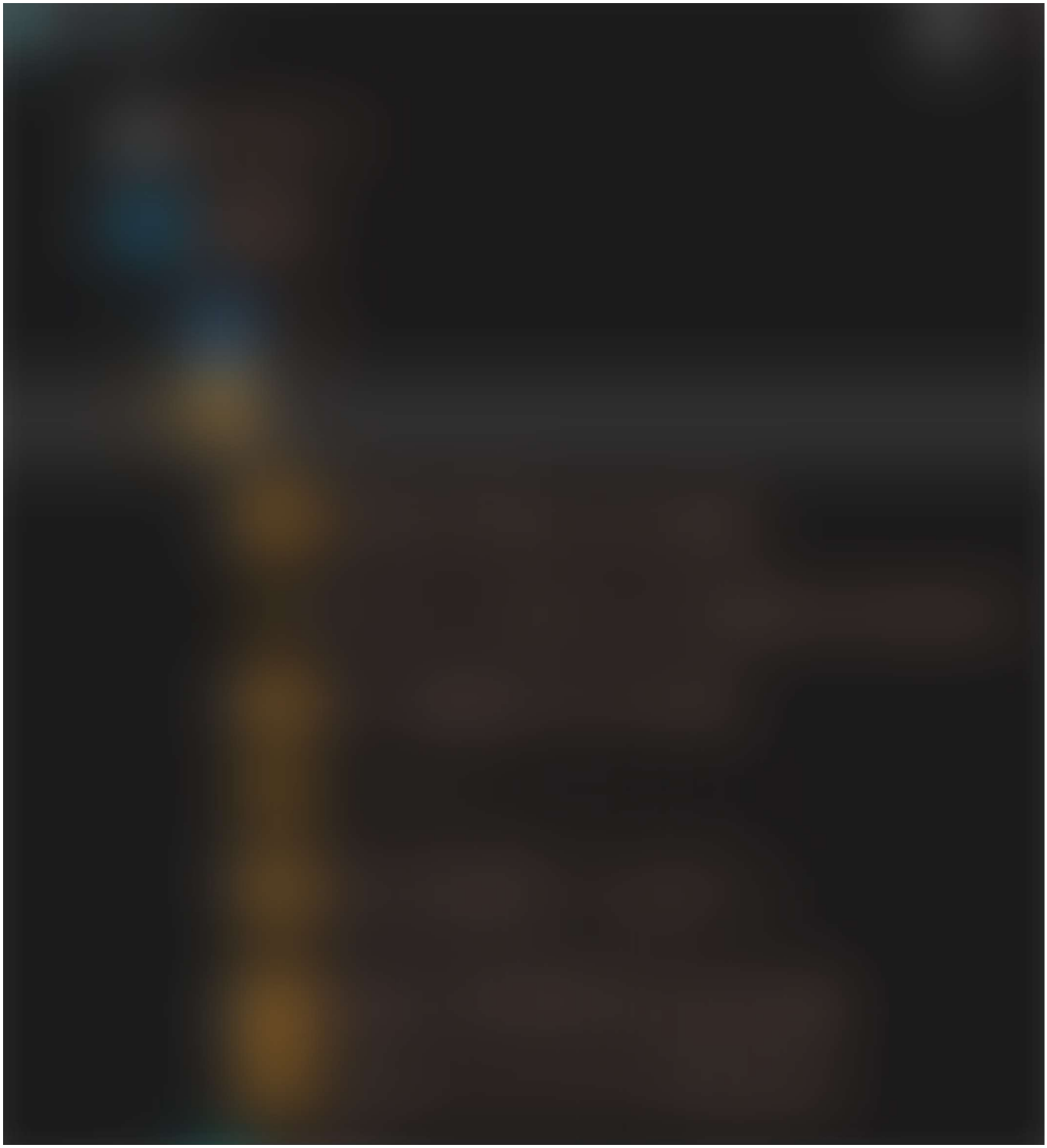


Figure 8. JS bundle after code splitting.

Module Federation

Can we do even more? The answer is yes and no.

Right now Webpack is at version “webpack”: “4.42.0”, for CRA (see node_modules/react-scripts/package.json). However, the current version of Webpack is

v5.4.0 as of October 2020 and it included tons of performance improvement. One major one involved Module Federation.

Module Federation allows importing remote Webpack builds to your application. with Webpack v5, we can have these chunks imported not just from our project but different origins (project). You can read more about it here: https://webpack.js.org/guides/build-performance/.

You can eject and do that on your own, and build on the work done so far, follow the GitHub PR.

