**[Front-End Performance Checklist 2017 (PDF, Apple Pages)](https://www.smashingmagazine.com/2016/12/front-end-performance-checklist-2017-pdf-pages/" \o "Read 'Front-End Performance Checklist 2017 (PDF, Apple Pages)')**

Are you using progressive booting already? What about **tree-shaking and code-splitting** in React and Angular? Have you set up Brotli or Zopfli compression, OCSP stapling and HPACK compression? Also, how about resource hints, client hints and CSS containment — not to mention IPv6, HTTP/2 and service workers?

Back in the day, performance was often a mere **afterthought**. Often deferred till the very end of the project, it would boil down to minification, concatenation, asset optimization and potentially a few fine adjustments on the server’s config file. Looking back now, things seem to have changed quite significantly.

Performance isn’t just a technical concern: It matters, and when baking it into the workflow, design decisions have to be informed by their performance implications. Performance has to be measured, monitored and refined continually, and the growing complexity of the web poses new challenges that make it hard to keep track of metrics, because metrics will vary significantly depending on the device, browser, protocol, network type and latency (CDNs, ISPs, caches, proxies, firewalls, load balancers and servers all play a role in performance).

So, if we created an overview of all the things we have to keep in mind when improving performance — from the very start of the process until the final release of the website — what would that list look like? Below you’ll find a (hopefully unbiased and objective) **front-end performance checklist for 2017** — an overview of the issues you might need to consider to ensure that your response times are fast and your website smooth.

(You can also just [download the checklist PDF](http://provide.smashingmagazine.com/performance-checklist/performance-checklist-1.0.pdf) (0.129 MB) or [download the checklist in Apple Pages](http://provide.smashingmagazine.com/performance-checklist/performance-checklist-1.0.pages) (0.236 MB). Happy optimizing!)

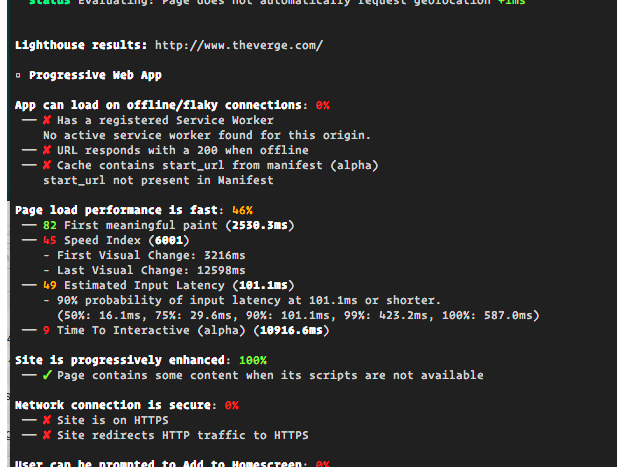
### Front-End Performance Checklist 2017 [Link](https://www.smashingmagazine.com/2016/12/front-end-performance-checklist-2017-pdf-pages/#front-end-performance-checklist-2017)

Micro-optimizations are great for keeping a performance on track, but it’s critical to have clearly defined targets in mind — measurable goals that would influence any decisions made throughout the process. There are a couple of different models, and the ones discussed below are quite opinionated — just make sure to set your own priorities early on.

#### GETTING READY AND SETTING GOALS [LINK](https://www.smashingmagazine.com/2016/12/front-end-performance-checklist-2017-pdf-pages/#getting-ready-and-setting-goals)

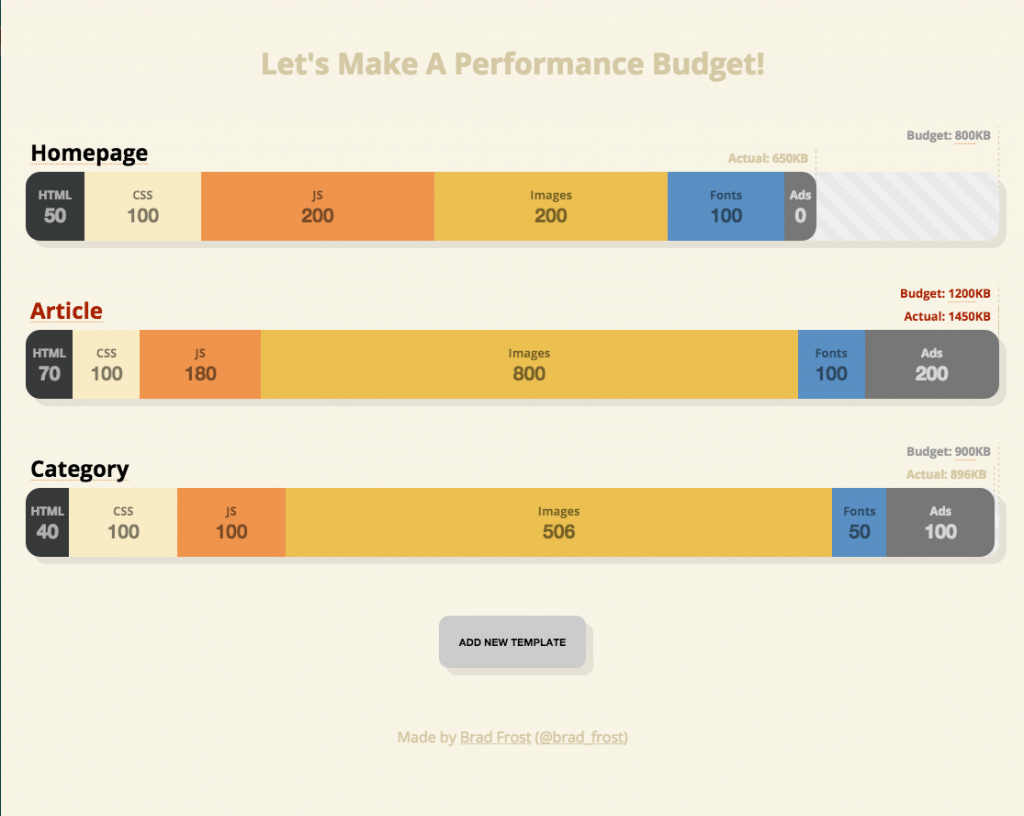
1. **Be 20% faster than your fastest competitor.**  
   According to [psychological research](https://www.smashingmagazine.com/2015/09/why-performance-matters-the-perception-of-time/#the-need-for-performance-optimization-the-20-rule), if you want users to feel that your website is faster than any other website, you need to be at least 20% faster. Full-page loading time isn’t as relevant as metrics such as start rendering time, the [first meaningful paint](https://developers.google.com/web/tools/lighthouse/audits/first-meaningful-paint) (i.e. the time required for a page to display its primary content) and the [time to interactive](https://developers.google.com/web/tools/lighthouse/audits/time-to-interactive) (the time at which a page — and primarily a single-page application — appears to be ready enough that a user can interact with it).

