FRONT-END PERFORMANCE CHECKLIST 2018

SMASHING MAGAZINE

Imprint

© 2018 Smashing Media AG, Freiburg, Germany
ISBN (PDF): 978-3-945749-61-6
Cover Design: Ricardo Gimenes
eBook Production: Cosima Mielke
Tools: Elja Friedman
Idea & Concept: Smashing Media AG

About This Book

Let’s make 2018… fast! This eBook summarizes every-thing you need to know to create fast experiences today.

TABLE OF CONTENTS

Introduction

Performance matters — we all know it. However, do we actually always know what our performance bottlenecks exactly are? Is it expensive JavaScript, slow web font delivery, heavy images, or sluggish rendering? Is it worth exploring tree-shaking, scope hoisting, code-splitting, and all the fancy loading patterns with intersection observer, clients hints, CSS containment, HTTP/2 and service workers? And, most importantly, where do we even start improving performance and how do we establish a performance culture long-term?

Back in the day, performance was often a mere afterthought. Often deferred till the very end of the project, it would boil down to minification, concatenation, asset optimization and potentially a few fine adjustments on the server's config file. Looking back now, things seem to have changed quite significantly.

Performance isn't just a technical concern: it matters, and when baking it into the workflow, design decisions have to be informed by their performance implications. Performance has to be measured, monitored and refined continually, and the growing complexity of the web poses new challenges that make it hard to keep track of metrics, because metrics will vary significantly depending on the device, browser, protocol, network type and latency (CDNs, ISPs, caches, proxies, firewalls, load balancers and servers all play a role in performance).

So, if we created an overview of all the things we have to keep in mind when improving performance — from the very start of the process until the final release of the website — what would that list look like? Below you'll find a (hopefully unbiased and objective) front-end performance checklist for 2018 — an overview of the issues you might need to consider to ensure that your response times are fast, user interaction is smooth and your sites don't drain user's bandwidth.

Getting Ready: Planning And Metrics

Micro-optimizations are great for keeping performance on track, but it's critical to have clearly defined targets in mind — *measurable* goals that would influence any decisions made throughout the process. There are a couple of different models, and the ones discussed below are quite opinionated — just make sure to set your own priorities early on.

Establish A Performance Culture

In many organizations, front-end developers know exactly what common underlying problems are and what loading patterns should be used to fix them. However, as long as there is no alignment between dev/design and marketing teams, performance isn't going to sustain long-term. Study common complaints coming into customer service and see how improving performance can help relieve some of these common problems.

Run performance experiments and measure outcomes — both on mobile and on desktop. It will help you build up a company-tailored case study with real data. Furthermore, using data from case studies and experiments published on WPO Stats[1] can help increase sensitivity for business about why performance matters, and what impact it has on user experience and business metrics. Stat-

1. https://wpostats.com/

ing that performance matters alone isn't enough though — you also need to establish some measurable and trackable goals and observe them.

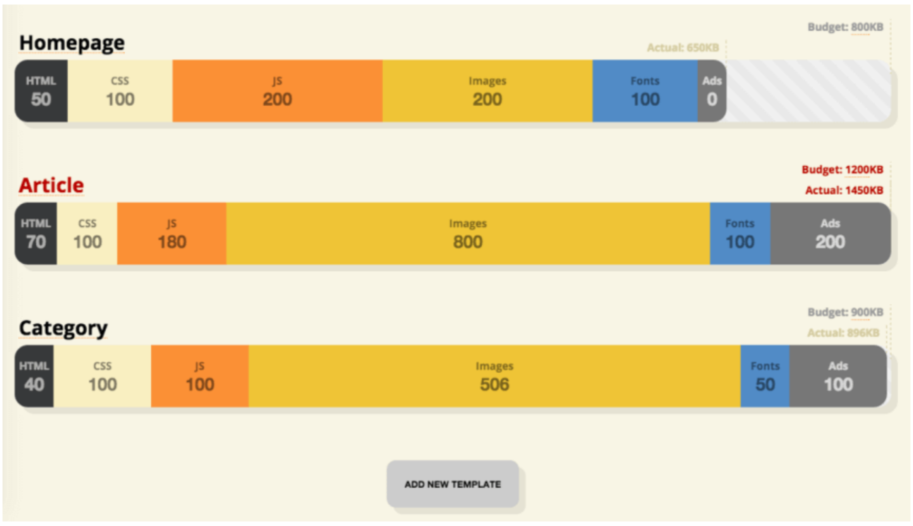
Be At Least 20% Faster Than Your Fastest Competitor

According to psychological research[2], if you want users to feel that your website is faster than your competitor's website, you need to be *at least* 20% faster. Study your main competitors, collect metrics on how they perform on mobile and desktop and set thresholds that would help you outpace them. To get accurate results and goals though, first study your analytics to see what your users are on. You can then mimic the 90th percentile's experience for testing. Collect data, set up a spreadsheet[3], shave off 20%, and set up your goals (i.e. performance budgets[4]) this way. Now you have something measurable to test against.

If you're keeping the budget in mind and trying to ship down just the minimal script to get a quick time-to-interactive, then you're on a reasonable path. Lara Hogan's guide on how to approach designs with a performance budget[5] can provide helpful pointers to designers and both Performance Budget Calculator[6] and Browser

2. https://www.smashingmagazine.com/2015/09/why-performance-matters-the-perception-of-time/#the-need-for-performance-optimization-the-20-rule
3. http://danielmall.com/articles/how-to-make-a-performance-budget/
4. http://bradfrost.com/blog/post/performance-budget-builder/
5. http://designingforperformance.com/weighing-aesthetics-and-performance/#approach-new-designs-with-a-performance-budget

Calories[7] can aid in creating budgets. (*Thanks to Karolina Szczur*[8] *for the heads up.*)



Performance budget builder[9] *by Brad Frost*

Beyond performance budgets, think about critical customer tasks that are most beneficial to your business. Set and discuss acceptable time thresholds for critical actions and establish "UX ready" user timing marks that the entire organization has agreed on. In many cases, user journeys will touch on the work of many different departments, so alignment in terms of acceptable timings will help support or prevent performance discussions down the road. Make sure that additional costs of added resources and features are visible and understood.

6. http://www.performancebudget.io/
7. https://browserdiet.com/calories/
8. https://medium.com/@fox/talk-the-state-of-the-web-3e12f8e413b3
9. http://bradfrost.com/blog/post/performance-budget-builder/

Also, as Patrick Meenan suggested, it's worth to plan out a loading sequence and trade-offs during the design process. If you prioritize early on which parts are more critical, and define the order in which they should appear, you will also know what can be delayed. Ideally, that order will also reflect the sequence of your CSS and JavaScript imports, so handling them during the build process will be easier. Also, consider what the visual experience should be in "in-between"-states, while the page is being loaded (e.g. when web fonts aren't loaded yet).

Planning, planning, planning. It might be tempting to get into quick "low-hanging-fruits"-optimizations early on — and eventually it might be a good strategy for quick wins — but it will be very hard to keep performance a priority without planning and realistic, company-tailored performance goals.

Choose The Right Metrics

Not all metrics are equally important[10]. Study what metrics matter most to your application: usually it will be related to how fast you can start render *most important* pixels (and what they are) and how quickly you can provide input responsiveness for these rendered pixels. This knowledge will give you the best optimization target for ongoing efforts. One way or another, rather than focusing on full page loading time (via *onLoad* and *DOMContentLoaded* timings, for example), prioritize page loading as perceived by your customers. That means focusing on

10. https://speedcurve.com/blog/rendering-metrics/

a slightly different set of metrics[11]. In fact, choosing the right metric is a process without obvious winners.

Below are some of the metrics worth considering:

- *First Meaningful Paint* (FMP, when primary content appears on the page),
- *Hero Rendering Times*[12] (when the page's important content has finished rendering),
- *Time to Interactive* (TTI, the point at which layout has stabilized, key webfonts are visible, and the main thread is available enough to handle user input — basically the time mark when a user can tap on UI and interact with it),
- *Input responsiveness* (how much time it takes for an interface to respond to user's action),
- *Perceptual Speed Index* (measures how quickly the page contents are visually populated; the lower the score, the better),
- Your custom metrics[13], as defined by your business needs and customer experience.

Steve Souders has a detailed explanation of each metrics[14]. While in many cases TTI and Input responsiveness

11. https://docs.google.com/presentation/d/1D4foHkE0VQdhcA5_hiesl8JhEGeTDRrQR4gipfJ8z7Y/present?slide=id.g21f3ab9dd6_0_33
12. https://speedcurve.com/blog/web-performance-monitoring-hero-times/
13. https://speedcurve.com/blog/user-timing-and-custom-metrics/
14. https://speedcurve.com/blog/rendering-metrics/

will be most critical, depending on the context of your application, these metrics might differ: e.g. for Netflix TV UI, key input responsiveness, memory usage and TTI[15] are more critical.

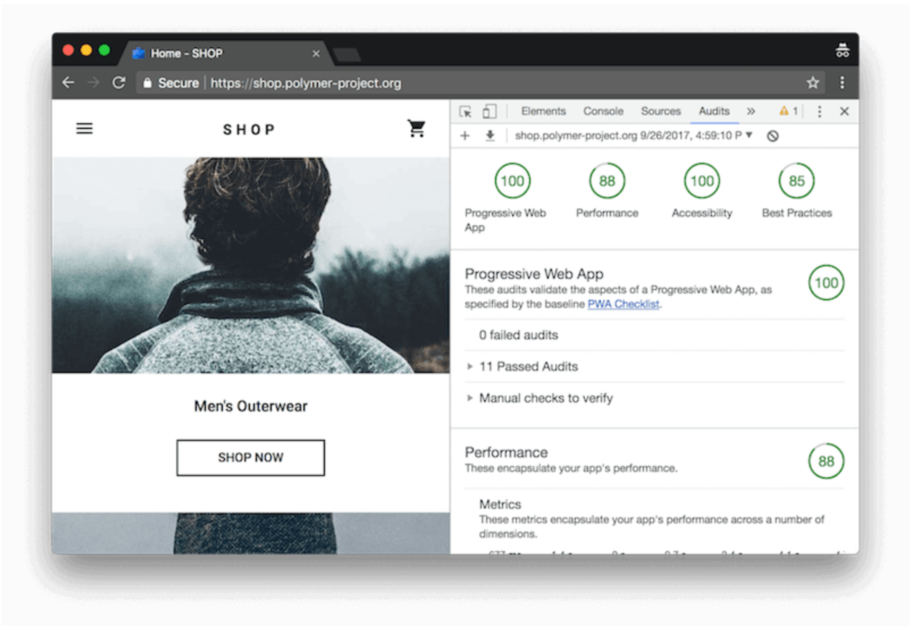
Gather Data On A Device Representative Of Your Audience

To gather accurate data, we need to thoroughly choose devices to test on. It's a good option to choose a Moto G4, a mid-range Samsung device and a good middle-of-the-road device like a Nexus 5X, perhaps in an open device lab[16]. If you don't have a device at hand, emulate mobile experience on desktop by testing on a throttled network (e.g. 150ms RTT, 1.5 Mbps down, 0.7 Mbps up) with a throttled CPU (5× slowdown). Eventually switch over to regular 3G, 4G and Wi-Fi. To make the performance impact more visible, you could even introduce 2G Tuesdays[17] or set up a throttled 3G network in your office[18] for faster testing.

Luckily, there are many great options that help you automate the collection of data and measure how your website performs over time according to these metrics. Keep in mind that a good performance metrics is a combination of passive and active monitoring tools:

15. https://medium.com/netflix-techblog/crafting-a-high-performance-tv-user-interface-using-react-3350e5a6ad3b
16. https://www.smashingmagazine.com/2016/11/worlds-best-open-device-labs/
17. https://www.theverge.com/2015/10/28/9625062/facebook-2g-tuesdays-slow-internet-developing-world
18. https://twitter.com/thommaskelly/status/938127039403610112

- **Passive monitoring tools** that simulate user interaction on request (*synthetic testing*, e.g. *Lighthouse*, *WebPageTest*) and
- **Active monitoring tools** that record and evaluate user interactions continuously (*Real User Monitoring*, e.g. *Speed-Curve*, *New Relic* — both tools provide synthetic testing, too).



These audits validate the aspects of a Progressive Web App, as specified by the baseline PWA Checklist.

These encapsulate your app's performance.

These metrics encapsulate your app's performance across a number of dimensions.

Lighthouse[19]*, a performance auditing tool integrated into DevTools.*

The former is particularly useful during development as it will help you stay on track while working on the product. The latter is useful for long-term maintenance as it will help you understand your performance bottlenecks as they are happening live — when users actually access the site. By tapping into built-in RUM APIs such as Navi-

19. https://developers.google.com/web/tools/lighthouse/

gation Timing, Resource Timing, Paint Timing, Long Tasks, etc., both passive and active performance monitoring tools together provide a complete picture of performance in your application. For instance, you could use PWMetrics[20], Calibre[21], SpeedCurve[22], mPulse[23] and Boomerang[24], Sitespeed.io[25], which all are fantastic options for performance monitoring.

Note: It's always a safer bet to choose network-level throttlers, external to the browser, as, for example, DevTools has issues interacting with HTTP/2 push, due to the way it's implemented. (*Thanks, Yoav!*)

RAIL[26], a user-centric performance model.

Share The Checklist With Your Colleagues

Make sure that the checklist is familiar to every member of your team to avoid misunderstandings down the line.

20. https://github.com/paulirish/pwmetrics
21. https://calibreapp.com
22. https://speedcurve.com/
23. https://www.soasta.com/performance-monitoring/
24. https://github.com/yahoo/boomerang
25. https://www-origin.sitespeed.io/
26. https://developers.google.com/web/fundamentals/performance/rail

Every decision has performance implications, and the project would hugely benefit from front-end developers properly communicating performance values to the whole team, so that everybody would feel responsibility for it, not just front-end developers. Map design decisions against performance budget and the priorities defined in the checklist.

Setting Realistic Goals

100-Millisecond Response Time, 60 fps

For an interaction to feel smooth, the interface has 100ms to respond to user's input. Any longer than that, and the user perceives the app as laggy. The RAIL, a user-centered performance model[27] gives you healthy targets: To allow for <100 milliseconds response, the page must yield control back to main thread at latest after every <50 milliseconds. Estimated Input Latency[28] tells us if we are hitting that threshold, and ideally, it should be below 50ms. For high pressure points like animation, it's best to do nothing else where you can and the absolute minimum where you can't.

