Front-End Performance Checklist 2019

*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 business buy-in, 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 and business metrics. Plan out a loading sequence and trade-offs during the design process.

☐ **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, a good middle-of-the-road device like a Nexus 5X and a slow device like Alcatel 1X or Nexus 2. 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). Then 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*).

☐ **Choose the right metrics.**
Not every metric 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, First Input Delay, Hero Rendering Times, First Meaningful Paint, Speed Index usually matter.

☐ **Set up "clean" and "customer" profiles for testing.**
Turn off anti-virus and background CPU tasks, remove background bandwidth transfers and test with a clean user profile without browser extensions to avoid skewed results. Study which extensions your customers use, and test with a dedicated "customer" profile as well.

Curated by Vitaly Friedman. Permanent URL: *www.smashed.by/perf-checklist.* January 7, 2019.

☐ **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.

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. Use idle time wisely, with the *Idle Until Urgent* approach.

☐ **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 2–3s. Put your effort into getting these values as low as possible.

☐ **Critical payload chunk = 14KB, critical file size budget < 170KB**
The first 14KB 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 (0.7-0.8MB decompressed) which already would take up to 1s to parse and compile at 400ms RTT on an average phone.
Make sure your budgets change based on network conditions and hardware limitations.

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. The only exception might be Webpack which provides many useful optimization techniques such as code-splitting. If it's not in use yet, make sure to look in detail into code-splitting and tree-shaking.

☐ **Use progressive enhancement as a default.**
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.

- ☐ **Evaluate each framework and each dependency.**
  Not every project needs a framework, not every page of a SPA needs to load the framework. Be deliberate in your choices. Evaluate 3rd-party JS by exploring features, accessibility, stability, performance, package ecosystem, community, learning curve, documentation, tooling, track record, team, compatibility and security. Gatsby.js (React), Preact CLI, and PWA Starter Kit provide reasonable defaults for fast loading out of the box on average mobile hardware.

- ☐ **Pick your battles wisely: React, Vue, Angular, 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. Look into the PRPL pattern and application shell architecture.

- ☐ **Optimize the performance of your APIs.**
  If many resources require data from an API, the API might become a performance bottleneck. Consider using GraphQL, a query language and a server-side runtime for executing queries by using a type system you define for your data. Unlike REST, GraphQL can retrieve all data in a single request, without over or under-fetching data as it typically happens with REST.

- ☐ **Will you be using AMP or Instant Articles?**
  You can achieve good performance without them, but AMP *might* 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, (e.g. image optimization in terms of formats, compression and resizing at the edge), support for servers workers 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).

☐ **Revisit the good ol' "cutting-the-mustard" technique.**
Send the core experience to legacy browsers and an enhanced experience to modern browsers. Use ES2015+ *<script type="module">* for loading JavaScript: modern browsers will interpret the script as a JavaScript module and run it as expected, while legacy browsers wouldn't recognize it and hence ignore it. 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).

☐ **Use tree-shaking, scope hoisting 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.

☐ **Can you offload JavaScript into a Web Worker or WebAssembly?**
As the code base evolves, UI performance bottlenecks will start showing up. It happens because DOM operations are running alongside your JS on the main thread. Consider moving these expensive operations to a background process that’s running on a different thread with web workers. Typical use case: prefetching data and PWAs. Consider compiling into WebAssembly which works best for computationally intensive web apps, such as web games.

☐ **Serve legacy code only to legacy browsers (differential serving).**
Use *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. Old browsers could load legacy builds with *script nomodule.* For lodash, use *babel-plugin-lodash* that will load only

modules that you are using in your source. Transform generic lodash requires to cherry-picked ones to avoid code duplication.

- [ ] **Identify and rewrite legacy code with incremental decoupling.**
  Revisit your dependencies and assess how much time would be required to refactor or rewrite legacy code that has been causing trouble lately. First, set up metrics that tracks if the ratio of legacy code calls is staying constant or going down, not up. Publicly discourage the team from using the library and make sure that your CI alerts developers if it's used in pull requests.

