Starting up Performance of Web Applications

1. Introduction

Performance is an essential but always under-appreciated part in nowadays web development. The share of economic activity online is growing; more than 5% of the developed world's economy is now on the internet. And our always-on, hyper-connected modern world means that user expectations are higher than ever. If your site does not respond instantly, or if your app does not work without acceptable delay, users quickly move on to your competitors.

For example, a study of Amazon almost 10 years ago proved that, even then, a 100-millisecond decrease in page-loading time translated to a 1% increase in its revenue. Another recent study highlighted the fact that more than half of site owners surveyed said they lost revenue or customers due to poor application performance.

Therefore, performance should be attached great attention when we develop applications. To improve it, we must know how to measure it first. According to the popular model RAIL, which stands for response, animation, idle and load, we should make users the focal point of our performance effort. A good web application should respond to users immediately, acknowledge user input in under 100ms, produce a frame in under 10ms when scrolling or animating, maximize main thread idle time and deliver interactive content in under 1000ms. [1]

2. Facts that influence starting up performance

There are mainly three parts that lead to defects in web application's start-up performance: network, scripting and rendering.

2.1 Network cost in web application start-up

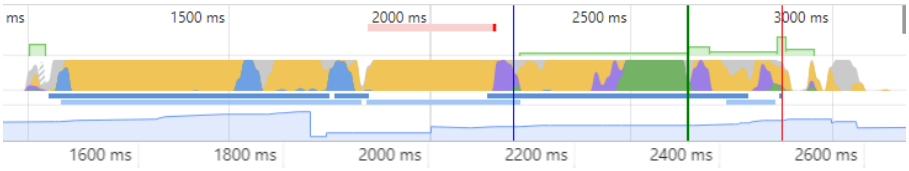
When a web application is opened, the first thing the browser does is to fetch files and resources to the user, during which network is the main reason that increase the loading time. According to a survey of GSMA this year, 45% of mobile connections occurred over 2G worldwide, and 75% of connections occurred on either 2G or 3G. Thus, we can see that the median user is on a slow network, especially the users on mobile end.

We can alleviate this by lots of methods. For example, only shipping the code a user needs with code-splitting framework like PRPL, minifying the codes with tools such as Uglify, compressing the files heavily with Gzip, removing unused code, preloading all resources in the main document or, even better, preloading with HTTP/2 Server Push. However, as HTTP/2 Push is not cache aware, it'll cause the network contention if without special process at the client side. One solution for that is using a Service Worker which intercepts each request to block those of them that aim for cached assets to prevent unnecessary cost.

2.2 Scripting cost in web application start-up

After browser obtains the resources, it'll begin to parse, compile and execute JavaScript codes, which is the heaviest task for the browser engine. We tested the course website through Chrome DevTools, and the result is represented in the chart below. As we can see, the yellow part, which stands for scripting, occupies over half of the entire browser processing time. Although the total

loading time, which is 1746.2ms, is acceptable, since this test is done on my PC with i7 Intel core CPU, devices with not as good processors may get a bad experience on this web site, especially for mobile devices, whose global average cost this year is only 245.1$ according to statistics on this site. We can see a more universal example in Fig 2. In this example[2], the scripting cost of loading "CNN.com" is compared between different devices, from desktop to mobile device. The result shows that desktop devices are the winner. And the cost varies a lot between high end mobile devices, such as iPhone 8, and average mobile devices, such as Moto G4: the average one took even 9s longer than the best to parse JavaScript!



(a) Timeline

Self Time	Total Time	Activity
1287.3 ms 73.7 %	1287.3 ms 73.7 %	Scripting
175.3 ms 10.0 %	175.3 ms 10.0 %	Painting
157.9 ms 9.0 %	157.9 ms 9.0 %	Rendering
125.7 ms 7.2 %	125.7 ms 7.2 %	Loading