Also, each frame of animation should be completed in less than 16 milliseconds, thereby achieving 60 frames per second (1 second ÷ 60 = 16.6 milliseconds) — preferably under 10 milliseconds. Because the browser needs time to paint the new frame to the screen your code should finish executing before hitting the 16.6 milliseconds mark. Be optimistic[29] and use idle time wisely. Obviously, these targets apply to runtime performance, rather than loading performance.

27. https://www.smashingmagazine.com/2015/10/rail-user-centric-model-performance/
28. https://developers.google.com/web/tools/lighthouse/audits/estimated-input-latency
29. https://www.smashingmagazine.com/2016/11/true-lies-of-optimistic-user-interfaces/

SpeedIndex < 1250, TTI < 5s on 3G, Critical File Size Budget < 170Kb

Although it might be very difficult to achieve, a good ultimate goal would be First Meaningful Paint under 1 second and a SpeedIndex[30] value under 1250. Considering the baseline being a $200 Android phone (e.g. Moto G4) on a slow 3G network, emulated at 400ms RTT and 400kbps transfer speed, aim for Time to Interactive under 5s[31], and for repeat visits, aim for under 2s.

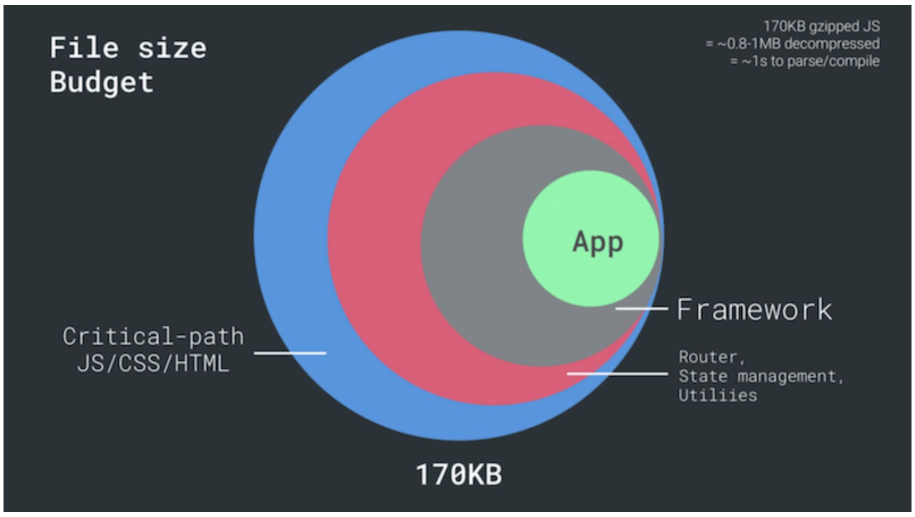
Notice that, when speaking about *Time To Interactive*, it's a good idea to distinguish between First Interactive and Consistency Interactive[32] to avoid misunderstandings down the line. The former is the earliest point after the main content has rendered (where there is at least a 5-second window where the page is responsive). The latter is the point where the page can be expected to always be responsive to input.

The first 14~15Kb of the HTML is the most critical payload chunk — and the only part of the budget that can be delivered in the first roundtrip (which is all you get in 1 second at 400ms RTT). In more general terms, to achieve the goals stated above, we have have to operate within a critical file size budget of max. 170Kb gzipped[33] (0.8–1MB decompressed) which already would take up to 1s (de-

30. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index
31. https://www.youtube.com/watch?v=_srJ7eHS3IM&feature=youtu.be&t=6m21s
32. https://calendar.perfplanet.com/2017/time-to-interactive-measuring-more-of-the-user-experience/
33. https://infrequently.org/2017/10/can-you-afford-it-real-world-web-performance-budgets/

pending on the resource type) to parse and compile on an average phone. Being slightly above that is fine, but push to get these values as low as possible.

You could also go beyond bundle size budget though. For example, you could set performance budgets on the activities of the browser's main thread, i.e. paint time before start render, or track down front-end CPU hogs[34]. Tools such as Calibre[35], SpeedCurve[36] and Bundlesize[37] can help you keep your budgets in check, and can be integrated into your build process.



From Fast By Default: Modern loading best practices[38] *by Addy Osmani (Slide 19).*

34. https://calendar.perfplanet.com/2017/tracking-cpu-with-long-tasks-api/
35. https://calibreapp.com/
36. https://speedcurve.com/
37. https://github.com/siddharthkp/bundlesize
38. https://speakerdeck.com/addyosmani/fast-by-default-modern-loading-best-practices

Defining The Environment

Choose And Set Up Your Build Tools

Don't pay too much attention to what's supposedly cool[39] these days. Stick to your environment for building, be it Grunt, Gulp, Webpack, Parcel, or a combination of tools. As long as you are getting results you need fast and you have no issues maintaining your build process, you're doing just fine.

Progressive Enhancement

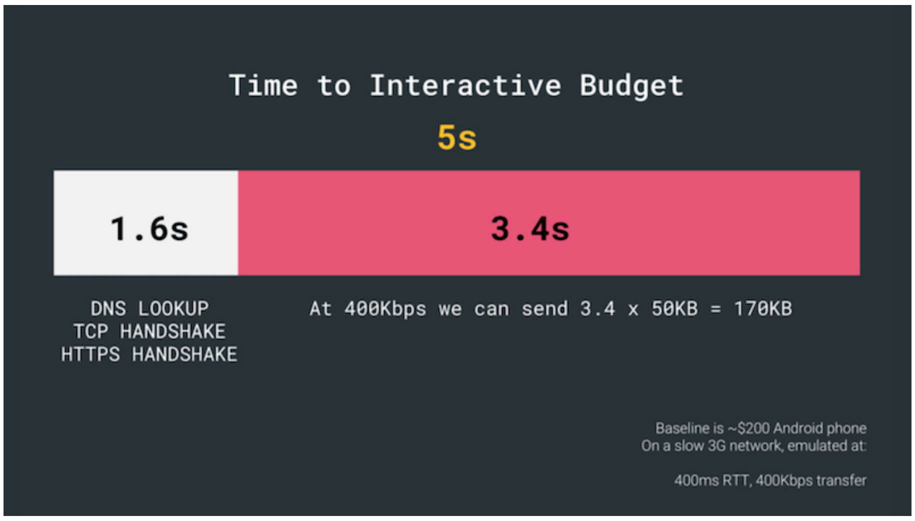
Keeping progressive enhancement[40] as the guiding principle of your front-end architecture and deployment is a safe bet. Design and build the core experience first, and then enhance the experience with advanced features for capable browsers, creating resilient[41] experiences. If your website runs fast on a slow machine with a poor screen in a poor browser on a suboptimal network, then it will only run faster on a fast machine with a good browser on a decent network.

Choose A Strong Performance Baseline

With so many unknowns impacting loading — the network, thermal throttling, cache eviction, third-party

39. https://24ways.org/2017/all-that-glisters/
40. https://www.aaron-gustafson.com/notebook/insert-clickbait-headline-about-progressive-enhancement-here/
41. https://resilientwebdesign.com/

scripts, parser blocking patterns, disk I/O, IPC jank, installed extensions, CPU, hardware and memory constraints, differences in L2/L3 caching, RTTS, images, web fonts loading behavior — JavaScript has the heaviest cost of the experience[42], next to web fonts blocking rendering by default and images often consuming too much memory. With the performance bottlenecks moving away from the server to the client[43], as developers, we have to consider all of these unknowns in much more detail.



From Fast By Default: Modern Loading Best Practices[44] *by Addy Osmani (Slides 18, 19).*

With a 170KB budget that already contains the critical-path HTML/CSS/JavaScript, router, state management, utilities, framework and the application logic, we have to thoroughly examine network transfer cost, the parse/

42. https://youtu.be/_srJ7eHS3IM?t=3m2s
43. https://calendar.perfplanet.com/2017/tracking-cpu-with-long-tasks-api/
44. https://speakerdeck.com/addyosmani/fast-by-default-modern-loading-best-practices

compile time and the runtime cost[45] of the framework of our choice.

As noted[46] by Seb Markbåge, a good way to measure start-up costs for frameworks is to first render a view, then delete it and then render again as it can tell you how the framework scales.

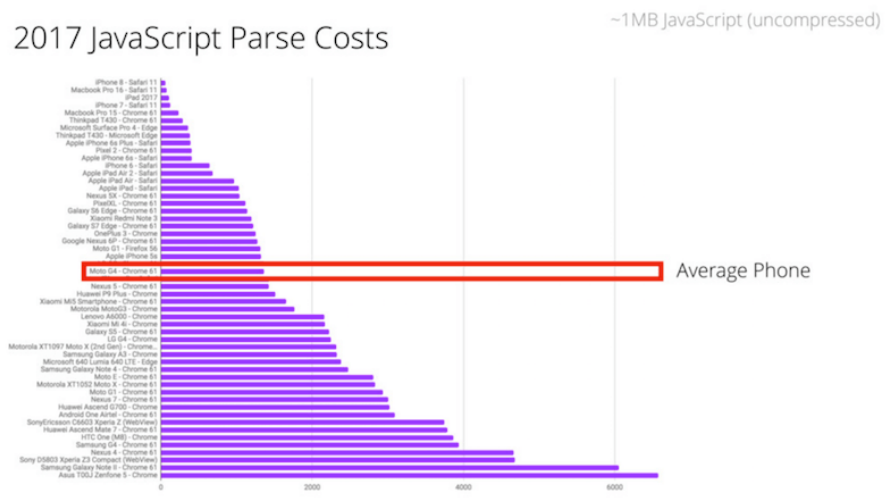
The first render tends to warm up a bunch of lazily compiled code, which a larger tree can benefit from when it scales. The second render is basically an emulation of how code reuse on a page affects the performance characteristics as the page grows in complexity.

Not every project needs a framework[47]. In fact, some projects can benefit from removing an existing framework[48] altogether. Once a framework is chosen, you'll be staying with it for at least a few years, so if you need to use one, make sure your choice is informed[49] and well considered[50]. It's a good idea to consider *at least* the total cost on size + initial parse times before choosing an option; lightweight options such as Preact[51], Inferno[52], Vue[53], Svelte[54] or Polymer[55] can get the job done just fine.

---

45. https://www.twitter.com/kristoferbaxter/status/908144931125858304
46. https://twitter.com/sebmarkbage/status/829733454119989248
47. https://twitter.com/jaffathecake/status/923805333268639744
48. https://twitter.com/jaffathecake/status/925320026411950080
49. https://www.youtube.com/watch?v=6I_GwgoGm1w
50. https://medium.com/@ZombieCodeKill/choosing-a-javascript-framework-535745d0ab90#.2op7rjakk
51. https://github.com/developit/preact
52. https://github.com/infernojs/inferno
53. https://vuejs.org/
54. https://svelte.technology/
55. https://github.com/Polymer/polymer

The size of your baseline will define the constraints for your application's code.



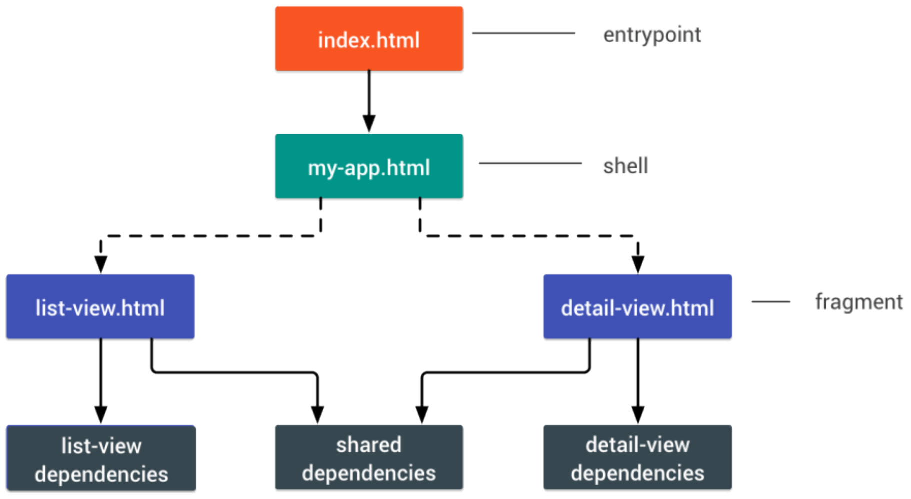
JavaScript parsing costs can differ significantly. From Fast By Default: Modern Loading Best Practices[56] *by Addy Osmani (Slide 10).*

Keep in mind that on a mobile device, you should be expecting a 4×–5× slowdown compared to desktop machines. Mobile devices have different GPUs, CPU, different memory, different battery characteristics. Parse times on mobile are 36% higher than on desktop[57]. So always test on an average device[58] — a device that is most representative of your audience.

Different frameworks will have different effects on performance and will require different strategies of optimization, so you have to clearly understand all of the nuts and bolts of the framework you'll be relying on. When

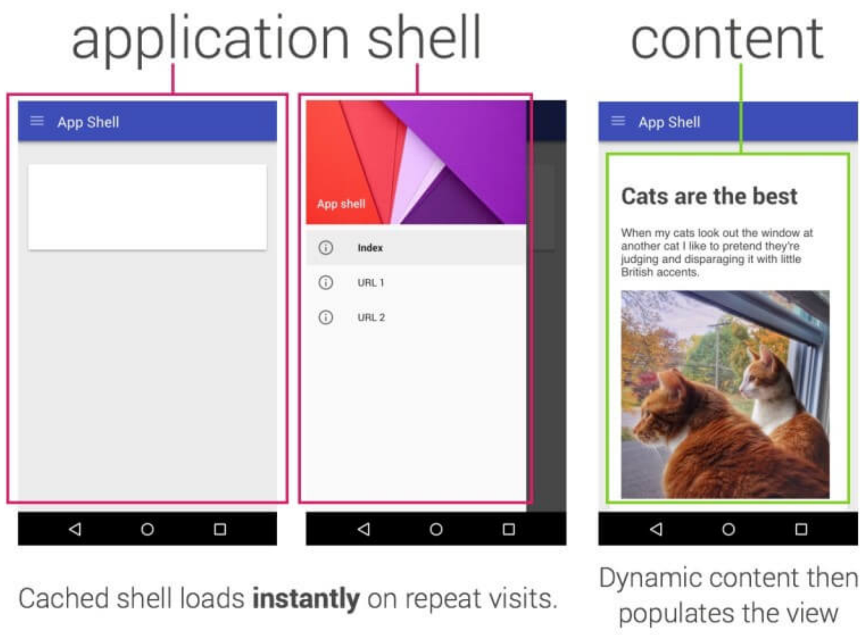
56. https://speakerdeck.com/addyosmani/fast-by-default-modern-loading-best-practices
57. https://github.com/GoogleChromeLabs/discovery/issues/1
58. https://www.webpagetest.org/easy-load

building a web app, look into the PRPL pattern[59] and application shell architecture[60]. The idea is quite straightforward: Push the minimal code needed to get interactive for the initial route to render quickly, then use service worker for caching and pre-caching resources and then lazy-load routes that you need, asynchronously.



