Front-End Performance Checklist 2018

*Below you'll find an overview of the **front-end performance issues** you might need to consider to ensure that your response times are fast and smooth.*

—

Get Ready: Planning and Metrics

☐ **Establish a performance culture.**
As long as there is no alignment between 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 problems. Build up a company-tailored case study with real data. Plan out a loading sequence and trade-offs during the design process.

☐ **Choose the right metrics.**
Not every metrics is equally important. Study what metrics matter most: usually it will be related to how fast you can start render *most important* pixels and how quickly you can provide input responsiveness. Prioritize page loading as perceived by your customers. Time to Interactive, Hero Rendering Times, First Meaningful Paint, Speed Index usually matter.

☐ **Be 20% faster than your fastest competitor.**
Gather data on a device representative of your audience. Prefer real devices to simulations. Choose a Moto G4, a mid-range Samsung device and a good middle-of-the-road device like a Nexus 5X. Alternatively, 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. Collect data, set up a spreadsheet, shave off 20%, and set up your goals (performance budgets).

☐ **Share the checklist with your colleagues.**
Make sure that the checklist is familiar to every member of your team. Every decision has performance implications, and your project would hugely benefit from front-end developers being actively involved. Map design decisions against the performance budget.

Curated by Vitaly Friedman. Permanent URL: *www.smashed.by/perf-checklist.* January 2, 2018.

Setting Realistic Goals

☐ **100-millisecond response time, 60 frames per second.**
Each frame of animation should complete in less than 16 milliseconds — ideally 10 milliseconds, thereby achieving 60 frames per second (1 second ÷ 60 = 16.6 milliseconds). Be optimistic and use the idle time wisely. For high pressure points like animation, it's best to do nothing else where you can and the absolute minimum where you can't. Estimated Input Latency should be below 50ms.

☐ **SpeedIndex < 1250, Time-To-Interactive < 5s on 3G.**
The goal is a First Meaningful Paint under 1 sec (on a fast connection) and a SpeedIndex value of under 1250 ms. Considering the baseline being a $200 Android phone on a slow 3G, emulated at 400ms RTT and 400kbps transfer speed, aim for Time to Interactive < 5s, and for repeat visits, under 2s. Put your effort into getting these values as low as possible.

☐ **Critical payload chunk = 15Kb, critical file size budget < 170Kb**
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. To achieve goals stated above, operate within a critical file size budget of max. 170Kb gzipped (0.8-1Mb decompressed) which already would take up to 1s to parse and compile on an average phone.

Defining the Environment

☐ **Choose and set up your build tools.**
Don't pay much attention to what's supposedly cool. As long as you are getting results fast and you have no issues maintaining your build process, you're doing just fine.

☐ **Progressive enhancement.**
Design and build the core experience first, and then enhance the experience with advanced features for capable browsers, creating resilient 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.**
JavaScript has the heaviest cost of the experience. With a 170KB budget that already contains the critical-path HTML/CSS/JavaScript, router, state management, utilities, framework and the app logic, thoroughly examine network transfer cost, the parse/compile time and the runtime cost of the framework of our choice.

☐ **Pick your battles wisely: Angular, React, Ember and co.**
Not every project needs a framework but if you need one, favor a framework that enables server-side rendering. Be sure to measure boot times in server- and client-rendered modes on mobile devices before settling on a framework. Understand the nuts and bolts of the framework you'll be relying on. Look into the PRPL pattern and application shell architecture.

☐ **Will you be using AMP or Instant Articles?**
You can achieve good performance without them, but AMP does provide a solid performance framework, with a free CDN, while Instant Articles will boost your visibility and performance on Facebook. You could build progressive web AMPs, too.

☐ **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, push it to a CDN and serve a static version from it, thus avoiding database requests (JAMStack). Double-check that your CDN performs content compression and conversion, smart HTTP/2 delivery and edge-side includes for you.

Build Optimizations

☐ **Set your priorities right.**
Run an inventory on all of your assets (JavaScript, images, fonts, third-party scripts, "expensive" modules on the page), and break them down in groups. Define the basic core experience (fully accessible core content for legacy browsers), the enhanced experience (an enriched, full experience for capable browsers) and the extras (assets that aren't absolutely required and that can be lazy-loaded, such as fonts, carousel scripts, video players, social media buttons).

