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 website’s response times are fast and smooth. Curated by Vitaly Friedman. Permanent URL: www.smashed.by/perf-checklist.

Get ready and set the goals

☐ Be 20% faster than your fastest competitor.
   Measure “Start Render” (WebPageTest) and “First meaningful paint” times (Lighthouse) on Moto G, a mid-range Samsung device and a good middle-of-the-road device like Nexus 4, preferably in an open device lab — on 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 the project will hugely benefit from front-end developers being actively involved. Map design decisions against performance budget.

☐ 100ms response time, 60fps.
   Each frame of animation should be completed in less than 16 milliseconds, thereby achieving 60 FPS (1 second ÷ 60 = 16.6 milliseconds). Be optimistic and use the idle time wisely.

☐ First Meaningful Paint <1.25s, SpeedIndex <1000.
   The goal is the Start Render time under 1s, the SpeedIndex value of under 1000 (on a fast connection), and for First meaningful paint, count on 1250ms at most. For mobile, the Start Render under 3s for 3G on a mobile device is acceptable. Put your efforts into getting these values as small as possible.

Defining the environment

☐ Choose and set up your build tools.
   Don’t pay too much attention to what’s supposed to be cool. As long as you are getting to 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 site runs fast on a slow machine with a poor screen in a poor browser on a suboptimal network, it can only run faster on a fast machine, with a good browser on a decent network.

Pick your battle wisely: Angular, React, Ember & 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 in on a framework. Understand all the nuts and bolts of the framework that you’ll be relying on. When building web apps, look into the PRPL Pattern and application shell architecture.

Google AMP or Facebook’s Instant Pages?
You can achieve a good performance without them, but AMP does provide a solid performance framework with a free CDN, while Instant Pages 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, avoiding database requests (JAM stack). Double check if 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 (for legacy browsers; fully accessible core content), Enhanced experience (for capable browsers; enriched, full experience) and Extras (assets which aren’t absolutely required and can be lazy-loaded, e.g. fonts, carousel scripts, video player, social media buttons).

Use “cutting-the-mustard”-technique.
Send core experience to legacy browsers and enhanced experience to modern browsers. Be strict in the loading of your assets: load Core immediately, Enhancement on `DOMContentLoaded` and Extras on `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 (FMP), but also include some
minimal JavaScript to keep Time to Interactive (TTI) close to 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 if 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 infinitely (if they are static). Use Cache-control: immutable designed for fingerprinted static resources to avoid revalidation.

☐ **Limit 3rd-party-libraries and load JavaScript asynchronously.**
As developers, we have to explicitly tell the browser not to wait and start render the page, with defer and async attributes in HTML. If you don't have to worry too much about IE<=9, prefer defer over async; otherwise use async. Use static social links buttons and static links to interactive maps instead of relying on 3rd-party libraries.

☐ **Are images properly optimized?**
Optimize images. As far as possible, use responsive images with srcset, sizes and <picture> element. Make use of 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 blury 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 used. Subset the fonts. Prefer WOFF2 and WOFF/OTF as fallback. Display the content in the fallback font 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 render 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 using conditional inlining approach. Alternatively, user 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 payload.**
Tree-shaking is a way to clean up your build process by including only code that is actually used in production. Code splitting splits your code base into “chunks” which are loaded on demand. Make use of both via WebPack. Also, use Rollup.js 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 you’re consistently hitting 60fps. If it’s not possible, having a consistent FPS instead of a mixed FPS ranging between 60 and 15fps is preferred. Use CSS will-change to inform the browser which elements will change.

**Warm-up the connection to speed up delivery.**
Use skeleton screens and lazy-load all expensive components, e.g. 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'll be better off with the latter. Downside: you have to migrate to HTTPS, and depending on how large your HTTP/1.1 user base is (users on legacy OSes or with legacy browsers), send different builds, which would require you to adapt a different build process.

**Set up a proper HTTP/2 deployment.**
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 fine balance for your site.

**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 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 what features support you should be expecting.

**Is Brotli/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 with 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 into 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 your 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 already adopted IPv6?**
Studies show that IPv6 makes sites 10-15 percent faster due to neighbor discovery (NDP) and route optimization. Update the DNS for IPv6 to stay bulletproof the 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 if your servers implement HPACK compression for HTTP response headers to reduce unnecessary overhead. Since 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 help check it.

☐ **Is service worker used for caching and network fallback?**
No performance optimization over network can be faster than a locally stored cache on user’s machine. If your website is running over HTTPS, cache static assets in a Service Worker cache and store offline fallbacks (or even offline pages) and retrieve them from 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?
It's not enough to test in Chrome and Firefox. Ensure to look into how your website works in proxy browsers and legacy browsers, e.g. UC Browser and Opera Mini. Measure average Internet speed in the 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 App.

Quick Wins

The list is quite comprehensive, and completing all of the optimizations might take quite a while. So if you had just 1h to get significant improvement, 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 Render time and SpeedIndex value on 3G and on cable connection.

1. Your goal is Start Render under 1s (cable) and 3s (on 3G), and SpeedIndex under 1000,
2. Prepare critical CSS for 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 JavaScripts,
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 a system font instead),
6. Optimize images and consider using WebP for critical pages, e.g. landing pages,
7. Check if HTTP Cache headers and security headers are set properly,
8. Enable Brotli/Zopfli compression on the server (if 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, 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 sharing insights publicly for everybody to use. You are truly smashing.