PRPL[61] stands for Pushing critical resource, Rendering initial route, Pre-caching remaining routes and Lazy-loading remaining routes on demand.

59. https://developers.google.com/web/fundamentals/performance/prpl-pattern/
60. https://developers.google.com/web/updates/2015/11/app-shell
61. https://developers.google.com/web/fundamentals/performance/prpl-pattern/


application shell
content
App Shell
App shell
Index
URL 1
URL 2
App Shell
Cats are the best
When my cats look out the window at another cat I like to pretend they're judging and disparaging it with little British accents.
Cached shell loads **instantly** on repeat visits.
Dynamic content then populates the view

An application shell[62] is the minimal HTML, CSS, and JavaScript powering a user interface.

Will You Be Using AMP Or Instant Articles?

Depending on the priorities and strategy of your organization, you might want to consider using Google's AMP[63] or Facebook's Instant Articles[64] or Apple's Apple News[65]. You can achieve good performance without them, but AMP does provide a solid performance framework with a free content delivery network (CDN), while Instant Articles will boost your visibility and performance on Facebook.

62. https://developers.google.com/web/updates/2015/11/app-shell
63. https://www.ampproject.org/
64. https://instantarticles.fb.com/
65. https://www.apple.com/news/

The main benefit of these technologies for users is *guaranteed performance*, so at times they might even prefer AMP-/Apple News/Instant Pages-links over "regular" and potentially bloated pages. For content-heavy websites that are dealing with a lot of third-party content, these options could help speed up render times dramatically.

A benefit for the website owner is obvious: discoverability of these formats on their respective platforms and increased visibility in search engines[66]. You could build progressive web AMPs[67], too, by reusing AMPs as a data source for your PWA. Downside? Obviously, a presence in a walled garden places developers in a position to produce and maintain a separate version of their content, and in case of Instant Articles and Apple News without actual URLs[68]. (*Thanks Addy, Jeremy!*)

Choose Your CDN Wisely

Depending on how much dynamic data you have, you might be able to "outsource" some part of the content to a static site generator[69], pushing it to a CDN and serving a static version from it, thus avoiding database requests. You could even choose a static-hosting platform[70] based

66. https://ethanmarcotte.com/wrote/ampersand/
67. https://www.smashingmagazine.com/2016/12/progressive-web-amps/
68. https://www.w3.org/blog/TAG/2017/07/27/distributed-and-syndicated-content-whats-wrong-with-this-picture/
69. https://www.smashingmagazine.com/2015/11/static-website-generators-jekyll-middleman-roots-hugo-review/
70. https://www.smashingmagazine.com/2015/11/modern-static-website-generators-next-big-thing/

on a CDN, enriching your pages with interactive components as enhancements (JAMStack[71]).

Notice that CDNs can serve (and offload) dynamic content as well. So, restricting your CDN to static assets is not necessary. Double-check whether your CDN performs compression and conversion (e.g. image optimization in terms of formats, compression and resizing at the edge), smart HTTP/2 delivery, edge-side includes, which assemble static and dynamic parts of pages at the CDN's edge (i.e. the server closest to the user), and other tasks.

71. https://jamstack.org/

Build Optimizations

Set Your Priorities Straight

It's a good idea to know what you are dealing with first. Run an inventory of all of your assets (JavaScript, images, fonts, third-party scripts and "expensive" modules on the page, such as carousels, complex infographics and multimedia content), and break them down in groups.

Set Up A Spreadsheet

Define the basic *core* experience for legacy browsers (i.e. fully accessible core content), the *enhanced* experience for capable browsers (i.e. the enriched, full experience) and the *extras* (assets that aren't absolutely required and can be lazy-loaded, such as web fonts, unnecessary styles, carousel scripts, video players, social media buttons, large images). We published an article on "Improving Smashing Magazine's Performance[72]," which describes this approach in detail.

Consider Using The "Cutting-The-Mustard" Pattern

Albeit quite old, we can still use the cutting-the-mustard technique[73] to send the core experience to legacy browsers and an enhanced experience to modern

72. https://www.smashingmagazine.com/2014/09/improving-smashing-magazine-performance-case-study/
73. http://responsivenews.co.uk/post/18948466399/cutting-the-mustard

browsers. Be strict in loading your assets: Load the core experience immediately, then enhancements, and then the extras. Note that the technique deduces device capability from browser version, which is no longer something we can do these days.

For example, cheap Android phones in developing countries mostly run Chrome and will cut the mustard despite their limited memory and CPU capabilities. That's where PRPL pattern[74] could serve as a good alternative. Eventually, using the Device Memory Client Hints Header[75], we'll be able to target low-end devices more reliably. At the moment of writing, the header is supported only in Blink (it goes for client hints[76] in general). Since Device Memory also has a JavaScript API which is already available in Chrome[77], one option could be to feature detect based on the API, and fallback to "cutting the mustard" technique only if it's not supported. (*Thanks, Yoav!*)

Parsing JavaScript Is Expensive, So Keep It Small

When dealing with single-page applications, you might need some time to initialize the app before you can render the page. Look for modules and techniques to speed up the initial rendering time (for example, here's how to debug React performance[78], and here's how to improve

74. https://developers.google.com/web/fundamentals/performance/prpl-pattern/
75. https://github.com/w3c/device-memory
76. https://caniuse.com/#search=client%20hints
77. https://developers.google.com/web/updates/2017/12/device-memory

performance in Angular[79]), because most performance issues come from the initial parsing time to bootstrap the app.

JavaScript has a cost[80], but it's not necessarily the file size that drains on performance. Parsing and executing times vary significantly depending on the hardware of a device. On an average phone (Moto G4), a parsing time alone for 1MB of (uncompressed) JavaScript will be around 1.3–1.4s, with 15–20% of all time on mobile spent on parsing. With compiling in play, just prep work on JavaScript takes 4s on average, with around 11s before First Meaningful Paint on mobile. Reason: parse and execution times can easily be 2–5x times higher[81] on low-end mobile devices.

An interesting way of avoiding parsing costs is to use binary templates[82] that Ember has recently introduced for experimentation. These templates don't need to be parsed. (*Thanks, Leonardo!*)

That's why it's critical to examine every single JavaScript dependency, and tools like webpack-bundle-analyzer[83], Source Map Explorer[84] and Bundle Buddy[85] can help you achieve just that. Measure JavaScript parse

78. https://building.calibreapp.com/debugging-react-performance-with-react-16-and-chrome-devtools-c90698a522ad
79. https://www.youtube.com/watch?v=p9vT0W31ym8
80. https://youtu.be/_srJ7eHS3IM?t=9m33s
81. https://medium.com/reloading/javascript-start-up-performance-69200f43b201
82. https://emberjs.com/blog/2017/10/10/glimmer-progress-report.html#toc_binary-templates
83. https://www.npmjs.com/package/webpack-bundle-analyzer
84. https://github.com/danvk/source-map-explorer
85. https://github.com/samccone/bundle-buddy

and compile times[86]. Etsy's DeviceTiming[87], a little tool allowing you to instruct your JavaScript to measure parse and execution time on any device or browser. Bottom line: while size matters, it isn't everything. Parse and compiling times don't necessarily increase linearly[88] when the script size increases.

Are You Using An Ahead-Of-Time Compiler?

Use an ahead-of-time compiler[89] to offload some of the client-side rendering[90] to the server[91] and, hence, output usable results quickly. Finally, consider using Optimize.js[92] for faster initial loading by wrapping eagerly invoked functions (it might not be necessary[93] any longer, though).

Are You Using Tree-Shaking, Scope Hoisting And Code-Splitting?

Tree-shaking[94] is a way to clean up your build process by only including code that is actually used in production

86. https://medium.com/reloading/javascript-start-up-performance-69200f43b201#7557
87. https://github.com/danielmendel/DeviceTiming
88. https://medium.com/reloading/javascript-start-up-performance-69200f43b201
89. https://www.lucidchart.com/techblog/2016/09/26/improving-angular-2-load-times/
90. https://www.smashingmagazine.com/2016/03/server-side-rendering-react-node-express/
91. http://redux.js.org/docs/recipes/ServerRendering.html
92. https://github.com/nolanlawson/optimize-js
93. https://twitter.com/tverwaes/status/809788255243739136

and eliminate unused exports in Webpack[95]. With Webpack 3 and Rollup, we also have scope hoisting[96] that allows both tools to detect where `import` chaining can be flattened and converted into one inlined function without compromising the code. With Webpack 4, you can now use JSON Tree Shaking[97] as well. UnCSS[98] or Helium[99] can help you remove unused styles from CSS.

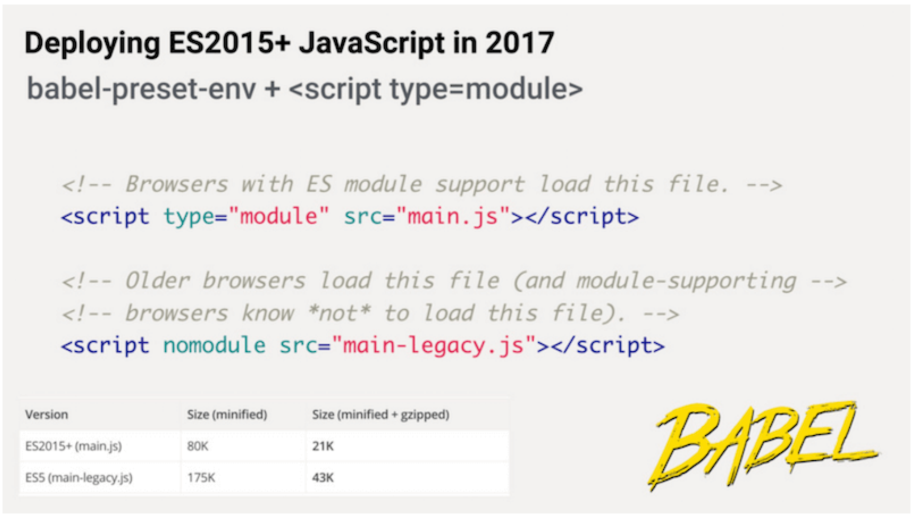
Also, you might want to consider learning how to write efficient CSS selectors[100] as well as how to avoid bloat and expensive styles[101]. Feeling like going beyond that? You can also use Webpack to shorten the class names and use scope isolation to rename CSS class names dynamically[102] at the compilation time.

Code-splitting[103] is another Webpack feature that splits your code base into "chunks" that are loaded on demand. Not all of the JavaScript has to be downloaded, parsed and compiled right away. Once you define split points in your code, Webpack can take care of the dependencies and outputted files. It enables you to keep the initial download small and to request code on demand,

94. https://medium.com/@roman01la/dead-code-elimination-and-tree-shaking-in-javascript-build-systems-fb8512c86edf
95. http://www.2ality.com/2015/12/webpack-tree-shaking.html
96. https://medium.com/webpack/brief-introduction-to-scope-hoisting-in-webpack-8435084c171f
97. https://react-etc.net/entry/json-tree-shaking-lands-in-webpack-4-0
98. https://github.com/giakki/uncss
99. https://github.com/geuis/helium-css
100. http://csswizardry.com/2011/09/writing-efficient-css-selectors/
101. https://benfrain.com/css-performance-revisited-selectors-bloat-expensive-styles/
102. https://medium.freecodecamp.org/reducing-css-bundle-size-70-by-cutting-the-class-names-and-using-scope-isolation-625440de600b
103. https://webpack.github.io/docs/code-splitting.html

when requested by the application. Also, consider using preload-webpack-plugin[104] that takes routes you code-split and then prompts browser to preload them using `<link rel="preload">` or `<link rel="prefetch">`.

Where to define split points? By tracking which chunks of CSS/JavaScript are used, and which aren't used. Umar Hansa explains[105] how you can use Code Coverage from Devtools to achieve it.



Deploying ES2015+ JavaScript in 2017

babel-preset-env + <script type=module>

```html
<!-- Browsers with ES module support load this file. -->
<script type="module" src="main.js"></script>

<!-- Older browsers load this file (and module-supporting -->
<!-- browsers know *not* to load this file). -->
<script nomodule src="main-legacy.js"></script>
```

Version	Size (minified)	Size (minified + gzipped)
ES2015+ (main.js)	80K	21K
ES5 (main-legacy.js)	175K	43K

From Fast By Default: Modern Loading Best Practices[106] by the one-and-only Addy Osmani. Slide 76.

If you aren't using Webpack, note that Rollup[107] shows significantly better results than Browserify exports. While we're at it, you might want to check out Rollupify[108], which converts ECMAScript 2015 modules into one

104. https://github.com/GoogleChromeLabs/preload-webpack-plugin
105. https://vimeo.com/235431630#t=11m37s
106. https://speakerdeck.com/addyosmani/fast-by-default-modern-loading-best-practices
107. http://rollupjs.org/

big CommonJS module — because small modules can have a surprisingly high performance cost[109] depending on your choice of bundler and module system.

Finally, with ES2015 being remarkably well supported in modern browsers[110], consider using `babel-preset-env`[111] to only transpile ES2015+ features unsupported by the modern browsers you are targeting. Then set up two builds[112], one in ES6 and one in ES5. We can use `script type="module"`[113] to let browsers with ES module support loading the file, while older browser could load legacy builds with `script nomodule`.

For lodash, use `babel-plugin-lodash`[114] that will load only modules that you are using in your source. This might save you quite a bit of JavaScript payload.