#6 Tree shaking

Tree shaking (https://webpack.js.org/guides/tree-shaking/) is a term used in JavaScript context for the removal of dead code. There is much to do when it comes to the removal of the dead code.

When I say dead code it can be two things:

1. **Never executed code —** code that can never be executed during run-time.
2. **The result never used —** code is executed but the result is never used.

For example in the App we built, I defined recoil inside of 'src/AppRouter.tsx'

```
<RecoilRoot>
    <Suspense fallback={<span>Loading...</span>}>
      <Switch>
        <Route exact path="/" component={App} />
        <Route exact path="/MyPage" component={MyPage} />
        <Redirect to="/" />
      </Switch>
    </Suspense>
</RecoilRoot>
```

but I am never using the feature for anything.

Now, if we dig inside our JS Bundles and see what's happening, we can see that Recoil is using almost 38.19kb. Figure 9;

```
$ source-map-explorer optimize-ts/build/static/js/[my chunk].chunk.js
```

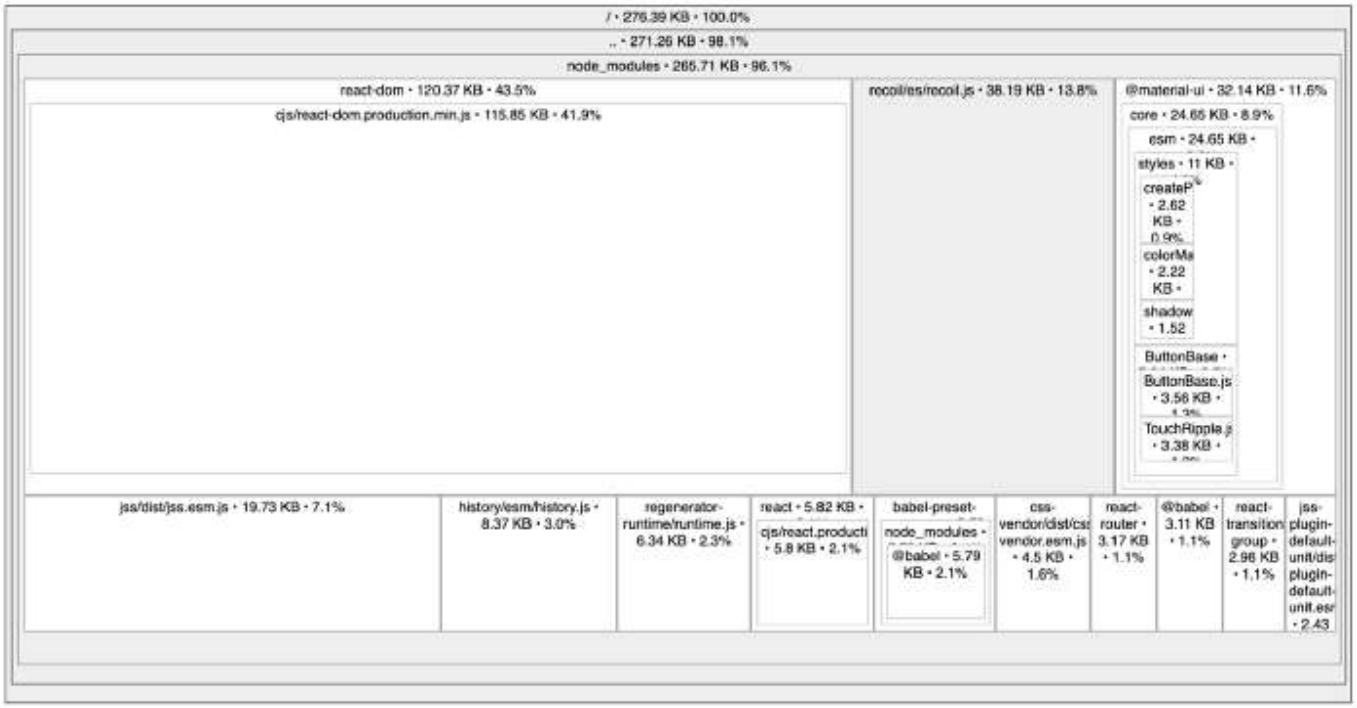
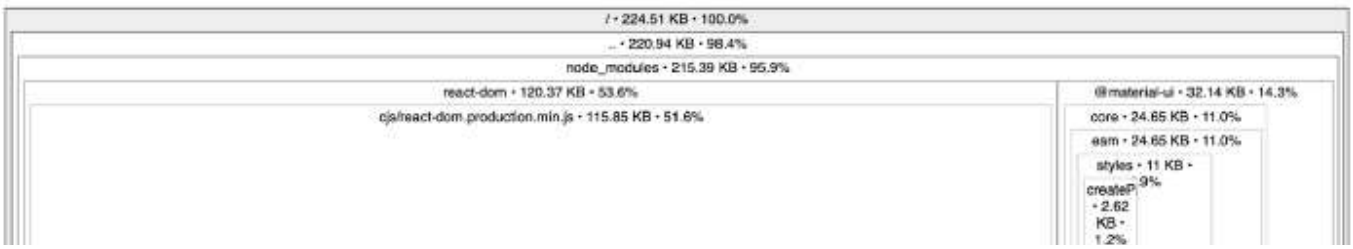