Measure start rendering (with [WebPagetest](http://www.webpagetest.org/)) and first meaningful paint times (with [Lighthouse](https://github.com/GoogleChrome/lighthouse)) on a Moto G, a mid-range Samsung device and a good middle-of-the-road device like a Nexus 4, preferably in an [open device lab](https://www.smashingmagazine.com/2016/11/worlds-best-open-device-labs/) — on regular 3G, 4G and Wi-Fi connections.



Lighthouse, a new performance auditing tool by Google.

Look at your analytics to see what your users are on. You can then mimic the 90th percentile’s experience for testing. Collect data, set up a [spreadsheet](http://danielmall.com/articles/how-to-make-a-performance-budget/), shave off 20%, and set up your goals (i.e. [performance budgets](http://bradfrost.com/blog/post/performance-budget-builder/)) this way. Now you have something measurable to test against. If you’re keeping the budget in mind and trying to ship down just the minimal script to get a quick time-to-interactive value, then you’re on a reasonable path.



[Performance budget builder](http://bradfrost.com/blog/post/performance-budget-builder/) by Brad Frost.

**Share the checklist with your colleagues.** Make sure that the checklist is familiar to every member of your team to avoid misunderstandings down the line. Every decision has performance implications, and the project would hugely benefit from front-end developers being actively involved when the concept, UX and visual design are decided on. Map design decisions against performance budget and the priorities defined in the checklist.

**100-millisecond response time, 60 frames per second.**  
The [RAIL performance model](https://www.smashingmagazine.com/2015/10/rail-user-centric-model-performance/) gives you healthy targets: Do your best to provide feedback in less than 100 milliseconds after initial input. To allow for <100 milliseconds response, the page must yield control back to main thread at latest after every <50 milliseconds. For high pressure points like animation, it’s best to do nothing else where you can and the absolute minimum where you can’t.

Also, each frame of animation should be completed in less than 16 milliseconds, thereby achieving 60 frames per second (1 second ÷ 60 = 16.6 milliseconds) — preferably under 10 milliseconds. Because the browser needs time to paint the new frame to the screen your code should finish executing before hitting the 16.6 milliseconds mark. [Be optimistic](http://info.meteor.com/blog/optimistic-ui-with-meteor-latency-compensation) and use the idle time wisely. Obviously, these targets apply to runtime performance, rather than loading performance.

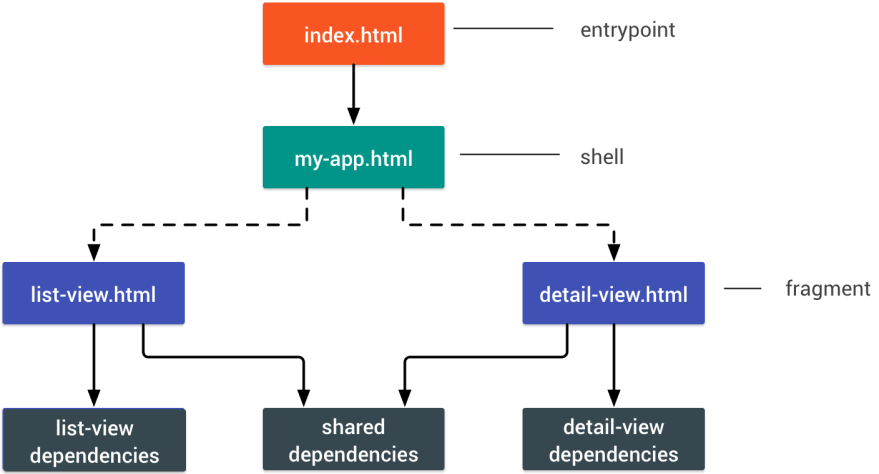
**First meaningful paint under 1.25 seconds, SpeedIndex under 1000.**  
Although it might be very difficult to achieve, your ultimate goal should be a start rendering time under 1 second and a [SpeedIndex](https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index) value under 1000 (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](https://www.soasta.com/blog/google-mobile-web-performance-study/). Being slightly above that is fine, but push to get these values as low as possible.

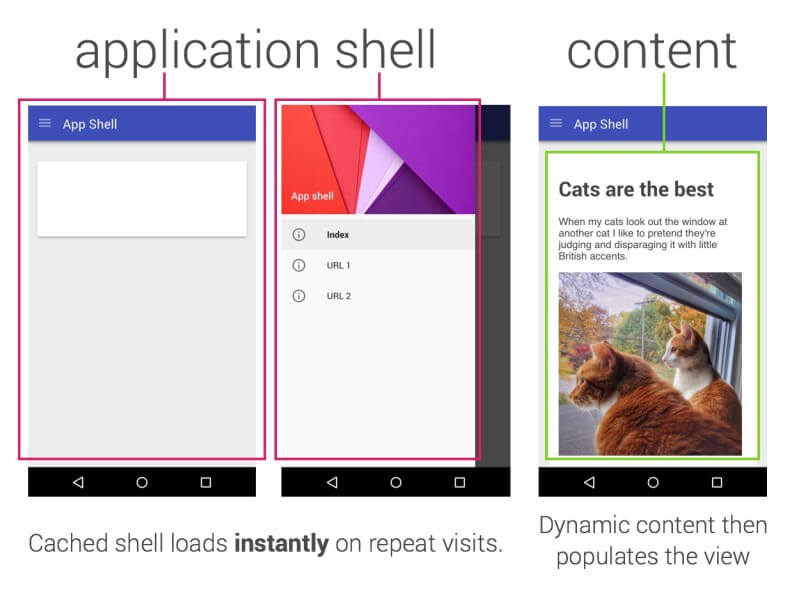
#### DEFINING THE ENVIRONMENT [LINK](https://www.smashingmagazine.com/2016/12/front-end-performance-checklist-2017-pdf-pages/#defining-the-environment)

**Choose and set up your build tools.**  
Don’t pay much attention to what’s supposedly cool these days. Stick to your environment for building, be it Grunt, Gulp, Webpack, PostCSS or a combination of tools. As long as you are getting results fast and you have no issues maintaining your build process, you’re doing just fine.

