Front-End Performance Checklist 2021

*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/G5 Plus, a mid-range Samsung device (Galaxy A50, S8), a good middle-of-the-road device like a Nexus 5X, Xiaomi Mi A3 or Xiaomi Redmi Note 7 and a slow device like Alcatel 1X or Cubot X19. Alternatively, emulate mobile experience on desktop by testing on a throttled network (e.g. 300ms RTT, 1.6 Mbps down, 0.8 Mbps up) with a throttled CPU (5× slowdown). Then switch over to regular 3G, slow 4G (e.g. 170ms RTT, 9 Mbps down, 9 Mbps up), 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, Largest Contentful Paint, Total Blocking Time and Cumulative Layout Shift usually matter. *Not:* First Meaningful Paint.

☐ **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.

- ☐ **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 distributing ownership across the entire team. 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.

- ☐ **LCP < 2.5s, FID < 100ms, CLS < 0.1, Time To Interactive < 5s on 3G.**
  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. Aim for *Largest Contentful Paint* < 2.5s and minimize *Total Blocking Time* and *Cumulative Layout Shift*. Put your effort into getting these values as low as possible.

- ☐ **Critical file size budget < 170KB**
  The first 14–16KB 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. 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 or Parcel which provide 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 third-party JS by exploring features, accessibility, stability, performance, package ecosystem, learning curve, documentation, tooling, track record, team, compatibility and security. Next.js (React), Gatsby (React), Vuepress (Vue) and Preact CLI provide reasonable defaults for fast loading out of the box on average mobile hardware.

- ☐ **Pick your battles wisely: React, Vue, Angular, Ember and Co.**
  Make sure that the framework of your choice provides server-side rendering or prerendering. Measure boot times in server- and client-rendered modes on mobile devices before deciding. Understand the nuts and bolts of the framework you'll be relying on. Look into PRPL pattern and app shell architecture. Notable options are Preact, Inferno, Vue, Svelte, Alpine, Polymer.

- ☐ **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.

Assets Optimizations

☐ **Use Brotli 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. Make sure that the server handles content negotiation for Brotli or gzip properly.

☐ **Use responsive images, AVIF and WebP.**
As far as possible, use responsive images with *srcset*, *sizes*, *<picture>* and *image-set*. AVIF > WebP. Serve AVIF and WebP formats with *<picture>* and a JPEG/PNG 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 (incl. AVIF). 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). Make sure *width* and *height* are defined for all images, add automatic image compression to your pull requests and look into lazy loading less critical images.

☐ **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. Whenever possible, serve the video in AV1 format, but provide a fallback as well. Resize videos and use adaptive media serving to provide the content that your customers can actually play without stalls on their devices.

☐ **Are web fonts optimized?**
Subset the fonts. Prefer WOFF2 and use WOFF as fallback. Display content in the fallback fonts right away, load fonts async, 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. Avoid the *local()* value for font declarations, but consider locally installed OS fonts. Don't forget to include *font-display: optional* and use *Font Load Events* for group repaints. Use *Web Font Reflow Count* and *Time To Real Italics* metrics.

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).

☐ **Use native JavaScript modules in production.**
Send the core experience to legacy browsers and an enhanced experience to modern browsers. Use ES2017+ *<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.

☐ **Executing JavaScript is expensive, so tame it.**
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, e.g. with progressive hydration and import on interaction (times are 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 (e.g. via Webpack). Use granular chunking and offload some of the client-side rendering to the server. Define split points by tracking which CSS/JS chunks are used, and which aren't. Consider code-splitting at the package level as well.

☐ **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 offloading computationally heavy tasks off to WebAssembly, which works best for computationally intensive web apps, such as games.

☐ **Serve legacy code only to legacy browsers (differential serving).**
Use *babel-preset-env* to only transpile ES2017+ features unsupported by the modern browsers you are targeting. Then set up two builds, one for modern, and one for legacy browsers. 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. Use the header *<link rel="modulepreload">* to initiate early (and high-priority) loading of *module* scripts.

☐ **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. *Bundlephobia* helps find the cost of adding an npm package to your bundle, *size-limit* extends basic bundle size check with details on JavaScript execution time. Use *Skypack* to discover community-curated packages with focus on quality and performance.

☐ **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. Also, consider *Quicklink*, *Instant.page* and *DNStradamus*. 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.

☐ **Optimize for your target JavaScript engines.**
Make use of *script streaming* for monolithic scripts, so they can be parsed on a separate background thread once downloading begins. Hook into V8's *code caching* as well, by splitting out libraries from code using them, or the other way around. Consider JIT Optimization Strategies for Firefox's Baseline Interpeter as well.