Figure 9. source map of our bundle with Recoil.

If we refactor the code and remove recoil and build again.

```
<Router>
    <Suspense fallback={<span>Loading...</span>}>
      <Switch>
        <Route exact path="/" component={App} />
        <Route exact path="/MyPage" component={MyPage} />
        <Redirect to="/" />
      </Switch>
    </Suspense>
</Router>
```

Now the code went down from 276 kb to 224.51 kb and Recoil is not included.



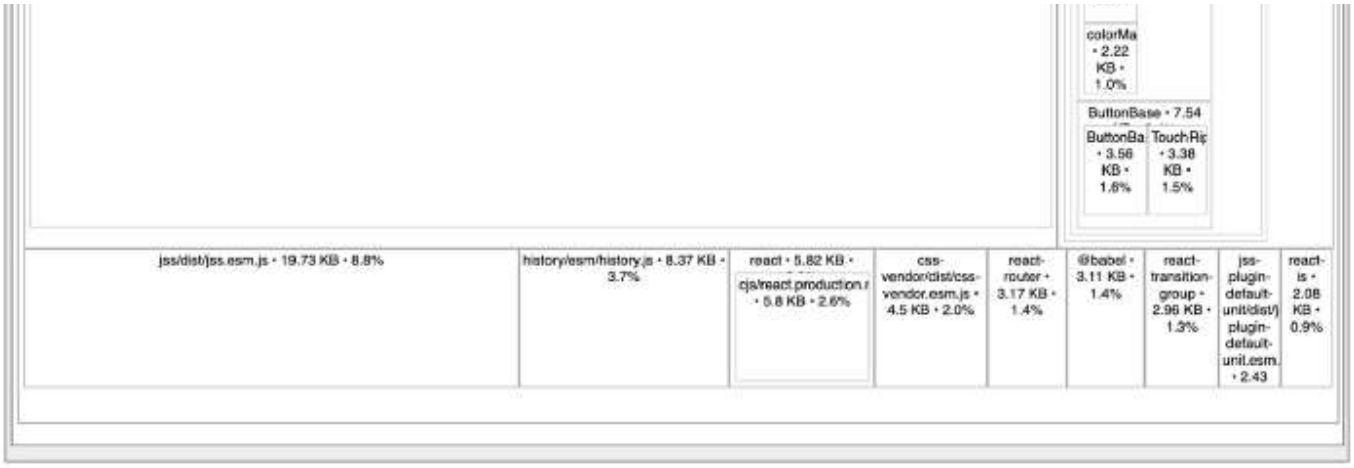


Figure 10. source map of our bundle without Recoil.

CRA includes Webpack out of the box to bundle our app with some settings already set for us.

You can see the optimization tag that includes `compression` and `OptimizeCSSAssetsPlugin`. If you need to make changes you would need to eject and maintain on your own. However even without ejecting there are things we can do.

To better understand what's happening under the hood in CRA open webpack config files, which are placed inside of 'react-scripts';

```
$ open node_modules/react-scripts/config/webpack.config.js
```

Or for dev server;

```
$ open node_modules/react-scripts/config/webpackDevServer.config.js
```

Let's talk about side effects.

> ***Note:*** *A side effect is a state change that is observable outside the called function other than its return value. Examples of a* ***side effect****: change value of an external variable or object property (global variable, or a variable in the parent function scope chain), an import*

> *statement with no specifiers (i.e.* `import 'someLib'`*), logging to the console, data fetching, setting up a subscription, or manually changing the DOM.*

If you using WebPack, we can instruct WebPack on how to handle libraries. In fact, the majority of npm libraries have side effects.

If you look inside the CRA WebPack `webpack.config.js` the libraries imported are set with the 'sideEffects: true' flag and this is set per library.

