Front-End Performance Checklist 2017

*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 and set goals

☐ **Be 20% faster than your fastest competitor.**
Measure "start rendering" (WebPageTest) and "first meaningful paint" times (Lighthouse) on a Moto G, a mid-range Samsung device and a good middle-of-the-road device like the Nexus 4, preferably in an open device lab — on regular 3G, 4G and Wi-Fi connections. 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.

☐ **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.

☐ **First meaningful paint under 1.25 seconds, SpeedIndex under 1000.**
The goal is a start rendering time under 1 second and a SpeedIndex value of under 1000 milliseconds (on a fast connection). For the first meaningful paint, count on 1250 milliseconds at most. For mobile, a start rendering time under 3 seconds for 3G on a mobile device is acceptable. Put your effort into getting these values as low as possible.

Curated by Vitaly Friedman. Permanent URL: *www.smashed.by/perf-checklist.* December 21, 2016.

Define 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.

☐ **Pick your battles wisely: Angular, React, Ember and co.**
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. When building web apps, look into the PRPL pattern and application shell architecture.

☐ **Google's AMP or Facebook's 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 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).

☐ **Use 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.

☐ **Consider micro-optimizations and progressive booting.**
You might need some time to initialize the app before you can render the page. Your goal: 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, code-splitting and an ahead-of-time compiler to offload some of the client-side rendering to the server.

☐ **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.

☐ **Limit third-party libraries, and 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.

☐ **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 and OTF as fallbacks. Display content in the fallback fonts right away, load fonts asynchronously (e.g. *loadCSS*), then switch the fonts, in that order. FOUT is better than FOIT. Consider locally installed OS fonts as well.

☐ **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.

☐ **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. Make use of both via WebPack. Also, use Rollup as a JavaScript module bundler.

☐ **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.

☐ **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*, *pretender* 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'll have 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 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.

☐ **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 Brotli or Zopfli compression in use?**
  Brotli, a new lossless data format, is widely supported in Chrome, Firefox and Opera. It's more effective than Gzip and Deflate (HTTPS only). The catch: Brotli doesn't come preinstalled on most servers today, and it's not easy to set up without self-compiling NGINX or Ubuntu. 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.

- ☐ **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.

Test and monitor

☐ **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?**
Having a private instance of WebPagetest is always beneficial for quick and unlimited tests. Set up continuous monitoring of performance budgets with automatic alerts. Set your own user-timing marks to measure and monitor business-specific metrics. Look into SpeedTracker, Lighthouse and Calibre.

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. Your goal is a start rendering time under 1 second on cable and 3 seconds on 3G, and a SpeedIndex value under 1000. 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.)
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*, *preload* and

prerender.

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 Anselm Hannemann, Patrick Hamann, Addy Osmani, Andy Davies, Tim Kadlec, Yoav Weiss, Rey Bango, Mariana Peralta, Jacob Groß, Tim Swalling, Bob Visser, Kev Adamson and Rodney Rehm for reviewing, as well as our fantastic community for sharing insights for everybody to use. You are truly smashing.