- [ ] **Identify and remove unused CSS/JavaScript.**
  CSS and JavaScript code coverage in Chrome allows you to learn which code has been executed/applied and which hasn't. Once you've detected unused code, find those modules and lazy load with *import()*. Then repeat the coverage profile and validate that it's now shipping less code on initial load. Use Puppeteer to programmatically collect code coverage.

- [ ] **Trim the size of your JavaScript dependencies.**
  There's a high chance you're shipping full JavaScript libraries when you only need a fraction. To avoid the overhead, consider using *webpack-libs-optimizations* that removes unused methods and polyfills during the build process. Add bundle auditing into your regular workflow.

- [ ] **Are you using predictive prefetching for JavaScript chunks?**
  Use heuristics to decide when to preload JavaScript chunks. *Guess.js* is a set of tools that use Google Analytics data to determine which page a user is mostly likely to visit next. Note: you might be prompting the browser to consume unneeded data and prefetch undesirable pages, so it's a good idea to be quite conservative in the number of prefetched requests.

- [ ] **Consider micro-optimizations and progressive booting.**
  Use server-side rendering to get a quick first meaningful paint, but also include some minimal JS 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. Always break up the execution of functions into separate, asynchronous tasks. Where possible use *requestIdleCallback*.

- [ ] **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).

☐ **Set HTTP cache headers 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. Check that you aren't sending unnecessary headers (such as *x-powered-by*, *pragma*, *x-ua-compatible*, *expires*).

Assets Optimizations

☐ **Use Brotli or Zopfli for plain text compression.**
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.

☐ **Use responsive images and WebP.**
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). Note: with WebP, you'll reduce the payload, but with JPEG you'll improve perceived performance, so users might see an actual image faster with a good ol' JPEG although WebP images might get faster through the network.

☐ **Are images properly optimized?**
Use *mozJPEG* for JPEG compression, *SVGO* for SVG compression, *Pingo* for PNGs — or *Squoosh* for all of them. To check the efficiency of your responsive markup, you can use *imaging-heap*. For critical images, use progressive JPEGs and blur out unnecessary parts (by applying a Gaussian blur filter) and remove contrast (you can reapply it with CSS filters).

☐ **Are videos properly optimized?**
Instead of animated GIFs, use either animated WebP (with GIF being a fallback) or looping inlined HTML5 videos. Make sure that your MP4s are processed with a multipass-encoding, blurred with the *freior iirblur effect* (if applicable) and *moov atom* metadata is moved to the head of the file, while your server accepts byte serving. Be prepared for the AV1 format which has good chances for becoming the ultimate standard for video on the web.

☐ **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. Ultimate solution: two-stage render, with a small *supersubset* first, and the rest of the family loaded async later. Preload 1–2 fonts of each family. Consider locally installed OS fonts. Don't forget to include *font-display: optional* and use *Font Load Events* for group repaints.

Delivery Optimizations

☐ **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*. With *defer*, browser doesn't execute scripts until HTML is parsed. So unless you need JS to execute before start render, it's better to use *defer*.

☐ **Lazy load expensive components with IntersectionObserver.**
lazy-load all expensive components, such as heavy JavaScript, videos, iframes, widgets, and potentially images. The most performant way to do so is by using the Intersection Observer. Also, watch out for the *lazyload* attribute that will allow us to specify which images and iframes should be lazy loaded, natively.

☐ **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.

☐ **Experiment with regrouping your CSS rules.**
Consider splitting the main CSS file out into its individual media queries. Avoid placing <*link rel="stylesheet"* /> before *async* snippets. If scripts don't depend on stylesheets, consider placing blocking scripts above blocking styles. If they do, split that JavaScript in two and load it either side of your CSS. Cache inlined CSS with a service worker and experiment with *in-body CSS*.