Having imports for libraries we are not using increases the size of our code (JS bundles). These imports should be removed.

Import required modules vs import statement with no specifiers

Let's take an example. If I am adding an import code that I am not using for instance and importing the feature I need, for instance, *'useRecoilValue'*.

```
// src/page/MyPage/MyPage.tsx

import { useRecoilValue } from 'recoil'
```

CRA react-script allow me to start and build prod, but I will be getting a warning message, Figure 11;

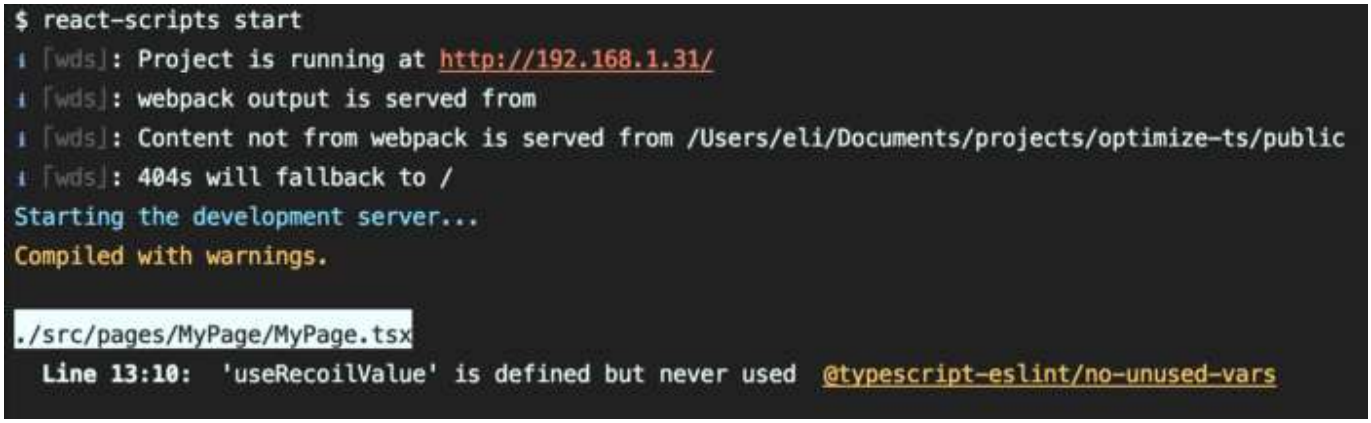


Figure 11. Warning for unused code.

That not good to have these warnings but it won't be the end of the world.

In this specific case, Terser (https://github.com/terser/terser) which WebPack is using is trying to figure this out and decided that the code is not needed, so although it creates a warning it doesn't include the library inside our build JS Bundles. The code is not included because the bundle is optimized and the library removed (you can verify this by checking the source maps).

But don't rely on the Terser to always be able to figure this out, when it can't it will include the deed code.

However, if I am using an import statement with no specifiers (`import 'recoil'`);

```
// src/page/MyPage/MyPage.tsx

import 'recoil'
```

The code will compile, build, and even ESlint for us, without any warnings, but it will include the entire Recoil library.

The reason is that Webpack treated the import statement with no specifiers (`import 'recoil'`) as a side effect and as if we intended to do that, so it will include Recoil in our source map.

> ***Note:*** *Only include the modules you require and avoid import statement with no specifiers.*

You can also fine-tune and add `"sideEffects"` property to your project's `package.json` file to tell WebPack how to handle side effects without ejecting, read more this and tree shaking over the WebPack docs here: https://webpack.js.org/guides/tree-shaking/

#7 Reduce Media Size

We talked about tree shaking and reducing the JS bundle size, however besides bundle size, other resources are often used in an App that can take lots of resources, these are media files such as images, videos, audio documents, and syncing with large data.

To optimize resources there are plenty of tools to use out there. If you on Adobe products you have Photoshop with the 'save for the web' option that you can ensure the size of

your images stay low. Adobe Premiere can encode your videos for different devices and with different settings and the list goes on and on.

You can use a library that checks the user network speed and deliver different resources based on the user's connection, that's a common practice for video delivery.

> *Ideally we want to upload our resources during run-time vs compile time, because we don't want the user to wait for a resource.*

Another thing is SVG. SVG is vector based and it's great. It gives the user that crisp graphic look in any resolution screen size, however, it comes with a price. The way React works to increase performance is to reduce the number of requests to the server.

