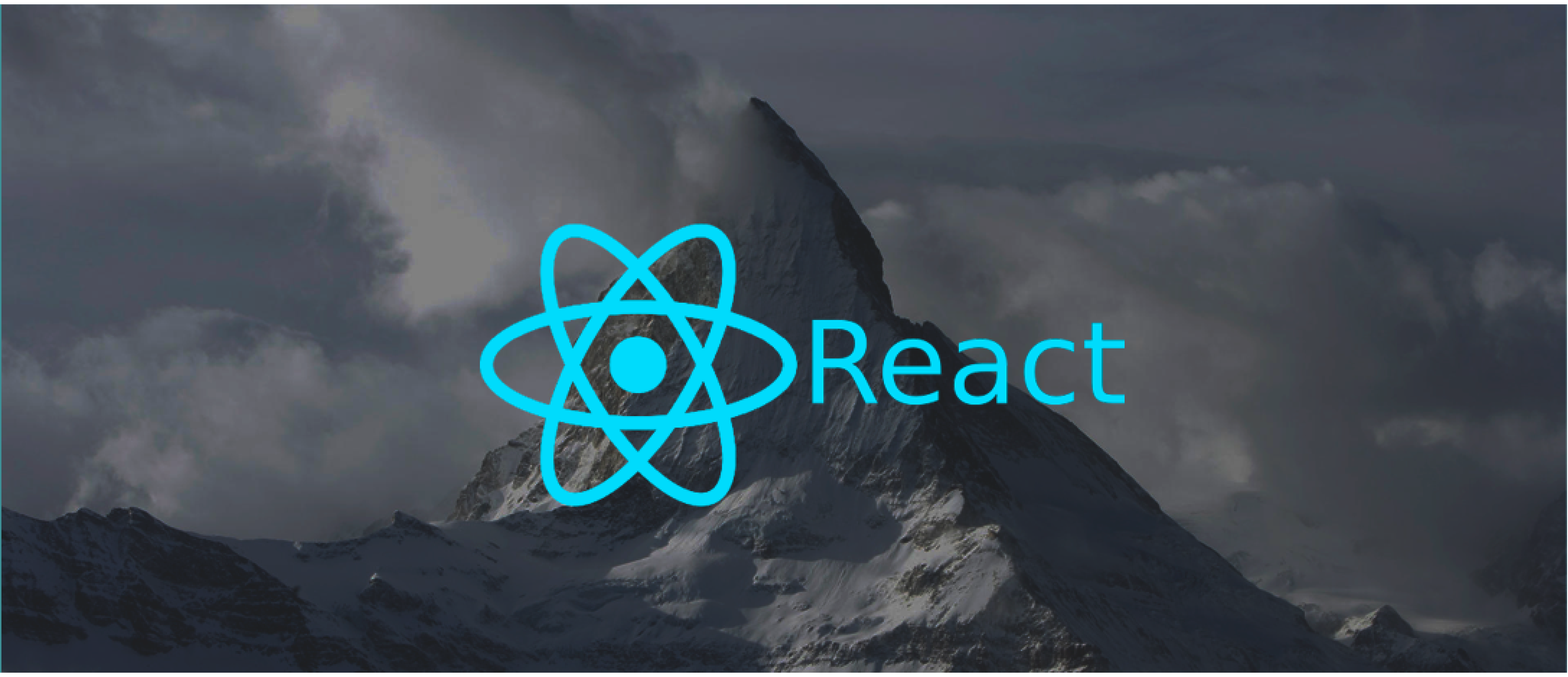
What to do when your React app feels slow

Aggelos Arvanitakis Follow
May 13, 2019 · 9 min read



I think this is Matterhorn

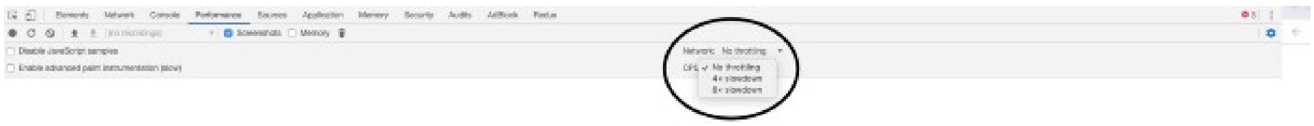
Intro

React is fast, but for some reason your app feels slow. It doesn't feel as smooth as it should given the fact that you are using "React", and you can't easily get your head around what's causing it. In this article I'll try and help you with situations like these, by giving you a list of steps to go through when you want to identify performance issues in a React app.