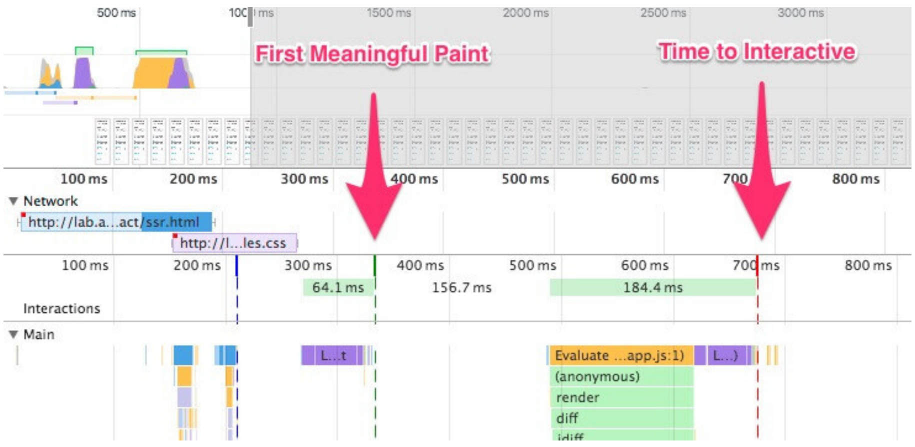
Take Advantage Of Optimizations For Your Target JavaScript Engine

Study what JavaScript engines dominate in your user base, then explore ways of optimizing for them. For example, when optimizing for V8 which is used in Blink-browsers, Node.js runtime and Electron, make use of script streaming[115] for monolithic scripts. It allows `async`

108. https://github.com/nolanlawson/rollupify
109. https://nolanlawson.com/2016/08/15/the-cost-of-small-modules/
110. http://kangax.github.io/compat-table/es6/
111. http://2ality.com/2017/02/babel-preset-env.html
112. https://gist.github.com/newyankeecodeshop/79f3e1348a09583faf62ed55b58d09d9
113. https://matthewphillips.info/posts/loading-app-with-script-module
114. https://github.com/lodash/babel-plugin-lodash
115. https://blog.chromium.org/2015/03/new-javascript-techniques-for-rapid.html

or `defer scripts` to be parsed on a separate background thread once downloading begins, hence in some cases improving page loading times by up to 10%. Practically, use `<script defer>`[116] in the `<head>`, so that the browsers can discover the resource[117] early and then parse it on the background thread.

Caveat: *Opera Mini doesn't support script deferment*[118]*, so if you are developing for India or Africa,* `defer` *will be ignored, resulting in blocking rendering until the script has been evaluated. (Thanks, Jeremy!)*



Progressive booting[119] *means using server-side rendering to get a quick first meaningful paint, but also include some minimal JavaScript to keep the time-to-interactive close to the first meaningful paint.*

116. https://medium.com/reloading/javascript-start-up-performance-69200f43b201#3498
117. https://medium.com/reloading/javascript-start-up-performance-69200f43b201#3498
118. https://caniuse.com/#search=defer
119. https://aerotwist.com/blog/when-everything-is-important-nothing-is/

Client-Side Rendering Or Server-Side Rendering?

In both scenarios, our goal should be to set up progressive booting[120]: Use server-side rendering to get a quick first meaningful paint, but also include some minimal necessary JavaScript to keep the time-to-interactive close to the first meaningful paint. If JavaScript is coming too late after the First Meaningful Paint, the browser might lock up the main thread[121] while parsing, compiling and executing late-discovered JavaScript, hence handcuffing the interactivity of site or application[122].

To avoid it, always break up the execution of functions into separate, asynchronous tasks, and where possible use `requestIdleCallback`. Consider lazy loading parts of the UI using WebPack's dynamic `import()` support[123], avoiding the load, parse, and compile cost until the users really need them. (*Thanks, Addy!*)

In its essence, Time to Interactive (TTI) tells us how the length of time between navigation and interactivity. The metric is defined by looking at the first five second window after the initial content is rendered, in which no JavaScript tasks take longer than 50ms. If a task over 50ms occurs, the search for a five second window starts over. As a result, the browser will first assume that it

120. https://aerotwist.com/blog/when-everything-is-important-nothing-is/
121. https://davidea.st/articles/measuring-server-side-rendering-performance-is-tricky
122. https://philipwalton.com/articles/why-web-developers-need-to-care-about-interactivity/
123. https://developers.google.com/web/updates/2017/11/dynamic-import

reached Interactive, just to switch to Frozen, just to eventually switch back to Interactive.

Once we reached Interactive, we can then, either on demand or as time allows, boot non-essential parts of the app. Unfortunately, as Paul Lewis noticed[124], frameworks typically have no concept of priority that can be surfaced to developers, and hence progressive booting is difficult to implement with most libraries and frameworks. If you have the time and resources, use this strategy to ultimately boost performance.

Do You Constrain The Impact Of Third-Party Scripts?

With all performance optimizations in place, often we can't control third-party scripts coming from business requirements. Third-party-scripts metrics aren't influenced by end user experience, so too often one single script ends up calling a long tail of obnoxious third-party scripts, hence ruining a dedicated performance effort. To contain and mitigate performance penalties that these scripts bring along, it's not enough to just load them asynchronously (probably via defer[125]) and accelerate them via resource hints such as `dns-prefetch` or `preconnect`.

As Yoav Weiss explained in his must-watch talk on third-party scripts[126], in many cases these scripts down-

124. https://aerotwist.com/blog/when-everything-is-important-nothing-is/#which-to-use-progressive-booting
125. https://www.twnsnd.com/posts/performant_third_party_scripts.html

load resources that are dynamic. The resources change between page loads, so we don't necessarily know which hosts the resources will be downloaded from and what resources they would be.

What options do we have then? Consider using service workers by racing the resource download with a timeout and if the resource hasn't responded within a certain timeout, return an empty response to tell the browser to carry on with parsing of the page. You can also log or block third-party requests that aren't successful or don't fulfill certain criteria.

Another option is to establish a Content Security Policy (CSP) to restrict the impact of third-party scripts, e.g. disallowing the download of audio or video. The best option is to embed scripts via `<iframe>` so that the scripts are running in the context of the iframe and hence don't have access to the DOM of the page, and can't run arbitrary code on your domain. Iframes can be further constrained using the `sandbox` attribute, so you can disable any functionality that iframe may do, e.g. prevent scripts from running, prevent alerts, form submission, plugins, access to the top navigation, and so on.

For example, it's probably going to be necessary to allow scripts to run with `<iframe sandbox="allow-scripts">`. Each of the limitations can be lifted via various `allow` values on the `sandbox` attribute (supported almost everywhere[127]), so constrain them to the bare minimum of what they should be allowed to do. Consider us-

126. http://conffab.com/video/taking-back-control-over-third-party-content/
127. https://caniuse.com/#search=sandbox

ing Safeframe[128] and Intersection Observer; that would enable ads to be iframed while still dispatching events or getting the information that they need from the DOM (e.g. ad visibility). Watch out for new policies such as Feature policy[129], resource size limits and CPU/Bandwidth priority to limit harmful web features and scripts that would slow down the browser, e.g. synchronous scripts, synchronous XHR requests, document.write and outdated implementations.

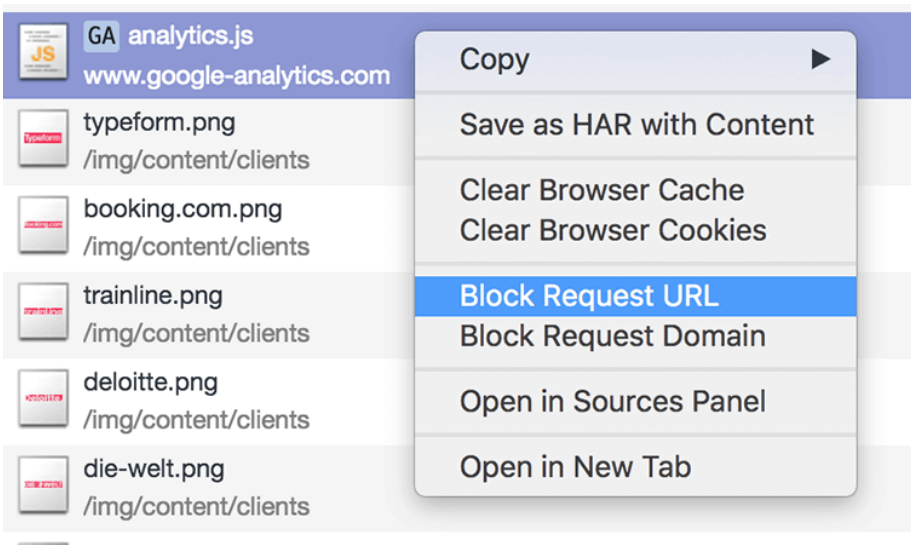


Image credit: Harry Roberts[130]

To stress-test third parties[131], examine bottom-up summaries in Performance profile page in DevTools, test what happens if a request is blocked or it has timed out —

128. https://github.com/InteractiveAdvertisingBureau/safeframe
129. https://wicg.github.io/feature-policy/
130. https://csswizardry.com/2017/07/performance-and-resilience-stress-testing-third-parties/#request-blocking
131. https://csswizardry.com/2017/07/performance-and-resilience-stress-testing-third-parties/

for the latter, you can use WebPageTest's Blackhole server `72.66.115.13` that you can point specific domains to in your `hosts` file. Preferably self-host and use a single hostname[132], but also generate a request map[133] that exposes fourth-party calls and detect when the scripts change.

Are HTTP Cache Headers Set Properly?

Double-check that `expires`, `cache-control`, `max-age` and other HTTP cache headers have been set properly. In general, resources should be cacheable either for a very short time (if they are likely to change) or indefinitely (if they are static) — you can just change their version in the URL when needed. Disable the `Last-Modified` header as any asset with it will result in a conditional request with an `If-Modified-Since`-header even if the resource is in cache. Same with `Etag`, though it has its uses.

Use `Cache-control: immutable`, designed for fingerprinted static resources, to avoid revalidation (as of December 2017, supported in Firefox, Edge and Safari[134]; in Firefox only on `https://` transactions). You can use Heroku's primer on HTTP caching headers[135], Jake Archibald's "Caching Best Practices[136]" and Ilya Grigorik's HTTP caching primer[137] as guides. Also, be wary of the

132. https://www.twnsnd.com/posts/performant_third_party_scripts.html
133. https://www.soasta.com/blog/10-pro-tips-for-managing-the-performance-of-your-third-party-scripts/
134. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control
135. https://devcenter.heroku.com/articles/increasing-application-performance-with-http-cache-headers
136. https://jakearchibald.com/2016/caching-best-practices/

vary header[138], especially in relation to CDNs[139], and watch out for the Key header[140] which helps avoiding an additional round trip for validation whenever a new request differs slightly, but not significantly, from prior requests. (*Thanks, Guy!*)

137. https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/http-caching?hl=en
138. https://www.smashingmagazine.com/2017/11/understanding-vary-header/
139. https://www.fastly.com/blog/getting-most-out-vary-fastly
140. https://www.greenbytes.de/tech/webdav/draft-ietf-httpbis-key-latest.html

Assets Optimizations

Is Brotli Or Zopfli Plain Text Compression In Use?

In 2015, Google introduced[141] Brotli[142], a new open-source lossless data format, which is now supported in all modern browsers[143]. In practice, Brotli appears to be more effective[144] than Gzip and Deflate. It might be (very) slow to compress, depending on the settings, but slower compression will ultimately lead to higher compression rates. Still, it decompresses fast.

Browsers will accept it only if the user is visiting a website over HTTPS though. What's the catch? Brotli still doesn't come preinstalled on some servers today, and it's not straightforward to set up without self-compiling NGINX or Ubuntu. Still, it's not that difficult[145]. In fact, some CDNs support it[146] and you can even enable Brotli even on CDNs that don't support it[147] yet (with a service worker).

141. https://opensource.googleblog.com/2015/09/introducing-brotli-new-compression.html
142. https://github.com/google/brotli
143. http://caniuse.com/#search=brotli
144. https://samsaffron.com/archive/2016/06/15/the-current-state-of-brotli-compression
145. https://www.smashingmagazine.com/2016/10/next-generation-server-compression-with-brotli/
146. https://community.akamai.com/community/web-performance/blog/2017/08/18/brotli-support-enablement-on-akamai
147. http://calendar.perfplanet.com/2016/enabling-brotli-even-on-cdns-that-dont-support-it-yet/

At the highest level of compression, Brotli is so slow that any potential gains in file size could be nullified by the amount of time it takes for the server to begin sending the response as it waits to dynamically compress the asset. With static compression, however, higher compression settings are preferred[148]. (*Thanks, Jeremy!*)

Alternatively, you could look into using Zopfli's compression algorithm[149], which encodes data to Deflate, Gzip and Zlib formats. Any regular Gzip-compressed resource would benefit from Zopfli's improved Deflate encoding, because the files will be 3 to 8% smaller than Zlib's maximum compression. The catch is that files will take around 80 times longer to compress. That's why it's a good idea to use Zopfli on resources that don't change much, files that are designed to be compressed once and downloaded many times.

If you can bypass the cost of dynamically compressing static assets, it's worth the effort. Both Brotli and Zopfli can be used for any plaintext payload — HTML, CSS, SVG, JavaScript, and so on.

The strategy? Pre-compress static assets with Brotli+Gzip[150] at the highest level and compress (dynamic) HTML on the fly with Brotli at level 1–4. Also, check for Brotli support on CDNs (e.g. *KeyCDN*, *CDN77*, *Fastly*). Make sure that the server handles content negotiation for Brotli or gzip properly. If you can't install/maintain Brotli on the server, use Zopfli.

148. https://css-tricks.com/brotli-static-compression/
149. https://blog.codinghorror.com/zopfli-optimization-literally-free-bandwidth/
150. https://css-tricks.com/brotli-static-compression/

Are Images Properly Optimized?

As far as possible, use responsive images[151] with `srcset`, `sizes` and the `<picture>` element. While you're at it, you could also make use of the WebP format[152] (supported in Chrome, Opera, Firefox soon[153]) by serving WebP images with the `<picture>` element and a JPEG fallback (see Andreas Bovens' code snippet[154]) or by using content negotiation (using `Accept` headers).