Importing images that are less than 10 kb returns a data URI instead of the actual SVG file, see React Docs (https://create-react-app.dev/docs/adding-images-fonts-and-files/). This is by default ignored in CRA development but you can see this behavior on the prod build, see 'IMAGE_INLINE_SIZE_LIMIT' https://create-react-app.dev/docs/advanced-configuration.

Having tons of SVG graphics around can easily bloat your App. It's better to gather it all as one jpg image file and use an image sprite. Loading a single image is quicker than loading individual images one by one.

#8 Prefetching

You might have used Higher-Order Components (HOC) in React to enhance a component capability (https://reactjs.org/docs/higher-order-components.html). We can take a similar approach when it comes to our JS bundle.

We want to first load the page and then retrieve the JS Bundle, so we get to display the page ASAP.

Let's take a look. If you build a prod version (`$ yarn build:serve`) and test in Chrome DevTools, take a look at the JS Bundle chunks hierarchy, they are at the top. Figure 12.

Figure 12. hierarchy is not set for JS Bundle.

We want to move these bundles to the bottom. To do that we can use Quicklink (https://github.com/GoogleChromeLabs/quicklink). Quicklink attempts to make navigations to subsequent pages load faster by using techniques to decide what to load first. Let's install it;

```
$ yarn add -D webpack-route-manifest
$ yarn add quicklink
```

Notice that your package.json was updated with this;

```
"devDependencies": {
  "webpack-route-manifest": "^1.2.0"
}
```

In our case, we use React CRA SPA, we will be using the React HOC where we want to add prefetching functionality to the pages we lazy load. To do that just an empty 'option' object and wrap our component with the `withQuicklink` HOC;

```
<Route exact path="/MyPage" component={withQuicklink(MyPage,
options)} />
```

Take a look at the refactored '*src/AppRouter.tsx*';

```
// src/AppRouter.tsx

import React, { FunctionComponent, lazy, Suspense } from 'react'
import { BrowserRouter as Router, Route, Switch, Redirect } from
```

```
'react-router-dom'
import { RecoilRoot } from 'recoil'
// @ts-ignore
// eslint-disable-next-line import/extensions
import { withQuicklink } from 'quicklink/dist/react/hoc.js'
import App from './App'
import ScrollToTop from './components/ScrollToTop/ScrollToTop'

// Lazy loading
const MyPage = lazy(() => import('./pages/MyPage/MyPage'))

const options = {
  origins: []
}
const AppRouter: FunctionComponent = () => {
  return (
    <Router>
      <ScrollToTop />
      <RecoilRoot>
        <Suspense fallback={<span>Loading...</span>}>
          <Switch>
            <Route exact path="/" component={App} />
            <Route exact path="/MyPage" component=
{withQuicklink(MyPage, options)} />
            <Redirect to="/" />
          </Switch>
        </Suspense>
      </RecoilRoot>
    </Router>
  )
}

export default AppRouter
```

After implementing the logic, as you can see the HOC worked and now our chunks are at the bottom, Figure 13.

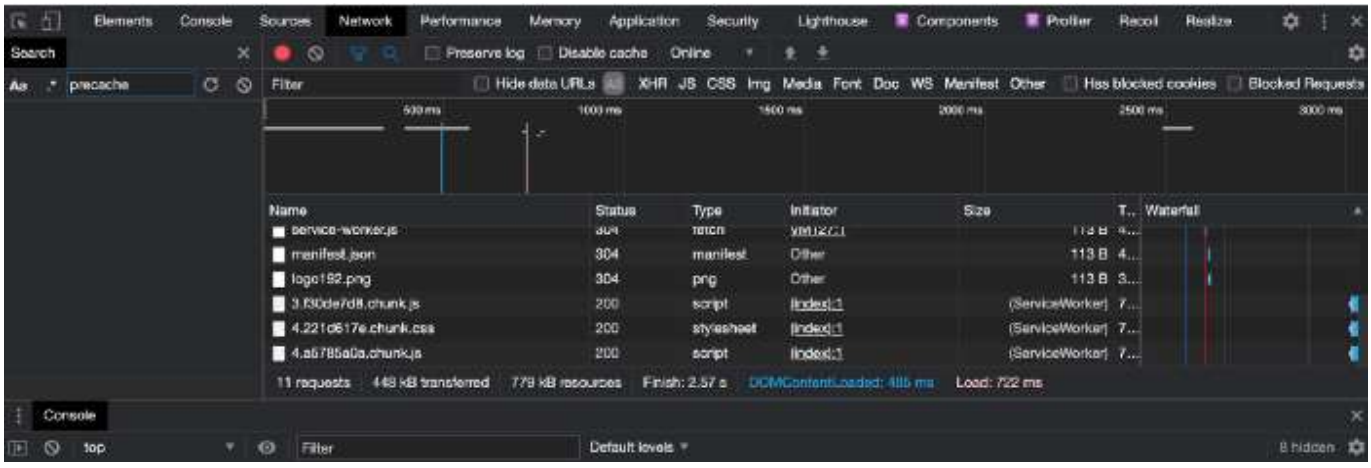


Figure 13. hierarchy set for About Page Component.