Approaching the issue

Mock the user's environment

The first thing that you should do is understand who your audience is. Different users have different devices, which means different CPU & GPU power. If you audience is mainly on mobile devices, then you have different thresholds than the ones you'd have if your audience was using your app mainly through a desktop. In addition, if you are targeting a big spectrum of age groups, chances are that there will be a lot more people with slow & low-end PCs, than if you were targeting the youth alone. CPU throttling is your friend here (found under the *“Performance”* tab within Chrome’s devtools).



CPU throttling within Chrome’s devtools

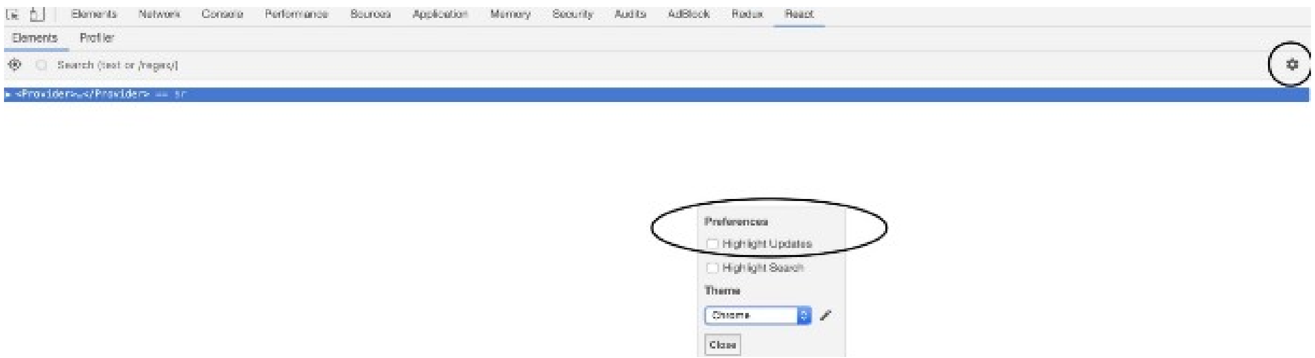
If your app is mobile-first, then a slowdown of 4x will do the trick; if it’s desktop-only, then perhaps a slowdown might not even be needed (depending on your PC). Before you classify something as a “slow”, make sure you mimic the average user’s environment in the best possible way.
Tip: Having the devtools open can slow down your app and you might see noticeable performance difference (especially in animations) if you have them open. Depending on your PC, there are cases where you might want to close them before you check whether something is slow or not. For example, when developing from a low-end PC while targeting high-end desktop users.

Check it in Production Mode

The second thing that you should check is whether you are using the production build of React. In the development build, React is a lot slower since it needs to analyse data and create stacks of calls for its warning messages. While these messages may be helpful they are also slowing down your app, so just make sure that your project has the React runtime running in production mode. Some things (like animations) that appear laggy on development, might be just fine in production mode. Bear that in mind.

Get a ballpark figure of what’s re-rendering too much

React devtools are your best friend here. If you don’t have them already installed, go ahead and install them from here. This includes two really important tools in order to analyse performance. The 1st tool is a “checkbox” that when checked, it will cause the React components that get re-rendered to have a flashing border in the screen.