(b) Time Cost of Each Phase

Fig. 1 Performance Panel for Course Website

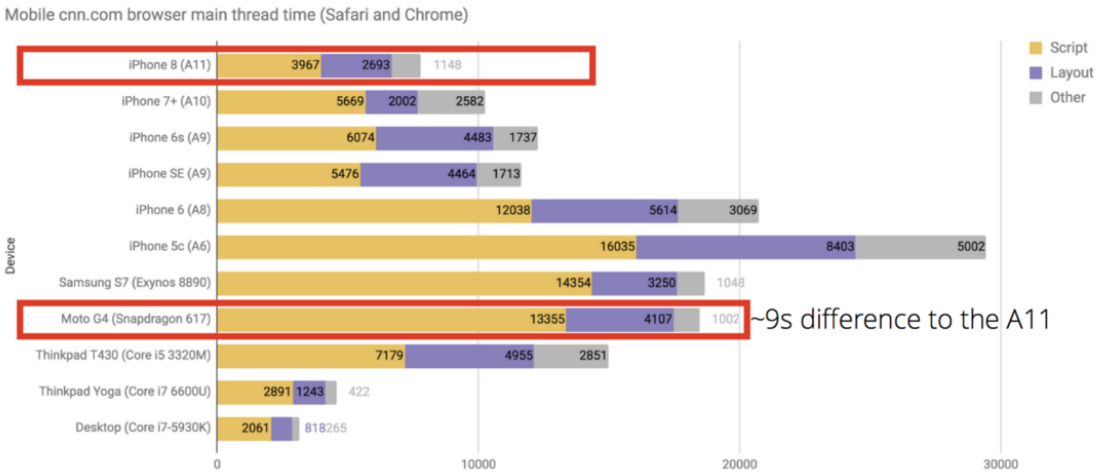


Fig. 2 JavaScript Parse Cost on Different Devices

From those examples we can conclude that it's significant to test on average hardware instead of just on your desktop or the phone in your pocket. We can easily obtain simulation of those environments in Chrome DevTool. To take more accurate simulation, you may refer to this tool. Developers have come up many ideas in reducing scripting cost:

a. **Ship less JavaScript**. The less script that requires parsing, the lower our overall time

spent in the parse & compile phases will be.

b. **Use code-splitting** to only ship the code a user needs for a route and lazy load the rest. This method has been mentioned above for alleviating network bound so you may find it familiar. When JavaScript is split into small chunks and loaded in different priorities, the burden of main thread will be much lighter.
c. Choose a framework that supports **ahead-of-time compilation** mode, like Angular AOT. It can convert codes into efficient JavaScript during the build phase before the browser downloads and runs that code.
d. **Progressive bootstrapping**. In other word, we can send down a minimally functional page, and lazy-load and unlock more features when more resources arrive. This will be further discussed in the net section.

2.3 Rendering cost in web application start-up

After finishing parsing JavaScript, the browser needs to render the assets resulted from parse and compile phase to the client's screen. The conventional method of getting the HTML up onto the screen was by using server-side rendering. It was a good choice when most webpages were just for displaying static images and text, with little in the way of interactive. Fast forward to today and that's no longer the case. A website today is more likely to be an application pretending as a website--it has much more complex features than it used to be. In this case, client-side rendering should be the better choice.

In client-side rendering, the server is only responsible for loading the bare minus of the website. Everything else is handled by client-side JavaScript libraries, for example, Angular, Vue.js or React. There's no denying that client-side rendering is hard to do well: bundling, transpiling, linting, cache busting and so much more facts must be handled when doing client-side rendering. These concern can lead to the delay of application's starting up to some extent.

However, it can also offer an even longer list of benefits that server-side rendering can't. For example, since the assets are rendered in browser, unnecessary requests for a full page can be avoided when only part of the webpage has changed. This is especially helpful in a world that's increasingly browsing via mobile networks with high latency. Besides, lazy loading is another amazing feature of client-side rendering, which can be used to save bandwidth & speed initial load. For example, additional records, images, and ads can be loaded as the user scrolls down, or as the user changes their search parameters, all without performing a full postback. Client-side rendering also supports rich, animated interactions, transformations, and transitions, for example, fading a row out on delete, or fading a dialog into view.[3]

3. Summary

In this paper, we analyzed why performance is important for web application, how to measure the performance of a web application, what facts that can influence performance and their corresponding solutions.

4. Reference

[1] Meggin Kearney, Measure Performance with the RAIL Model, September 26, 2017, link: https://developers.google.com/web/fundamentals/performance/rail.

[2] Addy Osmani, The Cost of JavaScript, November 15, 2017, link: https://medium.com/dev-channel/the-cost-of-javascript-84009f51e99e.
[3] Cory House, Here's Why Client-side Rendering Won, December 11, 2016, link: https://medium.freecodecamp.org/heres-why-client-side-rendering-won-46a349fadb52.