#9 Clean unused side effects event handlers

Set side effects in the 'useEffect' hook

For example, if we want to scroll to the top of every page update using the browser API. We don't even need to write a class we can just wrap that code inside of `useEffect` the hook in a React function.

```
// src/components/ScrollToTop/ScrollToTop.tsx

import { useEffect } from 'react'
import { useLocation } from 'react-router-dom'

export default function ScrollToTop() {
  const { pathname, search } = useLocation()

  useEffect(
    () => () => {
      try {
        window.scroll({
          top: 0,
          left: 0,
          behavior: 'smooth',
        })
      } catch (error) {
        // older browsers fallback
        window.scrollTo(0, 0)
      }
    },
    [pathname, search]
  )
  return null
}
```

> ***Note:*** *Side effects needs to be wrapped in* `useEffect`*. If not done right, side effects can run on every render and turn into a memory leak.*

We also need to refactor '*src/AppRouter.tsx*' to include the component inside of the React Router tag;

```
// src/AppRouter.tsx
```

```
import ScrollToTop from './components/ScrollToTop/ScrollToTop'

const AppRouter: FunctionComponent = () => {
  return (
    <Router>
      <ScrollToTop />
      ...
    </Router>
  )
}
```

This specific side effect doesn't need any clean-up because we are not attaching any events.

Side effect cleanup

Leaving event handlers around after a component is unmounted can cause memory leaks.

Luckily for us, React Components clean React-based event handlers once the component is unmounted automatically.

However, if we need to use the browser scroll API event listener;

```
window.addEventListener('scroll', scrollHandler)
```

You will have to remove the event manually, as React won't remove them for you.

```
window.removeEventListener('scroll', scrollHandler)
```

You can read more about this in the React docs (https://reactjs.org/docs/hooks-effect.html).

Install modules instead of global imports

Many libraries enable import modules instead of the entire library. Doing that can give reduce the bundle size need to run the App significantly.

For example, create a simple React with d3 library code that will draw a Rectangle.

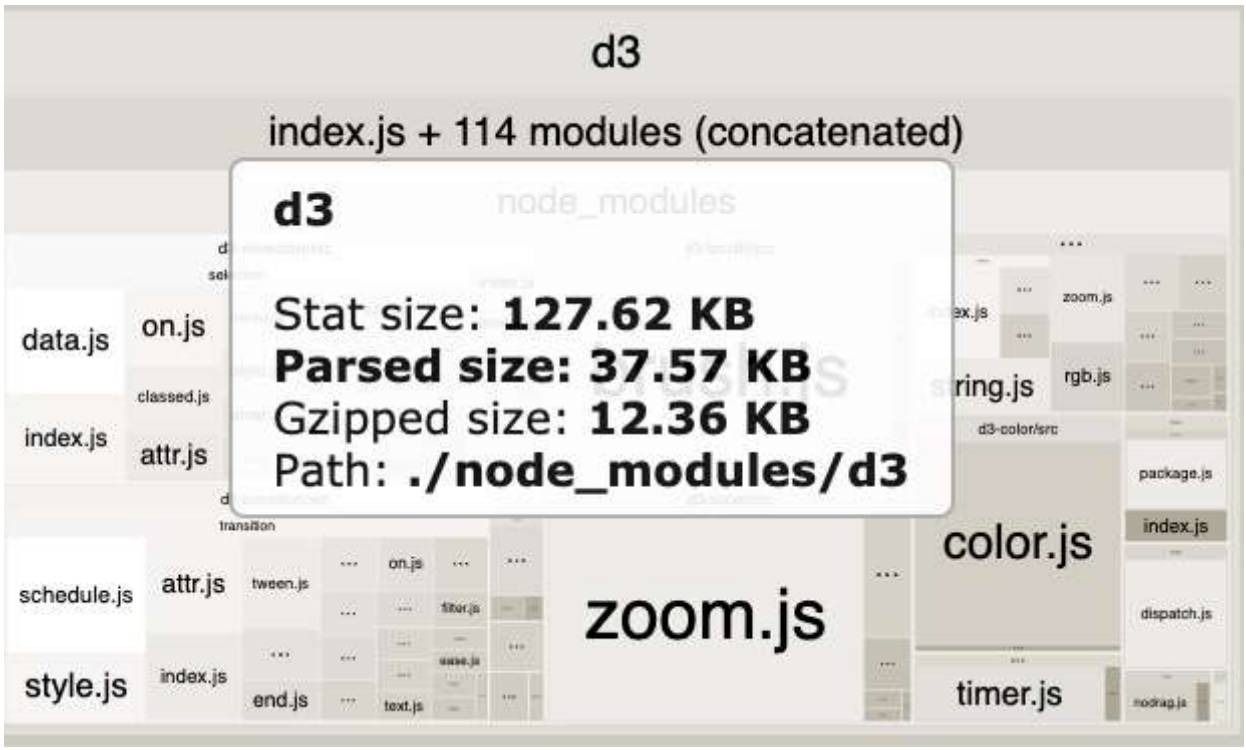
Install both the d3 global library as well as only the module we need (selection);

```
$ yarn add d3-selection @types/d3-selection

$ yarn add d3 @types/d3
```