**Progressive enhancement.**  
Keeping [progressive enhancement](https://www.aaron-gustafson.com/notebook/insert-clickbait-headline-about-progressive-enhancement-here/) as the guiding principle of your front-end architecture and deployment is a safe bet. Design and build the core experience first, and then enhance the experience with advanced features for capable browsers, creating [resilient](https://resilientwebdesign.com/) 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.

**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 (because changing that afterwards, due to performance issues, can be extremely hard). If you do use a JavaScript framework, make sure your choice [is informed](https://www.youtube.com/watch?v=6I_GwgoGm1w) and [well considered](https://medium.com/@ZombieCodeKill/choosing-a-javascript-framework-535745d0ab90#.2op7rjakk). Different frameworks will have different effects on performance and will require different strategies of optimization, so you have to clearly understand all of the nuts and bolts of the framework you’ll be relying on. When building a web app, look into the [PRPL pattern](https://developers.google.com/web/fundamentals/performance/prpl-pattern/) and [application shell architecture](https://developers.google.com/web/updates/2015/11/app-shell).

*PRPL stands for Pushing critical resource, Rendering initial route, Pre-caching remaining routes and Lazy-loading remaining routes on demand.*



*An*[*application shell*](https://developers.google.com/web/updates/2015/11/app-shell)*is the minimal HTML, CSS, and JavaScript powering a user interface.*

**AMP or Instant Articles?**  
Depending on the priorities and strategy of your organization, you might want to consider using Google’s [AMP](https://www.ampproject.org/) or Facebook’s [Instant Articles](https://instantarticles.fb.com/). You can achieve good performance without them, but AMP does provide a solid performance framework with a free content delivery network (CDN), while Instant Articles will boost your performance on Facebook. You could build [progressive web AMPs](https://www.smashingmagazine.com/2016/12/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](https://www.smashingmagazine.com/2015/11/static-website-generators-jekyll-middleman-roots-hugo-review/), pushing it to a CDN and serving a static version from it, thus avoiding database requests. You could even choose a [static-hosting platform](https://www.smashingmagazine.com/2015/11/modern-static-website-generators-next-big-thing/) based on a CDN, enriching your pages with interactive components as enhancements ([JAMStack](https://jamstack.org/)).

Notice that CDNs can serve (and offload) dynamic content as well? So, restricting your CDN to static assets is not necessary. Double-check whether your CDN performs content compression and conversion, smart HTTP/2 delivery, edge-side includes, which assemble static and dynamic parts of pages at the CDN’s edge (i.e. the server closest to the user), and other tasks.

#### BUILD OPTIMIZATIONS [LINK](https://www.smashingmagazine.com/2016/12/front-end-performance-checklist-2017-pdf-pages/#build-optimizations)

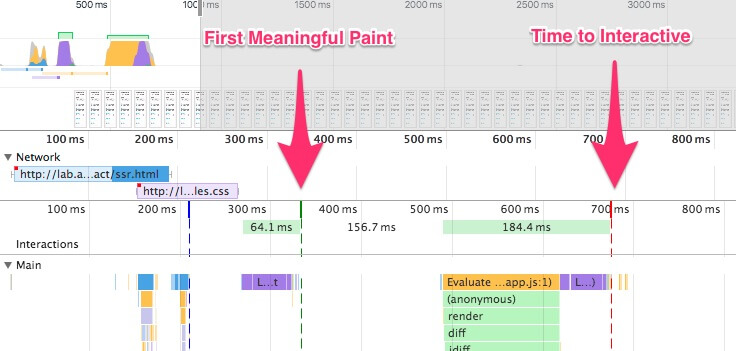
**Set your priorities straight.**  
It’s a good idea to know what you are dealing with first. Run an inventory of all of your assets (JavaScript, images, fonts, third-party scripts and “expensive” modules on the page, such as carousels, complex infographics and multimedia content), and break them down in groups.

Set up a spreadsheet. Define the basic **core** experience for legacy browsers (i.e. fully accessible core content), the **enhanced** experience for capable browsers (i.e. the enriched, full experience) and the **extras** (assets that aren’t absolutely required and can be lazy-loaded, such as web fonts, unnecessary styles, carousel scripts, video players, social media buttons, large images). We published an article on “[Improving Smashing Magazine’s Performance](https://www.smashingmagazine.com/2014/09/improving-smashing-magazine-performance-case-study/),” which describes this approach in detail.

**Use the “cutting-the-mustard” technique.**  
Use the [cutting-the-mustard technique](http://responsivenews.co.uk/post/18948466399/cutting-the-mustard) to send the core experience to legacy browsers and an enhanced experience to modern browsers. Be strict in loading your assets: Load the core experience immediately, the enhancements on DomContentLoaded and the extras on the loadevent.

Note that the technique deduces device capability from browser version, which is no longer something we can do these days. For example, cheap Android phones in developing countries mostly run Chrome and will cut the mustard despite their limited memory and CPU capabilities. Beware that, while we don’t really have an alternative, use of the technique has become more limited recently.