☐ **Find a way to marry client-side rendering and server-side rendering.**
Usually the goal is to find the optimum balance between client-side and server-side rendering. Consider prerendering if your pages don't change much, and defer the booting of frameworks

if you can. Stream HTML in chunks with server-side rendering, and implement progressive hydration for individual components with client-side rendering — and hydrate on visibility, interaction or during idle time to get the best of both worlds. (*Streaming Server-Side Rendering With Progressive Hydration*).

☐ **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*.

☐ **Always self-host third-party assets.**
Using a public CDN will *not* automatically lead to better performance. Even if two sites point to the exact same third party resource URL, the code is downloaded once per domain, and the cache is "sandboxed" to that domain. First-party assets are more likely to stay in the cache than third-party assets. Self-hosting is more reliable, secure, and better for performance.

☐ **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). Load third-party scripts only once the app has initialized. Watch out for anti-flicker snippets.

☐ **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* to avoid revalidation. Check that you aren't sending unnecessary headers (e.g. *x-powered-by*, *pragma*, *x-ua-compatible*, *expires*). Make use of zero RTT for repeat views via *stale-while-revalidate*.

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. Always prefer *defer* to *async*. With *defer*, browser doesn't execute scripts until HTML is parsed and all prior sync scripts have executed. Don't use both. 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 either with native lazy-loading (*loading and importance* attributes) or by using the Intersection Observer — the latter can be used for performant scrollytelling, parallax and ads tracking as well.

☐ **Defer rendering and decoding for expensive images.**
With *content-visibility: auto*, we can prompt the browser to skip the layout of the children while the container is outside of the viewport. Just make sure to use *contain-intrinsic-size* with a placeholder properly sized to avoid CLS. Also, prompt the browser to decode the image off the main thread with *<img decoding="async">*, to reduce CPU-time needed for the operation.

☐ **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 (stay under 14 KB). Consider the conditional inlining approach. Putting critical CSS in a separate file on the root domain has benefits, sometimes more than inlining due to caching.

☐ **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*. Dynamic styling can be expensive: check that your CSS-in-JS library optimizes the execution when your CSS has no dependencies on theme/props, don't over-compose styled components.

☐ **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.**
Customize the application and the payload to cost- and performance-constrained users with the *Save-Data* client hint request header. You could rewrite requests for high DPI images to low DPI images, remove web fonts and parallax, turn off video autoplay, or even change how you deliver markup. Use *Network Information API* to deliver variants of heavy components based on connectivity, and *Device Memory API* to adjust resources based on device memory.

- ☐ **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), *preload* (prefetch resources without executing them, among other things) and *prerender* (fetches resources in advance but doesn't execute JS or render any part of the page in advance). When using *preload*, *as* must be defined or nothing loads; preloaded fonts without *crossorigin* attribute will double fetch. With *preload*, there is a puzzle of priorities, so consider injecting *rel="preload"* elements into the DOM just before the external blocking scripts.

- ☐ **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 homepage. 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, consider service workers 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.**
  If needed, isolate expensive components with granular 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.

- ☐ **Prevent layout shifts (reflows) and repaints.**
  Reflows are caused by rescaled images and videos, web fonts, injected ads or late-discovered scripts that populate components with actual content. Set *width* and *height* attributes on images, so modern browsers allocate the box and reserve the space by default. Use an SVG

placeholder to reserve the display box in which the video and images will appear in. Use hybrid lazy-loading to load an external lazy-loading script only if native lazy-loading isn't supported. Group web font repaints and match line-height and spacing with *font-style-matcher*. Track the stability of the interface with Layout Instability API and *Cumulative Layout Shift (CLS)*.

Networking and HTTP/2

☐ **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 reduced the impact of SSL certificate revocation?**
Extended Validation (EV) certificates are expensive and time-consuming as they require a human to reviewing a certificate and ensuring its validity. *Domain Validation* (DV) certificates, on the other hand, are often provided for free. EV certificates do not fully support OCSP stapling, so always serve an OCSP stapled DV certificate and keep TLS certificates small.

☐ **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 TCP BBR in use?**
BBR is a relatively new TCP delay-controlled TCP flow control algorithm. It responds to actual congestion, rather than packet loss like TCP does, and as such, it's significantly faster, with higher throughput and lower latency. Enable BBR congestion control and set *tcp_notsent_lowat* to 16KB for HTTP/2 prioritization to work reliably on Linux 4.9 kernels and later.

☐ **Always prefer 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, and you could benefit from adapting 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. Split at the package level, or by tracking which chunks of CSS/JS are/aren't used. Separate vendor and client code, and separate vendor dependencies that change rarely and frequently. 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 best to send as many assets as possible over a single HTTP/2 connection.

☐ **Do your servers and CDNs support HTTP/2?**
Different servers and CDNs support HTTP/2 differently. Use *CDN Comparison* 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 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.