If we create the same code just changing it, we use the global library or the module. The code is almost identical, however the footprint change;

```
// src/component/Rectangle/Rectangle.tsx

import React, { useEffect, RefObject } from 'react'
import * as d3 from 'd3'

const Rectangle = () => {
  const ref: RefObject<HTMLDivElement> = React.createRef()

  useEffect(() => {
    draw()
  })

  const draw = () => {
    d3.select(ref.current).append('p').text('Hello World')
    d3.select('svg')
      .append('g')
      .attr('transform', 'translate(250, 0)')
      .append('rect').attr('width', 500)
      .attr('height', 500)
      .attr('fill', 'tomato')
  }

  return (
    <div className="Rectangle" ref={ref}>
      <svg width="500" height="500">
        <g transform="translate(0, 0)">
          <rect width="500" height="500" fill="green" />
        </g>
      </svg>
    </div>
  )
}

export default Rectangle
```

The d3 parsed size is 37.57 kb.



To better understand what the different sizes stand for, take a look below:

· **Stat size —** the size of the input, after webpack bundling, but before optimizations (such as minification).

· **Parsed size —** the size of the file on disk after optimizations. It is the effective size of the JavaScript code parsed by the client browser

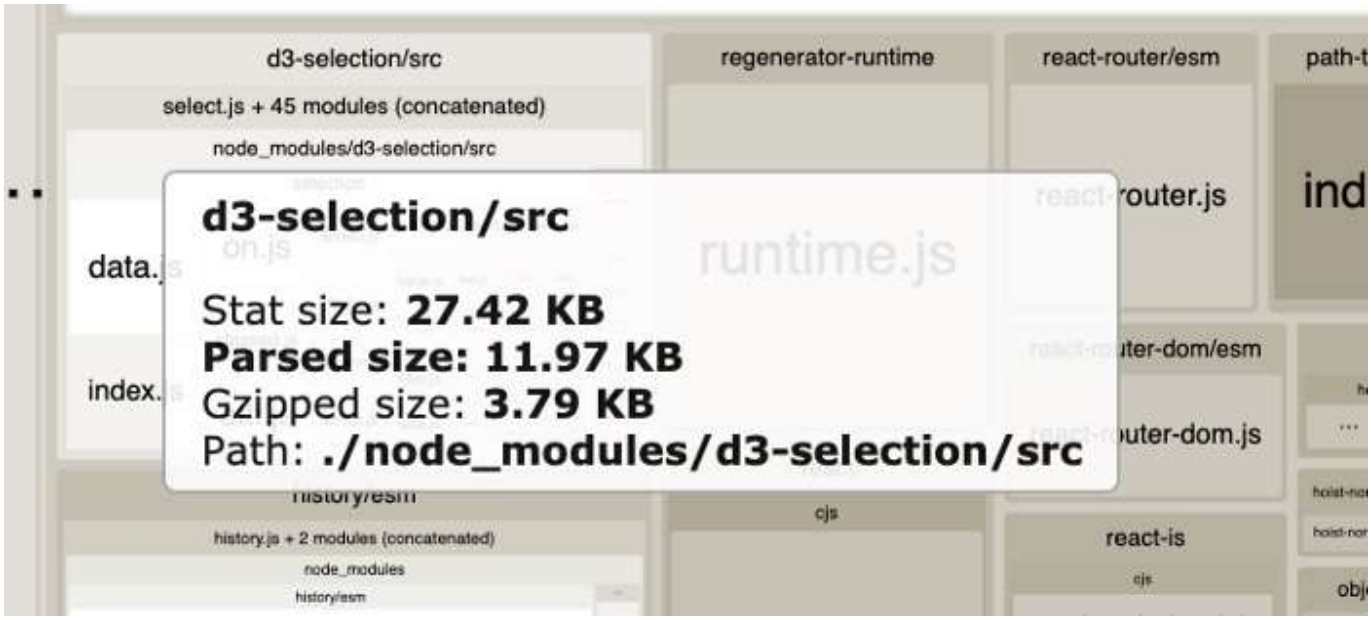
· **gzip size —** the size of the file after gzip usually transmitted over the network. Keep in mind that the gzip will need to unzip once reached the client (browser).