☐ **Consider the "cutting-the-mustard" technique.**
Send the core experience to legacy browsers and an enhanced experience to modern browsers. Be strict in the loading of assets: load the core immediately, enhancements on *DomContentLoaded* and extras on the *Load* event. But: cheap Android phones will cut the mustard despite their limited memory and CPU capabilities, so consider feature detect Device Memory JavaScript API and fall back to "cutting the mustard".

☐ **Parsing JavaScript is expensive, so keep it small.**
With SPAs, 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 (it can easily be 2–5x times higher on low-end mobile devices). Thoroughly examine every single JavaScript dependency to see where you are losing initial booting time.

☐ **Consider micro-optimizations and progressive booting.**
Use 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. Then, either on demand or as time allows, boot non-essential parts of the app. Display skeleton screens instead of loading indicators.

☐ **Use tree-shaking and code-splitting to reduce payloads.**
Tree-shaking is a way to clean up your build process by only including code that is actually used in production. Code-splitting splits your code base into "chunks" that are loaded on demand. Scope hoisting detects where *import* chaining can be flattened and converted into one inlined function without compromising the code. Make use of them via WebPack. Use an ahead-of-time compiler to offload some of the client-side rendering to the server.

☐ **Load JavaScript asynchronously.**
As developers, we have to explicitly tell the browser not to wait and to start rendering the page with the *defer* and *async* attributes in HTML. If you don't have to worry much about IE 9 and below, then prefer *defer* to *async;* otherwise, use *async*. Use static social-sharing buttons and static links to interactive maps, instead of relying on third-party libraries.

☐ **Do you constrain the impact of third-party scripts?**
Too often one single third-party script ends up calling a long tail of scripts. Consider using service workers by racing the resource download with a timeout. Establish a Content Security Policy (CSP) to restrict the impact of third-party scripts, e.g. disallowing the download of audio or video. Embed scripts via *iframe,* so scripts don't have access to the DOM. Sandbox them, too. To stress-test scripts, examine bottom-up summaries in Performance profile page (DevTools).

☐ **Are HTTP cache headers set properly?**
Double-check that expires, cache-control, max-age and other HTTP cache headers are 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). Use *cache-control: immutable*, designed for fingerprinted static resources, to avoid revalidation.

Assets Optimizations

☐ **Is Brotli or Zopfli text compression in use?**
Brotli, a new lossless data format, is now supported in all modern browsers. It's more effective than Gzip and Deflate, compresses very slowly, but decompresses fast. Pre-compress static assets with Brotli+Gzip at the highest level, compress (dynamic) HTML on the fly with Brotli at level 1–4. Check for Brotli support on CDNs, too. Alternatively, you can look into using

Zopfli on resources that don't change much — it encodes data to Deflate, Gzip and Zlib formats and is designed to be compressed once and downloaded many times.

☐ **Are images properly optimized?**
Optimize images. As far as possible, use responsive images with *srcset*, *sizes* and the <*picture*> element. Make use of the WebP format, by serving WebP images with <*picture*> and a JPEG fallback or by using content negotiation (using *Accept* headers). For critical images, use progressive JPEGs and blur out unnecessary parts (by applying a Gaussian blur filter).

☐ **Are web fonts optimized?**
Chances are high that the web fonts you are serving include glyphs and extra features that aren't really being used. Subset the fonts. Prefer WOFF2 and use WOFF as fallback. Display content in the fallback fonts right away, load fonts async (e.g. *loadCSS*), then switch the fonts, in that order. Consider locally installed OS fonts as well. Don't forget to include *font-display: optional* and if you can't serve fonts from your server, make sure to use *Font Load Events*.

Delivery Optimizations

☐ **Push critical CSS quickly.**
Collect all of the CSS required to start rendering the first visible portion of the page ("critical CSS" or "above-the-fold" CSS), and add it inline in the <*head*> of the page. Consider the conditional inlining approach. Alternatively, use HTTP/2 server push, but then you might need to create a cache-aware HTTP/2 server-push mechanism.

☐ **Using babel-preset-env to only transpile ES2015+ features.**
With ES2015 being well supported, consider using *babel-preset-env* to only transpile ES2015+ features unsupported by the modern browsers you are targeting. Then set up two builds, one in ES6 and one in ES5. Use *script type="module"* to let browsers with ES module support load the file, while older browser could load legacy builds with *script nomodule*.