**Consider micro-optimization and progressive booting.**  
In some apps, you might need some time to initialize the app before you can render the page. Display [skeleton screens](https://twitter.com/lukew/status/665288063195594752) instead of loading indicators. Look for modules and techniques to speed up the initial rendering time (for example, [tree-shaking](https://medium.com/@richavyas/aha-moments-from-ngconf-2016-part-1-angular-2-0-compile-cycle-6f462f68632e#.8b9afnsub) and [code-splitting](https://webpack.github.io/docs/code-splitting.html)), because most performance issues come from the initial parsing time to bootstrap the app. Also, use an [ahead-of-time compiler](https://www.lucidchart.com/techblog/2016/09/26/improving-angular-2-load-times/) to [offload some of the client-side rendering](https://www.smashingmagazine.com/2016/03/server-side-rendering-react-node-express/) to the [server](http://redux.js.org/docs/recipes/ServerRendering.html) and, hence, output usable results quickly. Finally, consider using [Optimize.js](https://github.com/nolanlawson/optimize-js) for faster initial loading by wrapping eagerly invoked functions (it [might not be necessary](https://twitter.com/tverwaes/status/809788255243739136) any longer, though).

[*Progressive booting*](https://aerotwist.com/blog/when-everything-is-important-nothing-is/)*means using 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.*

Client-side rendering or server-side rendering? In both scenarios, our goal should be to set up [progressive booting](https://aerotwist.com/blog/when-everything-is-important-nothing-is/): 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. We can then, either on demand or as time allows, boot non-essential parts of the app. Unfortunately, as [Paul Lewis noticed](https://aerotwist.com/blog/when-everything-is-important-nothing-is/#which-to-use-progressive-booting), frameworks typically have no concept of priority that can be surfaced to developers, and hence progressive booting is difficult to implement with most libraries and frameworks. If you have the time and resources, use this strategy to ultimately boost performance.

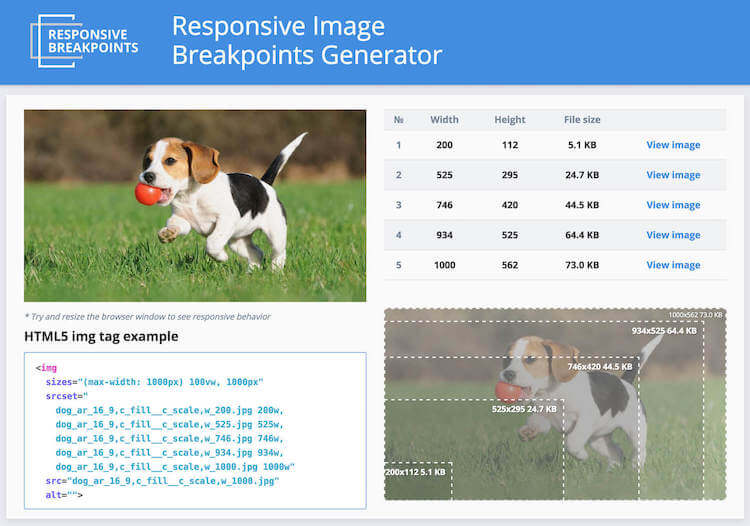
**Are HTTP cache headers set properly?**  
Double-check that expires, cache-control, max-age and other HTTP cache headers have been set properly. In general, resources should be cacheable either for a very short time (if they are likely to change) or indefinitely (if they are static) — you can just change their version in the URL when needed.

If possible, use Cache-control: immutable, designed for fingerprinted static resources, to avoid revalidation (as of December 2016, [supported only in Firefox](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control) on https:// transactions). You can use [Heroku’s primer on HTTP caching headers](https://devcenter.heroku.com/articles/increasing-application-performance-with-http-cache-headers), Jake Archibald’s “[Caching Best Practices](https://jakearchibald.com/2016/caching-best-practices/)” and Ilya Grigorik’s [HTTP caching primer](https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/http-caching?hl=en) as guides.

**Limit third-party libraries, and load JavaScript asynchronously.**  
When the user requests a page, the browser fetches the HTML and constructs the DOM, then fetches the CSS and constructs the CSSOM, and then generates a rendering tree by matching the DOM and CSSOM. If any JavaScript needs to be resolved, the browser won’t start rendering the page until it’s resolved, thus delaying rendering. As developers, we have to explicitly tell the browser not to wait and to start rendering the page. The way to do this for scripts is with the defer and async attributes in HTML.

In practice, it turns out we should [prefer defer to async](http://calendar.perfplanet.com/2016/prefer-defer-over-async/) (at a [cost to users of Internet Explorer](https://github.com/h5bp/lazyweb-requests/issues/42) up to and including version 9, because you’re likely to break scripts for them). Also, limit the impact of third-party libraries and scripts, especially with social sharing buttons and <iframe> embeds (such as maps). You can use [static social sharing buttons](https://www.savjee.be/2015/01/Creating-static-social-share-buttons/) (such as by [SSBG](https://simplesharingbuttons.com/)) and [static links to interactive maps](https://developers.google.com/maps/documentation/static-maps/intro) instead.

**Are images properly optimized?**  
As far as possible, use [responsive images](https://www.smashingmagazine.com/2014/05/responsive-images-done-right-guide-picture-srcset/) with srcset, sizes and the <picture> element. While you’re at it, you could also make use of the [WebP format](https://www.smashingmagazine.com/2015/10/webp-images-and-performance/) by serving WebP images with the <picture> element and a JPEG fallback (see Andreas Bovens’ [code snippet](https://dev.opera.com/articles/responsive-images/#different-image-types-use-case)) or by using content negotiation (using Accept headers). Sketch natively supports WebP, and WebP images can be exported from Photoshop using a [WebP plugin for Photoshop](http://telegraphics.com.au/sw/product/WebPFormat" \l "webpformat). [Other options are available](https://developers.google.com/speed/webp/docs/using), too.