☐ **Stream responses.**
Streams 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. Instead of serving an empty UI shell and letting JavaScript populate it, let the service worker construct a stream where the shell comes from a cache, but the body comes from the network. HTML rendered during the initial nav request can then take full advantage of the browser's streaming HTML parser.

☐ **Consider making your components connection-/device memory-aware.**
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. Use *Network Information API* to deliver variants of heavy components based on the network type. Dynamically adjust resources based on available device memory, too, with Device Memory API.

☐ **Warm up the connection to speed up delivery.**
Use resource hints to save time on *dns-prefetch* (DNS lookup in the background), *preconnect* (start the connection handshake (DNS, TCP, TLS)), *prefetch* (request a resource) and *preload* (prefetch resources without executing them, among other things). When using *preload*, *as* must be defined or nothing loads; preloaded fonts without *crossorigin* attribute will double fetch.

☐ **Use service workers for caching and network fallbacks.**
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 the user's machine, rather than going to the network. Store the app shell in the service worker's cache along with a few critical pages, such as offline page or frontpage. But: make sure the proper CORS response header exists for cross-origin resources, don't cache opaque responses and opt-in cross-origin image assets into CORS mode.

☐ **Use service workers on the CDN/Edge (e.g. for A/B testing).**
With CDNs implementing service workers on the server, considerm them to tweak performance on the edge as well. E.g. in A/B tests, when HTML needs to vary its content for different users, use service workers on the CDNs to handle the logic. Or stream HTML rewriting to speed up sites that use Google Fonts.

☐ **Optimize 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.

☐ **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 and animations.

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 it. Depending on how large your mobile user base is, you might need to send different builds, which would require you to adapt a different build process. (HTTP/2 is often slower on networks which have a noticeable packet loss rate.)

☐ **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.

☐ **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 which features you can expect to be supported. Enable BBR congestion control, set *tcp_notsent_lowat* to 16KB for HTTP/2 prioritization.

☐ **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.

☐ **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.

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.

☐ **Have you optimized your auditing and debugging workflow?**
Invest time to study debugging and auditing techniques in your debugger, WebPageTest, Lighthouse and supercharge your text editor. E,g, you could drive WebPageTest from a Google Spreadsheet and incorporate accessibility, performance and SEO scores into your Travis setup with Lighthouse CI or straight into Webpack.

☐ **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 among your user base to avoid big surprises. Test with network throttling, and emulate a high-DPI device. BrowserStack is fantastic, but test on real devices, too.

☐ **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 *SpeedTracker, Calibre*, *SpeedCurve* and others. Set your own user-timing marks to measure and 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 12 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 (0.7MB decompressed).
3. Trim, optimize, defer and lazy-load as many scripts as possible, check lightweight alternatives and limit the impact of third-party scripts.
4. Serve legacy code only to legacy browsers with *<script type="module">*.
5. Experiment with regrouping your CSS rules and test in-body CSS.
6. Add resource hints to speed up delivery with faster *dns-lookup*, *preconnect*, *prefetch* and *preload*.
7. Subset web fonts and load them asynchronously, and utilize font-display in CSS for fast first rendering.
8. Optimize images, and consider using WebP for critical pages (such as landing pages).
9. Check that HTTP cache headers and security headers are set properly.
10. Enable Brotli or Zopfli compression on the server. (If that's not possible, don't forget to enable Gzip compression.)
11. If HTTP/2 is available, enable HPACK compression, and start monitoring mixed-content warnings. Enable OCSP stapling.
12. If possible, cache assets such as fonts, styles, JavaScript and images in a service worker cache.

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, Peter Bowyer, Phil Walton, Mariana Peralta, Philipp Tellis, Ryan Townsend, Ingrid Bergman, Mohamed Hussain S H, Jacob Groß, Tim Swalling, Bob Visser, Kev Adamson, Adir Amsalem, Aleksey Kulikov and Rodney Rehm for reviewing the 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!