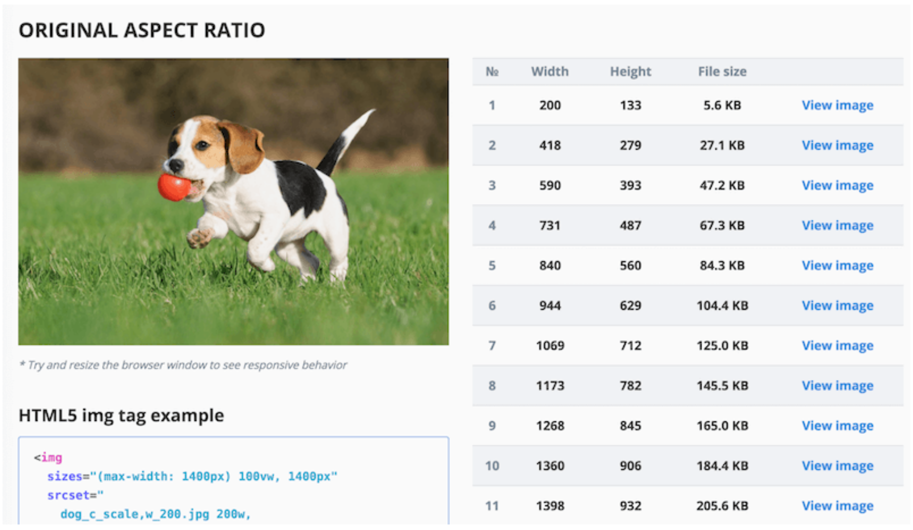
Sketch natively supports WebP, and WebP images can be exported from Photoshop using a WebP plugin for Photoshop[155]. Other options are available[156], too. If you're using WordPress or Joomla, there are extensions to help you easily implement support for WebP, such as Optimus[157] and Cache Enabler[158] for WordPress and Joomla's own supported extension[159] (via Cody Arsenault[160]).

You can also use client hints[161], which still have to gain some browser support[162]. Not enough resources to bake in sophisticated markup for responsive images? Use the

151. https://www.smashingmagazine.com/2014/05/responsive-images-done-right-guide-picture-srcset/
152. https://www.smashingmagazine.com/2015/10/webp-images-and-performance/
153. https://bugzilla.mozilla.org/show_bug.cgi?id=1294490
154. https://dev.opera.com/articles/responsive-images/#different-image-types-use-case
155. http://telegraphics.com.au/sw/product/WebPFormat#webpformat
156. https://developers.google.com/speed/webp/docs/using
157. https://wordpress.org/plugins/optimus/
158. https://wordpress.org/plugins/cache-enabler/
159. https://extensions.joomla.org/extension/webp/
160. https://css-tricks.com/comparing-novel-vs-tried-true-image-formats/
161. https://www.smashingmagazine.com/2016/01/leaner-responsive-images-client-hints/
162. http://caniuse.com/#search=client-hints

Responsive Image Breakpoints Generator[163] or a service such as Cloudinary[164] to automate image optimization. Also, in many cases, using `srcset` and `sizes` alone will reap significant benefits.

On Smashing Magazine, we use the postfix `-opt` for image names — for example, `brotli-compression-opt.png`; whenever an image contains that postfix, everybody on the team knows that the image has already been optimized.



ORIGINAL ASPECT RATIO

* Try and resize the browser window to see responsive behavior

HTML5 img tag example

```
<img
  sizes="(max-width: 1400px) 100vw, 1400px"
  srcset="
    dog_c_scale,w_200.jpg 200w,
```

№	Width	Height	File size	
1	200	133	5.6 KB	View image
2	418	279	27.1 KB	View image
3	590	393	47.2 KB	View image
4	731	487	67.3 KB	View image
5	840	560	84.3 KB	View image
6	944	629	104.4 KB	View image
7	1069	712	125.0 KB	View image
8	1173	782	145.5 KB	View image
9	1268	845	165.0 KB	View image
10	1360	906	184.4 KB	View image
11	1398	932	205.6 KB	View image

The Responsive Image Breakpoints Generator[165] automates images and markup generation.

163. http://www.responsivebreakpoints.com/
164. http://cloudinary.com/documentation/api_and_access_identifiers
165. http://www.responsivebreakpoints.com/

Take Image Optimization To The Next Level

When you're working on a landing page on which it's critical that a particular image loads blazingly fast, make sure that JPEGs are progressive and compressed with Adept[166], mozJPEG[167] (which improves the start rendering time by manipulating scan levels) or Guetzli[168], Google's new open source encoder focusing on perceptual performance, and utilizing learnings from Zopfli and WebP. The only downside[169]: slow processing times (a minute of CPU per megapixel). For PNG, we can use Pingo[170], and SVGO[171] or SVGOMG[172] for SVG.

Every single image optimization article would state it, but keeping vector assets clean and tight is always worth reminding. Make sure to clean up unused assets, remove unnecessary metadata and reduces the amount of path points in artwork (and thus SVG code). (*Thanks, Jeremy!*)

These optimizations so far cover just the basics. Addy Osmani has published a very detailed guide on Essential Image Optimization[173] that goes very deep into details of image compression and color management. For example, you could blur out unnecessary parts of the image (by applying a Gaussian blur filter to them) to reduce the file

166. https://github.com/technopagan/adept-jpg-compressor
167. https://github.com/mozilla/mozjpeg
168. https://github.com/google/guetzli
169. https://medium.com/@fox/talk-the-state-of-the-web-3e12f8e413b3
170. http://css-ig.net/pingo
171. https://www.npmjs.com/package/svgo
172. https://jakearchibald.github.io/svgomg/
173. https://images.guide/

size, and eventually you might even start removing colors or turn the picture into black and white to reduce the size even further. For background images, exporting photos from Photoshop with 0 to 10% quality can be absolutely acceptable as well.

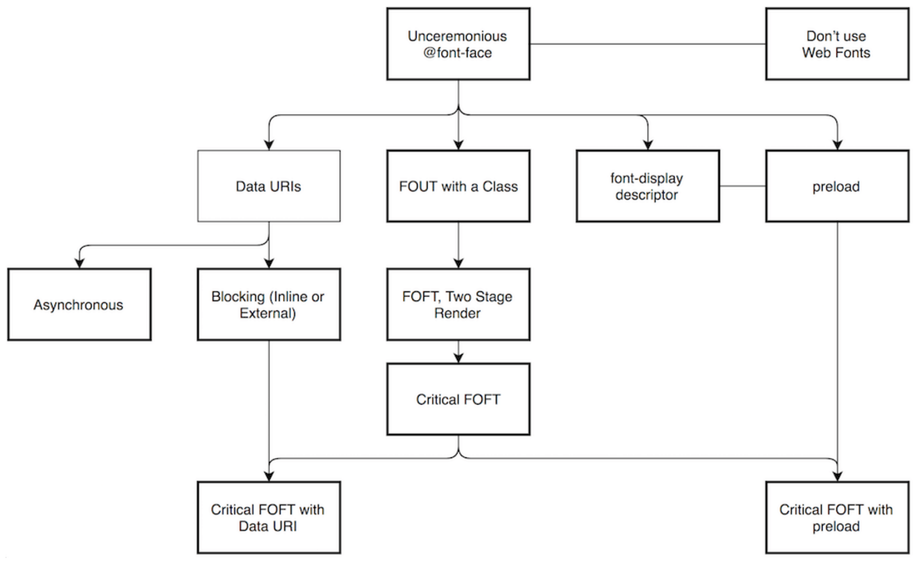
What about GIFs? Well, instead of loading heavy animated GIFs which impact both rendering performance and bandwidth, we could potentially use looping HTML5 videos[174], yet browser performance is slow with[175] `<video>`[176] and, unlike with images, browsers do not preload `<video>` content. At least we can add lossy compression to GIFs with Lossy GIF[177], gifsicle[178]or giflossy[179].

Good[180] news[181]: hopefully soon we'll be able to use `<img src=".mp4">` to load videos, and early tests show that `img` tags display 20× faster and decode 7× faster[182] than the GIF equivalent, in addition to being a fraction in file size.

Not good enough? Well, you can also improve perceived performance for images with the multiple[183] background[184] images[185] technique[186]. Keep in mind that play-

174. https://bitsofco.de/optimising-gifs/
175. https://calendar.perfplanet.com/2017/animated-gif-without-the-gif/#-but-we-already-have-video-tags
176. https://calendar.perfplanet.com/2017/animated-gif-without-the-gif/#-but-we-already-have-video-tags
177. https://kornel.ski/lossygif
178. https://github.com/kohler/gifsicle
179. https://github.com/pornel/giflossy
180. https://developer.apple.com/safari/technology-preview/release-notes/
181. https://bugs.chromium.org/p/chromium/issues/detail?id=791658
182. https://calendar.perfplanet.com/2017/animated-gif-without-the-gif/
183. http://csswizardry.com/2016/10/improving-perceived-performance-with-multiple-background-images/
184. https://jmperezperez.com/medium-image-progressive-loading-placeholder/

ing with contrast[187] and blurring out unnecessary details (or removing colors) can reduce file size as well. Ah, you need to enlarge a small photo without losing quality? Consider using Letsenhance.io[188].



Zach Leatherman's Comprehensive Guide to Font-Loading Strategies[189] *provides a dozen of options for better web font delivery.*

Are Web Fonts Optimized?

The first question that's worth asking if you can get away with using UI system fonts[190] in the first place. If it's not the case, chances are high that the web fonts you are serv-

185. https://manu.ninja/dominant-colors-for-lazy-loading-images#tiny-thumbnails
186. https://css-tricks.com/the-blur-up-technique-for-loading-background-images/
187. https://css-tricks.com/contrast-swap-technique-improved-image-performance-css-filters/
188. https://letsenhance.io
189. https://www.zachleat.com/web/comprehensive-webfonts/
190. https://www.smashingmagazine.com/2015/11/using-system-ui-fonts-practical-guide/

ing include glyphs and extra features and weights that aren't being used. You can ask your type foundry to subset web fonts or subset them yourself[191] if you are using open-source fonts (for example, by including only Latin with some special accent glyphs) to minimize their file sizes.

WOFF2 support[192] is great, and you can use WOFF and OTF as fallbacks for browsers that don't support it. Also, choose one of the strategies from Zach Leatherman's "Comprehensive Guide to Font-Loading Strategies[193]," (code snippets also available as Web font loading recipes[194]) and use a service worker cache to cache fonts persistently. Need a quick win? Pixel Ambacht has a quick tutorial and case study[195] to get your fonts in order.

If you can't serve fonts from your server and are relying on third-party hosts, make sure to use Font Load Events[196] (or Web Font Loader[197] for browsers not supporting it). FOUT is better than FOIT[198]; start rendering text in the fallback right away, and load fonts asynchronously — you could also use loadCSS[199] for that. You might be able to get away with locally installed OS

191. https://www.fontsquirrel.com/tools/webfont-generator
192. http://caniuse.com/#search=woff2
193. https://www.zachleat.com/web/comprehensive-webfonts/
194. https://github.com/zachleat/web-font-loading-recipes
195. https://pixelambacht.nl/2016/font-awesome-fixed/
196. https://www.igvita.com/2014/01/31/optimizing-web-font-rendering-performance/#font-load-events
197. https://github.com/typekit/webfontloader
198. https://www.filamentgroup.com/lab/font-events.html
199. https://github.com/filamentgroup/loadCSS

fonts[200] as well, or use variable[201] fonts[202] that are gaining traction[203], too.

What would make a bulletproof font loading strategy? Start with `font-display`, then fall back to the Font Loading API, *then* fall back to Bram Stein's Font Face Observer[204]. (*Thanks Jeremy!*) And if you're interested in measuring the performance of font loading from the user's perspective, Andreas Marschke explores performance tracking with Font API and UserTiming API[205].

Also, don't forget to include the `font-display: optional`[206] descriptor for resilient and fast font fallbacks, `unicode-range`[207] to break down a large font into smaller language-specific fonts, and Monica Dinculescu's font-style-matcher[208] to minimize a jarring shift in layout, due to sizing discrepancies between the two fonts.

200. https://www.smashingmagazine.com/2015/11/using-system-ui-fonts-practical-guide/
201. https://alistapart.com/blog/post/variable-fonts-for-responsive-design
202. https://www.smashingmagazine.com/2017/09/new-font-technologies-improve-web/
203. https://caniuse.com/#search=variable
204. https://github.com/bramstein/fontfaceobserver
205. https://www.andreas-marschke.name/posts/2017/12/29/Fonts-API-UserTiming-Boomerang.html
206. https://font-display.glitch.me/
207. https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2015/august/how-to-subset-fonts-with-unicode-range/
208. https://meowni.ca/font-style-matcher/

Delivery Optimizations

Do You Limit The Impact Of JavaScript Libraries, And Load Them Asynchronously?

When the user requests a page, the browser fetches the HTML and constructs the DOM, then fetches the CSS and constructs the CSSOM, and then generates a rendering tree by matching the DOM and CSSOM. If any JavaScript needs to be resolved, the browser won't start rendering the page until it's resolved, thus delaying rendering. As developers, we have to explicitly tell the browser not to wait and to start rendering the page. The way to do this for scripts is with the `defer` and `async` attributes in HTML.

In practice, it turns out we should prefer `defer` to `async`[209] (at a cost to users of Internet Explorer[210] up to and including version 9, because you're likely to break scripts for them). Also, as mentioned above, limit the impact of third-party libraries and scripts, especially with social sharing buttons and `<iframe>` embeds (such as maps). Size Limit[211] helps you prevent JavaScript libraries bloat[212]: If you accidentally add a large dependency, the tool will inform you and throw an error. You can use stat-

209. http://calendar.perfplanet.com/2016/prefer-defer-over-async/
210. https://github.com/h5bp/lazyweb-requests/issues/42
211. https://github.com/ai/size-limit
212. https://evilmartians.com/chronicles/size-limit-make-the-web-lighter

ic social sharing buttons[213] (such as by SSBG[214]) and static links to interactive maps[215] instead.

Are You Lazy-Loading Expensive Scripts With Intersection Observer?

If you need to lazy-load images, videos, ad scripts, A/B testing scripts or any other resources, you can use the shiny new Intersection Observer API[216] that provides a way to asynchronously observe changes in the intersection of a target element with an ancestor element or with a top-level document's viewport. Basically, you need to create a new IntersectionObserver object, which receives a callback function and a set of options. Then we add a target to observe.

The callback function executes when the target becomes visible or invisible, so when it intercepts the viewport, you can start taking some actions before the element becomes visible. In fact, we have a granular control over when the observer's callback should be invoked, with `rootMargin` (margin around the root) and threshold (a single number or an array of numbers which indicate at what percentage of the target's visibility we are aiming). Alejandro Garcia Anglada has published a handy tutorial[217] on how to actually implement it.

213. https://www.savjee.be/2015/01/Creating-static-social-share-buttons/
214. https://simplesharingbuttons.com
215. https://developers.google.com/maps/documentation/static-maps/intro
216. https://developer.mozilla.org/en-US/docs/Web/API/Intersection_Observer_API
217. https://medium.com/@aganglada/intersection-observer-in-action-efc118062366

You could even take it to the next level by adding progressive image loading[218] to your pages. Similarly to Facebook, Pinterest and Medium, you could load low quality or even blurry images first, and then as the page continues to load, replace them with the full quality versions, using the LQIP (Low Quality Image Placeholders) technique[219] proposed by Guy Podjarny.