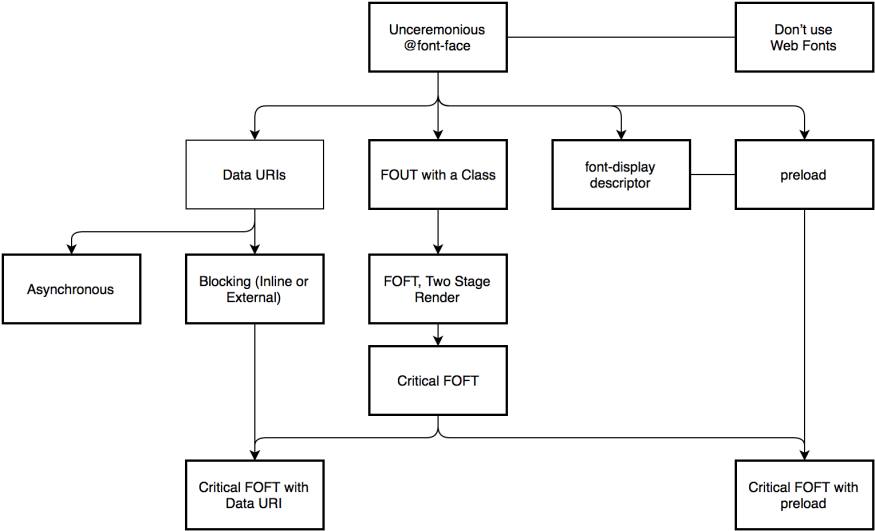
[*Responsive Image Breakpoints Generator*](http://www.responsivebreakpoints.com/)*automates images and markup generation.*

You can also use [client hints](https://www.smashingmagazine.com/2016/01/leaner-responsive-images-client-hints/), which are now [gaining browser support](http://caniuse.com/#search=client-hints). Not enough resources to bake in sophisticated markup for responsive images? Use the [Responsive Image Breakpoints Generator](http://www.responsivebreakpoints.com/) or a service such as [Cloudinary](http://cloudinary.com/documentation/api_and_access_identifiers) to automate image optimization. Also, in many cases, using srcset and sizes alone will reap significant benefits. On Smashing Magazine, we use the postfix -opt for image names — for example, brotli-compression-opt.png; whenever an image contains that postfix, everybody on the team knows that it’s been optimized.

1. **Take image optimization to the next level.**  
   When you’re working on a landing page on which it’s critical that a particular image loads blazingly fast, make sure that JPEGs are progressive and compressed with [mozJPEG](https://github.com/mozilla/mozjpeg) (which improves the start rendering time by manipulating scan levels), [Pingo](http://css-ig.net/pingo) for PNG, [Lossy GIF](https://kornel.ski/lossygif)for GIF and [SVGOMG](https://jakearchibald.github.io/svgomg/) for SVG. Blur out unnecessary parts of the image (by applying a Gaussian blur filter to them) to reduce the file size, and eventually you might even start to remove colors or make a picture black and white to reduce the size further. For background images, exporting photos from Photoshop with 0 to 10% quality can be absolutely acceptable as well.

Not good enough? Well, you can also improve perceived performance for images with the [multiple](http://csswizardry.com/2016/10/improving-perceived-performance-with-multiple-background-images/) [background](https://jmperezperez.com/medium-image-progressive-loading-placeholder/) [images](https://manu.ninja/dominant-colors-for-lazy-loading-images#tiny-thumbnails) [technique](https://css-tricks.com/the-blur-up-technique-for-loading-background-images/).

1. **Are web fonts optimized?**  
   Chances are high that the web fonts you are serving include glyphs and extra features that aren’t being used. You can ask your type foundry to subset web fonts or [subset them yourself](https://www.fontsquirrel.com/tools/webfont-generator) if you are using open-source fonts (for example, by including only Latin with some special accent glyphs) to minimize their file sizes. [WOFF2 support](http://caniuse.com/#search=woff2) is great, and you can use WOFF and OTF as fallbacks for browsers that don’t support it. Also, choose one of the strategies from Zach Leatherman’s “[Comprehensive Guide to Font-Loading Strategies](https://www.zachleat.com/web/comprehensive-webfonts/),” and use a service worker cache to cache fonts persistently. Need a quick win? Pixel Ambacht has a [quick tutorial and case study](https://pixelambacht.nl/2016/font-awesome-fixed/) to get your fonts in order.



*Zach Leatherman’s*[*Comprehensive Guide to Font-Loading Strategies*](https://www.zachleat.com/web/comprehensive-webfonts/)*provides a dozen of options for bettern web font delivery.*

If you can’t serve fonts from your server and are relying on third-party hosts, make sure to use [Web Font Loader](https://github.com/typekit/webfontloader). [FOUT is better than FOIT](https://www.filamentgroup.com/lab/font-events.html); start rendering text in the fallback right away, and load fonts asynchronously — you could also use [loadCSS](https://github.com/filamentgroup/loadCSS) for that. You might be able to [get away with locally installed OS fonts](https://www.smashingmagazine.com/2015/11/using-system-ui-fonts-practical-guide/) as well.

1. **Push critical CSS quickly.**  
   To ensure that browsers start rendering your page as quickly as possible, it’s become a [common practice](https://www.smashingmagazine.com/2015/08/understanding-critical-css/) to collect all of the CSS required to start rendering the first visible portion of the page (known as “critical CSS” or “above-the-fold CSS”) and add it inline in the <head> of the page, thus reducing roundtrips. Due to the limited size of packages exchanged during the slow start phase, your budget for critical CSS is around 14 KB. If you go beyond that, the browser will need addition roundtrips to fetch more styles. [CriticalCSS](https://github.com/filamentgroup/criticalCSS) and [Critical](https://github.com/addyosmani/critical)enable you to do just that. You might need to do it for every template you’re using. If possible, consider using the [conditional inlining approach](https://www.filamentgroup.com/lab/performance-rwd.html) used by the Filament Group.

With HTTP/2, critical CSS could be stored in a separate CSS file and delivered via a server push without bloating the HTML. The catch is that server pushing isn’t supported consistently and has some caching issues (see slide 114 onwards of [Hooman Beheshti’s presentation](http://www.slideshare.net/Fastly/http2-what-no-one-is-telling-you)). The effect could, in fact, [be negative](http://calendar.perfplanet.com/2016/http2-push-the-details/) and bloat the network buffers, preventing genuine frames in the document from being delivered. Server pushing is much [more effective on warm connections](https://docs.google.com/document/d/1K0NykTXBbbbTlv60t5MyJvXjqKGsCVNYHyLEXIxYMv0/edit) due to the TCP slow start. So, you might need to create a [cache-aware HTTP/2 server push mechanism](https://css-tricks.com/cache-aware-server-push/). Keep in mind, though, that the new [cache-digest specification](http://calendar.perfplanet.com/2016/cache-digests-http2-server-push/) will negate the need to manually build these “cache-aware” servers.