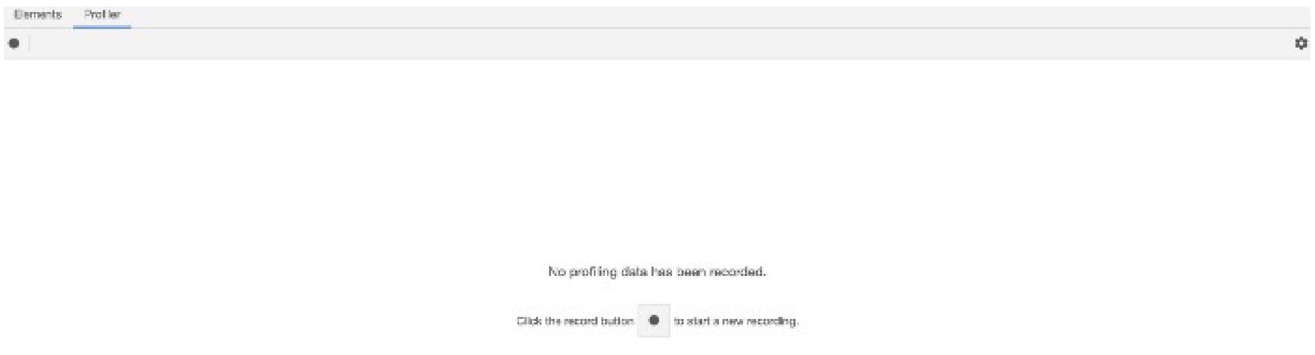


“Highlight updates” checkbox in Reaxt devtools

While this doesn’t do much, it does help you get a rough idea of what’s updating a lot and what’s not. With it activated, just go around your app and start doing “stuff” while seeing what gets flashy. Like golf, our goal here is to go roughly close to where the problem is located. I prefer to use this tool to get a glimpse of the overall app re-renders during the slow/lagging phase . This tool alone won’t help you solve your problems, but rather identify a **potential candidate** for perf issues. I say potential because, although most of the times a component that gets updated a lot will contribute to the overall lagginess, there are also times where a component might rarely get updated, but its rendering is still so expensive that it actually affects the overall performance of your app. To analyse all that we’ll use the React Profiler.

Get exact measurements with the React Profiler

The profiler is the main tool you should be using to understand the rendering of your app and can be found under the “Profiler” tab within the React devtools. This is available for projects using React ≥ 16.5, so make sure you don’t have an old version of React installed or the tab won’t show up at all.



No profiling data has been recorded.

Click the record button ● to start a new recording.

React Profiler tab

What it does, is fully analyse the number of re-renders that each component went though, while also measuring the time each render took (both on App and Component level). The profiler can help you understand whether something that re-renders a lot should be neglected (because its rendering time is negligible) or whether it should be optimised (since it might heavily contribute to the lagginess of your app). The documentation of this tool is superb and will get you fully familiar with it in no-time!

Now, as a rule of thumb, if the total rendering of the App is < 16ms, then you shouldn’t worry about optimising anything. Why 16ms? Well this is question for another article,

but it has to do with the way React Fiber works. The Fiber implementation of React takes advantage of the `requestIdleCallback()` API of the browsers, in order to defer non-essential tasks when the browser has some "idle time". That's why people were saying that the Fiber implementation will be faster; because instead of blocking the main thread, it would be able to potentially schedule work for later when the browser is less stressed. The maximum amount of time that the browser can spend on a task without losing any frames is roughly ~16ms, so that's why if your render takes less than that, you can safely assume that you are ok in terms of performance. Of course, a lot of the times an application render **might take much more and that might still be ok**. It all ties to how your app presents its UI. If you app has a very interactive UI, then losing some frames will be quickly noticeable by users. If, on the other hand, your app has mostly a static UI, then even an 80ms render might not appear laggy. It's all relative.
When isolating a component's rendering time, the profiler provides a way of quickly spotting which component is the most expensive (in terms of rendering time) through a flame chart. The more orange the colour is, the more expensive the component is. This information is useful, but it *doesn't and shouldn't automatically imply that this component needs to be optimised*. The flame chart is there to do a relative comparison of the rendering time for the components within a single render phase. It doesn't imply that this component's render is costly, but simply that it is **costlier** than another component's render.
The huge benefit of the Profiler is that you can see which components rendered while they shouldn't have (when their props have not changed). These components can easily benefit from render bail-out techniques, such as `PureComponent` and `memo()`. If you make correct use of them, then the components that should not have rendered will appear with a pale grey colour, indicating that React re-used the output from a previous render cycle without going through the reconciliation process for this component.