Opinions differ if the technique improves user experience or not, but it definitely improves time to first meaningful paint. We can even automate it by using SQIP[220] that creates a low quality version of an image as an SVG placeholder. These placeholders could be embedded within HTML as they naturally compress well with text compression methods. In his article, Dean Hume has described[221] how this technique can be implemented using Intersection Observer.

Browser support? Decent[222], with Chrome, Firefox, Edge and Samsung Internet being on board. WebKit status is currently in development[223]. Fallback? If we don't have support for intersection observer, we can still lazy load[224] a polyfill[225] or load the images immediately. And there is even a library[226] for it.

---

218. https://calendar.perfplanet.com/2017/progressive-image-loading-using-intersection-observer-and-sqip/
219. https://www.guypo.com/introducing-lqip-low-quality-image-placeholders/
220. https://github.com/technopagan/sqip
221. https://calendar.perfplanet.com/2017/progressive-image-loading-using-intersection-observer-and-sqip/
222. https://caniuse.com/#feat=intersectionobserver
223. https://webkit.org/status/#specification-intersection-observer
224. https://medium.com/@aganglada/intersection-observer-in-action-efc118062366
225. https://github.com/jeremenichelli/intersection-observer-polyfill
226. https://github.com/ApoorvSaxena/lozad.js

In general, it's a good idea to lazy-load all expensive components, such as fonts, JavaScript, carousels, videos and iframes. You could even adapt content serving based on effective network quality. Network Information API[227] and specifically `navigator.connection.effectiveType` (Chrome 62+) use RTT and downlink values to provide a slightly more accurate representation of the connection and the data that users can handle. You can use it to remove video autoplay, background images or web fonts entirely for connections that are too slow.

Do You Push Critical CSS Quickly?

To ensure that browsers start rendering your page as quickly as possible, it's become a common practice[228] to collect all of the CSS required to start rendering the first visible portion of the page (known as "critical CSS" or "above-the-fold CSS") and add it inline in the `<head>` of the page, thus reducing roundtrips. Due to the limited size of packages exchanged during the slow start phase, your budget for critical CSS is around 14 KB.

If you go beyond that, the browser will need additional roundtrips to fetch more styles. CriticalCSS[229] and Critical[230] enable you to do just that. You might need to do it for every template you're using. If possible, consider us-

227. https://googlechrome.github.io/samples/network-information/
228. https://www.smashingmagazine.com/2015/08/understanding-critical-css/
229. https://github.com/filamentgroup/criticalCSS
230. https://github.com/addyosmani/critical

ing the conditional inlining approach[231] used by the Filament Group.

With HTTP/2, critical CSS could be stored in a separate CSS file and delivered via a server push[232] without bloating the HTML. The catch is that server pushing is troublesome[233] with many gotchas and race conditions across browsers. It isn't supported consistently and has some caching issues (see slide 114 onwards of Hooman Beheshti's presentation[234]). The effect could, in fact, be negative[235] and bloat the network buffers, preventing genuine frames in the document from being delivered. Also, it appears that server pushing is much more effective on warm connections[236] due to the TCP slow start.

Even with HTTP/1, putting critical CSS in a separate file on the root domain has benefits[237], sometimes even more than inlining due to caching. Chrome speculatively opens a second HTTP connection to the root domain when requesting the page, which removes the need for a TCP connection to fetch this CSS. (*Thanks, Philip!*)

A few gotchas to keep in mind: unlike `preload` that can trigger preload from any domain, you can only push resources from your own domain or domains you are authoritative for. It can be initiated as soon as the server gets the very first request from the client. Server pushed

231. https://www.filamentgroup.com/lab/modernizing-delivery.html
232. https://www.filamentgroup.com/lab/modernizing-delivery.html
233. https://twitter.com/jaffathecake/status/867699157150117888
234. http://www.slideshare.net/Fastly/http2-what-no-one-is-telling-you
235. https://jakearchibald.com/2017/h2-push-tougher-than-i-thought/
236. https://docs.google.com/document/d/1K0NykTXBbbbTlv60t5MyJvXjqKGsCVNYHyLEXIxYMv0/edit
237. http://www.jonathanklein.net/2014/02/revisiting-cookieless-domain.html

resources land in the Push cache and are removed when the connection is terminated. However, since an HTTP/2 connection can be re-used across multiple tabs, pushed resources can be claimed by requests from other tabs as well. (*Thanks, Inian!*)

At the moment, there is no simple way for the server to know if pushed resources are already in one of user's caches[238], so resources will keep being pushed with every user's visit. So, you might need to create a cache-aware HTTP/2 server push mechanism[239]. If fetched, you could try to get them from a cache based on the index of what's already in the cache, avoiding secondary server pushes altogether.

Keep in mind, though, that the new `cache-digest` specification[240] negates the need to manually build such "cache-aware" servers, basically declaring a new frame type in HTTP/2 to communicate what's already in the cache for that hostname. As such, it could be particularly useful for CDNs as well.

For dynamic content, when a server needs some time to generate a response, the browser isn't able to make any requests since it's not aware of any sub-resources that the page might reference. For that case, we can warm up the connection and increase the TCP congestion window size, so that future requests can be completed faster. Also, all inlined assets are usually good candidates for server pushing. In fact, Inian Parameshwaran did a remarkable

238. https://blog.yoav.ws/tale-of-four-caches/
239. https://css-tricks.com/cache-aware-server-push/
240. http://calendar.perfplanet.com/2016/cache-digests-http2-server-push/

research comparing HTTP/2 Push vs. HTTP Preload[241], and it's a fantastic read with all the details you might need. Server Push or Not Server Push? Colin Bendell's "Should I Push?"[242] might point you in the right direction.

Bottom line: As Sam Saccone said[243], `preload` is good for moving the start download time of an asset closer to the initial request, while Server Push is good for cutting out a full RTT (or more[244], depending on your server think time) — if you have a service worker to prevent unnecessary pushing, that is.

Do You Stream Responses?

Often forgotten and neglected, streams[245] provide an interface for reading or writing asynchronous chunks of data, only a subset of which might be available in memory at any given time. Basically, they allow the page that made the original request to start working with the response as soon as the first chunk of data is available, and use parsers that are optimized for streaming to progressively display the content.

We could create one stream from multiple sources. For example, instead of serving an empty UI shell and letting JavaScript populate it, you can let the service worker construct a stream where the shell comes from a cache, but the body comes from the network. As Jeff Posnick

241. https://dexecure.com/blog/http2-push-vs-http-preload/
242. https://shouldipush.com/
243. https://medium.com/@samccone/performance-futures-bundling-281543d9a0d5
244. https://blog.yoav.ws/being_pushy/
245. https://streams.spec.whatwg.org/

noted[246], if your web app is powered by a CMS that server-renders HTML by stitching together partial templates, that model translates directly into using streaming responses, with the templating logic replicated in the service worker instead of your server. Jake Archibald's "The Year of Web Streams"[247] article highlights how exactly you could build it. Performance boost is quite noticeable[248].

One important advantage of streaming the entire HTML response is that HTML rendered during the initial navigation request can take full advantage of the browser's streaming HTML parser. Chunks of HTML that are inserted into a document after the page has loaded (as is common with content populated via JavaScript) can't take advantage of this optimization.

Browser support? Getting there[249] with Chrome 52+, Firefox 57+ (behind flag), Safari and Edge supporting the API and Service Workers being supported in all modern browsers[250].

Are You Saving Data With Save-Data?

Especially when working in emerging markets, you might need to consider optimizing experience for users who choose to opt into data savings. The Save-Data client hint request header[251] allows us to customize the applica-

246. https://developers.google.com/web/updates/2016/06/sw-readablestreams
247. https://jakearchibald.com/2016/streams-ftw/
248. https://www.youtube.com/watch?v=Cjo9iq8k-bc
249. https://caniuse.com/#search=streams
250. https://caniuse.com/#search=serviceworker

tion and the payload to cost- and performance-constrained users. In fact, you could rewrite requests for high DPI images to low DPI images[252], remove web fonts and fancy parallax effects, turn off video autoplay, server pushes or even change how you deliver markup.

The header is currently supported only in Chromium, on the Android version of Chrome or via the Data Saver extension on a desktop device. Finally, you can also use service workers and the Network Information API to deliver low/high resolution images based on the network type[253].

Do You Warm Up The Connection To Speed Up Delivery?

Use resource hints[254] to save time on `dns-prefetch`[255] (which performs a DNS lookup in the background), `preconnect`[256] (which asks the browser to start the connection handshake (DNS, TCP, TLS) in the background), `prefetch`[257] (which asks the browser to request a resource) and `preload`[258] (which prefetches resources without executing them, among other things).

Most of the time these days, we'll be using at least `preconnect` and `dns-prefetch`, and we'll be cautious

251. https://developers.google.com/web/updates/2016/02/save-data
252. https://css-tricks.com/help-users-save-data/
253. https://justmarkup.com/log/2017/11/network-based-image-loading/
254. https://w3c.github.io/resource-hints
255. http://caniuse.com/#search=dns-prefetch
256. http://www.caniuse.com/#search=preconnect
257. http://caniuse.com/#search=prefetch
258. https://www.smashingmagazine.com/2016/02/preload-what-is-it-good-for/

with using `prefetch` and `preload`; the former should only be used if you are very confident about what assets the user will need next (for example, in a purchasing funnel). Notice that `prerender` has been deprecated and is no longer supported.

Note that even with `preconnect` and `dns-prefetch`, the browser has a limit on the number of hosts it will look up/connect to in parallel, so it's a safe bet to order them based on priority. (*Thanks, Philip!*)

In fact, using resource hints is probably the easiest way to boost performance, and it works well indeed[259]. When to use what? As Addy Osmani has explained[260], we should preload resources that we have high-confidence will be used in the current page. Prefetch resources likely to be used for future navigations across multiple navigation boundaries, e.g. Webpack bundles needed for pages the user hasn't visited yet.

Addy's article on Loading Priorities in Chrome shows[261] how exactly Chrome interprets resource hints, so once you've decided which assets are critical for rendering, you can assign high priority to them. To see how your requests are prioritized, you can enable a "priority" column in the Chrome DevTools network request table (as well as Safari Technology Preview).

259. https://medium.com/reloading/preload-prefetch-and-priorities-in-chrome-776165961bbf
260. https://medium.com/reloading/preload-prefetch-and-priorities-in-chrome-776165961bbf
261. https://medium.com/reloading/preload-prefetch-and-priorities-in-chrome-776165961bbf

For example, since fonts usually are important assets on a page, it's always a good idea to request the browser to download fonts with `preload`[262]. You could also load JavaScript dynamically[263], effectively lazy-loading execution. Also, since `<link rel="preload">` accepts a `media` attribute, you could choose to selectively prioritize resources[264] based on `@media` query rules.

A few gotchas to keep in mind[265]: preload is good for moving the start download time of an asset[266] closer to the initial request, but preloaded assets land in the memory cache which is tied to the page making the request. It means that preloaded requests cannot be shared across pages. Also, `preload` plays well with the HTTP cache: a network request is never sent if the item is already there in the HTTP cache.

Hence, it's useful for late-discovered resources, a hero image loaded via background-image, inlining critical CSS (or JavaScript) and pre-loading the rest of the CSS (or JavaScript). Also, a `preload` tag can initiate a preload only after the browser has received the HTML from the server and the lookahead parser has found the `preload` tag. Preloading via the HTTP header is a bit faster since we don't to wait for the browser to parse the HTML to start the request. Early Hints[267] will help even further, enabling pre-

262. https://css-tricks.com/the-critical-request/#article-header-id-2
263. https://www.smashingmagazine.com/2016/02/preload-what-is-it-good-for/#dynamic-loading-without-execution
264. https://css-tricks.com/the-critical-request/#article-header-id-3
265. https://dexecure.com/blog/http2-push-vs-http-preload/
266. https://www.youtube.com/watch?v=RWLzUnESylc
267. https://tools.ietf.org/html/draft-ietf-httpbis-early-hints-05

load to kick in even before the response headers for the HTML are sent.

Beware: If you're using `preload`, `as` must be defined or nothing loads[268], plus preloaded fonts without the[269] `crossorigin`[270] attribute will double fetch[271].

Have You Optimized Rendering Performance?

Isolate expensive components with CSS containment[272] — for example, to limit the scope of the browser's styles, of layout and paint work for off-canvas navigation, or of third-party widgets. Make sure that there is no lag when scrolling the page or when an element is animated, and that you're consistently hitting 60 frames per second. If that's not possible, then at least making the frames per second consistent is preferable to a mixed range of 60 to 15. Use CSS' `will-change`[273] to inform the browser of which elements and properties will change.

Also, measure runtime rendering performance[274] (for example, in DevTools[275]). To get started, check Paul Lewis'

268. https://twitter.com/yoavweiss/status/873077451143774209
269. https://medium.com/reloading/preload-prefetch-and-priorities-in-chrome-776165961bbf
270. https://medium.com/reloading/preload-prefetch-and-priorities-in-chrome-776165961bbf
271. https://medium.com/reloading/preload-prefetch-and-priorities-in-chrome-776165961bbf
272. http://caniuse.com/#search=contain
273. http://caniuse.com/#feat=will-change
274. https://aerotwist.com/blog/my-performance-audit-workflow/#runtime-performance
275. https://developers.google.com/web/tools/chrome-devtools/rendering-tools/

free Udacity course on browser-rendering optimization[276] and Emily Hayman's article on Performant Web Animations and Interactions[277].

We also have a lil' article by Sergey Chikuyonok on how to get GPU animation right[278]. Quick note: changes to GPU-composited layers are the least expensive[279], so if you can get away by triggering only compositing via `opacity` and `transform`, you'll be on the right track.

Have You Optimized Rendering Experience?

While the sequence of how components appear on the page, and the strategy of how we serve assets to the browser matter, we shouldn't underestimate the role of perceived performance[280], too. The concept deals with psychological aspects of waiting, basically keeping customers busy or engaged while something else is happening. That's where perception management[281], preemptive

276. https://www.udacity.com/course/browser-rendering-optimization--ud860
277. https://blog.algolia.com/performant-web-animations/
278. https://www.smashingmagazine.com/2016/12/gpu-animation-doing-it-right/
279. https://blog.algolia.com/performant-web-animations/
280. https://www.smashingmagazine.com/2015/09/why-performance-matters-the-perception-of-time/
281. https://www.smashingmagazine.com/2015/11/why-performance-matters-part-2-perception-management/

start[282], early completion[283] and tolerance management[284] come into play.