1. **Use tree-shaking and code-splitting to reduce payloads.**  
   [Tree-shaking](https://medium.com/@roman01la/dead-code-elimination-and-tree-shaking-in-javascript-build-systems-fb8512c86edf) is a way to clean up your build process by only including code that is actually used in production. You can [use Webpack 2 to eliminate unused exports](http://www.2ality.com/2015/12/webpack-tree-shaking.html), and [UnCSS](https://github.com/giakki/uncss) or [Helium](https://github.com/geuis/helium-css) to remove unused styles from CSS. Also, you might want to consider learning how to [write efficient CSS selectors](http://csswizardry.com/2011/09/writing-efficient-css-selectors/) as well as how to [avoid bloat and expensive styles](https://benfrain.com/css-performance-revisited-selectors-bloat-expensive-styles/).

[Code-splitting](https://webpack.github.io/docs/code-splitting.html) is another Webpack feature that splits your code base into “chunks” that are loaded on demand. Once you define split points in your code, Webpack takes care of the dependencies and outputted files. It basically enables you to keep the initial download small and to request code on demand, when requested by the application.

Note that [Rollup](http://rollupjs.org/) shows significantly better results than Browserify exports. While we’re at it, you might want to check out [Rollupify](https://github.com/nolanlawson/rollupify), which converts ECMAScript 2015 modules into one big CommonJS module — because small modules can have a [surprisingly high performance cost](https://nolanlawson.com/2016/08/15/the-cost-of-small-modules/)depending on your choice of bundler and module system.

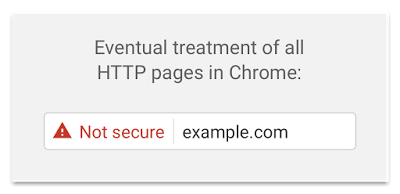
1. **Improve rendering performance.**  
   Isolate expensive components with [CSS containment](http://caniuse.com/#search=contain) — for example, to limit the scope of the browser’s styles, of layout and paint work for off-canvas navigation, or of third-party widgets. Make sure that there is no lag when 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 at least making the frames per second consistent is preferable to a mixed range of 60 to 15. Use CSS’ [will-change](http://caniuse.com/#feat=will-change) to inform the browser of which elements and properties will change.

Also, measure [runtime rendering performance](https://aerotwist.com/blog/my-performance-audit-workflow/#runtime-performance) (for example, [in DevTools](https://developers.google.com/web/tools/chrome-devtools/rendering-tools/)). To get started, check Paul Lewis’ free [Udacity course on browser-rendering optimization](https://www.udacity.com/course/browser-rendering-optimization--ud860). We also have a lil’ article by Sergey Chikuyonok on how to [get GPU animation right](https://www.smashingmagazine.com/2016/12/gpu-animation-doing-it-right/).

1. **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](https://w3c.github.io/resource-hints) to save time on [dns-prefetch](http://caniuse.com/" \l "search=dns-prefetch) (which performs a DNS lookup in the background), [preconnect](http://www.caniuse.com/" \l "search=preconnect) (which asks the browser to start the connection handshake (DNS, TCP, TLS) in the background), [prefetch](http://caniuse.com/" \l "search=prefetch)(which asks the browser to request a resource), [prerender](http://caniuse.com/" \l "search=prerender) (which asks the browser to render the specified page in the background) and [preload](https://www.smashingmagazine.com/2016/02/preload-what-is-it-good-for/) (which prefetches resources without executing them, among other things). Note that in practice, depending on browser support, you’ll prefer preconnect to dns-prefetch, and you’ll be cautious with using prefetch and prerender — the latter should only be used if you are very confident about where the user will go next (for example, in a purchasing funnel).

#### HTTP/2 [LINK](https://www.smashingmagazine.com/2016/12/front-end-performance-checklist-2017-pdf-pages/#http2)

1. **Get ready for HTTP/2.**  
   With Google [moving towards a more secure web](https://security.googleblog.com/2016/09/moving-towards-more-secure-web.html) and eventual treatment of all HTTP pages in Chrome as being “not secure,” you’ll need to decide on whether to keep betting on HTTP/1.1 or set up an [HTTP/2 environment](https://http2.github.io/faq/). HTTP/2 is [supported very well](http://caniuse.com/#search=http2); it isn’t going anywhere; and, in most cases, you’re better off with it. The investment will be quite significant, but you’ll need to move to HTTP/2 sooner or later. On top of that, you can get a [major performance boost](https://www.youtube.com/watch?v=RWLzUnESylc&t=1s&list=PLNYkxOF6rcIBTs2KPy1E6tIYaWoFcG3uj&index=25) with service workers and server push (at least long term).

[](https://security.googleblog.com/2016/09/moving-towards-more-secure-web.html)  
Eventually, Google plans to label all HTTP pages as non-secure, and change the HTTP security indicator to the red triangle that Chrome uses for broken HTTPS. ([Image source](https://security.googleblog.com/2016/09/moving-towards-more-secure-web.html))

The downsides are that you’ll have to migrate to HTTPS, and depending on how large your HTTP/1.1 user base is (that is, users on legacy operating systems or with legacy browsers), you’ll have to send different builds, which would require you to adapt a [different build process](https://rmurphey.com/blog/2015/11/25/building-for-http2). Beware: Setting up both migration and a new build process might be tricky and time-consuming. For the rest of this article, I’ll assume that you’re either switching to or have already switched to HTTP/2.

1. **Properly deploy HTTP/2.**  
   Again, [serving assets over HTTP/2](https://www.youtube.com/watch?v=yURLTwZ3ehk) requires a major overhaul of how you’ve been serving assets so far. You’ll need to find a fine balance between packaging modules and loading many small modules in parallel.