☐ **Get ready for HTTP/3.**
While in HTTP/2, multiple requests share a connection, in HTTP/3 requests also share a connection but stream independently, so a dropped packet no longer impacts all requests, just the one stream. With HTTP3's QUIC, TCP and TLS are combined and completed in just a single round trip, and from the second connection onward, we can already send and receive app layer data in the first round trip (*0-RTT*). Packaging still matters, so instead of sending a monolithic JS, send multiple JS-files in parallel. Expect an impact on loading times on mobile.

☐ **Do your servers and CDNs support HTTP/3?**
QUIC and HTTP/3 are better and more bulletproof: with faster handshakes, better encryption, more reliable independent streams, more encrypted, and with 0-RTT if the client previously had a connection with the server. However, it's quite CPU intensive (2-3x CPU usage for the same bandwidth). Check if your servers or CDNs support *HTTP over QUIC* (also known as *HTTP/3*), and if you can enable it.

☐ **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. Use a Perf Diagnostic CSS for quick check-ups.

- ☐ **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.

- ☐ **Have you tested the performance of your 404 pages?**
  Every time a client requests an asset that doesn't exist, they'll receive a 404 response — and often that response is huge. Make sure to examine and optimize the caching strategy for your 404 pages. Make sure to serve HTML to the browser only when it expects an HTML response, and return a small error payload for all other responses.

- ☐ **Have you tested the performance of your GDPR consent prompts?**
  Normally cookie consent prompts shouldn't have an impact on CLS, but sometimes they do, so consider using free and open source options *Osano* or *cookie-consent-box*. The consent is likely to change the impact of scripts on the overall performance, so set up and study a few different web performance profiles for different types of consent.

- ☐ **Have you tested the impact on accessibility?**
  Large pages and DOM manipulations with JavaScript will cause delays in screen reader announcements. Fast Time to Interactive means how much time passes by until the screen reader can announce navigation on a given page and a screen reader user can actually hit keyboard to interact.

- ☐ **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*, *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 **18 low-hanging fruits**. Obviously, before you start and once you finish, measure results, including Largest Contentful Paint and Time To Interactive on a 3G and cable connection.

1. Measure the real world experience and set appropriate goals. Aim to be at least 20% faster than your fastest competitor. Stay within Largest Contentful Paint < 2.5s, a First Input Delay < 100ms, Time to Interactive < 5s on slow 3G, for repeat visits, TTI < 2s. Optimize at least for First Contentful Paint and Time To Interactive.
2. Optimize images with *Squoosh*, *mozjpeg*, *guetzli*, *pingo* and *SVGOMG*, and serve AVIF/WebP with an image CDN.
3. Prepare critical CSS for your main templates, and inline them in the *<head>* of each template. For CSS/JS, operate within a critical file size budget of max. 170KB gzipped (0.7MB unzipped).
4. Trim, optimize, defer and lazy-load scripts. Invest in the config of your bundler to remove redundancies and check lightweight alternatives.
5. Always self-host your static assets. Always prefer to self-host 3rd-party assets. Limit their impact. Use facades, load widgets on interaction and beware of anti-flicker snippets.
6. Be selective when choosing a framework. For single-page-applications, identify critical pages and serve them statically, or at least prerender them, and use progressive hydration on component-level and import modules on interaction.
7. Client-side rendering alone isn't a good choice for performance. Prerender if your pages don't change much, and defer the booting of frameworks. Use streaming server-side rendering.
8. Serve legacy code only to legacy browsers with the *module/nomodule* pattern.
9. Experiment with regrouping your CSS rules and test in-body CSS.
10. Add resource hints to speed up delivery with *dns-lookup*, *preconnect*, *prefetch, preload, prerender*.
11. Subset web fonts and load them asynchronously, and utilize *font-display* in CSS for fast first rendering.
12. Check that HTTP cache headers and security headers are set properly.
13. Enable Brotli compression on the server. (If that's not possible, don't forget to enable Gzip compression.)
14. Enable TCP BBR congestion if your server is running on the Linux kernel version 4.9+.
15. Enable OCSP stapling and IPv6 if possible. Always serve an OCSP stapled DV certificate.
16. Enable HPACK compression for HTTP/2 and move to HTTP/3 if it's available.
17. Cache assets such as fonts, styles, JavaScript and images in a service worker cache.

18. Explore options to avoid rehydration, use progressive hydration and streaming server-side rendering for your single-page application.

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, Barry Pollard, Patrick Hamann, Gideon Pyzer, Andy Davies, Maria Prosvernina, Tim Kadlec, Rey Bango, Matthias Ott, Peter Bowyer, Phil Walton, Mariana Peralta, Pepijn Senders, Mark Nottingham, Jean Pierre Vincent, 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 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!