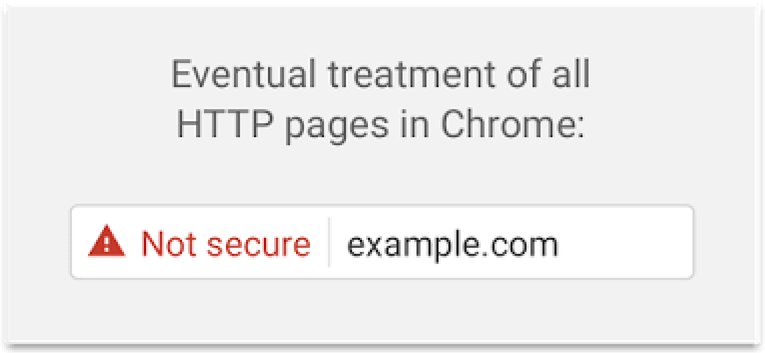
What does it all mean? While loading assets, we can try to always be one step ahead of the customer, so the experience feels swift while there is quite a lot happening in the background. To keep the customer engaged, we can use skeleton screens[285] (implementation demo[286]) instead of loading indicators, add transitions/animations and basically cheat the UX[287] when there is nothing more to optimize.

282. https://www.smashingmagazine.com/2015/11/why-performance-matters-part-2-perception-management/#preemptive-start
283. https://www.smashingmagazine.com/2015/11/why-performance-matters-part-2-perception-management/#early-completion
284. https://www.smashingmagazine.com/2015/12/performance-matters-part-3-tolerance-management/
285. https://twitter.com/lukew/status/665288063195594752
286. https://twitter.com/razvancaliman/status/734088764960690176
287. https://blog.stephaniewalter.fr/en/cheating-ux-perceived-performance-and-user-experience/

HTTP/2

Migrate To HTTPS, Turn On HTTP/2

With Google moving towards a more secure web[288] and eventual treatment of all HTTP pages in Chrome as being "not secure," a switch to HTTP/2 environment[289] is unavoidable. HTTP/2 is supported very well[290]; it isn't going anywhere; and, in most cases, you're better off with it. Once running on HTTPS already, you can get a major performance boost[291] with service workers and server push (at least long term).



Eventually, Google plans to label all HTTP pages as non-secure, and change the HTTP security indicator to the red triangle that Chrome uses for broken HTTPS. (Image source[292])

288. https://security.googleblog.com/2016/09/moving-towards-more-secure-web.html
289. https://http2.github.io/faq/
290. http://caniuse.com/#search=http2
291. https://youtu.be/RWLzUnESylc
292. https://security.googleblog.com/2016/09/moving-towards-more-secure-web.html

The most time-consuming task will be to migrate to HTTPS[293], and depending on how large your HTTP/1.1 user base is (that is, users on legacy operating systems or with legacy browsers), you'll have to send a different build for legacy browsers performance optimizations, which would require you to adapt a different build process[294]. Beware: Setting up both migration and a new build process might be tricky and time-consuming. For the rest of this article, I'll assume that you're either switching to or have already switched to HTTP/2.

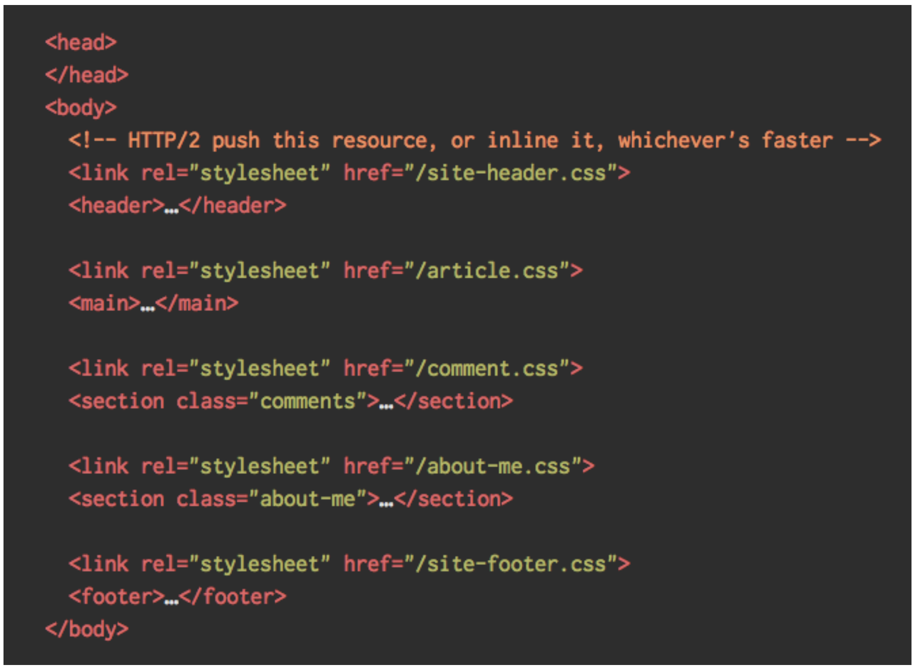
Properly Deploy HTTP/2

Again, serving assets over HTTP/2[295] requires a partial overhaul of how you've been serving assets so far. You'll need to find a fine balance between packaging modules and loading many small modules in parallel. In the end of the day, still the best request is no request[296], however the goal is to find a fine balance between quick first delivery of assets and caching.

On the one hand, you might want to avoid concatenating assets altogether, instead breaking down your entire interface into many small modules, compressing them as a part of the build process, referencing them via the "scout" approach[297] and loading them in parallel. A change in one file won't require the entire style sheet or

293. https://https.cio.gov/faq/
294. https://rmurphey.com/blog/2015/11/25/building-for-http2
295. https://www.youtube.com/watch?v=yURLTwZ3ehk
296. http://alistapart.com/article/the-best-request-is-no-request-revisited
297. https://rmurphey.com/blog/2015/11/25/building-for-http2

JavaScript to be re-downloaded. It also minimizes parsing time[298] and keeps the payloads of individual pages low.



```
<head>
</head>
<body>
  <!-- HTTP/2 push this resource, or inline it, whichever's faster -->
  <link rel="stylesheet" href="/site-header.css">
  <header>…</header>

  <link rel="stylesheet" href="/article.css">
  <main>…</main>

  <link rel="stylesheet" href="/comment.css">
  <section class="comments">…</section>

  <link rel="stylesheet" href="/about-me.css">
  <section class="about-me">…</section>

  <link rel="stylesheet" href="/site-footer.css">
  <footer>…</footer>
</body>
```

To achieve best results with HTTP/2, consider to load CSS progressively[299]*, as suggested by Chrome's Jake Archibald.*

On the other hand, packaging still matters[300]. First, compression will suffer. The compression of a large package will benefit from dictionary reuse, whereas small separate packages will not. There's standard work to address that, but it's far out for now. Secondly, browsers have not yet been optimized for such workflows. For example, Chrome will trigger inter-process communications[301]

298. https://css-tricks.com/musings-on-http2-and-bundling/
299. https://jakearchibald.com/2016/link-in-body/
300. http://engineering.khanacademy.org/posts/js-packaging-http2.htm
301. https://www.chromium.org/developers/design-documents/inter-process-communication

(IPCs) linear to the number of resources, so including hundreds of resources will have browser runtime costs.

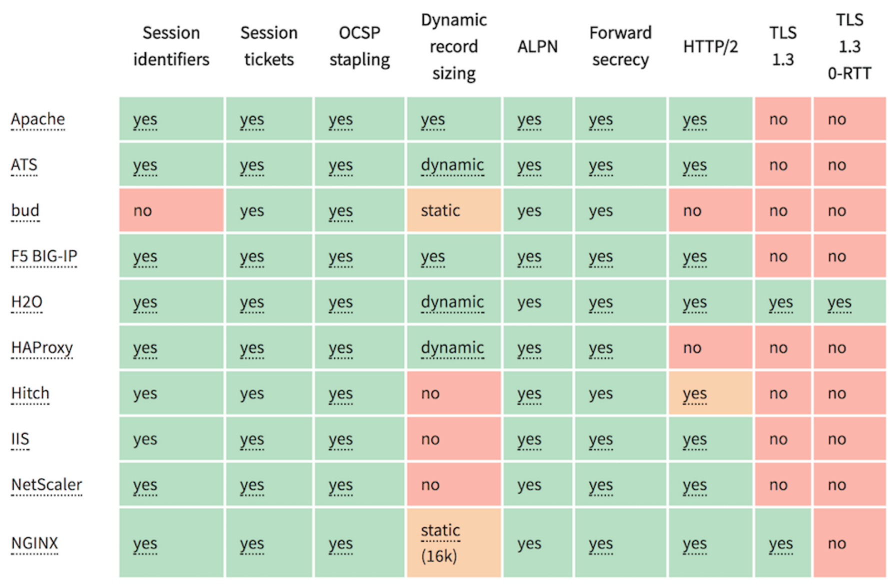
Still, you can try to load CSS progressively[302]. Obviously, by doing so, you are actively penalizing HTTP/1.1 users, so you might need to generate and serve different builds to different browsers as part of your deployment process, which is where things get slightly more complicated. You could get away with HTTP/2 connection coalescing[303], which allows you to use domain sharding while benefiting from HTTP/2, but achieving this in practice is difficult.

What to do? If you're running over HTTP/2, sending around 6–10 packages seems like a decent compromise (and isn't too bad for legacy browsers). Experiment and measure to find the right balance for your website.

Do Your Servers And CDNs Support HTTP/2?

Different servers and CDNs are probably going to support HTTP/2 differently. Use Is TLS Fast Yet?[304] to check your options, or quickly look up how your servers are performing and which features you can expect to be supported.

302. https://jakearchibald.com/2016/link-in-body/
303. https://daniel.haxx.se/blog/2016/08/18/http2-connection-coalescing/
304. https://istlsfastyet.com



	Session identifiers	Session tickets	OCSP stapling	Dynamic record sizing	ALPN	Forward secrecy	HTTP/2	TLS 1.3	TLS 1.3 0-RTT
Apache	yes	yes	yes	yes	yes	yes	yes	no	no
ATS	yes	yes	yes	dynamic	yes	yes	yes	no	no
bud	no	yes	yes	static	yes	yes	no	no	no
F5 BIG-IP	yes	yes	yes	yes	yes	yes	yes	no	no
H2O	yes	yes	yes	dynamic	yes	yes	yes	yes	yes
HAProxy	yes	yes	yes	dynamic	yes	yes	no	no	no
Hitch	yes	yes	yes	no	yes	yes	yes	no	no
IIS	yes	yes	yes	no	yes	yes	yes	no	no
NetScaler	yes	yes	yes	no	yes	yes	yes	no	no
NGINX	yes	yes	yes	static (16k)	yes	yes	yes	yes	no

Is TLS Fast Yet?[305] *allows you to check your options for servers and CDNs when switching to HTTP/2.*

Is OCSP Stapling Enabled?

By enabling OCSP stapling on your server[306], you can speed up your TLS handshakes. The Online Certificate Status Protocol (OCSP) was created as an alternative to the Certificate Revocation List (CRL) protocol. Both protocols are used to check whether an SSL certificate has been revoked. However, the OCSP protocol does not require the browser to spend time downloading and then searching a list for certificate information, hence reducing the time required for a handshake.

305. https://istlsfastyet.com
306. https://www.digicert.com/enabling-ocsp-stapling.htm

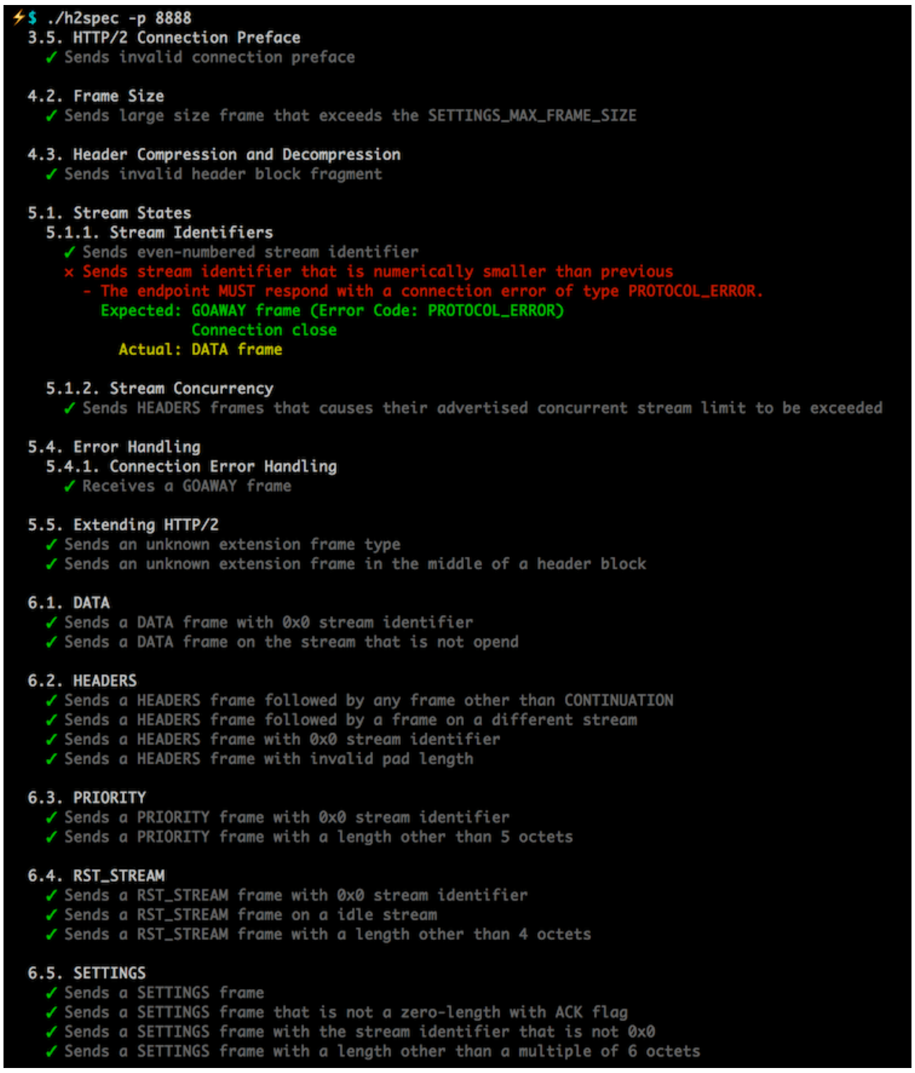
Have You Adopted IPv6 Yet?