On the one hand, you might want to avoid concatenating assets altogether, instead breaking down your entire interface into many small modules, compressing them as a part of the build process, referencing them via the [“scout” approach](https://rmurphey.com/blog/2015/11/25/building-for-http2) and loading them in parallel. A change in one file won’t require the entire style sheet or JavaScript to be redownloaded.

On the other hand, [packaging still matters](http://engineering.khanacademy.org/posts/js-packaging-http2.htm) because there are issues with sending many small JavaScript files to the browser. First, **compression will suffer**. The compression of a large package will benefit from dictionary reuse, whereas small separate packages will not. There’s standard work to address that, but it’s far out for now. Secondly, browsers have **not yet been optimized** for such workflows. For example, Chrome will trigger [inter-process communications](https://www.chromium.org/developers/design-documents/inter-process-communication) (IPCs) linear to the number of resources, so including hundreds of resources will have browser runtime costs.



1. To achieve best results with HTTP/2, consider to [load CSS progressively](https://jakearchibald.com/2016/link-in-body/), as suggested by Chrome’s Jake Archibald.

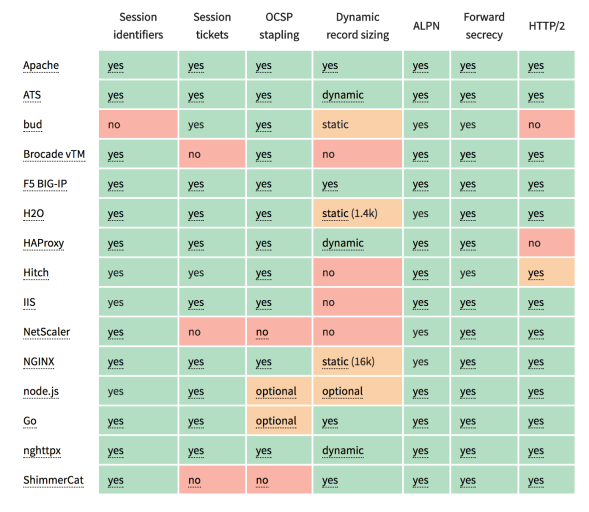
Still, you can try to [load CSS progressively](https://jakearchibald.com/2016/link-in-body/). Obviously, by doing so, you are actively penalizing HTTP/1.1 users, so you might need to generate and serve different builds to different browsers as part of your deployment process, which is where things get slightly more complicated. You could get away with [HTTP/2 connection coalescing](https://daniel.haxx.se/blog/2016/08/18/http2-connection-coalescing/), which allows you to use domain sharding while benefiting from HTTP/2, but achieving this in practice is difficult.

What to do? If you’re running over HTTP/2, 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.

1. **Make sure the security on your server is bulletproof.**  
   All browser implementations of HTTP/2 run over TLS, so you will probably want to avoid security warnings or some elements on your page not working. Double-check that your [security headers are set properly](https://securityheaders.io/), [eliminate known vulnerabilities](https://www.smashingmagazine.com/2016/01/eliminating-known-security-vulnerabilities-with-snyk/), and [check your certificate](https://www.ssllabs.com/ssltest/).

Haven’t migrated to HTTPS yet? Check [The HTTPS-Only Standard](https://https.cio.gov/faq/) for a thorough guide. Also, 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](https://www.owasp.org/index.php/HTTP_Strict_Transport_Security_Cheat_Sheet) and [Content Security Policy headers](https://content-security-policy.com/) are properly set.

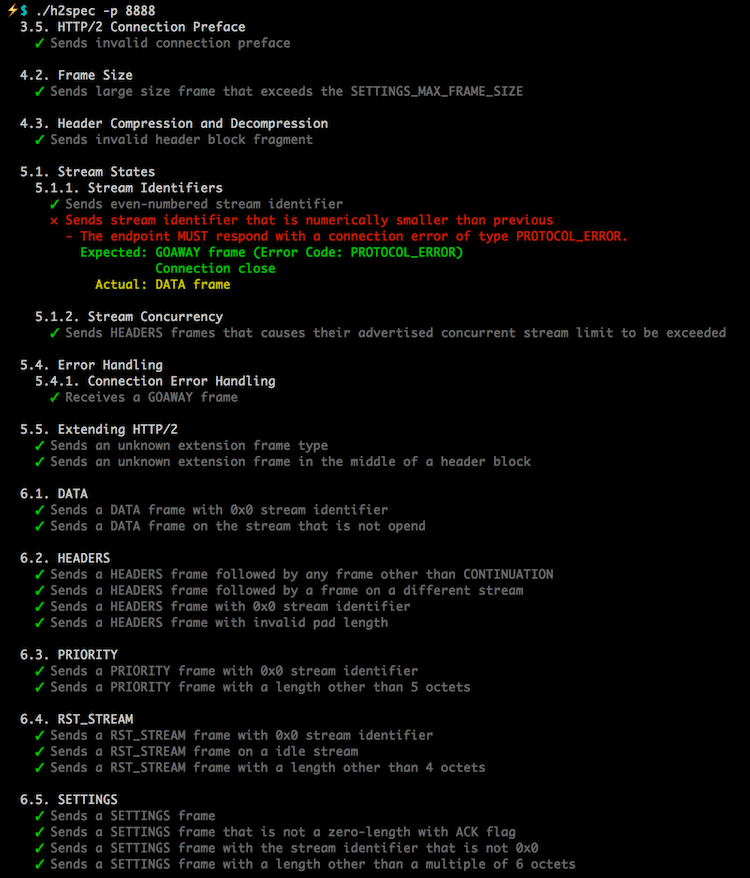
1. **Do your servers and CDNs support HTTP/2?**  
   Different servers and CDNs are probably going to support HTTP/2 differently. Use [Is TLS Fast Yet?](https://istlsfastyet.com/) to check your options, or quickly look up how your servers are performing and which features you can expect to be supported.

[](https://istlsfastyet.com/)  
[Is TLS Fast Yet?](https://istlsfastyet.com/) allows you to check your options for servers and CDNs when switching to HTTP/2.