In my opinion, the ideal way to use the profiler is:

- Isolate the actions that make your app non-performant. Is it after clicking on a certain element? Is it during the initial app mount? Is it when you are transitioning from a certain page to another one?

- Create a recording during which you reproduce the steps that make your app laggy. (Tip: Remove any console.logs and debuggers, since they will affect the measurements)

- Analyse the total number of renders. Did your app render more times than normal or the expected ones?

- Analyse the overall rendering time of each render phase. Just go through every render and check how much time did it take for your app to render. If a performance issue is present, you might a see a couple of renders that took more than 100ms (as mentioned this number is relative to the nature of your app).
- Go through the **expensive renders'** component stack, isolating the components that contributed the most to the total render time.
- Try and find subcomponents — under the aforementioned components — that can benefit from render-bail out techniques. Chances are that you can save some precious reconciliations by simply not re-rendering the things that haven't changed.
- Try and find subcomponents whose code can benefit from memoization techniques. Sometimes an array or an object might be unnecessarily re-calculated, wasting main thread resources.

As long as you found the black sheep part of your code, there are a lot of things that you can do to optimise it. My article on React Performance tricks attempts to cover some of them.

Before we close this section, it's important to notice that at the time of writing, there is a beta version of the new React devtools which will give you access to even more interesting features in order to debug your code. Stuff like live prop-changing, recording during initial mount, component render stack trace & history and screenshot capturing are all already implemented and are going to further facilitate React app debugging. I'd strongly suggest you follow its creator Brian Vaughn to get the latest updates!

Get a full understanding with Chrome's devtools

While the Profiler is awesome, it simply outputs how long a rendering took. Most of the times that's more than enough, but there are times where we need to fully understand what exactly is making this particular rendering so slow. The ideal tool for this, is the performance analysis that Chrome's devtools offer. This works exactly like the profiler, but instead of outputting the rendering time of each component, it will give you all the scripting, painting & rendering timings of the browser. To put it in layman's terms, it will tell you which functions where called, how long did the browser spent calculating the positions of elements on the screen, as well as how costly every paint was.

When it comes to React, the performance tab is "too analytical", including the internal function calls of React (like `performWork`). That's why when performing a React render analysis, the React Profiler is preferred over it, since it helps you make more sense of what's happening. The benefit of Chrome's devtools though, is that you can actually get a full breakdown of all the calls that occurred within a render. This might help you understand exactly what part of the component's — potentially big — code is causing the issue.

On top of that, you get access to more advanced browser timings, such as paint & render. A paint is when the browser actually has to create the graphics of your UI, while a render is when the browser has to calculate and potentially update the position of elements in the screen. Both processes are costly and should ideally be minimised if they *lengthy* and appear *too often* during a "laggy phase". Although it's natural for these operations to exist within a performance analysis (after all if they didn't exist, you wouldn't see anything on the screen), there are times where your React code could force them to fire way too often. Perhaps you might not know that changing the `overflow` prop of the `<body>` element (to enable/disable scrolling) will cause a re-calculation of all the components within the body. A subtle change like that might cause a big stress for the main thread (if your app has a lot of components) and potentially be the reason that your app feels slow when a modal opens or a sidebar kicks in. This is something that **only** this tab can find and that's why it's so important. Analysing the whys and hows of this behaviour is outside the context of this article, but feel free to read more about it here.

Conclusion

While I didn't address a ton of stuff (for example the problems of issuing multiple concurrent network requests), my purpose was to give the high-level steps that one should follow when attempting to debug a React app. Correctly mimic your user's environment, check that React is in production mode, optionally check if something re-renders a lot and then move on to the Profiler. The Profiler will help you get your app faster 95% of the times, since most of the performance bottlenecks are created from the way you structure your React code. If the profiler couldn't help you identify & solve the issue, then that's when you should move to the performance tab. The latter is guaranteed to give you the answer you're looking for.

Thanks for reading :)

P.S. 👋 Hi, I'm Aggelos! If you liked this, consider following me on twitter and sharing the story with your friends 😀

React JavaScript Redux Web Development Reactjs