Because we're running out of space with IPv4[307] and major mobile networks are adopting IPv6 rapidly (the US has reached[308] a 50% IPv6 adoption threshold), it's a good idea to update your DNS to IPv6[309] to stay bulletproof for the future. Just make sure that dual-stack support is provided across the network — it allows IPv6 and IPv4 to run simultaneously alongside each other. After all, IPv6 is not backwards-compatible. Also, studies show[310] that IPv6 made those websites 10 to 15% faster due to neighbor discovery (NDP) and route optimization.

Is HPACK Compression In Use?

If you're using HTTP/2, double-check that your servers implement HPACK compression[311] for HTTP response headers to reduce unnecessary overhead. Because HTTP/2 servers are relatively new, they may not fully support the specification, with HPACK being an example. H2spec[312] is a great (if very technically detailed) tool to check that. HPACK works[313].

307. https://en.wikipedia.org/wiki/IPv4_address_exhaustion
308. https://www.google.com/intl/en/ipv6/statistics.html#tab=ipv6-adoption&tab=ipv6-adoption
309. https://www.paessler.com/blog/2016/04/08/monitoring-news/ask-the-expert-current-status-on-ipv6
310. https://www.cloudflare.com/ipv6/
311. https://blog.cloudflare.com/hpack-the-silent-killer-feature-of-http-2/
312. https://github.com/summerwind/h2spec
313. https://www.keycdn.com/blog/http2-hpack-compression/



```
⚡$ ./h2spec -p 8888
  3.5. HTTP/2 Connection Preface
    ✓ Sends invalid connection preface

  4.2. Frame Size
    ✓ Sends large size frame that exceeds the SETTINGS_MAX_FRAME_SIZE

  4.3. Header Compression and Decompression
    ✓ Sends invalid header block fragment

  5.1. Stream States
    5.1.1. Stream Identifiers
      ✓ Sends even-numbered stream identifier
      × Sends stream identifier that is numerically smaller than previous
        - The endpoint MUST respond with a connection error of type PROTOCOL_ERROR.
          Expected: GOAWAY frame (Error Code: PROTOCOL_ERROR)
                    Connection close
            Actual: DATA frame

    5.1.2. Stream Concurrency
      ✓ Sends HEADERS frames that causes their advertised concurrent stream limit to be exceeded

  5.4. Error Handling
    5.4.1. Connection Error Handling
      ✓ Receives a GOAWAY frame

  5.5. Extending HTTP/2
    ✓ Sends an unknown extension frame type
    ✓ Sends an unknown extension frame in the middle of a header block

  6.1. DATA
    ✓ Sends a DATA frame with 0x0 stream identifier
    ✓ Sends a DATA frame on the stream that is not opend

  6.2. HEADERS
    ✓ Sends a HEADERS frame followed by any frame other than CONTINUATION
    ✓ Sends a HEADERS frame followed by a frame on a different stream
    ✓ Sends a HEADERS frame with 0x0 stream identifier
    ✓ Sends a HEADERS frame with invalid pad length

  6.3. PRIORITY
    ✓ Sends a PRIORITY frame with 0x0 stream identifier
    ✓ Sends a PRIORITY frame with a length other than 5 octets

  6.4. RST_STREAM
    ✓ Sends a RST_STREAM frame with 0x0 stream identifier
    ✓ Sends a RST_STREAM frame on a idle stream
    ✓ Sends a RST_STREAM frame with a length other than 4 octets

  6.5. SETTINGS
    ✓ Sends a SETTINGS frame
    ✓ Sends a SETTINGS frame that is not a zero-length with ACK flag
    ✓ Sends a SETTINGS frame with the stream identifier that is not 0x0
    ✓ Sends a SETTINGS frame with a length other than a multiple of 6 octets
```

H2spec (View large version[314]*) (Image source*[315]*)*

Make Sure The Security On Your Server Is Bulletproof

All browser implementations of HTTP/2 run over TLS, so you will probably want to avoid security warnings or

314. https://cloud.netlifyusercontent.com/assets/344dbf88-fdf9-42bb-adb4-46f01ee dd629/15891f86-c883-434a-8517-209273356ee6/h2spec-example-large-opt.png
315. https://github.com/summerwind/h2spec

some elements on your page not working. Double-check that your security headers are set properly[316], eliminate known vulnerabilities[317], and check your certificate[318]. Also, make sure that all external plugins and tracking scripts are loaded via HTTPS, that cross-site scripting isn't possible and that both HTTP Strict Transport Security headers[319] and Content Security Policy headers[320] are properly set.

Are Service Workers Used For Caching And Network Fallbacks?

No performance optimization over a network can be faster than a locally stored cache on user's machine. If your website is running over HTTPS, use the "Pragmatist's Guide to Service Workers[321]" to cache static assets in a service worker cache and store offline fallbacks (or even offline pages) and retrieve them from the user's machine, rather than going to the network. Also, check Jake's Offline Cookbook[322] and the free Udacity course "Offline Web Applications[323]." Browser support? As stated above, it's widely supported[324] (Chrome, Firefox, Safari TP, Sam-

316. https://securityheaders.io/
317. https://www.smashingmagazine.com/2016/01/eliminating-known-security-vulnerabilities-with-snyk/
318. https://www.ssllabs.com/ssltest/
319. https://www.owasp.org/index.php/HTTP_Strict_Transport_Security_Cheat_Sheet
320. https://content-security-policy.com/
321. https://github.com/lyzadanger/pragmatist-service-worker
322. https://jakearchibald.com/2014/offline-cookbook/
323. https://www.udacity.com/course/offline-web-applications--ud899
324. http://caniuse.com/#search=serviceworker

sung Internet, Edge 17+) and the fallback is the network anyway. Does it help boost performance? Oh yes, it does[325].

325. https://developers.google.com/web/showcase/2016/service-worker-perf

Testing And Monitoring

Have You Tested In Proxy Browsers And Legacy Browsers?

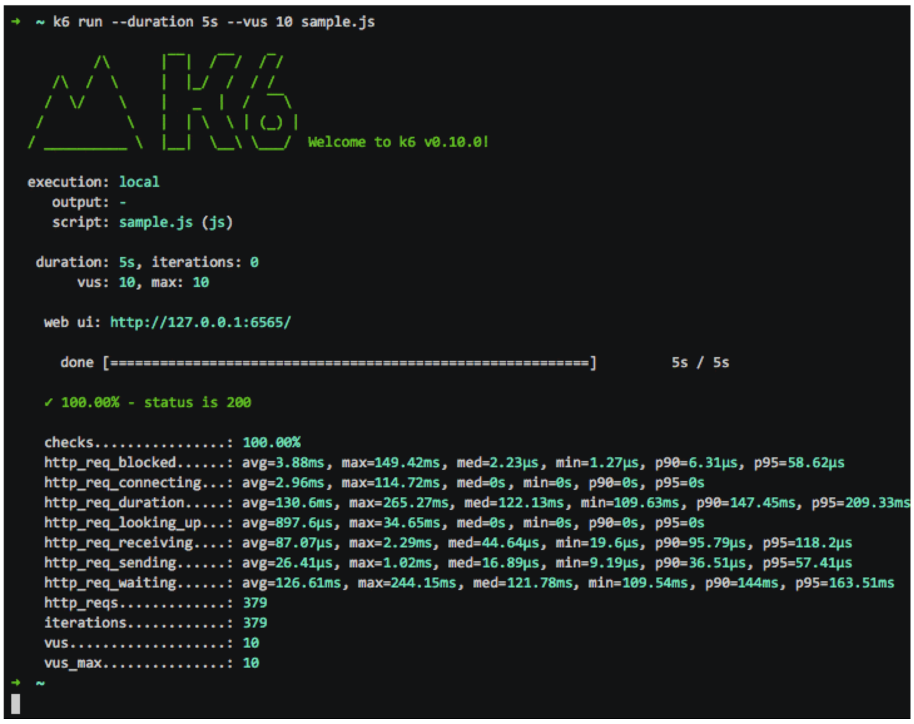
Testing in Chrome and Firefox is not enough. Look into how your website works in proxy browsers and legacy browsers. UC Browser and Opera Mini, for instance, have a significant market share in Asia[326] (up to 35% in Asia). Measure average Internet speed[327] in your countries of interest to avoid big surprises down the road. Test with network throttling, and emulate a high-DPI device. BrowserStack[328] is fantastic, but test on real devices as well.

Is Continuous Monitoring Set Up?

Having a private instance of WebPagetest[329] is always beneficial for quick and unlimited tests. However, a continuous monitoring tool with automatic alerts will give you a more detailed picture of your performance. Set your own user-timing marks to measure and monitor business-specific metrics. Also, consider adding automated performance regression alerts[330] to monitor changes over time.

---

326. http://gs.statcounter.com/#mobile_browser-as-monthly-201511-201611
327. https://www.webworldwide.io/
328. https://www.browserstack.com
329. http://www.webpagetest.org/
330. https://calendar.perfplanet.com/2017/automating-web-performance-regression-alerts/



```
➜  ~ k6 run --duration 5s --vus 10 sample.js

          /\      |‾‾|  /‾‾/  /‾/
     /\  /  \     |  |_/  /  / /
    /  \/    \    |      |  /  ‾‾\
   /          \   |  |‾\  \ | (_) |
  / __________ \  |__|  \__\ \___/  Welcome to k6 v0.10.0!

  execution: local
     output: -
     script: sample.js (js)

   duration: 5s, iterations: 0
        vus: 10, max: 10

    web ui: http://127.0.0.1:6565/

      done [==========================================================]         5s / 5s

    ✓ 100.00% - status is 200

    checks................: 100.00%
    http_req_blocked......: avg=3.88ms, max=149.42ms, med=2.23µs, min=1.27µs, p90=6.31µs, p95=58.62µs
    http_req_connecting...: avg=2.96ms, max=114.72ms, med=0s, min=0s, p90=0s, p95=0s
    http_req_duration.....: avg=130.6ms, max=265.27ms, med=122.13ms, min=109.63ms, p90=147.45ms, p95=209.33ms
    http_req_looking_up...: avg=897.6µs, max=34.65ms, med=0s, min=0s, p90=0s, p95=0s
    http_req_receiving....: avg=87.07µs, max=2.29ms, med=44.64µs, min=19.6µs, p90=95.79µs, p95=118.2µs
    http_req_sending......: avg=26.41µs, max=1.02ms, med=16.89µs, min=9.19µs, p90=36.51µs, p95=57.41µs
    http_req_waiting......: avg=126.61ms, max=244.15ms, med=121.78ms, min=109.54ms, p90=144ms, p95=163.51ms
    http_reqs.............: 379
    iterations............: 379
    vus...................: 10
    vus_max...............: 10
➜  ~
```

k6[331] allows you to write unit tests-alike performance tests.

Look into using RUM-solutions to monitor changes in performance over time. For automated unit-test-alike load testing tools, you can use k6[332] with its scripting API. Also, look into SpeedTracker[333], Lighthouse[334] and Calibre[335].

331. https://github.com/loadimpact/k6
332. https://github.com/loadimpact/k6
333. https://speedtracker.org
334. https://github.com/GoogleChrome/lighthouse
335. https://calibreapp.com

Quick Wins

This list is quite comprehensive, and completing all of the optimizations might take quite a while. So, if you had just 1 hour to get significant improvements, what would you do? Let's boil it all down to 10 low-hanging fruits. Obviously, before you start and once you finish, measure results, including start rendering time and SpeedIndex on a 3G and cable connection.

1. Measure the real world experience and set appropriate goals. A good goal to aim for is First Meaningful Paint < 1 s, a SpeedIndex value < 1250, Time to Interactive < 5s on slow 3G, for repeat visits, TTI < 2s. Optimize for start rendering time and time-to-interactive.
2. Prepare critical CSS for your main templates, and include it in the `<head>` of the page. (Your budget is 14 KB). For CSS/JS, operate within a critical file size budget of max. 170Kb gzipped[336] (0.8–1MB decompressed).
3. Defer and lazy-load as many scripts as possible, both your own and third-party scripts — especially social media buttons, video players and expensive JavaScript.
4. Add resource hints to speed up delivery with faster `dns-lookup`, `preconnect`, `prefetch` and `preload`.
5. Subset web fonts and load them asynchronously (or just switch to system fonts instead).

336. https://infrequently.org/2017/10/can-you-afford-it-real-world-web-performance-budgets/

6. Optimize images, and consider using WebP for critical pages (such as landing pages).
7. Check that HTTP cache headers and security headers are set properly.
8. Enable Brotli or Zopfli compression on the server. (If that's not possible, don't forget to enable Gzip compression.)
9. If HTTP/2 is available, enable HPACK compression and start monitoring mixed-content warnings. If you're running over LTS, also enable OCSP stapling.
10. Cache assets such as fonts, styles, JavaScript and images — actually, as much as possible! — in a service worker cache.

Off We Go!

Some of the optimizations might be beyond the scope of your work or budget or might just be overkill given the legacy code you have to deal with. That's fine! Use this checklist as a general (and hopefully comprehensive) guide, and create your own list of issues that apply to your context. But most importantly, test and measure your own projects to identify issues before optimizing. Happy performance results in 2018, everyone! ❧

A huge thanks to Guy Podjarny, Yoav Weiss, Addy Osmani, Artem Denysov, Denys Mishunov, Ilya Pukhalski, Jeremy Wagner, Colin Bendell, Mark Zeman, Patrick Meenan, Leonardo Losoviz, Andy Davies, Rachel Andrew, Anselm Hannemann, Patrick Hamann, Andy Davies, Tim Kadlec, Rey Bango, Matthias Ott, Mariana Peralta, Philipp Tellis, Ryan Townsend, Mohamed Hussain S H, Jacob Groß, Tim Swalling, Bob Visser, Kev Adamson, Aleksey Kulikov and Rodney Rehm for reviewing this article, as well as our fantastic community, which has shared techniques and lessons learned from its work in performance optimization for everybody to use. You are truly smashing!

About The Author

Vitaly Friedman (@smashingmag[337]) loves beautiful content and doesn't like to give in easily. When he is not writing or speaking at a conference, he's most probably running front-end/UX workshops and webinars[338]. He loves solving complex UX, front-end and performance problems. Get in touch[339].

337. https://twitter.com/smashingmag
338. https://www.smashingmagazine.com/smashing-workshops/
339. https://www.smashingmagazine.com/events/#in-house-workshop