1. **Is Brotli or Zopfli compression in use?**  
   Last year, Google [introduced](https://opensource.googleblog.com/2015/09/introducing-brotli-new-compression.html) [Brotli](https://github.com/google/brotli), a new open-source lossless data format, which is now [widely supported](http://caniuse.com/#search=brotli) in Chrome, Firefox and Opera. In practice, Brotli appears to be [more effective](https://samsaffron.com/archive/2016/06/15/the-current-state-of-brotli-compression) than Gzip and Deflate. It might be slow to compress, depending on the settings, and slower compression will ultimately lead to higher compression rates. Still, it decompresses fast. Because the algorithm comes from Google, it’s not surprising that browsers will accept it only if the user is visiting a website over HTTPS — and yes, there are technical reasons for that as well. The catch is that Brotli doesn’t come preinstalled on most servers today, and it’s not easy to set up without self-compiling NGINX or Ubuntu. However, you can [enable Brotli even on CDNs that don’t support it](http://calendar.perfplanet.com/2016/enabling-brotli-even-on-cdns-that-dont-support-it-yet/) yet (with a service worker).

Alternatively, you could look into using [Zopfli’s compression algorithm](https://blog.codinghorror.com/zopfli-optimization-literally-free-bandwidth/), which encodes data to Deflate, Gzip and Zlib formats. Any regular Gzip-compressed resource would benefit from Zopfli’s improved Deflate encoding, because the files will be 3 to 8% smaller than Zlib’s maximum compression. The catch is that files will take around 80 times longer to compress. That’s why it’s a good idea to use Zopfli on resources that don’t change much, files that are designed to be compressed once and downloaded many times.

1. **Is OCSP stapling enabled?**  
   By [enabling OCSP stapling on your server](https://www.digicert.com/enabling-ocsp-stapling.htm), you can speed up your TLS handshakes. The Online Certificate Status Protocol (OCSP) was created as an alternative to the Certificate Revocation List (CRL) protocol. Both protocols are used to check whether an SSL certificate has been revoked. However, 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.
2. **Have you adopted IPv6 yet?**  
   Because we’re [running out of space with IPv4](https://en.wikipedia.org/wiki/IPv4_address_exhaustion) and major mobile networks are adopting IPv6 rapidly (the US has [reached](https://www.google.com/intl/en/ipv6/statistics.html#tab=ipv6-adoption&tab=ipv6-adoption) a 50% IPv6 adoption threshold), it’s a good idea to [update your DNS to IPv6](https://www.paessler.com/blog/2016/04/08/monitoring-news/ask-the-expert-current-status-on-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. Also, [studies show](https://www.cloudflare.com/ipv6/) that IPv6 made those websites 10 to 15% faster due to neighbor discovery (NDP) and route optimization.
3. **Is HPACK compression in use?**  
   If you’re using HTTP/2, double-check that your servers [implement HPACK compression](https://blog.cloudflare.com/hpack-the-silent-killer-feature-of-http-2/) 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](https://github.com/summerwind/h2spec) is a great (if very technically detailed) tool to check that. [HPACK works](https://www.keycdn.com/blog/http2-hpack-compression/).

[](https://www.smashingmagazine.com/wp-content/uploads/2016/12/h2spec-example-large-opt.png)  
H2spec ([View large version](https://www.smashingmagazine.com/wp-content/uploads/2016/12/h2spec-example-large-opt.png)) ([Image source](https://github.com/summerwind/h2spec))

1. **Are service workers used for caching and network fallbacks?**  
   No performance optimization over a network can be faster than a locally stored cache on user’s machine. If your website is running over HTTPS, use the “[Pragmatist’s Guide to Service Workers](https://github.com/lyzadanger/pragmatist-service-worker)” to 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. Also, check Jake’s [Offline Cookbook](https://jakearchibald.com/2014/offline-cookbook/) and the free Udacity course “[Offline Web Applications](https://www.udacity.com/course/offline-web-applications--ud899).” Browser support? It’s [getting there](http://caniuse.com/#search=serviceworker), and the fallback is the network anyway.

#### TESTING AND MONITORING [LINK](https://www.smashingmagazine.com/2016/12/front-end-performance-checklist-2017-pdf-pages/#testing-and-monitoring)

1. **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 a tool such as [Report-URI.io](https://report-uri.io/). You can also use [Mixed Content Scan](https://github.com/bramus/mixed-content-scan) to scan your HTTPS-enabled website for mixed content.
2. **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 how to debug JavaScript and edit CSS styles. Umar Hansa recently prepared a (huge) [slidedeck](https://umaar.github.io/devtools-optimise-your-web-development-workflow-2016/" \l "/) and [talk](https://www.youtube.com/watch?v=N33lYfsAsoU) covering dozens of obscure tips and techniques to be aware of when debugging and testing in DevTools.
3. **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. UC Browser and Opera Mini, for instance, have a [significant market share in Asia](http://gs.statcounter.com/#mobile_browser-as-monthly-201511-201611) (up to 35% in Asia). [Measure average Internet speed](https://www.webworldwide.io/) in your countries of interest to avoid big surprises down the road. Test with network throttling, and emulate a high-DPI device. [BrowserStack](https://www.browserstack.com/) is fantastic, but test on real devices as well.
4. **Is continuous monitoring set up?**  
   Having a private instance of [WebPagetest](http://www.webpagetest.org/) 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 using [SpeedCurve](https://speedcurve.com/) to monitor changes in performance over time, and/or [New Relic](https://newrelic.com/browser-monitoring) to get insights that WebPagetest cannot provide. Also, look into [SpeedTracker](https://speedtracker.org/), [Lighthouse](https://github.com/GoogleChrome/lighthouse) and [Calibre](https://calibreapp.com/).

### 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 a 3G and cable connection.

1. Your goal is a start rendering time under 1 second on cable and under 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.

### Off We Go!

Some of the optimizations might be beyond the scope of your work or budget or might just be overkill given the legacy code you have to deal with. That’s fine! Use this checklist as a general (and hopefully comprehensive) guide, and create your own list of issues that apply to your context. But most importantly, test and measure your own projects to identify issues before optimizing. Happy performance results in 2017, everyone!