Now, let's change the library to only include 'd3-selection' module;

```
// src/component/Rectangle/Rectangle.tsx

import React, { useEffect, RefObject } from 'react'
import { select } from 'd3-selection'

const Rectangle = () => {
  const ref: RefObject<HTMLDivElement> = React.createRef()

  useEffect(() => {
```

```
      draw()
    })

    const draw = () => {
      select(ref.current).append('p').text('Hello World')
      select('svg')
        .append('g')
        .attr('transform', 'translate(250, 0)')
        .append('rect').attr('width', 500)
        .attr('height', 500)
        .attr('fill', 'tomato')
    }

    return (
      <div className="Rectangle" ref={ref}>
        <svg width="500" height="500">
          <g transform="translate(0, 0)">
            <rect width="500" height="500" fill="green" />
          </g>
        </svg>
      </div>
    )
  }

  export default Rectangle
```

The d3 parsed size was reduced from 37.57 kb to 11.97 kb!



My final Notes

It's great to know how much can be done to optimize our App. With some effort on our end, we can increase performance and improve user experience and the results are

noticeable even on a small App. Additionally, doing optimization is really gaining awareness of what's happening and how it all works as well as what needs to be improved.

With that being said, optimizing our App is all about testing and tweaking and testing results again, to fine-tune as there are use cases that it won't make sense to do all or any of these optimizing efforts. There is a trade-off for each feature we add.

You need to experiment and each feature needs to be checked on a case-to-case basis. The best approach is to record memory profiling, throttle the network connection, check bundles, go offline, try on different network speed to find out what is the best user experience.

Always keep in mind that your development and production builds are different than and don't assume it will just work the same. It's not one size fits all sort of things.

Let's Recap

In this article, I showed you how you can reduce the amount of memory your app is consuming to the minimum, reduce bundle file sizes, load resources only once in usage, decrease wait time to view content, increase performance and ensure it works anytime, anywhere even offline.

We also opened the hood to better understand what's happening; we installed Analyzer Bundle tools, as well as look at other settings, so you can configure your App better instead of using the default settings.

We broke down the process into the following;

1. **PureComponent and React.memo() —** gain performance.
2. **Lazy loading —** break your JS bundles and serve once needed avoiding wait time to see your content.
3. **Prerender —** almost static HTML.
4. **Precache —** have your App work offline.
5. **Code Splitting —** split JS Bundle even more.

6. **Tree shaking —** dead code removal configure 'package.json' file.
7. **Reduce Media size —** reduce the size of the media resources with image sprite.
8. **Prefetching —** set loading hierarchy.
9. **Clean unused side effects event handlers —** avoid memory leaks.
10. **Install modules instead of global imports —** reduce bundle size

You can download the final and complete project used for this article from here: https://github.com/EliEladElrom/react-tutorials > optimize-ts.

Where to go from here

As I said in the beginning, this subject is big enough to fill several articles and a whole book, but my aim was not to just give you a kitchen sink, but a good starting point and the main and most important ways you need to be aware of.

For instance, I didn't cover network traffic techniques. Additionally, if you render a large list you can use react-window (https://github.com/bvaughn/react-window), which will only render the *part* of a large data set (just enough to fill the viewport), react-infinite-scroll-component or react-paginate, and the list goes on and on.

Keeping your App free of lint errors and warnings can improve performance as well. One example is setting keys on UI elements to improve performance, check common ESLint / Lint Errors & how to fix articles.

Lastly, check my React + d3 optimizing article.

Optimizing React Tutorial React Reactjs Typescript