☐ **Improve rendering performance.**
Isolate expensive components with CSS containment. 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 making the frames per second consistent is at least preferable to a mixed range of 60 to 15. Use CSS *will-change* to inform the browser about which elements will change.

☐ **Lazy-load expensive scripts with Intersection Observer.**
Intersection Observer API 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. Browser support? Chrome, Firefox, Edge and Samsung Internet are on board. WebKit status is currently in development. Fallback? Lazy load a polyfill.

☐ **Have you optimized rendering experience?**
Don't underestimate the role of perceived performance. While loading assets, 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, use skeleton screens instead of loading indicators and add transitions/animations, for example.

☐ **Warm up the connection to speed up delivery.**
Use skeleton screens, and lazy-load all expensive components, such as fonts, JavaScript, carousels, videos and iframes. Use resource hints to save time on *dns-prefetch*, *preconnect*, *prefetch* and *preload*.

HTTP/2

☐ **Get ready for HTTP/2.**
HTTP/2 is supported very well and offers a performance boost. It isn't going anywhere, and in most cases, you're better off with the latter. The downsides are that you'll have to migrate to HTTPS, and depending on how large your HTTP/1.1 user base is (users on legacy OS' or with legacy browsers), you might need to send different builds, which would require you to adapt a different build process.

☐ **Properly deploy HTTP/2.**
You need to find a fine balance between packaging modules and loading many small modules in parallel. Break down your entire interface into many small modules; then group, compress and bundle them. 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.

☐ **Are you saving data with Save-Data?**
The *Save-Data* client hint request header allows us to customize the application and the payload to cost- and performance-constrained users. E.g, you could rewrite requests for high DPI images to low DPI images, remove web fonts and fancy parallax effects, turn off video autoplay, server pushes or even change how you deliver markup.

☐ **Make sure the security on your server is bulletproof.**
Double-check that your security headers are set properly, eliminate known vulnerabilities,

and check your certificate. 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 and Content Security Policy headers are properly set.

☐ **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?* to check your options, or quickly look up how your servers are performing and which features you can expect to be supported.

☐ **Is OCSP stapling enabled?**
By enabling OCSP stapling on your server, you can speed up TLS handshakes. 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.

☐ **Have you adopted IPv6 yet?**
Studies show that IPv6 makes websites 10 to 15% faster due to neighbor discovery (NDP) and route optimization. Update the DNS for IPv6 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.

☐ **Is HPACK compression in use?**
If you're using HTTP/2, double-check that your servers implement HPACK compression 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* is a great (if very technically detailed) tool to check that.

☐ **Are service workers being used for caching and network fallbacks?**
No performance optimization over a network can be faster than a locally stored cache on the user's machine. If your website is running over HTTPS, then 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.

Testing and Monitoring

☐ **Monitor mixed-content warnings.**
If you've recently migrated from HTTP to HTTPS, make sure to monitor both active and passive mixed-content warnings with tools such as Report-URI.io. You can also use Mixed Content Scan to scan your HTTPS-enabled website for mixed content.

- [ ] **Is your development workflow in DevTools optimized?**
  Pick a debugging tool and click on every single button. Make sure you understand how to analyze rendering performance and console output, and debug JavaScript and edit CSS styles.

- [ ] **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 (including UC Browser and Opera Mini). Measure average Internet speed in your countries of interest to avoid big surprises. Test with network throttling, and emulate a high-DPI device. BrowserStack is fantastic, but test on real devices as well.

- [ ] **Is continuous monitoring set up?**
  A good performance metrics is a combination of passive and active monitoring tools. Having a private instance of WebPagetest and using Lighthouse is always beneficial for quick tests, but also set up continuous monitoring with RUM tools such as Calibre, SpeedCurve and others. Set your own user-timing marks to measure an monitor business-specific metrics.

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 3G and cable connections.

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 (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).
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. If possible, cache assets such as fonts, styles, JavaScript and images — actually, as much as possible! — in a service worker cache.

Huge thanks to Yoav Weiss, Addy Osmani, Artem Denysov, Denys Mishunov, Ilya Pukhalski, Jeremy Wagner, Colin Bendell, Mark Zeman, Patrick Meenan, Leonardo Losoviz, Guy Podjarny, 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 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!